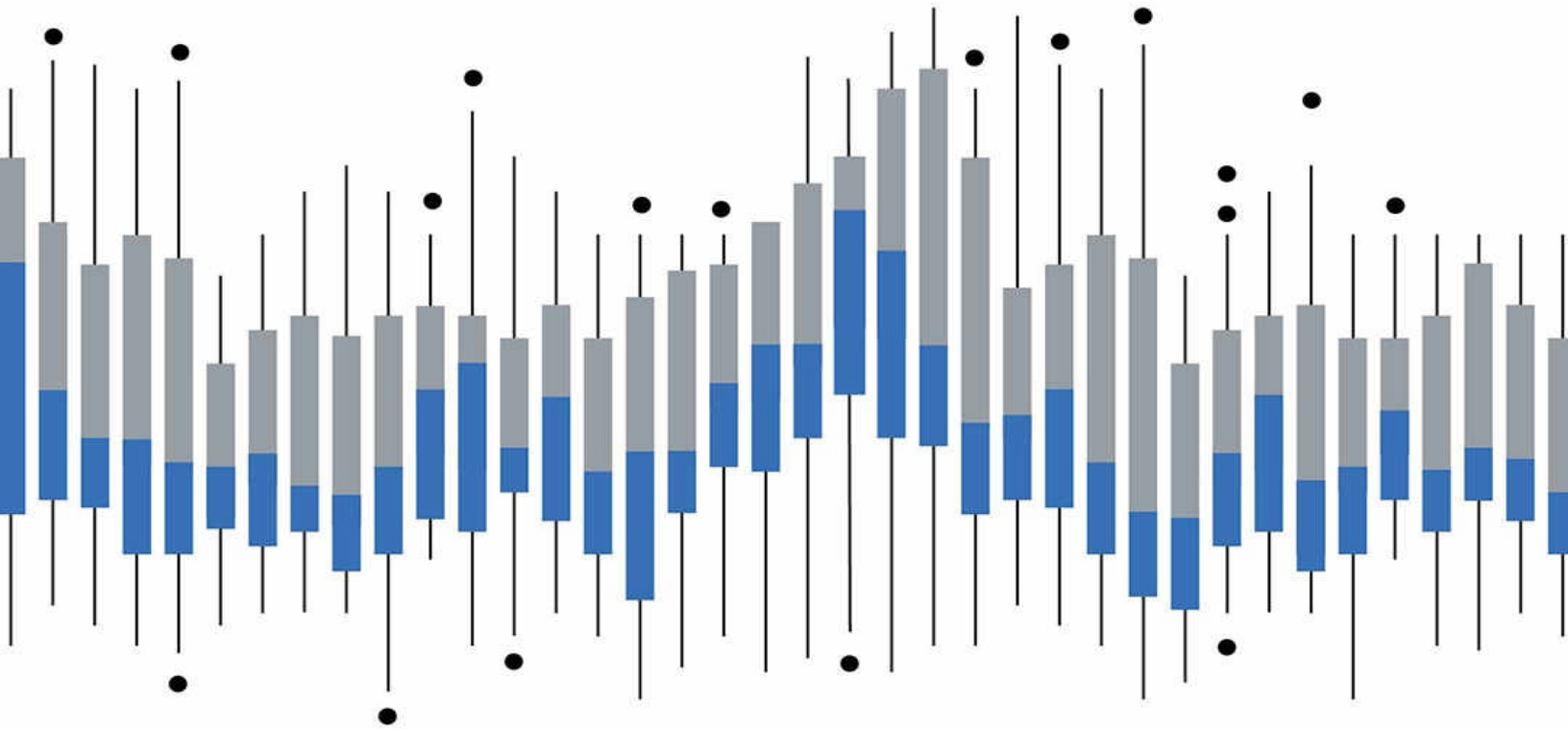
MARCELO S. PERLIN

PROCESSAMENTO E ANÁLISE DE DADOS FINANCEIROS E ECONÔMICOS COM O R

2ª edição



Processamento e Análise de Dados Financeiros e Econômicos com o R

Marcelo S. Perlin (marcelo.perlin@ufrgs.br)

2018-07-01

******ebook converter DEMO Watermarks*******

Processamento e Análise de Dados Financeiros e Econômicos com o R

******ebook converter DEMO Watermarks*******

Sobre Nova Edição

A primeira versão deste livro foi pioneira em apresentar um conteúdo introdutório sobre o uso do R em Finanças e Economia. Manter um conteúdo atualizado sobre a plataforma, porém, não é fácil! Desde o lançamento da obra, novos pacotes foram lançados e o R e Rstudio foram atualizados. Esta nova edição do livro complementa e estende o material anterior de forma abrangente, levando em conta novos procedimentos e tutoriais, novos pacotes e também o *feedback* recebido na primeira edição.

O formato do livro também mudou. A nova estrutura de capítulos alinha-se com as etapas de uma pesquisa, desde a obtenção dos dados, limpeza e estruturação, até a apresentação de resultados. Desta forma, o leitor pode avançar os capítulos de acordo com as etapas do seu próprio trabalho. Também buscou-se dar mais contexto sobre dados financeiros. No capítulo 5 trata-se sobre as principais fontes de dados, incluindo origem, contextualização e limitações. Isto permite ao leitor não somente aprender a baixar dados financeiros e econômicos da internet mas também a entender a particularidade de cada fonte. Um novo capítulo dedicado ao processo de reporte de resultados também foi adicionado, incluindo uma seção exclusiva para a tecnologia *RMarkdown*, a qual possibilita a criação de documentos dinâmicos.

Outra importante mudança no conteúdo do livro foi o alinhamento com o `tidyverse`, conjunto de funções e pacotes desenvolvidos por Hadley Wickhan. Esses facilitam imensamente o uso do R e não tenho dúvida que vieram para ficar. O código apresentado no livro foi totalmente modificado para dar preferência aos pacotes do `tidyverse`, principalmente na manipulação de dados. As funções nativas do R foram mantidas, porém são apresentadas de forma limitada.

Este livro foi escrito e será continuamente reformulado ao longo dos anos com bastante esforço por parte do autor. Vejo neste trabalho uma maneira de contribuir para a formação profissional de uma nova geração de pesquisadores e analistas de mercado. Espero que goste da obra e use-a para tornar o R um aliado no seu ciclo de trabalho.

******ebook converter DEMO Watermarks*******

Prefácio

Dado que se interessou por este livro, prevejo que você é um aluno de pós-graduação dando os primeiros passos em pesquisa empírica, ou é um profisional experiente procurando conhecer novas ferramentas para utilizar em seu trabalho e maximizar sua produtividade. Em ambos os casos, este livro é para você. **A finalidade e objeto deste trabalho é introduzir o leitor ao uso do R como ferramenta de computação e análise de dados, com uma ênfase especial para pesquisa empírica no tópico de Finanças e Economia.** Ao final deste livro você irá aprender como utilizar o R para importar dados financeiros, manipular estas informações e, por fim, reportar os resultados em um relatório.

Este livro é o resultado do meu trabalho como docente na Escola de Administração da Universidade Federal do Rio Grande do Sul. No programa de pós-graduação em Administração, eu leciono uma disciplina introdutória ao uso do R para resolver problemas de pesquisa na área de Finanças e Economia. Observando os alunos em sala de aula, percebo diariamente o impacto positivo que esse tipo de orientação tem em suas futuras carreiras profissionais, seja como pesquisadores acadêmicos, seja como analistas de dados em organizações públicas e privadas. Este livro é uma tentativa pessoal de registrar o meu conhecimento sobre a ferramenta e expandir o impacto de seu uso para um público maior e mais diversificado.

Outra motivação que tive para escrever o livro foi minha experiência na utilização de códigos disponibilizados por outros pesquisadores em pesquisas específicas. Na maioria das vezes, esses códigos são desorganizados, pouco claros e, possivelmente, funcionam apenas no computador do pesquisador que os escreveu! Após algumas tentativas frustradas, percebi que o trabalho de desvendar o código de outros professores levaria mais tempo do que desenvolver eu mesmo o procedimento, por mais complexo que ele fosse. Esses casos, aliás, ferem a ciência, pois um dos princípios básicos da pesquisa é a **replicabilidade** - isto é, uma rotina de pesquisa mal escrita irá reduzir a possibilidade de outras pessoas a utilizarem.

Assim como se espera que um artigo científico esteja bem escrito, também se deve esperar que o código por trás da respectiva pesquisa seja de qualidade. Porém, esse não é o caso na grande maioria das vezes. Com este livro, irei atacar esse problema, formalizando uma estrutura de código voltada à reprodutibilidade científica, focando em organização e usabilidade. O código resultante deve ser facilmente utilizável e extensível por outra pessoa. Nesse sentido, espero que as futuras gerações de pesquisadores estejam mais bem preparadas para compartilhar o seu trabalho.

Antes de prosseguir, um aviso. Não iremos trabalhar usos avançados do R. O conteúdo será limitado a exemplos simples e práticos de utilização do *software* para a construção de pesquisa baseada em dados financeiros e econômicos. De fato, um dos desafios na escrita deste livro foi definir o limite entre o material introdutório e o avançado. Procurei, sempre que possível, dosar gradualmente o nível de complexidade. Para leitores interessados em conhecer funções avançadas do programa e o seu funcionamento interno, sugiro a leitura do manual oficial do R (Teetor 2011) e as obras de Venables et al. (2004) e Wickham (2014a).

******ebook converter DEMO Watermarks*******

O livro está organizado com os seguintes capítulos:

Capítulo 1 - Introdução - Apresenta e justifica o R como plataforma de programação desenhada para resolver problemas relacionados a dados. Neste capítulo iremos apresentar os primeiros passos para o uso da ferramenta, incluindo a instalação dos programas necessários.

Capítulo 2 - Operações Básicas - Discute os comandos básicos do R e os recursos do RStudio. Estas são operações costumeiras e constituem a base de conhecimento para o uso eficiente do *software*. Inclui o tópico de criação de objetos, execução de *scripts*, formato local e internacional, recursos do RStudio e muito mais.

Capítulo 3 - Desenvolvendo Rotinas de Pesquisa - Discute as etapas e a estrutura de uma rotina de pesquisa, incluindo estágios, organização de pastas e políticas desejáveis para a melhor organização do trabalho.

Capítulo 4 - Importação e Exportação de Dados Locais - Este capítulo apresenta as funções mais comuns para importar e exportar dados de diferentes arquivos locais, includindo formatos nativos do R, planilhas eletrônicas, entre outros.

Capítulo 5 - Importação de Dados via Pacotes - Apresenta e discute o contexto de dados financeiros. Os principais pacotes disponíveis no CRAN para importar dados financeiros e econômicos são apresentados, incluindo pacotes desenvolvidos pelo próprio autor.

Capítulo 6 - Dataframes e outros Objetos - Discute o uso de dataframes para organizar e estruturar bases de dados em um formato eficiente. Inclui também a discussão sobre o uso de listas e matrizes.

Capítulo 7 - As Classes Básicas de Objetos - Apresenta as classes de objetos mais utilizadas, incluindo tipos numéricos, fatores, texto e Datas. Neste capítulo você aprenderá a usar as classes básicas para representar informações sobre seu problema de pesquisa e as possíveis manipulações para cada tipo de objeto.

Capítulo 8 - Programando com o R - Este capítulo apresenta as capacidades de programação do R, incluindo o uso de funções, *loops* e declarações condicionais. É dada especial ênfase ao uso das funções do R na manipulação de conjuntos de dados financeiros e econômicos.

Capítulo 9 - Estruturando e Limpando Dados - Apresenta a forma de estruturar e limpar dados com o R. Inclui a mudança de formato do dataframe (longo/largo), tratamento de *outliers*, desinflação de preços e mudança da frequência temporal de dados.

Capítulo 10 - Criando e Salvando Figuras - Trata do uso das funções do pacote `ggplot2` para criar visualizações do conjunto de dados, incluindo os casos mais comuns em finanças e economia, séries temporais e gráficos estatísticos.

Capítulo 11 - Modelos Econométricos no R - Apresenta o uso de seis modelos econométricos para pesquisa financeira e econômica. Inclui modelos lineares, GLM, dados em painel, Arima, Garch e modelos de Markov. Para cada tipo de modelo, aprenderemos como simular, estimar e prever. Sempre que apropriado, também aprenderemos a usar o R para efetivar testes estatísticos

******ebook converter DEMO Watermarks*******

relacionados aos modelos.

Capítulo 12 - Reportando Resultados e Criando Relatórios - Discute e apresenta maneiras de efetivar a última etapa da pesquisa: o reporte dos resultados obtidos. Isto inclui a apresentação e exportação de tabelas simples e estimações de modelos com pacotes específicos. Uma seção dedicada ao uso da tecnologia *RMarkdown*, mesclando código e texto, também é apresentada.

Capítulo 13 - A Performance de Investimentos no Brasil - O capítulo final apresenta um caso replicável de pesquisa, envolvendo todo o ciclo de análise, desde obtenção e estruturação dos dados até reporte dos resultados. Estuda-se a performance histórica de diferentes investimentos no mercado financeiro Brasileiro.

Sou um grande fã de código aberto e isso não será diferente neste livro. Todo o código utilizado aqui, incluindo exemplos separados por capítulos, está disponibilizado na internet. Também mantenho um site pessoal, onde busco escrever a respeito do uso do R em problemas específicos, entre outras coisas. Abaixo destaco os respectivos endereços:

Site do livro: https://sites.google.com/view/pafdr/home

Página pessoal e Blog: https://msperlin.github.io/

Espero que goste desta obra e encontre utilidade para o seu trabalho.

Boa leitura!

Marcelo S. Perlin

******ebook converter DEMO Watermarks*******

1 Introdução

Na era digital, informação é abundante e de fácil acesso. Nota-se, em todas as áreas de conhecimento, um crescimento substancial no registro de eventos em diferentes bancos de dados. A vida pessoal e profissional de qualquer cidadão é facilmente documentada no mundo digital. A cada segundo, volumosos bancos de dados de diferentes empresas e organizações são alimentados com novas informações. Essa tendência impõe uma alteração significativa na forma como organizações utilizam a análise de dados na tomada de decisões inteligentes. Sem dúvida, o período atual é bastante prolífico para profissionais com conhecimento e experiência na utilização das ferramentas corretas para a análise computacional de dados.

Em particular, a área de Economia e Finanças oferece grande variedade de informações disponíveis ao público. Instituições de pesquisa como IBGE, IPEA, Bancos Centrais, bolsas de valores e tantas outras, disponibilizam seus dados publicamente, seja por obrigatoriedade legal, seja para o próprio fomento da atividade de pesquisa. Há, ainda, outro fator importante que tem alavancado o mercado de análise de dados: o avanço tecnológico, acompanhado pela diminuição do custo computacional. Hoje em dia, computadores domésticos já possuem a capacidade de processar quantidades volumosas de dados em pouco tempo. Essa evolução do aparato computacional ocorreu mais rapidamente do que o acréscimo de complexidade das metodologias utilizadas, resultando em uma diminuição significativa do tempo necessário para a execução de uma pesquisa.

Os métodos de processamento também avançaram em termos de complexidade. No passado, uma simples planilha eletrônica poderia dar conta do trabalho. Hoje, a situação é bem diferente. Para áreas de conhecimento com aplicações práticas, como Economia e Finanças, é esperado que o aluno de pós-graduação tenha aprendido, no mínimo, uma linguagem de programação – como requerimento básico para a execução de sua pesquisa.

É nesse ambiente atual que se destaca o papel do R, uma linguagem de programação voltada para a resolução de problemas computacionais envolvendo análise, processamento, visualização e modelagem de dados. Nas próximas seções, explicarei o que é o R e quais são suas vantagens frente a outras alternativas.

******ebook converter DEMO Watermarks*******

1.1 O que é o R

O R é uma linguagem de programação voltada para a resolução de problemas estatísticos e para a visualização gráfica de dados. O código base do R foi inicialmente criado no laboratório da **Bell/AT&T** por **John Chambers** e seus colegas, com base na linguagem *S*. Esse código foi reaproveitado por dois acadêmicos, **Ross Ihaka** e **Robert Gentleman**, resultando na plataforma de programação que temos hoje. Para os curiosos, o nome R foi escolhido devido ao compartilhamento da primeira letra do nome de seus criadores.

Hoje, R é sinônimo de programação voltada à análise de dados, com uma larga base de usuários e funções bem estabelecidas. É muito provável que pesquisadores de áreas diversas, desde Economia até Biologia, ou mesmo Música, encontrem no R uma quantidade significativa de códigos que facilitem suas análises. No campo empresarial, grandes empresas como *Google* e *Microsoft* já o adotaram como a linguagem interna para a análise de dados. O R é atualmente mantido pelo **R Foundation** e o **R Consortium**, um esforço coletivo para financiar projetos de extensão da linguagem.

E o mais importante: **o R é totalmente livre** e disponível em vários sistemas operacionais. Seja você um usuário do Windows, do Linux/UNIX ou do Mac-OS, existe uma instalação do R para a sua plataforma, e os seus códigos devem rodar entre uma e outra com mínimas alterações.

******ebook converter DEMO Watermarks*******

1.2 Por que Escolher o R

O processo de aprendizado de uma nova linguagem de programação exige muito tempo e esforço. Portanto, é importante entender as razões por trás dessa escolha. Possivelmente você esteja se perguntado por que deve optar pelo R e investir tempo em sua aprendizagem, ao invés de escolher uma outra linguagem.

Em primeiro lugar, **o R é uma plataforma madura, estável, continuamente suportada e intensamente utilizada na indústria.** Ao escolher o R, você terá a bagagem computacional necessária não somente para uma carreira acadêmica em pesquisa científica, mas também para o trabalho em organizações privadas. Nesse sentido, com a escolha de outra linguagem de programação menos popular ou proprietária (comercial), é provável que tal linguagem não seja utilizada em um ambiente empresarial e isso pode limitar as suas oportunidades futuras de carreira. O conhecimento do R certamente aumenta a sua atratividade como um professional em Finanças e Economia. Além disso, o R é continuamente suportado pela comunidade, com rápida correção de *bugs* e adição de novas funcionalidades.

Aprender a linguagem do R é fácil. A experiência que tenho ensinando o R em sala de aula me permite afirmar que os alunos, mesmo aqueles sem experiência em programação, apresentam facilidade em aprender a linguagem e em utilizá-la para criar seus próprios códigos de pesquisa. A linguagem é intuitiva e certas normas e funções podem ser estendidas para diferentes casos. Após entender como o programa funciona, fica fácil descobrir novas funcionalidades partindo de uma lógica anterior. Essa notação compartilhada entre procedimentos facilita o aprendizado.

A interface do R e RStudio torna o uso da ferramenta bastante produtivo. A interface gráfica disponibilizada pelo RStudio facilita o uso do software, assim como a produtividade do usuário. Utilizando o ambiente de trabalho do R e RStudio, têm-se a disposição diversas ferramentas que facilitam e estendem o uso da plataforma.

Os pacotes do R permitem as mais diversas funcionalidades. Logo veremos que o R permite o uso de código de outros usuários, os quais podem ser localmente instalados na forma de um pacote com um simples comando. Esses estendem a linguagem básica do R e possibilitam as mais diversas funcionalidades. Podes, por exemplo, utilizar o R para escrever e publicar um livro, criar provas objetivas com conteúdo dinâmico, contar piadas e poemas (é sério!), mandar emails, acessar e coletar dados da internet, entre diversas outras funcionalidades.

O R tem compatibilidade com diferentes linguagens e sistemas operacionais. Se, por algum motivo, o usuário precisar utilizar código de outra linguagem de programação tal como *C++*, *Python*, *Julia*, é fácil integrar a mesma dentro de um programa do R. Diversos pacotes estão disponíveis para facilitar esse processo. Portanto, o usuário nunca fica restrito a uma única linguagem e tem alta flexibilidade para escolher as suas ferramentas.

O R é totalmente gratuito! O programa e todos os seus pacotes são completamente livres, não tendo custo algum de licença e distribuição. Portanto, você pode utilizá-lo e modificá-lo livremente no seu trabalho ou computador pessoal. Essa é uma razão muito forte para a adoção da linguagem em um ambiente empresarial, onde a obtenção de licenças individuais e coletivas

******ebook converter DEMO Watermarks*******

de outros softwares comerciais pode ter um alto custo financeiro.

******ebook converter DEMO Watermarks*******

1.3 Usos do R

O R é uma linguagem de programação completa e qualquer problema computacional pode ser resolvido com base nela. Dada a adoção do R por diferentes áreas de conhecimento, a lista de possibilidades é extensa. Para o caso de Finanças e Economia, destaco abaixo possíveis utilizações do programa:

- Importação, exportação, tratamento e armazenamento de dados financeiros e econômicos com base em arquivos locais ou da internet;
- Criação de rotinas para o cálculo e a administração do risco de uma carteira de investimento;
- Desenvolvimento de rotinas para a administração e execução de ordens financeiras no mercado de capitais;
- Criação de ferramentas para controle, avaliação e divulgação de índices econômicos sobre um país ou região;
- Execução de diversas possibilidades de pesquisa empírica através da estimação de modelos econométricos e testes de hipóteses;
- Criação de *websites* dinâmicos com o pacote `Shiny`, possibilitando que qualquer pessoa no mundo utilize uma ferramenta criada por você;
- Organização de um processo automatizado de criação e divulgação de relatórios técnicos com o pacote `knitr` e a tecnologia *RMarkdown*.

Além dos usos destacados anteriormente, o acesso público a pacotes desenvolvidos por usuários expande ainda mais essas funcionalidades. O site da CRAN (*Comprehensive R Archive Network*) oferece um *Task Views* do software para o tópico de Finanças. No link é possível encontrar os principais pacotes disponíveis para realizar algum tipo de operação na área de Finanças e Economia. Isso inclui a importação de dados financeiros da internet, a estimação de um modelo econométrico específico, cálculos de diferentes estimativas de risco, entre várias outras possibilidades. A leitura dessa página e o conhecimento desses pacotes são essenciais para aqueles que pretendem trabalhar com Finanças e Economia. Vale destacar, porém, que essa lista apresenta apenas os principais itens. A lista completa de pacotes é muito maior do que o apresentado no *Task Views*. O link para o site do CRAN está disponibilizado na página do livro.

******ebook converter DEMO Watermarks*******

1.4 Como Instalar o R

O R é instalado no seu sistema operacional como qualquer outro programa. A maneira mais direta e funcional de instalá-lo é ir ao website do R em https://www.r-project.org/ e clicar no *link CRAN* do painel *Download*, conforme mostrado na figura a seguir.

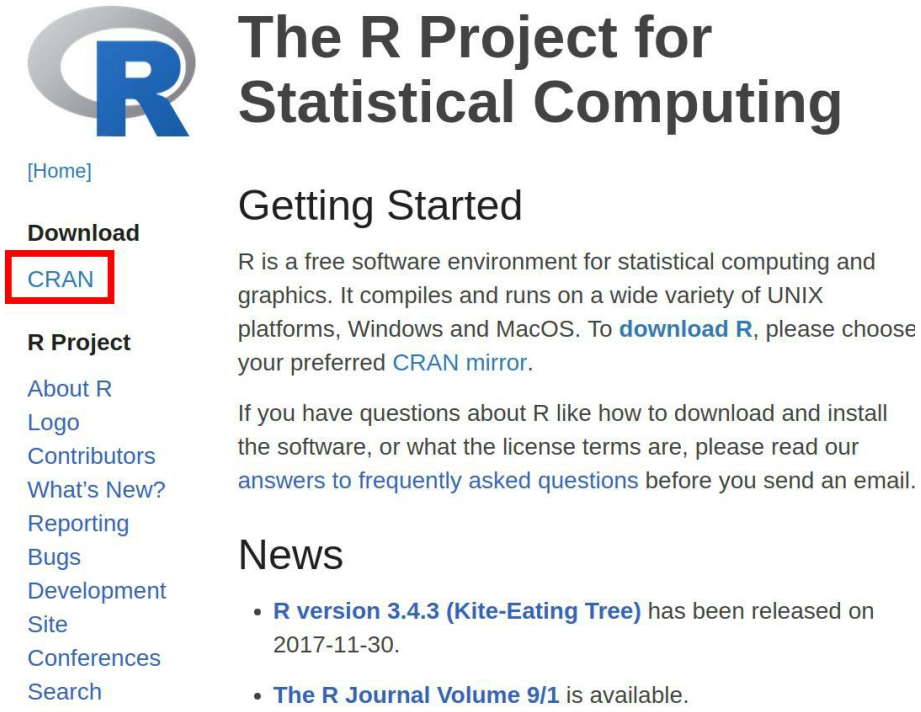


Figura 1.1 Página inicial para o download do R

A próxima tela apresenta a escolha do espelho para baixar os arquivos de instalação. O repositório do CRAN é espelhado em diversas partes do mundo, permitindo acesso rápido para os usuários. Para a grande maioria dos leitores deste livro, essa localidade deve ser o Brasil ou Portugal. Portanto, você pode escolher um dos links da instituição mais próxima, tal como o da UFPR (Universidade Federal do Paraná). Em caso de dúvida, escolha o repositório do RStudio (veja Figura), o qual automaticamente direciona para o local mais próximo.

******ebook converter DEMO Watermarks*******

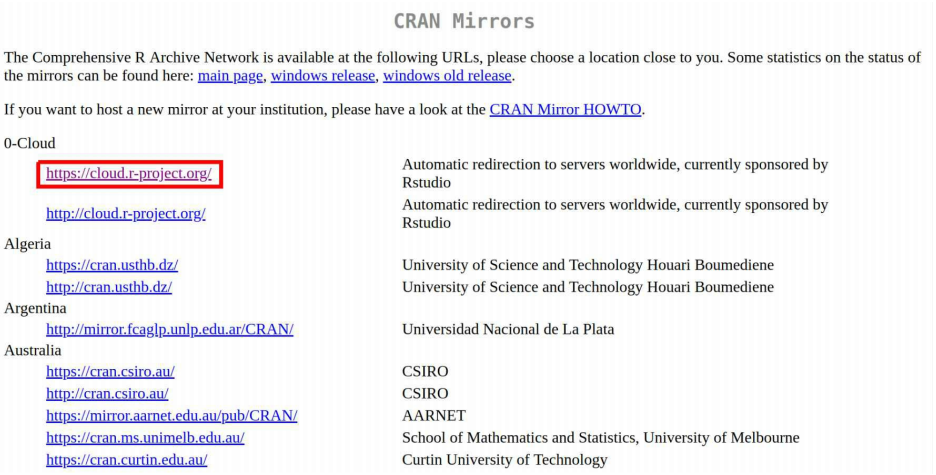


Figura 1.2 Tela com a escolha do espelho para o download

O próximo passo é selecionar o sistema operacional do computador. Devido à maior popularidade da plataforma *Windows*, a partir de agora daremos enfoque à instalação do R nesse sistema. As instruções de instalação nos demais sistemas operacionais podem ser facilmente encontradas na internet. Destaca-se que, independente da plataforma, o modo de uso do R é o mesmo. Existem, porém, algumas exceções, principalmente quando o R interage com o sistema de arquivos. Essas exceções serão destacadas no decorrer do livro. Assim, mesmo que você esteja utilizando Linux ou Mac, poderá tirar proveito do material aqui apresentado.

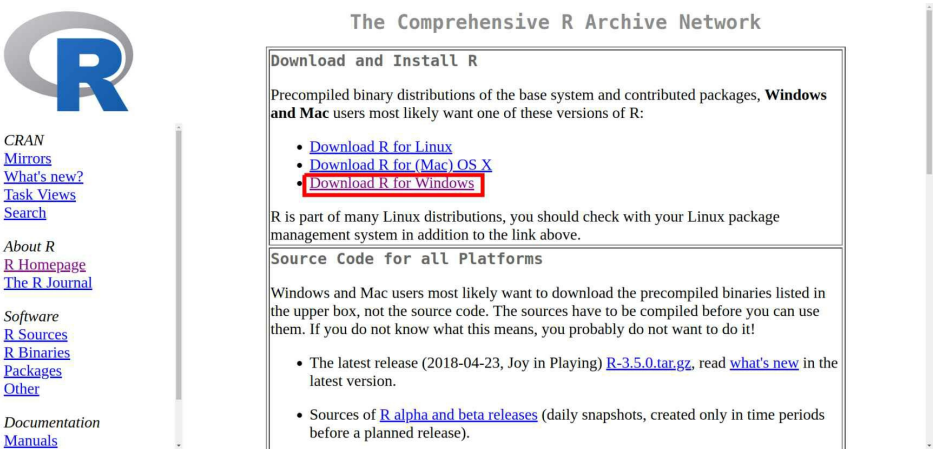


Figura 1.3 Tela com a escolha do sistema operacional

Após clicar no link *Download R for Windows*, a próxima tela irá mostrar as seguintes opções de *download*: *base*, *contrib*, *old.contrib* e *RTools*. Dentre as opções de *download*, a primeira (*base*) deve ser selecionada. O *link* acessa a instalação básica do R para *Windows*. Caso o usuário tenha interesse em criar e distribuir os seus próprios pacotes de R, deve ser instalado também o programa *RTools*. Para a maioria dos usuários, contudo, esse não deve ser o caso e, portanto, sugiro ignorar esse programa por enquanto.

******ebook converter DEMO Watermarks*******

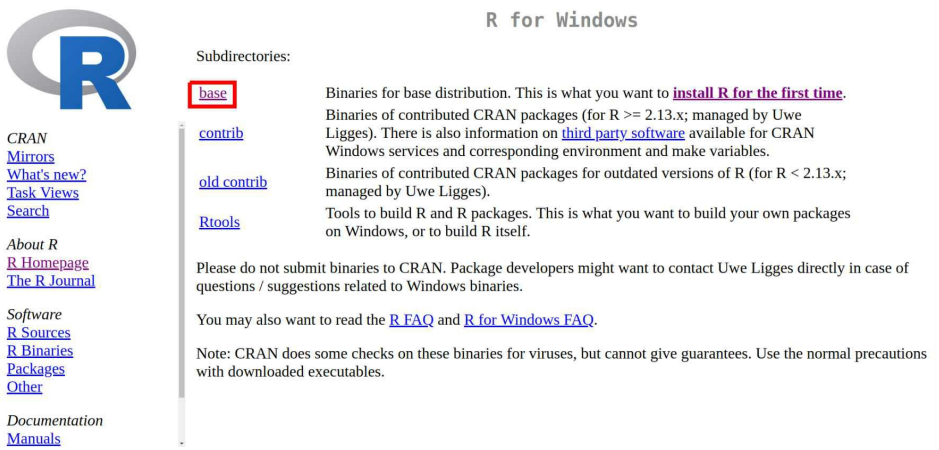


Figura 1.4 Tela com opções de instalação

Após clicar no link *base*, a próxima tela mostrará o link para o *download* do arquivo de instalação do R no *Windows*.

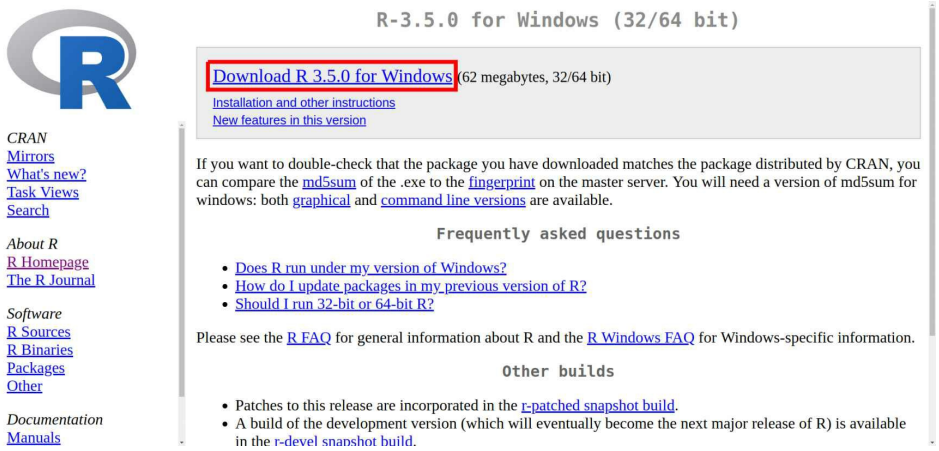


Figura 1.5 Tela para o download do R

Após baixar o arquivo, abra-o e siga os passos da tela de instalação do R. Escolha a língua inglesa em todas etapas do processo. O uso da língua inglessa não é acidental. Este é a melhor forma, mesmo para iniciantes, de se aprender a usar o R. É possível instalar uma versão em português porém isso limita o potencial da ferramenta. Caso não for fluente em inglês, não se preocupe, o vocabulário necessário é básico. Neste momento, nenhuma outra configuração especial é necessária. Sugiro manter todas as escolhas padrão selecionadas e simplesmente ir aceitando as telas de diálogo. Após a instalação do R, partimos para a instalação do RStudio.

******ebook converter DEMO Watermarks*******

1.5 Instalando o RStudio

A instalação do R inclui a sua própria interface gráfica, um programa que facilita a edição e execução de nossos *scripts*. Essa, porém, possui várias limitações. O RStudio é um *software* que torna o uso e o visual do R muito mais prático e eficiente. Uma forma de entender essa relação é com uma analogia com carros. Enquanto o R é o motor da linguagem de programação, o RStudio é a carroceria e o painel de instrumentos, os quais melhoram significativamente a experiência de uso. Além de apresentar um visual mais atrativo, o RStudio também é acrescido de várias funcionalidades que facilitam a vida do usuário, possibilitando a construção de projetos e pacotes do próprio R, a criação de documentos dinâmicos (*Sweave/knitr*) e a interface com edição de textos em *LaTeX*, entre várias outras. Assim como o R, **o RStudio também é gratuito** e pode ser utilizado no ambiente empresarial.

A instalação do RStudio é mais simples do que a do R. Os arquivos estão disponíveis no endereço disponibilizado no site do livro. Após acessar a página, clique em *Download RStudio* e depois em *Download Rstudio Desktop*. Logo após, basta selecionar o arquivo relativo ao sistema operacional em que você irá trabalhar. Provavelmente, essa opção será *Windows Vista/7/8/10*. Note que, assim como o R, o RStudio também está disponível para diferentes plataformas.

Destaco que o uso do RStudio não é essencial para desenvolver programas no R. Outros softwares de interface estão disponíveis e podem ser utilizados. Porém, dada minha experiência atual, o RStudio é o programa de interface que oferece a maior variedade de funcionalidades para essa linguagem, além de ser amplamente utilizado, o que justifica a sua escolha.

******ebook converter DEMO Watermarks*******

1.6 Uso do `tidyverse`

Logo veremos que o R possui um sistema de pacotes que estendem as funcionalidade do programa. Dentre essas, existe um grupo de pacotes desenvolvidos por Hadley Wickham e diversos outros autores que representam o `tidyverse`. As funções desses pacotes substituem as funções nativas do R em procedimentos comuns, tal como leitura de arquivos e manipulação de objetos. A inovação é que as funções seguem uma filosofia chamada `tidydata` (Wickham 2014b), sendo, muitas vezes, mais fáceis de usar do que as funções nativas.

A minha experiência com o `tidyverse` é bastante positiva. Prevejo que o uso do `tidyverse` irá se solidificar ainda mais no futuro. Devido a isso, neste livro irei dar prioridade para as funções e pacotes do `tidyverse` em relação a funções nativas do R. Irei, porém, sempre destacar as funções equivalentes no sistema nativo. Isto é importante pois nunca se sabe quais funções serão encontradas no código de outras pessoas. Essas serão sinalizadas com o prefixo `base::`, como em `base::paste`. Explicando melhor, lê-se o código em `base::paste` como função `paste` do pacote `base`. Essa notação não é acidental. Logo veremos como ela se encaixa na linguagem do R.

******ebook converter DEMO Watermarks*******

1.7 Recursos na Internet

A comunidade R é viva e envolvente. Na internet é possível encontrar uma diversidade de material sobre o uso do R. Diversos usuários, assim como o próprio autor do livro, publicam material sobre o uso R em seus blogs. Isso inclui anúncios de pacotes, publicações sobre análise de dados na vida real, curiosidades, novidades e tutoriais. R-Bloggers é um site que agrega esses blogs em um único local, tornando mais fácil para qualquer um acessar e participar. O conteúdo do R-Bloggers, porém, é todo em inglês.

Recentemente, uma versão local de agregamento de blogs sobre R por autores Brasileiros e com conteúdo em português foi organizada por Daniel Falbel e Julio Trecenti. O acesso é disponível no site https://brbloggers.com.br/. Outra fonte de informações é a lista de blogs compilada e organizada por Marcos Vital em https://marcosvital.github.io/blogs-de-R-no-Brasil/. Eu recomendo a inscrição no feed do R-Bloggers e BR-Bloggers. Não só você será informado sobre o que está acontecendo no universo do R, mas também aprenderá muito lendo artigos e os códigos de outros usuários.

Aprender e usar R pode ser uma experiência social. Várias conferências e grupos de usuários estão disponíveis em muitos países, incluindo o Brasil. O grupo *R Brasil - Programadores* no Facebook é bastante ativo, com um grande número de participantes. Recomendo fortemente a inscrição neste grupo para acompanhar e discutir questões relacionadas ao uso do R. Diversas conferências locais sobre o R são divulgadas nesse grupo.

******ebook converter DEMO Watermarks*******

1.8 Organização e Material do Livro

Este livro tem uma abordagem prática no uso do R e será acompanhado por uma série de códigos que irão exemplificar e mostrar para o leitor as funcionalidades do programa. Para tirar o máximo de proveito do material, sugiro que você primeiro busque entender o código mostrado e, somente então, tente utilizá-lo em seu próprio computador. O índice remissivo disponibilizado no final do livro serve como uma mapa de uso das funções. Toda vez que uma função é chamada no livro, um registro do número da página é criado no índice remissivo. Esse indica, também, o pacote que a função pertence. Podes utilizar este mapa para localizar o uso de qualquer função ou pacote no decorrer do livro.

Aprender a programar em uma nova linguagem é como aprender uma língua estrangeira: o uso no dia-a-dia é de extrema importância para criar fluência. Sempre que possível, sugiro que o leitor teste o código no seu computador e *brinque* com o mesmo, modificando os exemplos dados e verificando o efeito das modificações nas saídas do programa. Cada capítulo apresenta no seu final uma lista de exercícios, incluindo questões do tipo desafio. Podes testar as suas habilidades de programação resolvendo as atividades propostas. Vale relembrar que todo o código deste livro está disponibilizado na internet. Não precisas, portanto, escrever o código diretamente do livro. Podes copiar e colar do código fonte disponibilizado no site.

No decorrer da obra, toda demonstração de código terá duas partes: o código em si e sua saída do R. Essa saída nada mais é do que o resultado dos comandos na tela do programa. Todas as entradas e saídas de código serão sinalizadas no texto com um formato especial. Veja o exemplo a seguir:

```
# create a list
object.x <- list(var1 = 'abc', var2 = 1:5)

# print to prompt
print(object.x)
```

```
## $var1
## [1] "abc"
##
## $var2
## [1] 1 2 3 4 5
```

No caso anterior, os textos `object.x <- list(var1 = 'abc', var2 = 1:5)` e `print(object.x)` são os códigos de entrada. A saída do programa é a apresentação na tela dos elementos de `object.x`, com o símbolo antecessor `##`. Por enquanto não se preocupe em entender e reproduzir o código utilizado acima. Iremos tratar disso no próximo capítulo.

Note que faço uso da língua inglesa no código, tanto para a nomeação de objetos quanto para os comentários. Isso não é acidental. O uso da língua inglesa facilita o desenvolvimento de código ao evitar caracteres latinos, além de ser uma das línguas mais utilizadas no mundo. Portanto, é importante já ir se acostumando com esse formato. O vocabulário necessário, porém, é limitado. De forma alguma precisarás ter fluência em inglês para entender o código.

******ebook converter DEMO Watermarks*******

O código também pode ser espacialmente organizado usando novas linhas. Esse é um procedimento comum em torno de argumentos de funções. O próximo pedaço de código é equivalente ao anterior, e executará exatamente da mesma maneira. Observe como usei uma nova linha para alinhar verticalmente os argumentos da função `list`. Você verá em breve que, ao longo do livro, esse tipo de alinhamento vertical é constantemente usado.

```
# create a list
object.x <- list(var1 = 'abc',
                 var2 = 1:5)

# print to prompt
print(object.x)
```

```
## $var1
## [1] "abc"
##
## $var2
## [1] 1 2 3 4 5
```

O código também segue uma estrutura bem definida. Uma das decisões a ser feita na escrita de códigos de computação é a forma de nomear os objetos e como lidar com a estrutura do texto do código em geral. É recomendável seguir um padrão limpo de código, de forma que o mesmo seja fácil de ser mantido ao longo do tempo e de ser entendido por outros usuários. Para este livro, foi utilizado uma mistura de escolhas pessoais do autor com o estilo de código sugerido pelo Google em https://google.github.io/styleguide/Rguide.xml. O usuário, porém, é livre para escolher a estrutura que achar mais eficiente. Sugiro que se conheçam essas normas de estrutura de código no R, de forma a maximizar a usabilidade do mesmo. Iremos voltar a esse assunto no próximo capítulo.

******ebook converter DEMO Watermarks*******

2 Operações Básicas no R

Antes de começar a desenvolver o seu código, é necessário entender a forma de trabalhar com o R e o RStudio. Isso inclui os componentes da linguagem, os diferentes tipos de objetos e as operações que podem ser realizadas com base nos comandos existentes. Entender o uso do R e como tirar vantagem da sua linguagem é essencial para o desenvolvimento do material futuro. Este é o principal capítulo para aqueles que estão se familiarizando com o uso da ferramenta.

Neste capítulo iremos percorrer os passos iniciais sobre o ponto de vista de alguém que nunca trabalhou com o R e, possivelmente, nunca teve contato com outra linguagem de programação. Aqueles já familiarizados com o programa irão encontrar pouca informação nova e, portanto, sugiro a leitura da próxima seção. É recomendado, porém, que no mínimo esse usuário verifique os temas tratados para confirmar ou não o seu conhecimento sobre as funcionalidades do programa. Muitas das sugestões apresentadas aqui tem potencial de aumentar significativamente a sua produtividade no trabalho.

******ebook converter DEMO Watermarks*******

2.1 Como o R Funciona?

A maior dificuldade que um usuário iniciante possui ao começar a desenvolver rotinas com o R é a forma de trabalho. A nossa interação com computadores foi simplificada ao longo dos anos e atualmente estamos confortáveis com o formato de interação do tipo *aponte e clique*. Isto é, caso se queira efetuar alguma operação no computador, basta apontar o *mouse* para o local específico na tela e clicar o botão do mouse ou teclado que realize tal operação. Uma série de passos nesse sentido permite a execução de tarefas complexas no computador. Mas não se engane, essa forma de interação no formato *aponte e clique* é apenas uma camada por cima do que realmente acontece no computador. Por trás de todo *clique* existe um comando sendo executado, seja na abertura de um arquivo *pdf*, direcionamento do *browser* para uma página na internet ou qualquer outra operação cotidiana de trabalho. Esse comando foi criado pelo desenvolvedor do programa para rodar dentro do seu sistema operacional.

Enquanto esse formato de interação visual e motora tem seus benefícios ao facilitar e popularizar o uso de computadores, é pouco flexível e eficaz quando se trabalha com procedimentos computacionais. Ao conhecer os possíveis comandos disponíveis ao usuário, é possível criar um arquivo contendo alguns comandos em sequência e, futuramente, simplesmente pedir que o computador **execute** esse arquivo com os nossos procedimentos. **Uma rotina de computador é nada mais do que um texto que instrui, de forma clara e objetiva, o que o computador deve fazer**. Investe-se certo tempo para a criação do programa, porém, no futuro, esse irá executar sempre da mesma maneira o procedimento gravado. No médio e longo prazo, existe um ganho significativo de tempo entre o uso de uma rotina do computador e uma interface do tipo *aponte e clique*.

Além disso, o risco de erro humano na execução do procedimento é quase nulo, pois os comandos e a sua sequência estão registrados no arquivo texto e irão ser executados sempre da mesma maneira. Da mesma forma, esse aglomerado de comandos pode ser compartilhado com outras pessoas, as quais podem replicar os resultados em seus computadores. Essa é uma das grandes razões que justificam a popularização de programação na realização de pesquisa em dados. Todos os procedimentos executados podem ser replicados pelo uso de um *script*.

No uso do R, o ideal é mesclar o uso do mouse com a utilização de comandos. O R e o RStudio possuem algumas funcionalidades através do *mouse*, porém a sua capacidade é otimizada quando os utilizamos via inserção de comandos específicos. Quando um grupo de comandos é realizado de uma maneira inteligente, temos um *script* do R que deve preferencialmente produzir algo importante para nós no final de sua execução. Em Finanças, isso pode ser o valor atualizado de um portfólio de investimento, o cálculo do risco de uma carteira, a performance histórica de uma estratégia de investimento, o resultado de uma pesquisa acadêmica, entre diversas outras possibilidades.

O R também possibilita a exportação de arquivos, tal como figuras a serem inseridas em um relatório técnico ou informações em um arquivo texto. De fato, o próprio relatório técnico pode ser dinamicamente criado dentro do R através da tecnologia *RMarkdown*. Por exemplo, este livro que estás lendo foi escrito utilizando o pacote `bookdown` (Xie 2016). O livro é compilado com a

******ebook converter DEMO Watermarks*******

execução dos códigos e as suas saídas são registradas em texto. Todas as figuras e os dados do livro podem ser atualizados com a execução de um simples comando. Usuários que possuem interesse em construir documentos cujos dados e figuras devem ser atualizados periodicamente, sugiro fortemente o estudo das ferramentas destacadas anteriormente. No capítulo 12 iremos mostrar uma introdução a essas ferramenta de construção de documentos dinâmicos.

******ebook converter DEMO Watermarks*******

2.2 Objetos e Funções

No R, **tudo é um objeto, e cada tipo de objeto tem suas propriedades**. Por exemplo, o valor de um índice de inflação pode ser representado como um objeto do tipo vetor numérico, em que cada elemento desse vetor é a inflação acumulada até um período de tempo. As datas relacionadas com esses valores podem ser representadas como texto (*character*) ou a própria classe `Date`. Por fim, podemos representar conjuntamente os dados de inflação e as datas armazenando-os em um objeto único do tipo *dataframe*, o qual nada mais é do que uma tabela com linhas e colunas. Esses objetos fazem parte do ecossistema do R e é através da manipulação destes que tiramos o máximo proveito do *software*.

Enquanto representamos dados como um objeto específico no R, um tipo especial de objeto é a função, a qual representa um procedimento preestabelecido que está disponível para o usuário. O R possui uma grande quantidade de funções, as quais possibilitam que o usuário realize uma vasta gama de procedimentos. Por exemplo, os comandos básicos do R, não incluindo demais pacotes, somam um total de 1217 funções.

Cada função possui um próprio nome. Por exemplo, a função `mean` é um procedimento que calcula a média de valores utilizados como *input*. Caso quiséssemos calcular o valor médio da sequência *1,2,3,4,5*, basta inserir no *prompt* o seguinte comando e apertar *enter*:

```
mean(1:5, na.rm = TRUE)
```

```
## [1] 3
```

O comando `1:5`, utilizado anteriormente, cria uma sequência com início em 1 e final em 5 (maiores detalhes sobre a criação de sequências serão dados em seção futura). Observe que a função `mean` é utilizada com parênteses de início e fim. Esses parênteses servem para destacar as entradas (*inputs*), isto é, as informações enviadas para a função produzir alguma coisa. Observe que cada entrada é separada por uma vírgula, tal como em `MinhaFct(input1, input2, input3, ...)`. No caso do código anterior, note que usamos a opção `na.rm = TRUE`. Essa é uma instrução específica para a função `mean` ignorar elementos do tipo NA (*not available*), caso eles existam. Por enquanto, essa breve introdução a funções já serve o seu propósito. No capítulo 8 iremos tratar deste assunto com maior profundidade.

******ebook converter DEMO Watermarks*******

2.3 O Formato Brasileiro

Antes de começar a explicar o uso do R e RStudio, é importante ressaltar algumas regras de formatação de números e códigos para o caso brasileiro.

decimal: O decimal no R é definido pelo ponto (.), tal como em `2.5` e não vírgula, como em `2,5`. Esse é o padrão internacional, e a diferença para a notação brasileira gera muita confusão. Alguns softwares, tal como o Microsoft Excel, fazem essa conversão automaticamente no momento da importação dos dados. Porém isso não ocorre na maioria dos casos. Como regra geral, utilize vírgulas apenas para separar os termos de entradas em uma função. Em nenhuma situação deve-se utilizar a vírgula como separador de casas decimais. Sempre dê prioridade para o formato internacional, pois esse será compatível com a grande maioria dos dados. Outros pesquisadores também podem ter dificuldade de entender o seu código, caso utilize a notação local para o decimal.

caracteres latinos: devido ao seu padrão internacional, o R apresenta problemas para entender caracteres latinos, tal como cedilha e acentos. Caso possa evitar, não utilize esses tipos de caracteres no código para nomeação de variáveis ou arquivos. Nos textos (objeto de caracteres), é possível utilizá-los desde que a codificação do objeto esteja correta (UTF-8 ou Latin1). Recomenda-se que o código do R seja escrito na língua inglesa. Isso automaticamente elimina o uso de caracteres latinos e facilita a usabilidade do código por outras pessoas que não entendam a língua portuguesa. Destaca-se que essa foi a escolha utilizada para o livro. Os nomes dos objetos nos exemplos estão em inglês, assim como também os comentários do código.

formato das datas: as datas no R são formatadas de acordo com o padrão ISO 8601, seguindo o padrão `YYYY-MM-DD`, onde `YYYY` é o ano em quatro números, `MM` é o mês e `DD` é o dia. Por exemplo, uma data em ISO 8601 é 2018-06-10. No Brasil, as datas são formatadas como `DD/MM/YYYY`. Reforçando a regra, sempre dê preferência ao padrão internacional. Vale salientar que a conversão entre um formato e outro é bastante fácil e será apresentada em capítulo futuro.

No momento de instalação do R, diversas informações sobre o formato local do seu computador são importadas. Para saber qual o formato que o R está configurado, digite o seguinte comando no *prompt* (canto esquerdo inferior do RStudio) e aperte *enter*:

```
# get local format
Sys.localeconv()
```

```
##     decimal_point     thousands_sep          grouping
##               "."                ""                ""
##   int_curr_symbol   currency_symbol mon_decimal_point
##           "BRL "               "R$"               ","
## mon_thousands_sep      mon_grouping     positive_sign
##               "."         "\003\003"                ""
##     negative_sign   int_frac_digits       frac_digits
##               "-"               "2"               "2"
##     p_cs_precedes    p_sep_by_space     n_cs_precedes
##               "1"               "1"               "1"
##    n_sep_by_space       p_sign_posn       n_sign_posn
##               "1"               "1"               "1"
```

******ebook converter DEMO Watermarks*******

A saída de `Sys.localeconv()` mostra como o R interpreta pontos decimais e o separador de milhares, entre outras coisas. Como você pode ver no resultado anterior, este livro foi compilado usando a notação brasileira de moeda, mas usa o ponto para decimais. Conforme mencionado anteriormente, é boa política de trabalho seguir a notação internacional, especialmente para o ponto decimal. Se necessário, você pode (temporariamente) alterar seu formato local para a notação internacional usando o seguinte comando.

```
# set local format to English
Sys.setlocale("LC_ALL", "English")
```

Caso desejes incorporar esta mudança de forma permanente, deves buscar entender e modificar como o seu sistema operacional registra o formato local, ou então incorporar o comando anterior na inicialização do R. Essa mudança, porém, não é essencial. Se trabalhares com o R no formato padrão de instalação, o formato internacional será utilizado nos itens mais importantes e provavelmente não terás problema com isso.

******ebook converter DEMO Watermarks*******

2.4 Tipos de Arquivos

Assim como outros programas, o R possui um ecossistema de arquivos. Cada tipo de extensão tem uma finalidade diferente. Na grande maioria dos casos, o trabalho se concentrará em uma extensão apenas: arquivos *.R*. A seguir apresenta-se uma descrição de diversas extensões de arquivos. Os itens da lista estão ordenados por ordem de importância e uso. Note que omitimos arquivos de figuras tal como *.png*, *.jpg*, *.gif* entre outros, pois estes não são exclusivos do R.

Arquivos com extensão *.R*: Representam arquivos do tipo texto contendo diversas instruções para o R. Esses são os arquivos que conterão o código da pesquisa. Também pode ser chamado de um *script* ou rotina de pesquisa. Exemplos: Minha_pesquisa.R, Minhas_Funções.R.

Arquivos com extensão *.RData* e *.rds*: armazenam dados nativos do R. Esses arquivos servem para salvar objetos criados em diferentes sessões. Por exemplo, podes guardar o resultado de uma pesquisa em uma tabela, a qual é salva em um arquivo com extensão *.RData* ou *.rds*. A diferença entre os dois formatos anteriores é que o *RData* possibilita armazenar mais de um objeto do R, enquanto o *.rds* armazena apenas um. No capítulo 4 iremos tratar desse assunto com maior profundidade. Os arquivos salvos nesse formato podem ser futuramente carregados em outra sessão do R. Exemplos: Meus_dados.RData, Resultados_Pesquisa.rds.

Arquivos com extensão *.Rmd, .md* e *.Rnw*: representam arquivos para a edição de documentos dinâmicos relacionados a linguagem *Rmarkdown* e *markdown*. O uso desses arquivos permite a criação de documentos onde texto e código são integrados. No capítulo 12 iremos ver a forma de usar o *RMarkdown* no RStudio.

Arquivos com extensão *.Rproj*: contém informações para a edição de projetos no RStudio. O uso desse tipo de arquivo facilita a criação de projetos no R, tal com um novo pacote ou um *script* de pesquisa. O uso desse arquivo possibilita configurar um projeto qualquer e também facilita a utilização de ferramentas de controle de código, tal como controle de versões. O seu uso, porém, não é essencial. Para aqueles com interesse em conhecer esta funcionalidade, sugiro a leitura do manual do RStudio. Exemplo: Meu_Projeto.Rproj.

******ebook converter DEMO Watermarks*******

2.5 Explicando a Tela do RStudio

Após instalar os dois programas, R e RStudio, abra o programa RStudio. Procure o seu ícone na área de trabalho ou via menu *Iniciar*. Destaca-se que o R também possui um programa de interface e isso muitas vezes gera confusão. Deve-se procurar o atalho correto para o RStudio. A janela resultante deve ser igual a figura 2.1, apresentada a seguir.

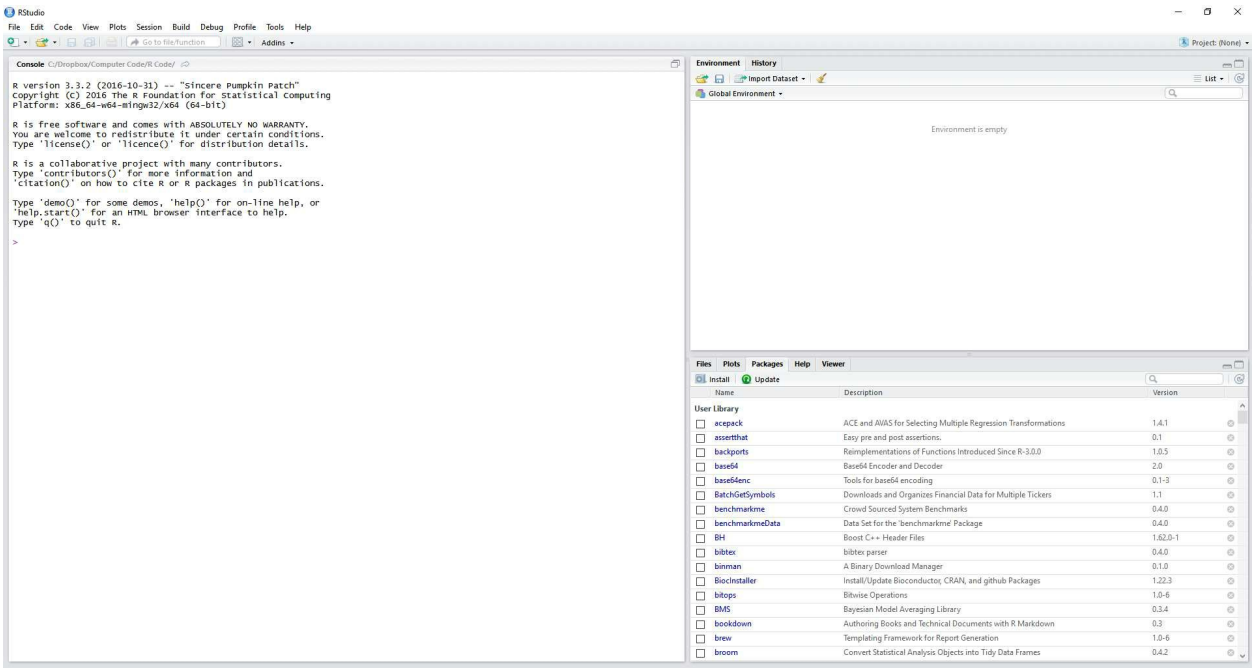


Figura 2.1 A tela do RStudio

Observe que o RStudio automaticamente detectou a instalação do R e inicializou a sua tela, no lado esquerdo.

Caso não visualizar na tela do RStudio algo como:

```
R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
```

******ebook converter DEMO Watermarks*******

```
    Type 'q()' to quit R.
```

o R não foi instalado corretamente. Repita os passos de instalação do capítulo anterior e confirme o aparecimento da respectiva mensagem de inicialização no lado esquerdo inferior do RStudio.

Como um primeiro exercício, clique em *file*, *New File* e *R Script*. Deve aparecer um editor de texto no lado esquerdo da tela. É nesse editor que iremos inserir os nossos comandos, os quais são executados de cima para baixo, na mesma direção em que normalmente o lemos. Note que essa direção de execução introduz uma dinâmica de recursividade: cada comando depende do comando executado nas linhas anteriores. Após realizar os passos definidos anteriormente, a tela resultante deve ser semelhante à apresentada na figura 2.2.

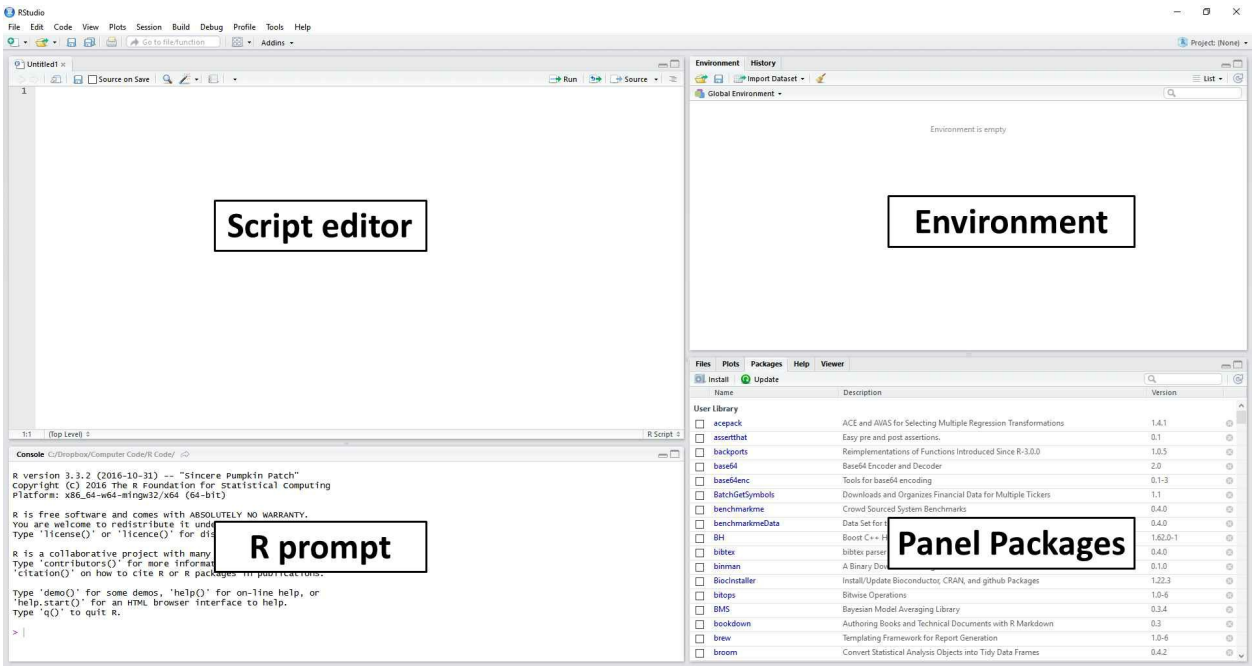


Figura 2.2 Explicando a tela do RStudio

Nesse ponto, uma sugestão é modificar o esquema de cores do RStudio para uma configuração mais escura. Não é uma questão de gosto estético mas sim de saúde física. Dado que irás passar bastante tempo na frente do computador, vale a pena modificar as cores para aliviar o constante brilho da tela. Dessa forma, conseguirás trabalhar por mais tempo sem forçar a sua visão. Podes configurar o esquema de cores do RStudio indo na opção *Tools*, *Global Options* e então em *Appearance*. Um esquema de cores escuras que pessoalmente gosto e sugiro é o *Ambience*.

Voltando ao RStudio, os principais itens/painéis são:

Editor de scripts (*Script editor*): localizado no lado esquerdo e acima da tela. Esse painel é utilizado para escrever *scripts* inteiros. Passaremos a maior parte do tempo escrevendo códigos nesse editor.

Console do R (*R prompt*): localizado no lado esquerdo e abaixo do editor de *scripts*. Apresenta o *prompt* do R, o qual também pode ser utilizado para dar comandos para o R. A principal

******ebook converter DEMO Watermarks*******

função do *prompt* é testar código e apresentar os resultados dos comandos inseridos no editor de *scripts*.

Área de trabalho (*Environment*): localizado no lado direito e superior da tela. Mostra todos os objetos, incluindo variáveis e funções atualmente disponíveis para o usuário. Observe também a presença do painel *History*, o qual mostra o histórico dos comandos já executados.

** Pacotes (*Panel Packages*):** mostra os pacotes instalados e carregados pelo R. Um pacote é nada mais que um módulo no R. Podemos instalar diferentes módulos para diferentes tarefas. Logo em seguida iremos destacar com mais detalhes o uso de pacotes no R e RStudio. Observe a presença de quatro abas: *Files*, para carregar e visualizar arquivos do sistema; *Plots*, para visualizar figuras; *Help*, para acessar o sistema de ajuda do R e *Viewer*, para mostrar resultados dinâmicos e interativos, tal como uma página da internet.

Como um exercício introdutório, vamos inicializar duas variáveis. Dentro do console do R (lado esquerdo inferior), digite os seguintes comandos e aperte *enter* ao final de cada linha. O símbolo `<-` é nada mais que a junção de `<` com `-`. O símbolo `'` representa uma aspa simples e sua localização no teclado Brasileiro é no botão abaixo do *escape* (*esc*), lado esquerdo superior do teclado.

```
# set x and y
x <- 1
y <- 'text'
```

Após a execução, dois objetos devem aparecer no painel *Environment*, um chamado `x` com o valor 1, e outro chamado `y` com o conjunto de caracteres `text`. O histórico de comandos na aba *History* também foi atualizado com os comandos utilizados anteriormente.

Agora, vamos mostrar na tela os valores de `x`. Para isso, digite o seguinte comando no *prompt* e aperte *enter* novamente:

```
# print it
print(x)
```

```
## [1] 1
```

A função `print` é uma das principais funções para mostrarmos valores no *prompt* do R. O texto apresentado como `[1]` indica o índice do primeiro número da linha. Para verificar isso, digite o seguinte comando, o qual irá mostrar vários números na tela:

```
# print vector from 50 to 100
print(50:100)
```

```
##  [1]  50  51  52  53  54  55  56  57  58  59  60  61  62  63
## [15]  64  65  66  67  68  69  70  71  72  73  74  75  76  77
## [29]  78  79  80  81  82  83  84  85  86  87  88  89  90  91
## [43]  92  93  94  95  96  97  98  99 100
```

Nesse caso, utilizamos o símbolo `:` em `50:100` para criar uma sequência iniciando em 50 e terminando em 100. Observe que temos os valores 1, 15, 29 e 43. Esses representam os índices do primeiro elemento apresentado na linha. Por exemplo, o décimo quinto elemento do vetor

******ebook converter DEMO Watermarks*******

criado é o valor 64.

******ebook converter DEMO Watermarks*******

2.6 Executando Códigos em um *Script*

Agora, vamos juntar todos os códigos digitados anteriormente e colar na tela do editor (lado esquerdo superior), assim como mostrado a seguir:

```
# set objects
x <- 1
y <- 'my text'

# print it
print(x)
print(1:50)
```

Após colar todos os comandos no editor, salve o arquivo *.R* em alguma pasta pessoal. Esse arquivo, o qual no momento não faz nada especial, registrou os passos de um algoritmo simples que cria vários objetos e mostra os seus valores. Futuramente esse irá ter mais forma, com a importação de dados, manipulação e modelagem dos mesmos e saída de tabelas e figuras.

No RStudio existem alguns atalhos predefinidos para executar códigos que economizam bastante tempo. Para executar um *script* inteiro, basta apertar `control + shift + s`. Esse é o comando *source*. Com o RStudio aberto, sugiro testar essa combinação de teclas e verificar como o código digitado anteriormente é executado, mostrando os valores no *prompt* do R. Visualmente, o resultado deve ser próximo ao apresentado na figura 2.3.

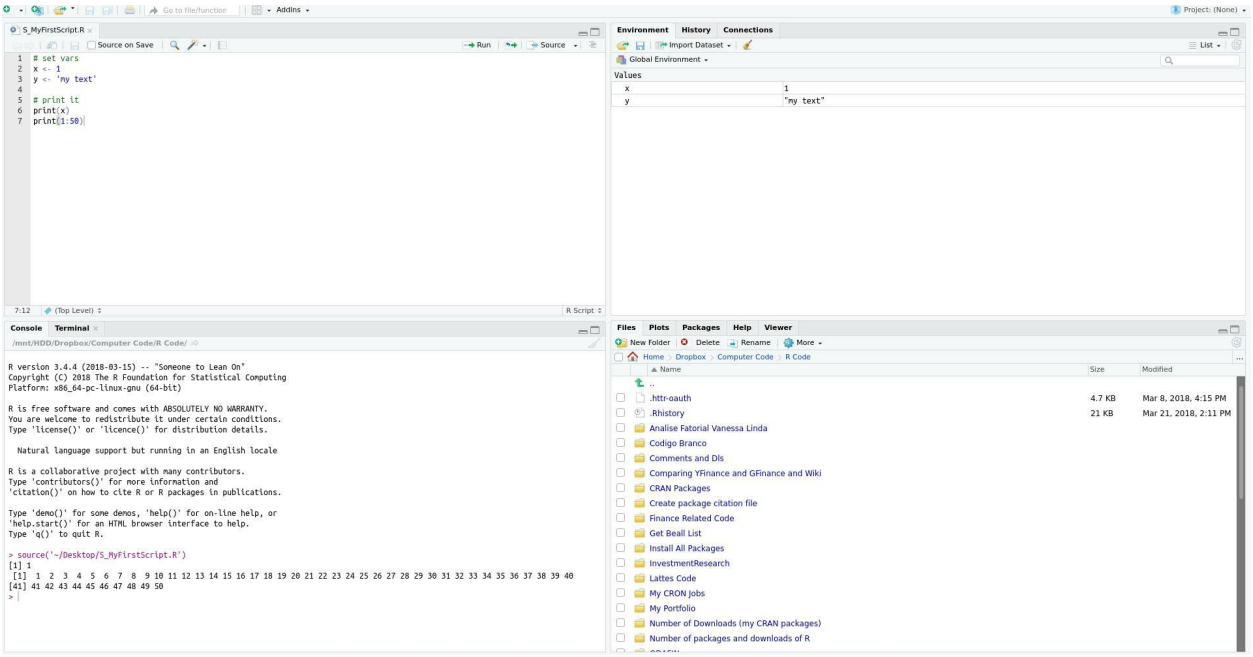


Figura 2.3 Exemplo de Rotina no R

Outro comando muito útil é a execução por linha. Nesse caso não é executado todo o arquivo, mas somente a linha em que o cursor se encontra. Basta apertar `control+enter`. Esse atalho é bastante útil no desenvolvimento de rotinas pois permite que cada linha seja testada antes de

******ebook converter DEMO Watermarks*******

executar o programa inteiro. Como um exemplo de uso, aponte o cursor para a linha `print(x)` e pressione `control + enter`. Verás que o valor de `x` é mostrado na tela inferior. A seguir destaco esses e outros atalhos do RStudio, os quais também são muito úteis.

- **control+shift+s** executa o arquivo atual do RStudio, sem mostrar comandos no *prompt* (sem eco);
- **control+shift+enter**: executa o arquivo atual, mostrando comandos na tela (com eco);
- **control+enter**: executa a linha selecionada, mostrando comandos na tela;
- **control+shift+b**: executa os códigos do início do arquivo até a linha atual onde o cursor se encontra;
- **control+shift+e**: executa os códigos da linha onde o cursor se encontra até o final do arquivo.

Sugere-se que esses atalhos sejam memorizados e utilizados. Isso facilita bastante o uso do programa. Para aqueles que gostam de utilizar o *mouse*, uma maneira alternativa para rodar o código do *script* é apertar o botão *source*, localizado no canto direito superior do editor de rotinas. Isto é equivalente ao atalho `control+shift+s`.

Caso desejes rodar um código salvo em um arquivo *.R*, podes utilizar o comando `source`. Por exemplo, imagine que tens um *script* principal com a pesquisa e outro *script* que realiza alguma operação de suporte tal como a importação de dados para o R. Essas operações foram desmembradas em arquivos diferentes como uma forma de organizar o código.

Para executar o *script* de suporte, basta chamá-lo no *script* principal com o comando `source`, como no código a seguir:

```
# source (execute) file
source('import-data.R')
```

Nesse caso, o *script* em `import-data.R` será executado de forma equivalente a abrir o mesmo manualmente e apertar *control+shift+s*.

Como podemos ver, existem diversas maneiras de executar uma rotina de pesquisa. Na prática, porém, iras centralizar o uso em dois comandos apenas: `control+shift+s` para rodar o *script* inteiro e `control+enter` para rodar por linha.

******ebook converter DEMO Watermarks*******

2.7 Testando Código

O desenvolvimento do código segue um ciclo. Primeiro você escreverá uma nova linha de comando em uma rotina. Essa linha será testada com o atalho `control + enter`, verificando-se a ocorrência de erros e as saídas na tela. Caso não houver erro e o resultado for igual ao esperado, parte-se para a próxima linha de código.

Um ciclo de trabalho fica claro, a escrita do código da linha atual é seguida pela execução, seguido da verificação de resultados, modificação caso necessário e assim por diante. Esse é um processo normal e esperado. Dado que uma rotina é lida e executada de cima para baixo, você precisa ter certeza de que cada linha de código está corretamente especificada antes de passar para a próxima.

Quando você está tentando encontrar um erro em um *script* preexistente, o R oferece algumas ferramentas para controlar e avaliar sua execução. Isso é especialmente útil quando você possui um código longo e complicado. A ferramenta de teste mais simples e fácil de utilizar que o RStudio oferece é o ponto de interrupção do código. No RStudio, você pode clicar no lado esquerdo do editor e aparecerá um círculo vermelho, como na Figura 2.4.

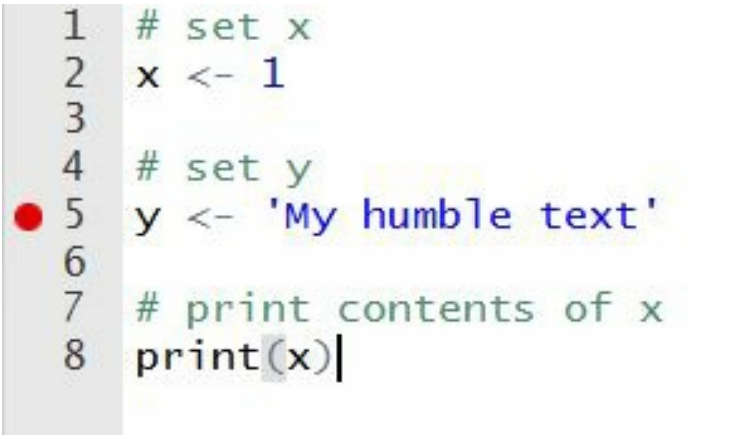


Figura 2.4 Exemplo de debug

O círculo vermelho indica um ponto de interrupção do código que forçará o R a parar a execução nessa linha. Após isso, você pode verificar o conteúdo dos objetos existentes. Quando a execução atinge o ponto de interrupção, o *prompt* mudará para `browser[1]>` e você poderá verificar o conteúdo dos objetos. No console, você tem a opção de continuar a execução para o próximo ponto de interrupção ou interrompê-la. O mesmo resultado pode ser alcançado usando a função `browser`. Dê uma olhada:

```
# set x
x <- 1

# set y
browser()
y <- 'My humble text'
```

******ebook converter DEMO Watermarks*******

```
# print contents of x
print(x)
```

O resultado prático do código anterior é o mesmo que utilizar o círculo vermelho do RStudio, figura 2.4. Porém, o uso do `browser` permite mais controle sobre onde a execução deve ser interrompida.

******ebook converter DEMO Watermarks*******

2.8 Criando Objetos Simples

Um dos comandos mais básicos no R é a definição de objetos. Como foi mostrado nas seções anteriores, pode-se definir um objeto com o uso do comando `<-`, o qual, para o português, é traduzido para o verbo *defina* (*assign* em inglês). Considere o seguinte código:

```
# set x
x <- 123

# set x, y and z in one line
my.x <- 1 ; my.y <- 2; my.z <- 3
```

Lê-se esse código como *x é definido como 123*. A direção da seta define onde o valor será armazenado. Por exemplo, utilizar `123 -> x` também funcionaria, apesar de ser uma sintaxe pouco utilizada ou recomendada. Note que também é possível escrever diversos comandos na mesma linha com o uso da semi-vírgula (`;`).

O uso do símbolo `<-` para a definição de objetos é específico do R. Na época da concepção da linguagem *S*, de onde o R foi baseado, existiam teclados com uma tecla que definia diretamente o símbolo de seta. Essa escolha foi definida pelo fácil acesso ao símbolo. Atualmente, no *Windows*, o atalho para esse símbolo é `alt + -`.

É possível também usar o símbolo `=` para definir objetos assim como o `<-`, porém a utilização do `=` com esse fim específico não é recomendada. O símbolo de igualdade tem uso especial dentro da definição de argumentos de uma função. Esse caso será explicado e demonstrado em capítulos futuros.

O nome dos objetos é importante no R. Tirando alguns casos específicos, o usuário pode nomear os objetos como quiser. Essa liberdade, porém, pode ser um problema. É desejável sempre dar nomes curtos que façam sentido ao conteúdo do objeto e que sejam simples de entender. Isso facilita o entendimento do código por outros usuários e faz parte das normas sugeridas para a estruturação do código. Observe que todos os objetos criados nesse livro possuem nomenclatura em inglês e formatação específica, onde espaços entre substantivos e verbos são substituídos por ponto, tal como em `my.x` e `my.file`. Outra possibilidade é utilizar o traço baixo (_), tal como `my_x` e `my_file`. Não tem nenhum ganho computacional entre um e outro. Deves escolher um que aches mais atrativo visualmente. Aqui, o mais importante é a consistência do formato. Sempre mantenha o mesmo padrão em todo o código.

O R executa o código procurando objetos e funções disponíveis. Se tentarmos acessar um objeto que não existe, o R irá dar como resultado uma mensagem de erro, como a seguir:

```
print(z)
```

```
## Error in print(z): object 'z' not found
```

Isso ocorre pois o objeto `z` não existe no *enviromnent* atual do R. Se criarmos uma variável `z` como `z <- 123` e repetirmos o comando `print(z)`, não teremos a mesma mensagem de erro.

******ebook converter DEMO Watermarks*******

Um ponto importante aqui é a definição de objetos de classes diferentes com o uso de símbolos específicos. O uso de aspas duplas (`" "`) ou simples (`' '`) define objetos da classe texto enquanto números são definidos pelo próprio valor. Conforme será mostrado, cada objeto no R tem uma classe e cada classe tem um comportamento diferente. Portanto, objetos criados com o uso de aspas pertencem à classe *character*. Para confirmar isso, verifique o próximo código.

```
# set vars
x <- 1
y <- '1'

# display classes
class(x)
```

```
## [1] "numeric"
```

```
class(y)
```

```
## [1] "character"
```

As saídas anteriores mostram que a variável `x` é do tipo numérico, enquanto a variável `y` é do tipo texto (*character*). Ambas fazem parte das classes básicas de objetos no R. Iremos estudar este assunto mais profundamente no capítulo 7.

******ebook converter DEMO Watermarks*******

2.9 Criando Vetores

Nos exemplos anteriores criamos objetos simples tal como `x <- 1` e `x <- 'abc'`. Enquanto isso é suficiente para demonstrar os comandos básicos do R, na prática tais comandos são bastante limitados, uma vez que um problema real de análise de dados certamente irá ter um maior volume de informações a serem processadas.

Um dos procedimentos mais utilizados no R é a criação de vetores atômicos. Esses são objetos que guardam uma série de elementos. Todos os elementos de um vetor atômico devem possuir a mesma classe, o que justifica a sua propriedade *atômica*. Um exemplo seria representar no R uma série de preços diários de uma ação. Tal série possui vários valores numéricos que formam um vetor da classe numérica.

Vetores atômicos são criados no R através do uso do comando `c`, o qual é oriundo do verbo em inglês *combine*. Por exemplo, caso eu quisesse *combinar* os valores 1, 2 e 3 em um objeto/vetor, eu poderia fazê-lo através do seguinte comando:

```
# set vector
x <- c(1,2,3)
print(x)
```

```
## [1] 1 2 3
```

Esse comando funciona da mesma maneira para qualquer número de elementos. Caso necessário, poderíamos criar um vetor com mais elementos simplesmente adicionando valores após o `3`, tal como em `x <- c(1, 2, 3, 4, 5)`.

O uso do comando `c` não é exclusivo para vetores numéricos. Por exemplo, poderíamos criar um vetor de outra classe de dados, tal como *character*:

```
y <- c('text 1', 'text 2', 'text 3', 'text 4')
print(y)
```

```
## [1] "text 1" "text 2" "text 3" "text 4"
```

A única restrição no uso do comando `c` é que todos os itens do vetor tenham a mesma classe. Se inserirmos dados de classes diferentes, o R irá tentar transformar os itens para a mesma classe seguindo uma lógica própria, onde a classe mais complexa sempre tem preferência. Caso ele não consiga transformar todos os elementos para uma classe só, uma mensagem de erro será retornada. Observe no próximo exemplo como os valores numéricos no primeiro e segundo elemento de `x` são transformados para a classe de caracteres.

```
x <- c(1, 2, '3')
print(x)
```

```
## [1] "1" "2" "3"
```

Isso ocorre pois a classe de texto é mais complexa que a classe numérica. O uso do comando `class` confirma que o objeto `x` é da classe `character`:

******ebook converter DEMO Watermarks*******

```r
class(x)
```

```
## [1] "character"
```

Outra utilização do comando `c` é a combinação de vetores. De fato, isto é exatamente o que fizemos ao executar o código `c(1, 2, 3)`. Neste caso, cada vetor possuía um elemento. Podemos realizar o mesmo com vetores maiores. Veja a seguir:

```r
# set x and y
x <- c(1, 2, 3)
y <- c(4, 5)

# print combination of x and y
print(c(x, y))
```

```
## [1] 1 2 3 4 5
```

Portanto, o comando `c` possui duas funções principais: criar e combinar vetores.

******ebook converter DEMO Watermarks*******

2.10 Conhecendo os Objetos Criados

Após a execução de diversos comandos no editor ou *prompt*, é desejável saber quais são os objetos criados pelo código. É possível descobrir essa informação simplesmente olhando para o lado direito superior do RStudio, na aba da área de trabalho. Porém, existe um comando que sinaliza a mesma informação no *prompt*. Com o fim de saber quais são as variáveis atualmente disponíveis na memória do R, pode-se utilizar o comando `ls`. Observe o exemplo a seguir:

```
# set vars
x <- 1
y <- 2
z <- 3

# show current objects
ls()
```

```
## [1] "x" "y" "z"
```

Os objetos `x`, `y` e `z` foram criados e estavam disponíveis no ambiente de trabalho atual, juntamente com outros objetos. Para descobrir os valores dos mesmos, basta digitar os nomes dos objetos e apertar `enter` no *prompt*:

```
x
```

```
## [1] 1
```

```
y
```

```
## [1] 2
```

```
z
```

```
## [1] 3
```

Destaca-se que digitar o nome do objeto na tela tem o mesmo resultado que utilizar a função `print`. De fato, ao executar o nome de uma variável, internamente o R passa esse objeto para a função `print`.

No R, conforme mostrado, todos os objetos pertencem a alguma classe. Para descobrir a classe de um objeto, basta utilizar a função `class`. Observe no exemplo a seguir que `x` é um objeto da classe numérica e `y` é um objeto da classe de texto (*character*).

```
# set vars
x <- 1
y <- 'a'

# check classes
class(x)
```

```
## [1] "numeric"
```

```
class(y)
```

******ebook converter DEMO Watermarks*******

```
## [1] "character"
```

Outra maneira de conhecer melhor um objeto é verificar a sua representação em texto. Todo objeto no R possui uma representação textual e a verificação desta é realizada através da função `str`:

```
# print textual representation of a vector
print(str(1:10))
```

```
##  int [1:10] 1 2 3 4 5 6 7 8 9 10
## NULL
```

Essa função é particularmente útil quando se está tentando entender os detalhes de um objeto mais complexo, tal como uma tabela. A utilidade da representação textual é que nela aparece o tamanho do objeto e suas classes internas. Nesse caso, o objeto `1:10` é da classe *integer* e possui dez elementos. No `tidyverse`, a função equivalente é `dplyr::glimpse`. Veja o exemplo a seguir para um vetor atômico:

```
# glimpse a vector
dplyr::glimpse(1:10)
```

```
##  int [1:10] 1 2 3 4 5 6 7 8 9 10
```

No código anterior, usamos o operador `::` para explicitamente dizer ao R que a função `glimpse` pertence ao pacote `dplyr`, o qual faz parte do `tidyverse`. Caso não conseguir rodar o exemplo anterior, provavelmente não tens o pacote `dplyr` instalado em sua máquina. Iremos ver o tópico de instalação de pacotes logo em seguida.

******ebook converter DEMO Watermarks*******

2.11 Mostrando e Formatando Informações na Tela

Como já vimos, é possível mostrar o valor de uma variável na tela de duas formas, digitando o nome dela no *prompt* ou então utilizando a função `print`. Explicando melhor, a função `print` é voltada para a apresentação de objetos e pode ser customizada para qualquer tipo. Por exemplo, caso tivéssemos um objeto chamado `table` que representasse um objeto tabular, poderíamos criar uma função chamada `print.table` que irá mostrar uma tabela na tela com um formato especial tal como número de linhas, nomes das colunas, etc. A função `print`, portanto, pode ser customizada para cada classe de objeto.

Porém, existem outras funções específicas para apresentar texto (e não objetos) no *prompt*. A principal delas é `cat` (*Concatenate and Print*). Essa toma como *input* um texto, processa-o para símbolos específicos e o apresenta na tela. Essa função é muito mais poderosa e personalizável do que `print`.

Por exemplo, caso quiséssemos mostrar na tela o texto `'O valor de x é igual a 2'`, poderíamos fazê-lo da seguinte forma:

```
# set var
x <- 2

# print with cat()
cat('The value of x is', x)
```

```
## The value of x is 2
```

Função `cat` também funciona para vetores:

```
# set vec
x <- 2:5

# print with cat()
cat('The values in x are: ', x)
```

```
## The values in x are:  2 3 4 5
```

A customização da saída da tela é possível através de comandos específicos. Por exemplo, se quiséssemos quebrar a linha da tela, poderíamos fazê-lo através do uso do caractere reservado `\n`:

```
# set char
my.text <- ' First line,\nSecond Line,\nThird Line'

# print with new lines
cat(my.text)
```

```
##  First line,
## Second Line,
## Third Line
```

Observe que o uso do `print` não resultaria no mesmo efeito, uma vez que esse comando

******ebook converter DEMO Watermarks*******

apresenta o texto como ele é, sem processar para efeitos específicos:

```
print(my.text)
```

```
## [1] " First line,\nSecond Line,\nThird Line"
```

Outro exemplo no uso de comandos específicos para texto é adicionar um espaçamento *tab* no texto apresentado com o símbolo `\t`. Veja a seguir:

```
# set char with \t
my.text <- 'A->\t<-B'

# print with cat()
cat(my.text)
```

```
## A->  <-B
```

Vale destacar que, na grande maioria dos casos de pesquisa, será necessário apenas o uso de `\n` para formatar textos de saída. Outras maneiras de manipular a saída de texto no *prompt* com base em símbolos específicos são encontradas no manual oficial do R.

Parte do processo de apresentação de texto na tela é a customização do mesmo. Para isto, existem duas funções muito úteis: `paste` e `format`.

A função `paste` *cola* uma série de caracteres juntos. É uma função muito útil, a qual será utilizada intensamente para o resto dos exemplos deste livro. Observe o código a seguir:

```
# set chars
my.text.1 <- 'I am a text'
my.text.2 <- 'very beautiful'
my.text.3 <- 'and informative.'

# using paste and cat
cat(paste(my.text.1, my.text.2, my.text.3))
```

```
## I am a text very beautiful and informative.
```

O resultado anterior não está muito longe do que fizemos no exemplo com a função `print`. Note, porém, que a função `paste` adiciona um espaço entre cada texto. Caso não quiséssemos esse espaço, poderíamos usar a função `paste0`:

```
# using paste0
cat(paste0(my.text.1, my.text.2, my.text.3))
```

```
## I am a textvery beautifuland informative.
```

Outra possibilidade muito útil no uso do `paste` é modificar o texto entre a junção dos itens a serem colados. Por exemplo, caso quiséssemos adicionar uma vírgula (`,`) entre cada item, poderíamos fazer isso através do uso do argumento `sep`, como a seguir:

```
# using custom separator
cat(paste(my.text.1, my.text.2, my.text.3, sep = ', '))
```

```
## I am a text, very beautiful, and informative.
```

******ebook converter DEMO Watermarks*******

Caso tivéssemos um vetor atômico com os elementos da frase em um objeto apenas, poderíamos atingir o mesmo resultado utilizando `paste` o argumento *collapse*:

```
# using paste with collapse argument
my.text <-c('Eu sou um texto', 'muito bonito', 'e charmoso.')
cat(paste(my.text, collapse = ', '))
```

```
## Eu sou um texto, muito bonito, e charmoso.
```

Prosseguindo, o comando `format` é utilizado para formatar números e datas. É especialmente útil quando formos montar tabelas e buscarmos apresentar os números de uma maneira visualmente atraente. Por definição, o R apresenta uma série de dígitos após a vírgula:

```
# cat without formatting
cat(1/3)
```

```
## 0.3333333
```

Caso quiséssemos apenas dois dígitos aparecendo na tela, utilizaríamos o seguinte código:

```
# cat with format and two digits
cat(format(1/3, digits=2))
```

```
## 0.33
```

******ebook converter DEMO Watermarks*******

2.12 Conhecendo o Tamanho dos Objetos

Na prática de programação com o R, é muito importante saber o tamanho das variáveis que estão sendo utilizadas. Isso serve não somente para auxiliar o usuário na verificação de possíveis erros do código, mas também para saber o tamanho necessário em certos procedimentos de iteração tal como *loops*, os quais serão tratados em capítulo futuro.

No R, o tamanho do objeto pode ser verificado com o uso de quatro principais funções: `length`, `nrow`, `ncol` e `dim`.

A função `length` é destinada a objetos com uma única dimensão, tal como vetores atômicos:

```
# set x
x <- c(2,3,3,4,2,1)

# get length x
n <- length(x)

# display message
cat(paste('The length of x is', n))
```

```
## The length of x is 6
```

Para objetos com mais de uma dimensão, por exemplo matrizes e dataframes, utilizam-se as funções `nrow`, `ncol` e `dim` para descobrir o número de linhas (primeira dimensão) e o número de colunas (segunda dimensão). Veja a diferença a seguir.

```
# set matrix and print it
x <- matrix(1:20, nrow = 4, ncol = 5)
print(x)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
```

```
# find number of rows, columns and elements
my.nrow <- nrow(x)
my.ncol <- ncol(x)
my.length <- length(x)

# print message
cat(paste('\nThe number of lines in x is ', my.nrow))
```

```
##
## The number of lines in x is  4
```

```
cat(paste('\nThe number of columns in x is ', my.ncol))
```

```
##
## The number of columns in x is  5
```

```
cat(paste('\nThe number of elements in x is ', my.length))
```

******ebook converter DEMO Watermarks*******

```
##
## The number of elements in x is  20
```

Já a função `dim` mostra a dimensão do objeto, resultando em um vetor numérico como saída. Essa deve ser utilizada quando o objeto tiver mais de duas dimensões. Na prática, esses casos são raros. Um exemplo para a variável `x` é dado a seguir:

```
print(dim(x))
```

```
## [1] 4 5
```

Para o caso de objetos com mais de duas dimensões, podemos utilizar a função `array` para criá-los e `dim` para descobrir o seu tamanho:

```
# set array with dimension
my.array <- array(1:9, dim = c(3,3,3))

# print it
print(my.array)
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
# print its dimension
print(dim(my.array))
```

```
## [1] 3 3 3
```

Uma observação importante aqui é que as funções `length`, `nrow`, `ncol` e `dim` não servem para descobrir o número de letras em um texto. Esse é um erro bastante comum. Por exemplo, caso tivéssemos um objeto do tipo texto e usássemos a função `length`, o resultado seria o seguinte:

```
# set char object
my.char <- 'abcde'

# find its length (and NOT number of characters)
print(length(my.char))
```

```
## [1] 1
```

******ebook converter DEMO Watermarks*******

Isso ocorre pois a função `length` retorna o número de elementos. Nesse caso, `my.char` possui apenas um elemento. Para descobrir o número de caracteres no objeto, utilizamos a função `nchar`, conforme a seguir:

```
# using nchar for number of characters
print(nchar(my.char))
```

```
## [1] 5
```

Conforme foi destacado anteriormente, cada objeto tem suas propriedades e funções específicas para manipulação.

******ebook converter DEMO Watermarks*******

2.13 Selecionando Elementos de um Vetor Atômico

Após a criação de um vetor atômico de qualquer classe, é possível que se esteja interessado em apenas um ou alguns elementos desse mesmo vetor. Por exemplo, caso estivéssemos buscando atualizar o valor de um portfólio de investimento, o nosso interesse dentro de um vetor contendo preços de uma ação é somente para o preço mais recente. Todos os demais preços não seriam relevantes para a nossa análise e, portanto, poderiam ser ignorados.

Esse processo de seleção de *pedaços* de um vetor atômico é chamado de indexação e é executado através do uso de colchetes `[]`. Observe o exemplo de código a seguir:

```
# set my.x
my.x <- c(1, 5, 4, 3, 2, 7, 3.5, 4.3)
```

Se quiséssemos apenas o terceiro elemento de `my.x`, utilizaríamos o operador de colchete da seguinte forma:

```
# get third element of my.x
elem.x <- my.x[3]
print(elem.x)
```

```
## [1] 4
```

Também podemos utilizar o comando `length`, apresentado anteriormente, para acessar o último elemento do vetor:

```
# get last element of my.x
last.elem <- my.x[length(my.x)]

# print it
print(last.elem)
```

```
## [1] 4.3
```

No caso de estarmos interessado apenas no último e penúltimo valor de `my.x` utilizaríamos o operador de sequência (`:`):

```
# get last and second last elements
piece.x.1 <- my.x[ (length(my.x)-1):length(my.x) ]

# print it
print(piece.x.1)
```

```
## [1] 3.5 4.3
```

Uma propriedade única da linguagem R é que, caso for acessado uma posição que não existe no vetor, o programa retorna o valor `NA` (*not available*). Veja a seguir, onde tenta-se obter o quarto valor de um vetor com apenas três elementos.

```
# set vec
my.vec <- c(1,2,3)
```

******ebook converter DEMO Watermarks*******

```
# find fourth element (NA returned!)
print(my.vec[4])
```

```
## [1] NA
```

É importante conhecer esse comportamento do R, pois o não tratamento desses erros pode gerar problemas difíceis de identificar em um código mais complexo. Em outras linguagens de programação, a tentativa de acesso a elementos não existentes geralmente retorna um erro e cancela a execução do resto do código. No caso do R, dado que o acesso a elementos inexistentes não gera erro ou mensagem de *warning*, é possível que isso gere um problema em outras partes do *script*. **O usuário deve prestar atenção toda vez que surgirem valores `NA` de forma inesperada nos objetos criados.** Uma inspeção nos índices dos vetores pode ser necessária.

O uso de indexadores é muito útil quando se está procurando por itens de um vetor que satisfaçam alguma condição. Por exemplo, caso quiséssemos todos os valores de `my.x` que são maiores que 3, utilizaríamos o seguinte comando:

```
# get all values higher than 3
piece.x.2 <- my.x[my.x>3]

# print it
print(piece.x.2)
```

```
## [1] 5.0 4.0 7.0 3.5 4.3
```

É possível também indexar por mais de uma condição através dos operadores de lógica `&` (*e*) e `|` (*ou*). Por exemplo, caso quiséssemos os valores de `my.x` maiores que 2 e menores que 4, usaríamos o seguinte comando:

```
# get all values higher than 2 AND lower than 4
piece.x.3 <- my.x[ (my.x>2) & (my.x<4) ]

# print it
print(piece.x.3)
```

```
## [1] 3.0 3.5
```

Da mesma forma, havendo interesse nos itens que são menores que 3 ou maiores que 6, teríamos:

```
# get all values lower than 3 OR higher than 6
piece.x.4 <- my.x[ (my.x<3)|(my.x>6) ]

# print it
print(piece.x.4)
```

```
## [1] 1 2 7
```

A indexação lógica também funciona com a interação de diferentes variáveis, isto é, podemos utilizar uma condição lógica em uma variável para selecionar itens em outra:

```
# set my.x and my.y
my.x <- c(1,4,6,8,12)
my.y <- c(-2,-3,4,10,14)

# find elements in my.x where my.y are positive
```

******ebook converter DEMO Watermarks*******

```
my.piece.x <- my.x[ my.y > 0 ]

# print it
print(my.piece.x)

## [1]  6  8 12
```

Olhando mais de perto o processo de indexação, vale salientar que, quando utilizamos uma condição de indexação dos dados, esta-se criando uma variável do tipo lógica. Essa toma apenas dois valores: `TRUE` (verdadeiro) ou `FALSE` (falso). É fácil perceber isso quando criamos o teste lógico em um objeto e o mostramos na tela:

```
# set logical object
my.logical <- my.y > 0

# print it
print(my.logical)

## [1] FALSE FALSE  TRUE  TRUE  TRUE

# show its class
class(my.logical)

## [1] "logical"
```

As propriedades e operações com vetores lógicos serão explicadas em capítulo futuro.

******ebook converter DEMO Watermarks*******

2.14 Limpando a memória

Após a criação de diversas variáveis, o ambiente de trabalho do R pode ficar cheio de conteúdo já utilizado e dispensável. Nesse caso, é desejável limpar a memória do programa. Geralmente isso é realizado no começo de um *script*, de forma que toda vez que o script for executado, a memória estará totalmente limpa antes de qualquer cálculo. Além de desocupar a memória do computador, isso ajuda a evitar possíveis erros no código. Na grande maioria dos casos, a limpeza do ambiente de trabalho deve ser realizada apenas uma vez.

Por exemplo, dada uma variável `x`, podemos excluí-la da memória com o comando `rm`, conforme mostrado a seguir:

```
# set x and y
x <- 1
y <- 2

# print all existing objects
ls()

# remove x from memory
rm('x')

# print objects again
ls()
```

Observe que o objeto `x` não estará mais mais disponível após o uso do comando `rm('x')`.

Em situações práticas, porém, é desejável limpar toda a memória utilizada por todos os objetos criados no R. Pode-se atingir esse objetivo com o seguinte código:

```
# clean up workspace (all existing objects)
rm(list=ls())
```

O termo `list` é um argumento da função `rm`, o qual define quais objetos serão eliminados. Já o comando `ls()` mostra todas os objetos disponíveis atualmente. Portanto, ao encadear ambos os comandos, limpamos da memória **todos** os objetos disponíveis para o R. Como comentado, uma boa política de programação é sempre iniciar o *script* limpando a memória do R. **Atenção: apenas limpe toda a memória do R se já tiver salvo os resultados de interesse ou se puder replicá-los**.

******ebook converter DEMO Watermarks*******

2.15 Mostrando e Mudando o Diretório de Trabalho

Assim como outros softwares, **o R sempre trabalha em algum diretório**. É com base nesse diretório que o R procura arquivos para importar dados. É nesse mesmo diretório que o R salva arquivos, caso não definirmos um endereço no computador explicitamente. Essa saída pode ser um arquivo de uma figura, o arquivo de texto ou uma planilha eletrônica. Como boa prática de criação e organização de *scripts*, deve-se sempre mudar o diretório de trabalho para onde o arquivo do *script* está localizado. Em sua inicialização, o R possui como diretório *default* a pasta de documentos do usuário.

Para mostrar o diretório atual de trabalho, basta utilizar a função `getwd`:

```
# get current directory
my.dir <- getwd()

# print it
print(my.dir)
```

```
## C:/MyRBook-pt/Edition2/Book Content/
```

O resultado do código anterior mostra a pasta onde este livro foi escrito. Esse é o diretório onde os arquivos do livro foram compilados.

A mudança de diretório de trabalho é realizada através do comando `setwd`. Por exemplo, caso quiséssemos mudar o nosso diretório de trabalho para *C:/Minha pesquisa/*, basta digitar no *prompt*:

```
# set dir
my.d <- 'C:/Minha Pesquisa/'
setwd(my.d)
```

Enquanto para casos simples, como o anterior, lembrar o nome do diretório é fácil, em casos práticos o diretório de trabalho pode ser em um lugar mais profundo da raiz de diretórios do sistema de arquivos. Nessa situação, uma estratégia eficiente para descobrir a pasta de trabalho é utilizar um explorador de arquivos, tal como o *Explorer* no Windows. Abra esse aplicativo e vá até o local onde quer trabalhar com o seu *script*. Após isso, coloque o cursor na barra de endereço e selecione todo o endereço. Aperte *control+c* para copiar o endereço para a área de transferência. Volte para o seu código e cole o mesmo no código. **Atenção nesta etapa, o Windows utiliza a barra invertida para definir endereços no computador, enquanto o R utiliza a barra normal**. Caso tente utilizar a barra invertida, um erro será mostrado na tela. Veja o exemplo a seguir.

```
my.d <- 'C:\Minha pesquisa\'
setwd(my.d)
```

O erro terá a seguinte mensagem:

```
Error: '\M' is an unrecognized escape in character string starting "'C:\M"
Error in setwd(my.d) : object 'my.d' not found
```

******ebook converter DEMO Watermarks*******

A justificativa para o erro é que a barra invertida `\` é um caractere reservado no R e não pode ser utilizado isoladamente. Caso precises, podes definí-lo no objeto de texto com dupla barra, tal como em `\\`. Veja no exemplo a seguir, onde a dupla barra é substituída por uma barra única:

```
# set char with \
my.char <- 'using \\'

# print it
cat(my.char)

## using \
```

A solução do problema é simples. Após copiar o endereço, modifique todas as barras para a barra normal, assim como no código a seguir:

```
my.d <- 'C:/Minha pesquisa/'
setwd(my.d)
```

É possível também utilizar barras invertidas duplas `\\` na definição de diretórios, porém não se recomenda essa formatação, pois não é compatível com outros sistemas operacionais.

Outro ponto importante aqui é o uso de endereços relativos. Por exemplo, caso esteja trabalhando em um diretório que contém um subdiretório chamado `Data`, podes entrar nele com o seguinte código:

```
# change to subfolder
setwd('Data')
```

Outra possibilidade pouco conhecida no uso de `setwd` é que é possível entrar em níveis inferiores do sistema de diretórios com `..`, tal como em:

```
# change to previous level
setwd('..')
```

Portanto, caso estejas trabalhando no diretório `C:/My Research/` e executar o comando `setwd('..')`, o diretório atual de trabalho viraria `C:/`, um nível inferior a `C:/My Research/`.

Uma maneira mais moderna, e pouco conhecida, de definir o diretório de trabalho é usar as funções internas do RStudio. Este é um conjunto de funções que só funcionam dentro do RStudio e fornecem diversas informações sobre o arquivo sendo editado. Para descobrir o caminho do arquivo atual que está sendo editado no RStudio e configurar o diretório de trabalho para lá, você pode escrever:

```
my.path <- dirname(rstudioapi::getActiveDocumentContext()$path)
setwd(my.path)
```

Dessa forma, o *script* mudará o diretório para sua própria localização. Apesar de não ser um código exatamente bonito, ele é bastante funcional. Caso copie o arquivo para outro diretório, o valor de `my.path` muda para o novo diretório. Esteja ciente, no entanto, de que esse truque só funciona no editor de rotinas do RStudio e dentro de um arquivo salvo. O código não funcionará a partir do *prompt*.

******ebook converter DEMO Watermarks*******

Outro truque bastante útil para definir diretórios de trabalho no R é usar o símbolo ~. Esse define a pasta `'Documentos'` no *Windows*, a qual é única para cada usuário. Portanto, ao executar `setwd('~')`, irás direcionar o R a uma pasta de fácil acesso e livre modificação pelo usuário atual do computador.

******ebook converter DEMO Watermarks*******

2.16 Comentários no Código

Comentários são definidos usando o símbolo #. Qualquer texto a direita desse símbolo não será processados pelo R. Note que até a cor do código muda no editor. Isso dá liberdade para escrever o que for necessário dentro do *script*. Um exemplo:

```
# This is a comment
# This is another comment
x <- 'abc' # this is another comment, but mixed with code

my.l <- list(var1 = 1:10,   # set var 1
             var2 = 2:5)    # another var
```

Os comentários são uma maneira de comunicar qualquer informação importante que não pode ser inferida diretamente do código. O uso correto de comentários é tão importante quanto o código em si. Quando bem feitos, aumentam a reprodutibilidade, organização e entendimento do código. Em geral, você deve evitar o uso de comentários que são muito óbvios ou muito genéricos. Por exemplo:

```
# read a .csv file
df <- read.csv ('MyDataFile.csv')
```

Como você pode ver, é bastante óbvio que a linha `df <- read.csv('MyDataFile.csv')` está lendo um arquivo .csv. O nome da função, `read.csv` já afirma isso. Então, o comentário não foi bom pois não adicionou novas informações ao usuário. Uma melhor abordagem seria definir o autor, a descrição da funcionalidade do *script* e explicar melhor a origem e a última atualização do arquivo de dados. Vamos dar uma olhada:

```
# Script for reproducing results of JOHN (2018)
# Author: Mr Researcher (dontspamme@emailprovider.com)
# Last script update: 2018-01-10
#
# File downloaded from www.sitewithdatafiles.com/data-files/
# The description of the data goes here
#
# Last file update: 2017-12-05
df <- read.csv('MyDataFile.csv')
```

Com esses comentários, o usuário saberá o propósito do *script*, quem o escreveu e a data da última edição. A origem do arquivo e a data de atualização mais recente também estão disponíveis. Se o usuário quiser atualizar os dados, tudo o que ele tem a fazer é ir ao mencionado site e baixar o novo arquivo. Isso facilitará o uso futuro e o compartilhamento do *script*.

Outro uso de comentários é definir seções no código, como em:

```
# Script for reproducing results of JOHN (2018)
# Author: Mr Researcher (dontspamme@emailprovider.com)
# Last script update: 2018-01-10
#
# File downloaded from www.sitewithdatafiles.com/data-files/
# The description of the data goes here
```

******ebook converter DEMO Watermarks*******

```
#
# Last file update: 2017-12-05

# Clean data -------------------------
# - remove outliers
# - remove unnecessary columns

# Create descriptive tables ----------


# Estimate models --------------------


# Report results ---------------------
```

O uso de uma longa linha de traços (-) é intencional. Isto faz com que o RStudio identifique as seções do código e apresente no espaço abaixo do editor de rotinas um atalho para acessar as correspondentes linhas de cada seção. Teste você mesmo, copie e cole o código acima em um novo script do R, salve o mesmo, e verás que as seções aparecem em um botão entre o editor e o *prompt*. Desta forma, uma vez que você precisa mudar uma parte específica do código, você pode procurar a seção rapidamente.

Quando começar a compartilhar código com outras pessoas, logo perceberás que os comentários são essenciais e esperados. Eles ajudam a transmitir informações que não estão disponíveis no código. Uma nota aqui, ao longo do livro você verá que os comentários do código são, na maior parte do tempo, bastante óbvios. Isso foi intencional, pois mensagens claras e diretas são importantes para novos usuários, os quais fazem parte da audiência deste livro.

******ebook converter DEMO Watermarks*******

2.17 Cancelando a Execução de um Código

Toda vez que o R estiver executando algum código, uma sinalização visual no formato de um pequeno círculo vermelho no canto direito do *prompt* irá aparecer. Caso conseguir ler, o texto indica o termo *stop*. Esse símbolo não somente indica que o programa ainda está rodando mas também pode ser utilizado para cancelar a execução de um código. Para isso, basta clicar no botão. Outra maneira de cancelar uma execução é apontar o mouse no *prompt* e pressionar a tecla *Esc*, no canto esquerdo superior do teclado.

Para testar o cancelamento de código, copie e cole o código a seguir em um *script* do RStudio. Após salvar, rode o mesmo com `control+shift+s`.

```
for (i in 1:100) {
  cat('\nRunning code (please make it stop by hitting esc!)')
  Sys.sleep(1)
}
```

O código anterior usa um comando especial do tipo `for` para mostrar a mensagem a cada segundo. Neste caso, o código demorará 100 segundos para rodar. Caso não desejes esperar, aperte `esc` para cancelar a execução. Por enquanto, não se preocupe com as funções utilizadas no exemplo. Iremos discutir o uso do comando `for` no capítulo 8.

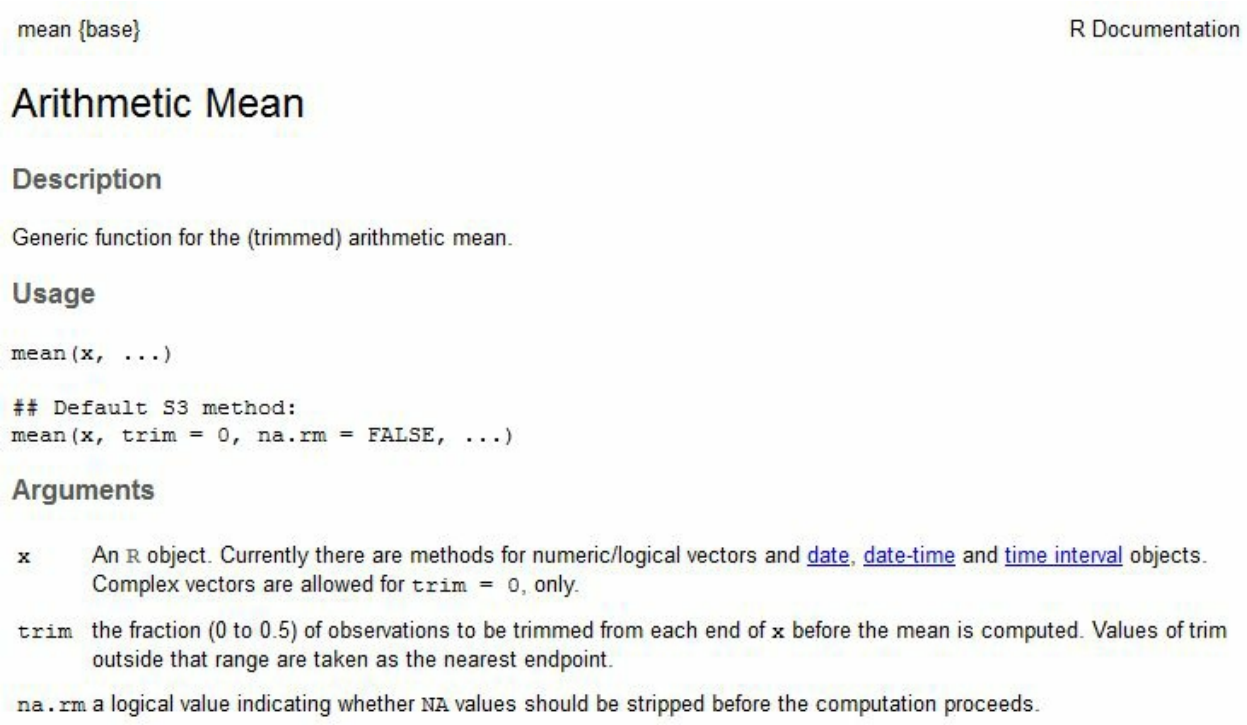
******ebook converter DEMO Watermarks*******

2.18 Procurando Ajuda

Uma tarefa muito comum no uso do R é procurar ajuda. Até mesmo usuários avançados comumente procuram ajuda sobre tarefas específicas no programa, seja para entender detalhes sobre algumas funções ou estudar um novo procedimento. O uso do sistema de ajuda do R faz parte do cotidiano.

É possível buscar ajuda utilizando tanto o painel de *help* do RStudio como diretamente do *prompt*. Para isso, basta digitar o ponto de interrogação junto ao objeto sobre o qual se deseja ajuda, tal como em `?mean`. Nesse caso, o objeto `mean` é uma função e o uso do comando irá abrir o painel de ajuda sobre ela.

No R, toda tela de ajuda de uma função é igual, conforme se vê na Figura 2.5 apresentada a seguir. Esta mostra uma descrição da função `mean`, seus argumentos de entrada explicados e também o seu objeto de saída. A tela de ajuda segue com referências e sugestões para outras funções relacionadas. Mais importante, os exemplos de uso da função aparecem por último e podem ser copiados e colados para acelerar o aprendizado no uso da função.



mean {base} R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

x	An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.
trim	the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
na.rm	a logical value indicating whether NA values should be stripped before the computation proceeds.

Figura 2.5 Tela de ajuda da função mean

Caso quiséssemos procurar um termo nos arquivos de ajuda, bastaria utilizar o comando `??"standard deviation"`. Essa operação irá procurar a ocorrência do termo em todos os pacotes do R e é muito útil para aprender como realizar alguma operação, nesse caso o cálculo de desvio-padrão.

******ebook converter DEMO Watermarks*******

Como sugestão, o ponto inicial e mais direto para aprender uma nova função é observando o seu exemplo de uso, localizada no final da página de ajuda. Com isto, podes verificar quais tipos de objetos de entrada a mesma aceita e qual o formato e o tipo de objeto na sua saída. Após isso, leia atentamente a tela de ajuda para entender se a mesma faz exatamente o que esperas e quais são as suas opções de uso nas respectivas entradas. Caso a função realizar o procedimento desejado, podes copiar e colar o exemplo de uso para o teu próprio *script*, ajustando onde for necessário.

Outra fonte muito importante de ajuda é a própria internet. Sites como *stackoverflow* e *mailing lists* específicos do R, cujo conteúdo também está na internet, são fontes preciosas de informação. Havendo alguma dúvida que não foi possível solucionar via leitura dos arquivos de ajuda do R, vale o esforço de procurar uma solução via mecanismo de busca na internet. Em muitas situações, o seu problema, por mais específico que seja, já ocorreu e já foi solucionado por outros usuários.

Caso estiver recebendo uma mensagem de erro enigmática, vale copiar e colar a mesma para uma pesquisa no Google. Aqui apresenta-se outro benefício do uso da língua inglesa. É mais provável que encontres a solução se o erro for escrito inglês, dado o maior número de usuários. Caso não encontrar uma solução desta forma, podes inserir uma pergunta no stackoverflow ou no grupo Brasileiro do R no Facebook. Uma sugestão aqui é que poste junto um exemplo reproduzível do seu problema, de forma a facilitar a reconstituição da mensagem de erro. Veja um tutorial sobre isto neste link.

******ebook converter DEMO Watermarks*******

2.19 Pacotes do R

Um dos grandes benefícios do uso do R é o seu acervo de pacotes. Esses representam um conjunto de procedimentos agrupados em uma coleção de funções e voltados para a resolução de um problema qualquer. O R tem em sua essência uma filosofia de colaboração. Usuários disponibilizam os seus códigos para outras pessoas utilizarem. E, mais importante, **todos os pacotes são gratuitos**, assim como o R. Por exemplo, considere um caso em que está interessado em baixar dados da internet sobre o desemprego histórico no Brasil. Para isso, basta procurar e instalar o pacote específico que realiza esse procedimento.

Esses pacotes podem ser instalados de diferentes fontes, com as principais sendo **CRAN** (*The Comprehensive R Archive Network*), **R-Forge** e **Github**. A cada dia aumenta a quantidade e diversidade de pacotes existentes para o R. O próprio autor deste livro possui diversos pacotes disponíveis no CRAN, cada um para resolver algum problema diferente. Na grande maioria, são pacotes para importar e organizar dados financeiros.

O CRAN é o repositório oficial do R e é livre. Qualquer pessoa pode enviar um pacote e todo código enviado está disponível na internet. Existe, porém, um processo de avaliação que o código passa e certas normas rígidas devem ser respeitadas sobre o formato do código, o manual do usuário e a forma de atualização do pacote. Para quem tiver interesse, um tutorial claro e fácil de seguir é apresentado no site http://r-pkgs.had.co.nz/intro.html. As regras completas estão disponíveis no site do CRAN - https://cran.r-project.org/web/packages/policies.html. A adequação do código a essas normas é responsabilidade do desenvolvedor e gera um trabalho significativo, principalmente na primeira submissão.

A lista completa de pacotes disponíveis no CRAN, juntamente com uma breve descrição, pode ser acessada no link *packages* do site do R - https://cran.r-project.org/. Uma maneira prática de verificar a existência de um pacote para um procedimento específico é carregar a página anterior e procurar no seu navegador de internet a palavra-chave que define o seu procedimento. Caso existir o pacote com a palavra-chave, a procura acusará o encontro do termo na descrição do pacote.

Outra fonte importante para o encontro de pacotes é o *Task Views*, em que são destacados os principais pacotes de acordo com a área e o tipo de uso. Veja a tela do *Task Views* na Figura .

******ebook converter DEMO Watermarks*******



CRAN Task Views

Bayesian	Bayesian Inference
ChemPhys	Chemometrics and Computational Physics
ClinicalTrials	Clinical Trial Design, Monitoring, and Analysis
Cluster	Cluster Analysis & Finite Mixture Models
DifferentialEquations	Differential Equations
Distributions	Probability Distributions
Econometrics	Econometrics
Environmetrics	Analysis of Ecological and Environmental Data
ExperimentalDesign	Design of Experiments (DoE) & Analysis of Experimental Data
ExtremeValue	Extreme Value Analysis
Finance	Empirical Finance
Genetics	Statistical Genetics
Graphics	Graphic Displays & Dynamic Graphics & Graphic Devices & Visualization
HighPerformanceComputing	High-Performance and Parallel Computing with R
MachineLearning	Machine Learning & Statistical Learning
MedicalImaging	Medical Image Analysis
MetaAnalysis	Meta-Analysis
Multivariate	Multivariate Statistics
NaturalLanguageProcessing	Natural Language Processing
NumericalMathematics	Numerical Mathematics
OfficialStatistics	Official Statistics & Survey Methodology
Optimization	Optimization and Mathematical Programming
Pharmacokinetics	Analysis of Pharmacokinetic Data
Phylogenetics	Phylogenetics, Especially Comparative Methods
Psychometrics	Psychometric Models and Methods
ReproducibleResearch	Reproducible Research
Robust	Robust Statistical Methods

Figura 2.6 Tela do Task Views

Ao contrário do CRAN, o R-Forge e o Github não possuem restrição quanto ao código enviado e, devido a isso, tendem a ser escolhidos por alguns desenvolvedores. A responsabilidade de uso, porém, é do próprio usuário. Na prática, é muito comum os desenvolvedores manterem uma versão em desenvolvimento no Github e outra oficial no CRAN. Quando a versão em desenvolvimento atinge um estágio de maturidade, a mesma é enviada ao CRAN.

O mais interessante no uso de pacotes é que estes podem ser acessados e instalados diretamente no R via a internet. Para saber qual é a quantidade atual de pacotes no CRAN, digite e execute os seguintes comandos no *prompt*:

```
# find current available packages
df.cran.pkgs <- available.packages()

# get size of matrix
n.cran.packages <- nrow(df.cran.pkgs)

# print it
print(n.cran.packages)

## [1] 12588
```

Atualmente, 2018-06-10 14:55:36, existem 12588 pacotes disponíveis nos servidores do CRAN.

Também se pode verificar a quantidade de pacotes localmente instalados com o comando `installed.packages()`:

```
# get number of local (installed) packages
n.local.packages <- nrow(installed.packages())

# print it
print(n.local.packages)
```

******ebook converter DEMO Watermarks*******

```
## [1] 438
```

Nesse caso, o computador em que o livro foi escrito possui 438 pacotes do R instalados. Note que, apesar do autor ser um experiente programador do R, apenas uma pequena fração do pacotes disponíveis no CRAN está sendo usado! Esse valor é provavelmente diferente do seu. Sugere-se rodar esses códigos para verificar o número de pacotes instalados na sua máquina.

2.19.1 Instalando Pacotes do CRAN

Para instalar um pacote, basta utilizar o comando `install.packages`. Como exemplo, vamos instalar um pacote que será utilizado nos capítulos futuros, o `readr`:

```
# install pkg readr
install.packages('readr')
```

Copie e cole este comando no *prompt* e pronto! O R irá baixar os arquivos necessários e instalar o pacote `readr` e suas dependências. Após isto, as funções relativas ao pacote estarão prontas para serem usadas. Observe que definimos o nome do pacote na instalação como se fosse texto, com o uso das aspas (`"`). Caso o pacote instalado seja dependente de outro pacote, o R automaticamente instala o mesmo. Assim, todos os requerimentos para o uso do respectivo pacote já serão satisfeitos e tudo funcionará perfeitamente. É possível, porém, que um pacote tenha uma dependência externa. Como um exemplo, o pacote `RndTexExams` depende da existência de uma instalação do LaTex. Geralmente essa é anunciada na sua descrição e um erro é sinalizado na execução do programa quando o LaTex não é encontrado. Fique atento, portanto, a esses casos.

Aproveitando o tópico, sugiro que o leitor já instale todos os pacotes do `tidyverse` com o seguinte código:

```
# install pkgs from tidyverse
install.packages('tidyverse')
```

2.19.2 Instalando Pacotes do Github

Para instalar um pacote diretamente do Github, é necessário instalar antes o pacote *devtools*, disponível no CRAN:

```
# install devtools
install.packages('devtools')
```

Após isto, utilize função `devtools::install_github` para instalar um pacote diretamente do Github. No exemplo a seguir instalamos a versão em desenvolvimento do pacote `ggplot2`, cuja versão oficial também está disponível no CRAN:

```
# install ggplot2 from github
devtools::install_github("hadley/ggplot2")
```

Observe que o nome do usuário do repositório também é incluído. No caso anterior, o nome *hadley* pertence ao desenvolvedor do `ggplot2`, Hadley Wickham. No decorrer do livro notará

******ebook converter DEMO Watermarks*******

que esse nome aparecerá diversas vezes, dado que Hadley é um prolífico e competente desenvolvedor de diversos pacotes do R.

Um aviso aqui é importante. **Os pacotes do github não são moderados. Qualquer pessoa pode enviar código para lá e o conteúdo não é checado de forma independente.** Nunca instale pacotes do github sem conhecer os autores. Apesar de improvável, é possível que esses possuam código malicioso.

2.19.3 Carregando Pacotes

Dentro de uma rotina de pesquisa, utilizamos a função `library` para carregar um pacote. Cada pacote deve ser carregado a cada sessão. Alguns pacotes, tal como o `base` com os comandos básicos do R, são inicializados automaticamente. A grande maioria, porém, deve ser carregada no início dos *scripts*. Veja o exemplo a seguir:

```
# load dplyr
library(dplyr)
```

A partir disso, todas as funções do pacote estarão disponíveis para o usuário. Note que não é necessário utilizar aspas (") ao carregar o pacote. Caso utilize uma função específica do pacote e não deseje carregar todo ele, pode fazê-lo através do uso do símbolo especial `::`, conforme o exemplo a seguir.

```
# call fct fortune() from pkg fortune
fortunes::fortune(10)
```

```
##
## Overall, SAS is about 11 years behind R and S-Plus in
## statistical capabilities (last year it was about 10 years
## behind) in my estimation.
##    -- Frank Harrell (SAS User, 1969-1991)
##       R-help (September 2003)
```

Nesse caso, utilizamos a função `fortune` do próprio pacote `fortunes`, o qual mostra na tela uma frase possivelmente engraçada escolhida do *mailing list* do R. Nesse caso, selecionamos a mensagem número 10. Se não tiver disponível o pacote, o R mostrará a seguinte mensagem de erro:

```
## Error in library("fortune") : there is no package called "fortune"
```

Para resolver, utilize o comando `install.packages("fortunes")` para instalar o pacote no seu computador. Execute o código `fortunes::fortune(10)` no *prompt* para confirmar a instalação. Toda vez que se deparar com essa mensagem de erro, deves instalar o pacote que está faltando.

Outra maneira de carregar um pacote é através da função `require`. Essa tem um comportamento diferente da função `library` e deve ser utilizada dentro da definição de funções. Caso o usuário crie uma função customizada que necessite de procedimentos de um pacote em particular, o mesmo deve carregar o pacote no escopo da função. Por exemplo, veja o código a seguir, em que criamos uma função dependente do pacote `quantmod`:

******ebook converter DEMO Watermarks*******

```
my.fct <- function(x){
  require(quantmod)

  df <- getSymbols(x, auto.assign = F)
  return(df)
}
```

Nesse caso, a função `getSymbols` faz parte do pacote `quantmod`. Não se preocupe agora com a estrutura utilizada para criar uma função no R. Essa será explicada em capítulo futuro.

2.19.4 Atualizando Pacotes

Ao longo do tempo, é natural que os pacotes disponibilizados no CRAN sejam atualizados para acomodar novas funcionalidades ou se adaptar a mudanças em suas dependências, como uma mudança no código base do R. Assim, é recomendável que os usuários atualizem os seus pacotes instalados para uma nova versão através da internet. Esse procedimento é bastante fácil. Uma maneira direta de atualizar pacotes é clicar no botão *update* no painel de pacotes no canto direito inferior do RStudio, conforme mostrado na figura 2.7.

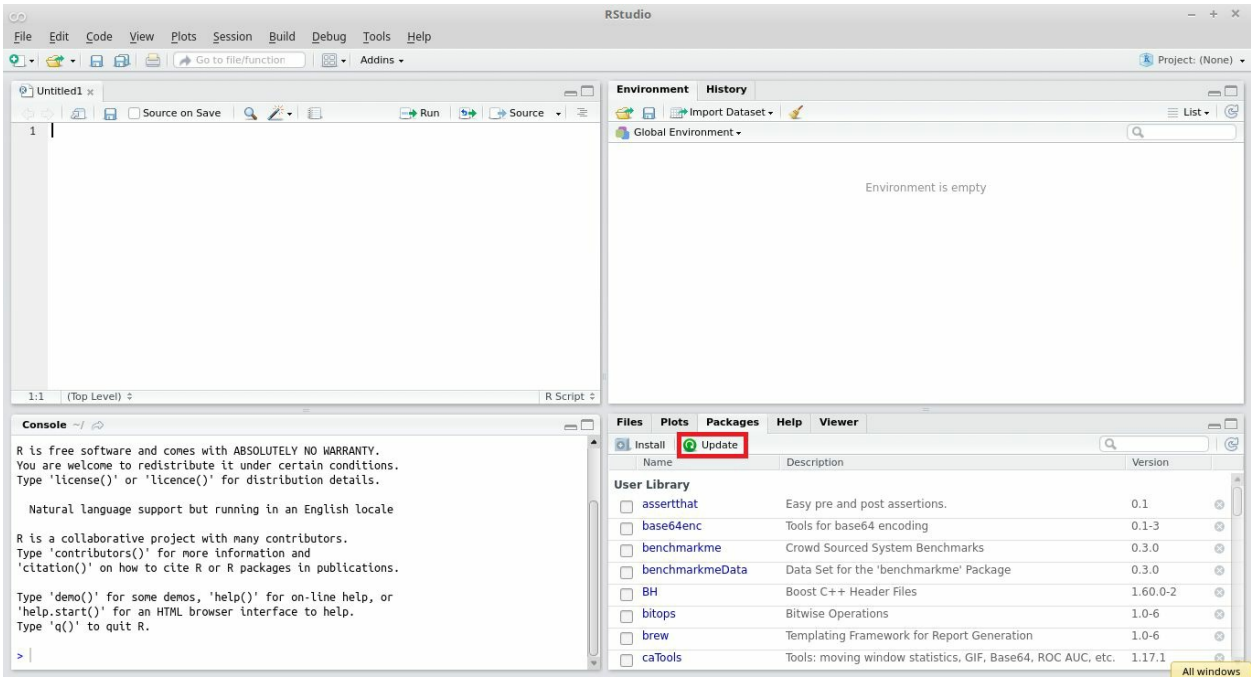


Figura 2.7 Atualizando pacotes no R

A atualização de pacotes através do *prompt* também é possível. Para isso, basta utilizar o comando `update.packages`, conforme mostrado a seguir.

```
update.packages()
```

O comando `update.packages()` compara a versão dos pacotes instalados em relação a versão disponível no CRAN. Caso tiver alguma diferença, a nova versão é instalada. Após a execução do comando, todos os pacotes estarão atualizados com a versão disponível nos servidores do

******ebook converter DEMO Watermarks*******

CRAN.

******ebook converter DEMO Watermarks*******

2.20 Utilizando *Code Completion* com a Tecla *tab*

Um dos recursos mais úteis do RStudio é o preenchimento automático de código (*code completion*). Essa é uma ferramenta de edição que facilita o encontro de nomes de objetos, nome de pacotes, nome de arquivos e nomes de entradas em funções. O seu uso é muito simples. Após digitar um texto qualquer, basta apertar a tecla *tab* e uma série de opções aparecerá. Veja a Figura 2.8 apresentada a seguir, em que, após digitar a letra *f* e apertar *tab*, aparece uma janela com uma lista de objetos que iniciam com a respectiva letra.

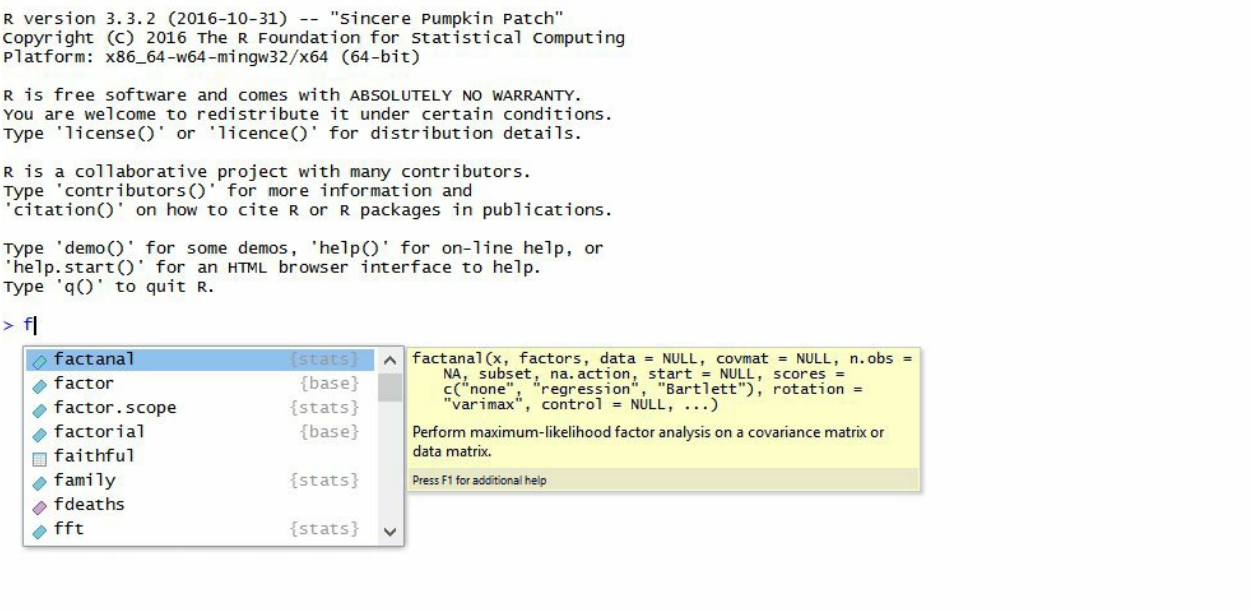


Figura 2.8 Uso do autocomplete para objetos

Essa ferramenta também funciona para pacotes. Para verificar, digite `library(r)` no *prompt* ou no editor, coloque o cursor entre os parênteses e aperte *tab*. O resultado deve ser algo parecido com a figura 2.9, mostrada a seguir.

******ebook converter DEMO Watermarks*******

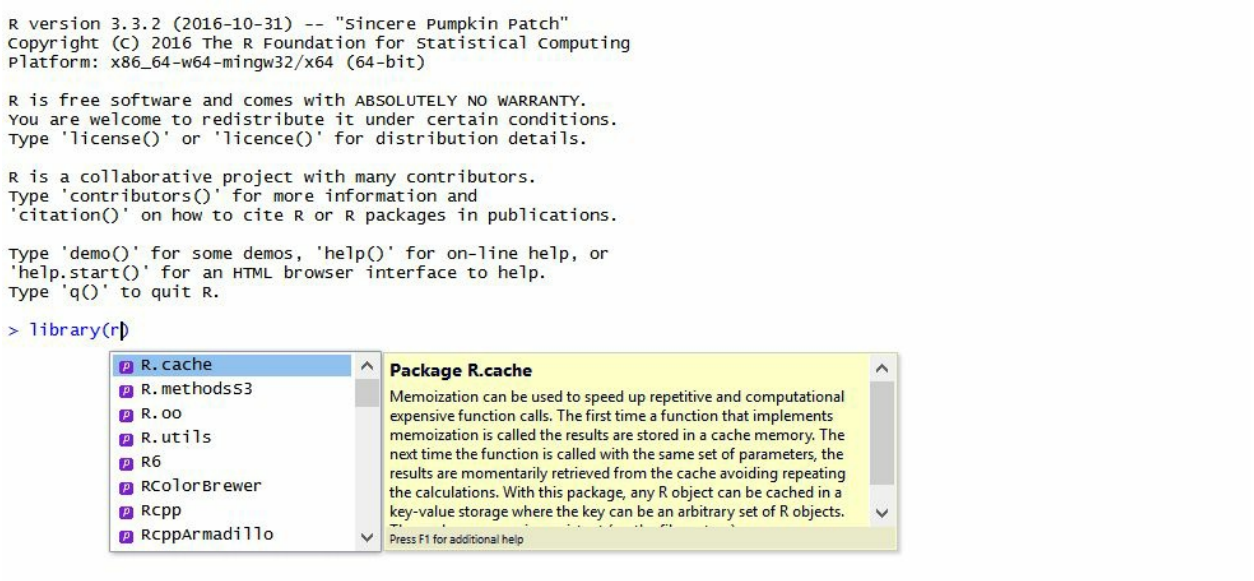


Figura 2.9 Uso do autocomplete para pacotes

Observe que uma descrição do pacote ou objeto também é oferecida. Isso facilita bastante o dia a dia, pois a memorização das funcionalidades e dos nomes dos pacotes e os objetos do R não é uma tarefa fácil. O uso do *tab* diminui o tempo de investigação dos nomes e evita possíveis erros de digitação na definição destes.

O uso dessa ferramenta torna-se ainda mais benéfico quando os objetos são nomeados com algum tipo de padrão. No restante do livro observarás que os objetos tendem a ser nomeados com o prefixo *my.*, como em *my.x*, *my.num*. O uso desse padrão facilita o encontro futuro do nome dos objetos, pois basta digitar *my.*, apertar *tab* e uma lista de todos os objetos criados pelo usuário aparecerá.

Outro uso do *tab* é no encontro de arquivos e pastas no computador. Basta criar uma variável como `my.file <- " "`, apontar o cursor para o meio das aspas e apertar a tecla *tab*. Uma tela com os arquivos e pastas do diretório atual de trabalho aparecerá, conforme mostrado na figura 2.10. Nesse caso específico, o R estava direcionado para a minha pasta de códigos, em que é possível enxergar diversos trabalhos realizados no passado.

******ebook converter DEMO Watermarks*******

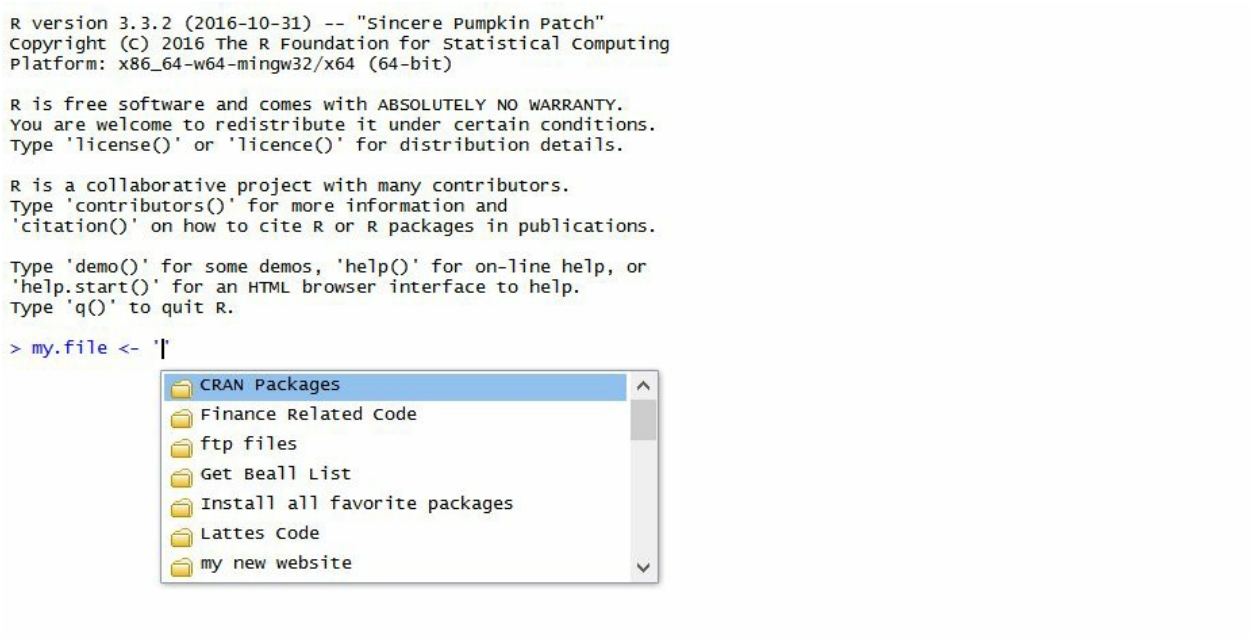


Figura 2.10 Uso do autocomplete para arquivos

Uma dica aqui é utilizar o *tab* com a raiz do computador. Assumindo que o disco do seu computador está alocado para `C:/`, digite `my.file <- "C:/"` e pressione *tab* após o símbolo `/`. Uma tela com os arquivos da raiz do computador aparecerá no RStudio. Podes facilmente navegar o sistema de arquivos utilizando as setas e *enter*.

O *autocomplete* também funciona para encontrar e definir as entradas de uma função. Por se tratar de um tópico mais avançado, deixamos o seu uso e demonstração para o capítulo 8.

******ebook converter DEMO Watermarks*******

2.21 Interagindo com Arquivos e o Sistema Operacional

Em muitas situações de uso do R será necessário interagir com os arquivos do computador, seja criando novas pastas, descompactando e compactando arquivos, listando e removendo arquivos do disco rígido do computador ou qualquer outro tipo de operação. Na grande maioria dos casos, o interesse é na manipulação de arquivos contendo dados.

2.21.1 Listando Arquivos e Pastas

Para listar arquivos do computador, basta utilizar o função `list.files`. O argumento `path` define o diretório para listar os arquivos. Na construção deste livro foi criado um diretório chamado *data*. Esse contém todos os dados necessários para recriar os exemplos do livro. Pode-se verificar os arquivos nessa pasta com o seguinte código:

```
my.f <- list.files(path = "data", full.names = TRUE)
print(my.f)
```

```
##  [1] "data/AdjustedPrices-InternacionalIndices.RDATA"
##  [2] "data/BGS_Cache"
##  [3] "data/BovStocks_2011-12-01_2016-11-29.csv"
##  [4] "data/BovStocks_2011-12-01_2016-11-29.RData"
##  [5] "data/example_gethfdata.RDATA"
##  [6] "data/FileWithLatinChar_ISO-8859-9.txt"
##  [7] "data/FileWithLatinChar.txt"
##  [8] "data/FileWithLatinChar_UTF-8.txt"
##  [9] "data/grunfeld.csv"
## [10] "data/HFData_6_Assets_15 min.RData"
## [11] "data/HFData.csv"
## [12] "data/HFData_equity_6_Assets_15 min_2016-09-12_2016-09-30.RData"
## [13] "data/Ibov.csv"
## [14] "data/IbovStocks_long.csv"
## [15] "data/Ibov_xls.xlsx"
## [16] "data/Ibov_xlsx.xlsx"
## [17] "data/MktIndices_and_Symbols.csv"
## [18] "data/MySQLiteDatabase.SQLITE"
## [19] "data/Ret_BR.csv"
## [20] "data/SP500_2011-11-13_2016-11-11.csv"
## [21] "data/SP500.csv"
## [22] "data/SP500-Excel.xlsx"
## [23] "data/SP500-Stocks_long.csv"
## [24] "data/SP500-Stocks_wide.csv"
## [25] "data/SP500-Stocks-WithRet.RData"
## [26] "data/SP500-Stocks-WithRet.rds"
## [27] "data/TDData.csv"
## [28] "data/temp.csv"
## [29] "data/temp.fst"
## [30] "data/temp_fst.fst"
## [31] "data/temp.RData"
## [32] "data/temp.rds"
## [33] "data/temp_rds.rds"
## [34] "data/temp.txt"
## [35] "data/temp_writexl.xlsx"
## [36] "data/temp.xlsx"
## [37] "data/temp_xlsx.xlsx"
```

******ebook converter DEMO Watermarks*******

```
## [38] "data/temp_xts.RData"
```

Observe que nesse diretório encontram-se vários arquivos *.csv*, *.rds* e *.xlsx*. Esses contêm dados que serão utilizados em capítulos futuros a respeito da importação e exportação de dados. Recomenda-se utilizar o argumento `full.names` como `TRUE`, o qual faz com que o retorno da função `list.files` contenha o caminho completo do arquivo. Isso facilita, por exemplo, uma possível importação de dados em que é necessário indicar não somente o nome do arquivo, mas a sua localização completa no computador. Destaca-se que também é possível listar os arquivos de forma recursiva, isto é, listar os arquivos de subpastas do endereço original. Para verificar, tente utilizar o seguinte código no seu computador:

```
# list all files recursively
list.files(path = getwd(), recursive = T, full.names = TRUE)
```

O comando anterior irá listar todos os arquivos existentes na pasta atual e subpastas de trabalho. Dependendo de onde o comando foi executado, pode levar um certo tempo para o término do processo. Caso precisar cancelar a execução, aperte *esc* no teclado.

Para listar pastas (diretórios) do computador, basta utilizar o comando `list.dirs`. Veja a seguir.

```
# list directories
my.dirs <- list.dirs(recursive = F)
print(my.dirs)
```

```
##  [1] "./BGS_Cache"
##  [2] "./_book"
##  [3] "./_bookdown_files"
##  [4] "./data"
##  [5] "./DFP Cache Folder"
##  [6] "./eqs"
##  [7] "./fig_ggplot"
##  [8] "./figs"
##  [9] "./ftp files"
## [10] "./html_files"
## [11] "./latex_files"
## [12] "./many_datafiles"
## [13] "./Other scripts"
## [14] "./ProcAnFinDataR_pt_ed_2_cache"
## [15] "./ProcAnFinDataR_pt_ed_2_files"
## [16] "./.Rproj.user"
## [17] "./Scripts"
## [18] "./tabs"
## [19] "./TD Files"
```

No caso anterior, o comando lista todos os diretórios do trabalho atual sem recursividade. A saída do comando mostra os diretórios que utilizei para escrever este livro. Isso inclui o diretório de saída do livro (`./_book`), o diretório com os dados (`./data`), entre outros. Nesse mesmo diretório, encontram-se os capítulos do livro, organizados por arquivos e baseados na linguagem *RMarkdown* (`.Rmd`). Para listar somente os arquivos com extensão `.Rmd`, utiliza-se o argumento `pattern` da função `list.files`, como a seguir:

```
list.files(pattern = "*.Rmd$")
```

```
##  [1] "00a-Sobre-NovaEdicao.Rmd"
```

******ebook converter DEMO Watermarks*******

```
##  [2] "00b-Prefacio.Rmd"
##  [3] "01-Introducao.Rmd"
##  [4] "02-Operacoes-Basicas.Rmd"
##  [5] "03-Scripts-Pesquisa.Rmd"
##  [6] "04-Importacao-Exportacao-Local.Rmd"
##  [7] "05-Importacao-Internet.Rmd"
##  [8] "06-Objetos-Armazenamento.Rmd"
##  [9] "07-Objetos-Basicos.Rmd"
## [10] "08-Programacao-com-R.Rmd"
## [11] "09-Limpando-Estruturando-Dados.Rmd"
## [12] "10-Figuras.Rmd"
## [13] "11-Modelagem.Rmd"
## [14] "12-Reportando-resultados.Rmd"
## [15] "13-Exemplos-Pesquisa.Rmd"
## [16] "14-Referencias.Rmd"
## [17] "_BemVindo.Rmd"
## [18] "index.Rmd"
## [19] "ProcAnFinDataR_pt_ed_2.Rmd"
```

O texto `*.Rmd$` orienta o R a procurar todos arquivos que terminam o seu nome com o texto *.Rmd*. Os símbolos `'*''` e `'$'` são operadores específicos para o encontro de padrões em texto em uma linguagem chamada regex (*regular expressions*). Os arquivos apresentados anteriormente contêm todo o conteúdo deste livro, incluindo este próprio parágrafo, localizado no arquivo `02-OperacoesBasicas.Rmd`!

2.21.2 Apagando Arquivos e Diretórios

A remoção de arquivos é realizada através do comando `file.remove`:

```
# create temporary file
my.file <- 'data/tempfile.csv'
write.csv(x = data.frame(x=1:10),
          file = my.file)

# delete it
file.remove(my.file)
```

```
## [1] TRUE
```

Lembre-se que deves ter permissão do seu sistema operacional para apagar um arquivo. Para o nosso caso, o retorno `TRUE` mostra que a operação teve sucesso.

Para deletar diretórios e todos os seus elementos, utilizamos `unlink`:

```
# create temp dir
dir.create('temp')

# fill it with file
my.file <- 'temp/tempfile.csv'
write.csv(x = data.frame(x=1:10),
          file = my.file)

unlink(x = 'temp', recursive = TRUE)
```

A função, neste caso, não retorna nada. Podes checar se o diretório existe com `dir.exists`:

******ebook converter DEMO Watermarks*******

```
dir.exists('temp')
```

```
## [1] FALSE
```

2.21.3 Baixando Arquivos da Internet

O R pode baixar arquivos da Internet diretamente no código. Isso é realizado com a função `download.file`. Veja o exemplo a seguir, onde baixamos uma planilha de Excel do site da Microsoft:

```
# set link
link.dl <- 'go.microsoft.com/fwlink/?LinkID=521962'
local.file <- 'temp_file.xlsx' # name of local file

download.file(url = link.dl,
              destfile = local.file)
```

O uso de `download.file` é bastante prático quando se está trabalhando com dados da Internet que são constantemente atualizados. Basta baixar e atualizar o arquivo com dados no início do *script*. Poderíamos continuar a rotina lendo o arquivo baixado e realizando a nossa de análise.

Um exemplo nesse caso é a tabela de empresas listadas na bolsa divulgada pela CVM (comissão de valores mobiliários). A tabela está disponível em um arquivo zipado no site. Podemos baixar o arquivo, descompactá-lo e depois ler a tabela para analisar os dados.

```
library(readr)
library(dplyr)

# set destination link and file
my.link <- 'http://sistemas.cvm.gov.br/cadastro/SPW_CIA_ABERTA.ZIP'
my.destfile <- 'temp.zip'

# download file
download.file(my.link, destfile = my.destfile)

# read it
df.cvm <- read_delim(my.destfile,
                     delim = '\t',
                     locale = locale(encoding = 'Latin1'))

# delete temp file
file.remove(my.destfile)
```

```
## [1] TRUE
```

```
# check available columns
print(names(df.cvm))
```

```
##  [1] "CD_CVM"             "DENOM_SOCIAL"
##  [3] "DENOM_COMERC"       "SETOR_ATIV"
##  [5] "PF_PJ"              "CNPJ"
##  [7] "DT_REG"             "DT_CONST"
##  [9] "DT_CANCEL"          "MOTIVO_CANCEL"
## [11] "SIT_REG"            "DT_INI_SIT"
## [13] "SIT_EMISSOR"        "DT_INI_SIT_EMISSOR"
## [15] "CATEG_REG"          "DT_INI_CATEG"
```

******ebook converter DEMO Watermarks*******

```
## [17] "AUDITOR"            "CNPJ_AUDITOR"
## [19] "TP_ENDER"           "LOGRADOURO"
## [21] "COMPL"              "BAIRRO"
## [23] "CIDADE"             "UF"
## [25] "PAIS"               "CD_POSTAL"
## [27] "TEL"                "FAX"
## [29] "EMAIL"              "TP_RESP"
## [31] "RESP"               "DT_INI_RESP"
## [33] "LOGRADOURO_RESP"    "COMPL_RESP"
## [35] "BAIRRO_RESP"        "CIDADE_RESP"
## [37] "UF_RESP"            "PAIS_RESP"
## [39] "CEP_RESP"           "TEL_RESP"
## [41] "FAX_RESP"           "EMAIL_RESP"
```

Existem diversas informações interessantes nestes dados. E, mais importante, o arquivo está sempre atualizado. O código anterior estará sempre buscando os dados mais recentes a cada execução.

2.21.4 Utilizando Arquivos e Diretórios Temporários

Um aspecto interessante do R é que ele possui um folder temporário que é criado na inicialização do programa. Esse folder serve para guardar quaisquer arquivos descartáveis gerados pelo R. A cada nova sessão do R, um novo diretório temporário é criado. O endereço atual desse diretório é verificado com `tempdir`:

```
windows.tempdir <- tempdir()
print(windows.tempdir)
```

```
## C:\Users\MARCEL~1\AppData\Local\Temp\Rtmp8Emo2H
```

O último texto do diretório, neste caso `'Rtmp8Emo2H'` é aleatóriamente definido e irá trocar a cada nossa sessão do R. Ao reinicializarmos o computador, todas pastas temporárias serão deletadas.

A mesma dinâmica é encontrada para nomes de arquivos. Caso queira, por algum motivo, utilizar um nome temporário e aleatório para algum arquivo, utilize `tempfile`:

```
windows.tempfile <- tempfile()
cat(windows.tempfile)
```

```
## C:\Users\MARCEL~1\AppData\Local\Temp\Rtmp8Emo2H\file33f84f149a2b
```

Um caso prático no uso de `tempfile` é o *download* da planilha Excel anterior. Dado que iremos sobrescrever o arquivo a cada execução de código, não precisamos manter o mesmo na pasta de trabalho. Podemos então baixar o arquivo na pasta temporária, a qual será apagada futuramente. Veja a seguir:

```
# set link
link.dl <- 'go.microsoft.com/fwlink/?LinkID=521962'
local.file <- paste0(tempfile(),'.xlsx')

download.file(url = link.dl,
              destfile = local.file)
```

******ebook converter DEMO Watermarks*******

Nesse caso, não precisamos apagar o arquivo pois o mesmo será removido da memória do computador quando reinicializarmos o sistema.

******ebook converter DEMO Watermarks*******

2.22 Exercícios

1. Crie um novo *script*, salve o mesmo em uma pasta pessoal e defina dois objetos: um contendo uma sequência entre 1 e 100 e outro com o texto do seu nome (ex. `'Ricardo'`). Execute o código com os atalhos no teclado.

2. No *script* anterior, use o objeto de texto criado e mostre a seguinte frase no *prompt* do R: `"My name is ...."`.

3. Verique qual o diretório de trabalho atual da sua sessão do R. Modifique o diretório de trabalho para a área de *Desktop* e mostre na tela a seguinte mensagem: `'My desktop address is ....'`.

4. Utilize o R para baixar o arquivo compactado com o material do livro, disponível em https://msperlin.github.io/content/pafdr%20files/R%20code%20and%20data%20(zip).zip. Salve o mesmo como um arquivo na pasta temporária da sessão (veja função `tempdir`).

5. Utilize a função `unzip` para descompactar o arquivo baixado na questão anterior e, após a operação, utilize função `file.remove` para remover o arquivo zipado do computador, após a sua descompactação. Quantos arquivos estão disponíveis no arquivo zipado?

6. Toda vez que o usuário instala um pacote do R, os arquivos desse são armazenados localmente em uma pasta específica do computador. Utilizando comando `Sys.getenv('R_LIBS_USER')` e `list.dirs`, liste todos os diretórios desta pasta. Quantos pacotes estão disponíveis nesta pasta?

7. No mesmo assunto do exercício anterior, liste todos os arquivos em todas as subpastas do diretório contendo os arquivos dos diferentes pacotes. Em média, quantos arquivos são necessários para cada pacote?

8. Instale pacote `GetTDData` no seu computador. Leia o manual do pacote e utilize a função `GetTDData::GetIbovStocks` para baixar a curva de juros atual no sistema financeiro Brasileiro. Verifique o exemplo de uso da função para entender como utilizá-la.

9. Utilizando pacote `devtools`, instale a versão de desenvolvimento do pacote `ggplot2` no repositório de Hadley Hickman. Carregue o pacote usando `library` e crie uma figura simples com o código `qplot(y = rnorm(10), x = 1:10)`.

10. **DESAFIO** - Utilizando sua capacidade de programação, verifique no seu computador qual pasta, a partir do diretório de raiz, possui o maior número de arquivos. Apresente na tela do R as cinco pastas com maior número de arquivos.

******ebook converter DEMO Watermarks*******

******ebook converter DEMO Watermarks*******

3 Desenvolvendo Rotinas de Pesquisa

No capítulo anterior aprendemos a utilizar o R e RStudio para operações básicas tal como a criação de *scripts*, manipulação de objetos no R, mudança de diretório, entre outras. Antes de passarmos para a importação de dados, é necessário discutirmos como estruturar e organizar uma rotina de pesquisa.

Este aspecto é importante pois uma estrutura de trabalho organizada facilita o uso e compartilhamento do código no futuro. Neste capítulo iremos tratar das etapas de pesquisa e a organização de arquivos no computador, incluindo dados e rotinas. O principal objetivo aqui é mostrar e justificar um formato de trabalho que facilite o desenvolvimento e compartilhamento de código.

******ebook converter DEMO Watermarks*******

3.1 Etapas de uma Pesquisa

Um *script* de pesquisa pode ser organizado em quatro etapas consecutivas:

1. **Importação dos dados**: Dados crus (originais) são importados de algum local, seja da internet, seja de arquivos físicos disponíveis no computador.

2. **Limpeza e estruturação dos dados**: Dados importados na fase anterior são processados em uma etapa de limpeza e estruturação. Isto é, registros anormais e erros em observações são eliminados ou tratados. Novas variáveis de interesse são criadas com base nos dados importados. A estrutura dos dados originais também pode ser refeita de acordo com a necessidade. O resultado final dessa etapa deve ser, preferencialmente, uma tabela final com todos os dados necessários para a análise do problema.

3. **Modelagem e teste de hipóteses**: O grupo de dados criado no passo anterior é então analizado em algum tipo de estimação de modelo ou teste de hipótese. Essa etapa refere-se ao próprio problema da pesquisa. Na grande maioria dos casos, essa é a fase que exigirá maior trabalho, visto que representa o coração da pesquisa. Essa etapa termina com a criação de objetos que serão exportados na etapa seguinte.

4. **Reportando os resultados**: A última etapa é a criação dos objetos de interesse a serem reportados no artigo ou relatório. Esses objetos geralmente são referentes a tabelas e figuras, que podem ser exportados como arquivos externos e futuramente importados em um relatório.

Cada uma das etapas anteriores pode ser estruturada em um único arquivo *.R* ou em vários arquivos separados. O uso de vários arquivos é preferível quando as etapas de cada pesquisa demandarem muito tempo de processamento. Por exemplo, na importação e organização de uma base de dados de grande volume, vale a pena separar os procedimentos em arquivos diferentes. Um caso prático seria a análise de dados de transações financeiras no mercado de capitais onde, como exemplo, um único dia de dados pode chegar a ocupar 500 MB de espaço no computador. A importação e limpeza desses dados leva muito tempo de processamento.

Uma estratégia inteligente de organização seria inserir os procedimentos de importação e limpeza em um arquivo *.R* e, na última linha do código, salvar os dados limpos em um arquivo externo. A etapa posterior de modelagem e teste de hipóteses importa os dados já salvos, sem precisar limpar novamente os dados originais a cada vez que houver uma modificação nos testes da pesquisa. Essa simples organização dos arquivos salva bastante tempo na prática de pesquisa, pois, geralmente, a fase de modelagem e pesquisa é a que exige maiores modificações no desenvolvimento.

Caso você for trabalhar com diversos arquivos, uma sugestão é criar uma estrutura de nomenclatura que defina as etapas da pesquisa. Um exemplo seria nomear o código de importação de dados como `01-Import-and-clean-data.R`, o código de modelagem como `02-build-report-models.R` e assim sucessivamente. O efeito prático é que o uso de um contador na primeira letra do nome do arquivo faz com que a ordem de execução do programa fique bastante

******ebook converter DEMO Watermarks*******

clara. Indo além, você pode criar um *script* mestre chamado `00-run-it-all.R` que roda (comando `source`) todos as outras rotinas. Assim, toda vez que realizar uma atualização nos dados originais, você pode simplesmente executar `00-run-it-all.R` e terá os novos resultados, sem necessitar rodar cada *script* individualmente.

******ebook converter DEMO Watermarks*******

3.2 A Estrutura de Diretórios

Uma estrutura de organização de diretórios também beneficia a reprodutibilidade e organização da pesquisa. Para rotinas simples, com uma base de dados única e um baixo número de procedimentos, não é necessário dispender muito tempo pensando na organização dos arquivos nos diretórios. Para programas mais complexos, onde existem diversas etapas de pesquisa e diversas bases de dados, uma segmentação dos diretórios é não somente recomendada como essencial.

Uma estrutura de diretórios que considero eficiente é criar um diretório único para a pesquisa e, dentro dele, criar subdiretórios para cada elemento de entrada e saída. Por exemplo, você pode criar um subdiretório chamado `data`, onde todos os dados originais serão guardados, um diretório `figs` e um `tabs`, para onde figuras e tabelas com resultados de interesse serão exportadas. Para funções utilizadas na pesquisa, você pode também criar um diretório chamado `R-Fcts`. Todos os *scripts* principais da pesquisa, e nada mais, devem ficar na raiz do diretório de pesquisa. Um exemplo da estrutura de arquivos seria:

```
/My Research about capital markets/
    /data/
        datafile1.csv
        datafile2.csv
        datafile2.csv
    /figs/
        MyBeautifulfig.png
    /tabs/
        Table_with_publishable_results.xlsx
    /R-Fcts/
        fct_estimate_model.R
        fct_get_results.R
        fct_read_my_files.R
    00-run-it-all.R
    01-import-and-clean-data.R
    02-run-research.R
```

Destaco que o formato anterior torna o código da pesquisa auto-contido, isto é, todos os arquivos necessários para a pesquisa estão disponibilizados no mesmo diretório. Os benefícios desse tipo de formatação do diretório de trabalhos são claros. Caso você precise compartilhar o código com outros pesquisadores, bastará comprimir o diretório para um formato tal como *.zip* e enviar para o destinatário. O mesmo, ao observar essa estrutura de arquivos, entenderá imediatamente como mudar os dados originais e a ordem de execução dos arquivos, além de compreender onde as saídas estarão registradas. Essa organização também facilita o uso futuro do código criado.

Seguindo a sugestão de um *script* mestre, um exemplo comentado para o conteúdo do arquivo `00-run-it-all.R` seria:

```
# clean up workspace
rm(list=ls())

# close all figure windows created with x11()
graphics.off()
```

******ebook converter DEMO Watermarks*******

```
# load packages
library(pkg1)
library(pkg2)
library(pkg3)

# change directory
my.d <- 'C:/CHANGE_DIR_HERE'
setwd(my.d)

# list  functions in 'R-Fcts'
my.R.files <- list.files(path='R-Fcts',
                         pattern = '*.R',
                         full.names=TRUE)

# Load all functions in R
sapply(my.R.files,source)

# Import data script
source('01-import-and-clean-data.R')

# run models and report results
source('02-run-research.R')
```

Para o código anterior, as funções ainda não conhecidas são `graphics.off` e `sapply`. A primeira fecha todas janelas de gráficos abertas. Essas tendem a acumular no decorrer do trabalho e devem ser fechadas no início de um novo *script*. O comando `sapply` aplica uma função, nesse caso `source`, para uma série de elementos. O efeito prático em `sapply(my.R.files, source)` é que todos arquivos com extensão *.R* localizados na pasta `R-Fct` serão executados. Nos capítulos 10 e 8, respectivamente, iremos tratar destas funções.

Note que `00-run-it-all.R` é facilmente compartilhável. Para rodar o código anterior em um outro computador, basta trocar o diretório em `my.d`. A instalação dos pacotes utilizados na pesquisa também pode ser necessária. Caso o leitor quiser ir um passo além, pode também utilizar a função `file.copy` para copiar todos os arquivos de figuras para a pasta de escrita do artigo ou documento acadêmico. A partir disso, crie um link no texto para cada arquivo copiado anteriormente. Como exemplo, no LaTex você pode incluir um arquivo de figura com o comando `\includegraphics{filenamehere}`. Pode também criar um link direto entre o arquivo de escrita e a figura da pesquisa, apesar de esse método não ser recomendado, uma vez que ele cria uma dependência externa ao arquivo de escrita. Em ambas as formas, todas as figuras da pesquisa serão automaticamente atualizadas no texto e estarão sincronizadas com os arquivos provenientes do código da pesquisa. Para tabelas, a importação não é tão simples, pois uma tabela pode ser escrita em diversos formatos. Existem, porém, pacotes específicos para lidar com isso. No capítulo 12 estudaremos a forma correta de reportar resultados utilizando os pacotes `xtable` (Dahl 2016), `texreg` (Leifeld 2013), entre outros.

Outra maneira de trabalhar com um diretório de pesquisa é criando um projeto no RStudio. Para isso, abra o RStudio e vá em *File*, *New Project*, *New Directory*, *New Project* e escolha um diretório e nome do projeto. O RStudio irá então criar um arquivo com extensão *.RProj* no diretório escolhido. Toda vez que quiser trabalhar no projeto, basta abrir o arquivo de projeto no RStudio.

******ebook converter DEMO Watermarks*******

Os benefícios desta abordagem é que não é necessário mudar o diretório no código. O projeto automaticamente muda o mesmo para o local do arquivo *.RProj*. Diversas informações são salvas, incluindo rotinas em edição, histórico de comandos do projeto, entre outras coisas. Além disso, opções customizadas para o projeto podem ser feitas. Por curiosidade, o arquivo com extensão *.RProj* é puro texto e editável. Podes abrir o mesmo em qualquer editor de texto do seu sistema operacional. Caso o fizer, verás que o mesmo guarda diversas opções do R.

******ebook converter DEMO Watermarks*******

3.3 Pontos Importantes em uma Pesquisa

Aproveitando o tópico de execução de pesquisa, vou colocar aqui algumas sugestões para a realização de pesquisas com o R. Deixo claro que essas são posições pessoais, oriundas da minha experiência como pesquisador. Muitos pontos levantados aqui são específicos para o ambiente acadêmico, porém podem ser facilmente estendíveis para a prática de pesquisa fora das universidades.

Em primeiro lugar, conheça os seus dados! Entendo que o primeiro instinto ao se deparar com uma nova base de dados é instantaneamente importá-la no R e sair realizando análises. Aqui, um certo nível de cautela é necessário. Toda vez que se deparar com um conjunto de dados novos, se pergunte o quanto você **realmente** conhece esses dados:

- Como os dados foram coletados? Para que fim?
- Como estes dados se comparam com dados já utilizados em outros trabalhos?
- Existe alguma possibilidade de viés na forma de coleta dos dados?

Lembre-se que o propósito final de qualquer pesquisa é a comunicação. Certamente irás reportar os resultados para pessoas que irão ter algum tipo de opinião informada sobre a pesquisa. É provável que os avaliadores terão mais experiência que você no assunto, incluindo sobre as fontes e individualidades dos dados. Não desejo para ninguém estar em uma situação onde um esforço de pesquisa, com investimento de 3 a 6 meses de trabalho entre programação e escrita, é anulado por um simples lapso na checagem dos dados. Infelizmente, isso não é incomum.

Portanto, seja muito cauteloso sobre os dados que estás utilizando. Um detalhe que passa despercebido pode invalidar toda uma pesquisa. Caso tiver sorte e a base de dados vier acompanhada de um manual escrito, destrinche o mesmo até os últimos detalhes. Elenque as principais dúvidas em relação aos dados e, em caso das informações não estarem claras, não seja tímido em enviar os questionamentos para o responsável.

O segundo ponto é o código. Após terminar de ler este livro, terás o conhecimento para realizar pesquisa com o R. O computador será um poderoso aliado em tornar realidade as suas ideias de pesquisa, por mais gigantescas e mirabolantes que forem. Porém, **um grande poder vem acompanhado de grande responsabilidade**. Um erro de código pode facilmente inviabilizar ou tendenciar a sua pesquisa.

Lembre que analisar dados é a sua profissão e a sua reputação é o seu maior ativo. Caso não tenhas confiança no código produzido, não publique ou comunique os seus resultados. O código de sua pesquisa é de total responsabilidade sua e de mais ninguém. Verifique o mesmo quantas vezes for necessário. Seja, sempre, o avaliador mais criterioso do seu trabalho:

- As estatísticas descritivas das variáveis relatam fielmente a base de dados?
- Existe alguma relação entre as variáveis que pode ser verificada na tabela descritiva?
- Os resultados encontrados fazem sentido para a literatura atual do assunto? Caso não, como explicá-los?
- É possível que um *bug* no código tenha produzido o resultado encontrado?

******ebook converter DEMO Watermarks*******

Ainda me surpreendo como pesquisas submetidas a respeitados periódicos podem ser negadas a publicação baseado em uma simples análise da tabela descritiva dos dados construídos. Erros básicos de cálculos de variáveis são facilmente encontrados para um olho treinado, que sabe onde procurar. Esse processo de avaliação contínua da sua pesquisa não somente o deixará mais forte como pesquisador(a) mas também servirá de treino para a prática de avaliação de pares, muito utilizada na pesquisa acadêmica. Caso não tenhas confiança suficiente para reportar os resultados, teste o seu código ostensivamente. Caso já o tenha feito e ainda não estás confiante, identifique as linhas de código que tens mais dificuldade e busque ajuda com um colega ou o seu orientador, caso existir. Este último é um forte aliado que pode ajudá-lo com a sua maior experiência.

Todos o trabalho de pesquisa é, de certa forma, baseado em trabalhos já existentes. Atualmente é extremamente difícil realizar algum tipo de pesquisa que seja totalmente inovadora. O conhecimento é construído na forma de blocos, um encima do outro. Sempre existe uma parcela de literatura que deve ser consultada. Particularmente para o caso de pesquisa em dados, deves sempre comparar os seus resultados com os resultados já apresentados na literatura do assunto, principalmente quando é um estudo replicado. Caso os resultados principais não forem semelhantes aos encontrados na literatura, questione-se o porquê disso. Será que um erro de código pode ter criado esse resultado?

Deixo claro que é possível sim que resultados de uma pesquisa sejam diferentes dos da literatura, porém, o contrário é mais provável. O conhecimento disso demanda cuidado com o seu código. *Bugs* e erros de código são bastante comuns, principalmente nas primeiras versões das rotinas. É importante reconhecer este risco e saber administrá-lo.

******ebook converter DEMO Watermarks*******

3.4 Exercícios

1. Imagine uma pesquisa envolvendo a análise do seu orçamento doméstico ao longo do tempo. Dados estão disponíveis em planilhas eletrônicas separadas por mês, durante 10 anos. O objetivo da pesquisa é entender se é possível a compra de um imóvel daqui a 5 anos. A partir disso, detalhe em texto os elementos em cada etapa do estudo, desde importação dos dados até a construção do relatório.

2. Com base no estudo proposto anteriormente, crie uma estrutura de diretórios em seu computador para acomodar a pesquisa. Em um arquivo texto na pasta raiz da pesquisa, assinale arquivos fictícios para cada subdiretório (veja estrutura de diretórios no início da seção 3.2). Note que a criação dos diretórios pode ser realizada pelo próprio R.

******ebook converter DEMO Watermarks*******

4 Importação e Exportação de Dados Locais

Conforme vimos no capítulo 3, a primeira etapa de uma pesquisa é a importação de dados. Nesta seção iremos aprender a importar e exportar dados contidos em arquivos locais. No próximo capítulo aprenderemos a importar dados de forma mais dinâmica, utilizando pacotes especializados e uma conexão a Internet.

Existem diferentes formatos de armazenamento de dados em arquivos. É impossível tratar todos os casos, porém iremos traçar uma lista abrangente com os seguintes formatos e extensões de arquivos:

- Dados delimitados em texto (*csv*);
- Microsoft Excel (*xls*, *xlsx*);
- Arquivos nativos do R (*RData* e *rds*)
- Formato `fst` (*fst*)
- SQLite (*SQLITE*)
- Texto não estruturado (*txt*).

Cada um desses possui sua própria particularidade. Os benefícios entre um formato e outro mudam de acordo com a situação. Na última seção do capítulo iremos discorrer sobre o processo de seleção de formato. Em geral, o formato mais flexível e utilizado é *.csv*.

A primeira lição na importação de dados para o R é que o local do arquivo deve ser indicado explicitamente no código. Este endereço é passado para a função que irá ler o arquivo. Veja a definição a seguir:

```
my.file <- 'C:/Data/MyData.csv'
```

Note o uso de barras (`/`) para designar o diretório do arquivo. Referências relativas também funcionam, tal como em:

```
my.file <- 'Data/MyData.csv'
```

Neste caso, assume-se que na pasta atual de trabalho existe um diretório chamado `Data` e, dentro desse, um arquivo denominado `MyData.csv`. Se o endereço do arquivo é simplesmente o seu nome, assume-se que o mesmo encontra-se na raiz da pasta de trabalho. Para verificar o endereço atual de trabalho, utilize a função `getwd`.

Um ponto bastante importante aqui é que os dados serão importados e exportados no R como objetos do tipo dataframe. Isto é, uma tabela contida em um arquivo Excel ou *.csv* se transformará em um objeto no ambiente de trabalho do R após a importação. Quando exportarmos dados, o formato mais comum é esse mesmo objeto, um dataframe. Convenientemente, dataframes são nada mais que tabelas, com linhas e colunas. Cada coluna do dataframe terá a sua própria classe, sendo as mais comuns:

- Numérica (*numeric*)

******ebook converter DEMO Watermarks*******

- Texto (*character*)
- Fator (*factor*)
- Data (*Date*)

É de fundamental importância que os dados sejam representados na classe correta. Uma vasta quantidade de erros podem ser evitados pela simples checagem das classes das colunas no dataframe resultante do processo de importação. Por enquanto somente é necessário entender esta propriedade básica de dataframes. Estudaremos esse objeto mais profundamente no capítulo 6 e as classes básicas de objetos no capítulo 7. Vamos partir para o estudo das funções e pacotes para a importação e exportação de cada formato de arquivo.

******ebook converter DEMO Watermarks*******

4.1 Arquivos *csv*

Considere o arquivo de dados no formato `csv` chamado `'Ibov.csv'`, localizado na pasta `data` no diretório atual de trabalho. Esse arquivo contém dados diários do índice Ibovespa de 2010-01-04 até 2018-06-08. Observe que se trata de um simples arquivo de texto estruturado em linhas e colunas. O mesmo pode ser aberto e modificado em qualquer editor de texto. A primeira linha apresenta os nomes das colunas -- linha também chamada de cabeçalho (*header*) -- e todas as colunas são separadas por vírgulas (,). Esse é um padrão internacional para esse tipo de arquivo.

No caso brasileiro, esse padrão internacional gera confusão, pois a vírgula também pode indicar valores decimais em dados numéricos. Para evitar problemas na importação dos dados, caso você encontre um arquivo onde os valores decimais são definidos por vírgula, abra-o em um editor de texto qualquer e substitua todos os valores de vírgula por ponto (.). Isso deixa o arquivo em um formato internacional. No Windows, programas como o Notepad e o Notepad++ realizam essa tarefa facilmente.

Destaca-se que, no formato brasileiro, é comum utilizar a semi-vírgula (`;`) para separar colunas, enquanto no formato internacional o mais comum é a vírgula (`,`). Caso não seja possível converter os dados para o formato internacional, pode-se utilizar funções específicas que assumem que o indicador de decimal é a vírgula e o separação de colunas é o ponto-e-vírgula (`;`).

Como sugestão para evitar problemas, antes de prosseguir para a importação de dados em um arquivo *.csv*, deve-se abrir o arquivo em um editor de texto qualquer e verificar:

1. A existência de texto antes dos dados e a necessidade de ignorar algumas linhas iniciais;
2. A existência ou não dos nomes das colunas;
3. O símbolo separador de colunas;
4. O símbolo de decimal, o qual deve ser o ponto (`.`).

Na maioria dos casos, os dados podem ser importados diretamente. Porém, os itens anteriores devem ser checados para evitar problemas na importação.

4.1.1 Importação de Dados

O R possui funções nativas para importar dados de arquivos *.csv*, `read.csv` e `read.csv2`. Porém, esse é um dos casos em que o `tidyverse` oferece melhores opções, com maior velocidade de leitura e melhor tratamento das classes de colunas. Portanto, utilizamos a função `read_csv` do pacote `readr` para ler os dados do arquivo data/Ibov.csv. Veja um exemplo de uso:

```
library(readr)

# set file
my.f <- 'data/Ibov.csv'

# read data
my.df.ibov <- read_csv(my.f)

## Parsed with column specification:
```

******ebook converter DEMO Watermarks*******

```
## cols(
##   ref.date = col_date(format = ""),
##   price.close = col_integer()
## )
```

Observe como o código anterior apresentou a mensagem "Parsed with column specification: ...". Essa mensagem mostra como a função identifica as classes das colunas lendo as primeiras 1000 linhas do arquivo. Regras inteligentes tentam prever a classe com base no conteúdo importado. Podemos usar essas informações em nosso próprio código copiando o texto e atribuindo-o a uma variável:

```
# set cols from readr import message
my.cols <- cols(
  price.close = col_integer(),
  ref.date = col_date(format = "")
)

# read file with readr::read_csv
my.df.ibov <- read_csv(my.f,
                       col_types = my.cols)
```

Agora, vamos checar a classe das nossas colunas. Para isso, utilizamos a função `glimpse` do pacote `dplyr`:

```
# print column classes
dplyr::glimpse(my.df.ibov)
```

```
## Observations: 2,102
## Variables: 2
## $ ref.date    <date> 2010-01-04, 2010-01-05, 2010-01-06...
## $ price.close <int> 70045, 70240, 70729, 70451, 70263, ...
```

Verificando a importação, temos duas colunas. A primeira é `ref.date`, com classe `date`. A segunda é `price.close`, da classe `int` (*integer*) - número inteiro.

No uso de `readr::read_csv`, um possível conjunto de etapas é:

1. ler o arquivo sem argumentos em `read_csv`,
2. copiar o texto das classes de coluna padrão da mensagem de saída
3. adicionar o texto anterior como argumento `col_types`

O conjunto de passos anterior é suficiente para a grande maioria dos casos. O uso da mensagem com as classes das colunas é particularmente útil quando o arquivo importado tem várias colunas e a definição manual de cada classe exige muita digitação.

Indo além, função `read_csv` possui diversas outras opções tais como:

- modificação do formato dos dados de importação (opção `locale`)
- pular *n* linhas antes da importação (opção `skip`)
- definição customizada para valores NA (opção `na`)

entre várias outras possibilidades. O pacote `readr` também possui várias outras funções para

******ebook converter DEMO Watermarks*******

situações específicas de importação. Caso a função `read_csv` não resolva o seu problema na leitura de algum arquivo texto estruturado, certamente outra função desse pacote resolverá.

4.1.2 Exportação de Dados

Para escrever um arquivo *.csv*, basta utilizar a função `readr::write_csv`. No próximo exemplo iremos criar dados artificiais, salvar em um dataframe e exportar para um arquivo *.csv*. Veja a seguir:

```
library(readr)

# set number of observations
N <- 100

# create dataframe with random data
my.df <- data.frame(y = runif(N),
                    z = rep('a',N))

# write to file
f.out <- 'data/temp.csv'
write_csv(x = my.df, path = f.out)
```

No exemplo anterior, salvamos o `dataframe` chamado `my.df` para o arquivo `temp.csv`, localizado na pasta `data` do diretório de trabalho. Podemos verificar o arquivo importando o seu conteúdo:

```
my.df.import <- read_csv(f.out,
                         col_types = cols(y = col_double(),
                                          z = col_character() ) )
print(head(my.df.import))
```

```
## # A tibble: 6 x 2
##       y z
##   <dbl> <chr>
## 1 1.000 a
## 2 0.666 a
## 3 0.760 a
## 4 0.379 a
## 5 0.318 a
## 6 0.329 a
```

O resultado está conforme o esperado, um dataframe com duas colunas, a primeira com números e a segunda com texto.

******ebook converter DEMO Watermarks*******

4.2 Arquivos *Excel* (*xls* e *xlsx*)

Em Finanças e Economia, é bastante comum encontrarmos dados salvos em arquivos do tipo Microsoft Excel, com extensão *xls* ou *xlsx*. Apesar de não ser um formato de armazenamento de dados eficiente, esse é um programa de planilha bastante popular devido às suas funcionalidades. É muito comum que informações sejam armazenadas e distribuídas dessa forma. Por exemplo: dados históricos do Tesouro Direto são disponibilizados como arquivos *.xls* no site do tesouro nacional. A CVM (Comissão de Valores Mobiliários) e ANBIMA (Associação Brasileira das Entidades dos Mercados Financeiro e de Capitais) também tem preferência por esse tipo de formato em alguns dados publicados em seu *site*.

4.2.1 Importação de Dados

O R não possui uma função nativa para importar dados do Excel e, portanto, deve-se instalar e utilizar certos pacotes para realizar essa operação. Existem diversas opções, porém, os principais pacotes são `XLConnect` (Mirai Solutions GmbH 2016), `xlsx` (Dragulescu 2014) e `readxl` (Wickham 2016a).

Apesar de os pacotes anteriores terem objetivos semelhantes, cada um tem suas peculiaridades. Caso a leitura de arquivos do Excel seja algo importante no seu trabalho, aconselho-o fortemente a estudar as diferenças entre esses pacotes. Nesta seção, daremos prioridade para funções do pacote `readxl`, que é um dos mais fáceis e diretos de utilizar, além de não necessitar de outros softwares instalados (tal como o *Java*). Para instalar o referido pacote, basta utilizar a função `install.packages`:

```
install.packages('readxl')
```

Imagine agora a existência de um arquivo chamado `Ibov_xls.xlsx` que contenha os mesmos dados do Ibovespa. A importação das informações contidas nesse arquivo para o R seria realizada através da função `read_excel`:

```
library(readxl)
library(dplyr)

# set file
my.f <- 'data/Ibov_xlsx.xlsx'

# read xlsx into dataframe
my.df <- read_excel(my.f, sheet = 'Sheet1')

# glimpse contents
glimpse(my.df)
```

```
## Observations: 2,102
## Variables: 2
## $ ref.date    <dttm> 2010-01-04, 2010-01-05, 2010-01-06...
## $ price.close <dbl> 70045, 70240, 70729, 70451, 70263, ...
```

Observe que, nesse caso, as datas já foram importadas com a formatação correta na classe `dttm`

******ebook converter DEMO Watermarks*******

(*datetime*). Essa é uma vantagem ao utilizar arquivos do Excel: a classe dos dados do arquivo original é levada em conta no momento da importação. O lado negativo desse formato é a baixa portabilidade dos dados e o maior tempo necessário para a execução da importação. Como regra geral, dados importados do Excel apresentarão um tempo de carregamento mais alto do que dados importados de arquivos *.csv*.

4.2.2 Exportação de Dados

A exportação para arquivo Excel também é fácil. Assim como para a importação, não existe uma função nativa do R que execute esse procedimento. Podemos, porém, utilizar pacotes `xlsx` (Dragulescu 2014) e `writexl` (Ooms 2017) para esse objetivo. Uma diferença aqui é que o pacote `xlsx` oferece mais funcionalidade mas exige a instalação do Java no sistema operacional. No caso do Windows, basta visitar o site do Java e instalar o software na versão 64 bits (opção *Windows Off-line (64 bits)*). Logo após, instale o pacote `xlsx` normalmente no R com o comando `install.packages('xlsx')`.

Vamos começar com um exemplo para `xlsx`

```
library(xlsx)

# set number of rows
N <- 50

# create random dataframe
my.df <- data.frame(y = seq(1,N),
                    z = rep('a',N))

# write to xlsx
f.out <- 'data/temp.xlsx'
write.xlsx(x = my.df,
           file = f.out,
           sheetName = "my df")
```

Note que uma diferença nos argumentos da função `write.xlsx` é que é necessário incluir o nome da aba do arquivo Excel onde os dados da tabela serão exportados. Para exportar várias informações para um mesmo arquivo, é necessário utilizar o argumento `append` da função `write.xlsx`. Caso contrário, a função irá criar um novo arquivo em cada chamada da mesma. Veja o exemplo a seguir, onde exportamos dois dataframes para duas abas diferentes do mesmo arquivo Excel:

```
# set number of rows
N <- 25

# create random dfs
my.df.A <- data.frame(y = seq(1,N),
                      z = rep('a',N))

my.df.B <- data.frame(z = rep('b',N))

# write both df to single file
f.out <- 'data/temp.xlsx'
write.xlsx(x = my.df.A,
           file = f.out,
```

******ebook converter DEMO Watermarks*******

```
            sheetName = "my df A")

write.xlsx(x = my.df.B,
            file = f.out,
            sheetName = "my df B",
            append = TRUE )
```

Após a exportação, podes verificar o conteúdo dos arquivos abrindo-o no Excel.

O diferencial do pacote `writexl` em relação a `xlsx` é a não necessidade do Java, e a rapidez de execução. O lado negativo é que não permite a escolha das abas para a cópia. Veja a seguir:

```
library(writexl)
# set number of rows
N <- 25

# create random dfs
my.df.A <- data.frame(y = seq(1,N),
                      z = rep('a',N))

write_xlsx(x = my.df.A,
            path = f.out)
```

Para comparar o desempenho, vamos verificar a diferença de tempo de execução entre um e outro:

```
library(writexl)
library(readxl)
library(xlsx)

# set number of rows
N <- 2500

# create random dfs
my.df.A <- data.frame(y = seq(1,N),
                      z = rep('a',N))

# set files
my.file.1 <- 'data/temp_writexl.xlsx'
my.file.2 <- 'data/temp_xlsx.xlsx'

# test export
time.write.writexl <- system.time(write_xlsx(x = my.df.A,
                                             path = my.file.1 ))

time.write.xlsx <- system.time(write.xlsx(x = my.df.A,
                                          file = my.file.2 ))

# test read
time.read.readxl <- system.time(read_xlsx(path = my.file.1 ))
time.read.xlsx <- system.time(read.xlsx(file = my.file.2,
                                        sheetIndex = 1 ))
```

Após a execução, vamos verificar a diferença de tempo:

```
# results
my.formats <- c('xlsx', 'readxl')
```

******ebook converter DEMO Watermarks*******

```
results.read <- c(time.read.xlsx[3], time.read.readxl[3])
results.write<- c(time.write.xlsx[3], time.write.writexl[3])

# print text
my.text <- paste0('\nTime to WRITE dataframe with ',
                  my.formats, ': ',
                  results.write, ' seconds', collapse = '')
cat(my.text)

my.text <- paste0('\nTime to READ dataframe with ',
                  my.formats, ': ',
                  results.read, ' seconds', collapse = '')
cat(my.text)

##
## Time to WRITE dataframe with xlsx: 2.235 seconds
## Time to WRITE dataframe with readxl: 0.00999999999999979 seconds
## Time to READ dataframe with xlsx: 3.549 seconds
## Time to READ dataframe with readxl: 0.032 seconds
```

Como podemos ver, mesmo para dados de pouco volume, um dataframe com 2500 linhas e 2 colunas, a diferença de tempo de execução é significativa. Caso estiveres trabalhando com grandes planilhas, o uso de pacotes `readxl` e `writexl` é fortemente recomendado. Porém, como já mostrado anteriormente, as funções de `xlsx` oferecem algumas funcionalidades extras.

******ebook converter DEMO Watermarks*******

4.3 Arquivos *RData/rds*

O R possui dois formatos nativos para salvar objetos de sua área de trabalho para um arquivo local com extensão *RData* ou *rds*. O grande benefício, em ambos os casos, é que o arquivo resultante é compacto e o seu acesso é muito rápido. A desvantagem é que os dados perdem portabilidade para outros programas. A diferença entre um formato e outro é que arquivos *RData* podem salvar mais de um objeto, enquanto o formato *.rds* salva apenas um.

4.3.1 Importação de Dados

Para carregar os dados de um aquivo `RData`, utilizamos a função `load`:

```
# set file path (objects my.x and my.y)
my.file <- 'data/temp.RData'

# load all contents into workspace
load(file = my.file)
```

O arquivo data/temp.RData continha dois objetos, `my.x` e `my.y`, os quais foram carregados na memória do R.

O processo de importação para arquivos *.rds* é muito semelhante. A diferença é no uso da função `readRDS`:

```
# set file path
my.file <- 'data/temp.rds'

# load content into workspace
my.x <- readRDS(file = my.file)
```

Comparando o código entre o uso de arquivos `.RData` e `.rds`, note que um benefício no uso de `.rds` é a explícita definição do objeto na área de trabalho. Isto é, o conteúdo de `my.file` em `readRDS` é explicitamente salvo em `my.x`. Quando usamos a função `load`, no código não fica claro qual o nome do objeto que foi importado. Isso é particularmente inconveniente quando é necessário modificar o nome do objeto importado.

Como sugestão, dê preferência ao uso do formato *.rds*. Entendo que esse é mais prático, deixando o código mais claro. A diferença de velocidade de acesso e gravação entre um e outro é mínima. O benefício de importar vários objetos em um mesmo arquivo com o formato `RData` torna-se irrelevante quando no uso de objetos do tipo lista, os quais podem incorporar outros objetos no seu conteúdo. Veremos mais detalhes sobre este tipo de objeto no capítulo 6

4.3.2 Exportação de Dados

Para criar um novo arquivo *RData*, utilizamos a função `save`. Veja o exemplo a seguir, onde criamos um arquivo *RData* com dois objetos:

```
# set vars
```

******ebook converter DEMO Watermarks*******

```
my.x <- 1:100
my.y <- 1:100

# write to RData
my.file <- 'data/temp.RData'
save(list = c('my.x', 'my.y'),
     file = my.file)
```

Podemos verificar a existência do arquivo com a função `file.exists`:

```
file.exists(my.file)
```

```
## [1] TRUE
```

Observe que o arquivo data/temp.RData está disponível na pasta `data`.

Já para arquivos *.rds*, salvamos o objeto com função `saveRDS`:

```
# set data and file
my.x <- 1:100
my.file <- 'data/temp.rds'

# save as .rds
saveRDS(object = my.x,
        file = my.file)

# read it
my.x2 <- readRDS(file = my.file)

# test equality
print(identical(my.x, my.x2))
```

```
## [1] TRUE
```

O comando `identical` testa a igualdade entre os objetos e mostra que são exatamente iguais.

******ebook converter DEMO Watermarks*******

4.4 Arquivos *fst* (pacote `fst`)

Uma recente inovação no ambiente R diz respeito ao armazenamento de dados locais com o formato *fst*. Esse foi especialmente desenhado para possibilitar a gravação e leitura de dados tabulares de forma rápida e com mínimo uso do espaço no disco. O uso deste formato é particularmente benéfico quando se está trabalhando com volumosas bases de dados.

4.4.1 Importação de Dados

O uso é bastante simples. Utilizamos a função `read_fst` para ler arquivos *fst*.

```
library(fst)

my.file <- 'data/temp.fst'
my.df <- read_fst(my.file)

glimpse(my.df)
```

```
## Observations: 1,000
## Variables: 1
## $ x <dbl> 0.747546662, 0.009745144, 0.385196564, 0.8220...
```

Assim como para os demais casos, os dados estão disponíveis na área de trabalho após a importação.

4.4.2 Exportação de Dados

Utilizamos a função `write_fst` para gravar arquivos no formato *fst*, :

```
library(fst)

# create dataframe
N <- 1000
my.file <- 'data/temp.fst'
my.df <- data.frame(x = runif(N))

# write to fst
write_fst(x = my.df, path = my.file)
```

Como um teste do potencial do formato `fst`, a seguir vamos cronometrar o tempo de leitura e gravação entre `fst` e `rds` para um dataframe com grande quantidade de dados: 2,500,000 linhas e 2 colunas. Iremos reportar também o tamanho do arquivo resultante.

```
library(fst)

# set number of rows
N <- 2500000

# create random dfs
my.df <- data.frame(y = seq(1,N),
                    z = rep('a',N))
```

******ebook converter DEMO Watermarks*******

```
# set files
my.file.1 <- 'data/temp_rds.rds'
my.file.2 <- 'data/temp_fst.fst'

# test write
time.write.rds <- system.time(saveRDS(my.df, my.file.1 ))
time.write.fst <- system.time(write_fst(my.df, my.file.2 ))

# test read
time.read.rds <- system.time(readRDS(my.file.1))
time.read.fst <- system.time(read_fst(my.file.2))

# test file size (MB)
file.size.rds <- file.size(my.file.1)/1000000
file.size.fst <- file.size(my.file.2)/1000000
```

Após a execução, vamos verificar o resultado:

```
# results
my.formats <- c('.rds', '.fst')
results.read <- c(time.read.rds[3], time.read.fst[3])
results.write<- c(time.write.rds[3], time.write.fst[3])
results.file.size <- c(file.size.rds , file.size.fst)

# print text
my.text <- paste0('\nTime to WRITE dataframe with ',
                  my.formats, ': ',
                  results.write, ' seconds', collapse = '')
cat(my.text)

my.text <- paste0('\nTime to READ dataframe with ',
                  my.formats, ': ',
                  results.read, ' seconds', collapse = '')
cat(my.text)

my.text <- paste0('\nResulting FILE SIZE ',
                  my.formats, ': ',
                  results.file.size, ' MBs', collapse = '')
cat(my.text)

##
## Time to WRITE dataframe with .rds: 1.338 seconds
## Time to WRITE dataframe with .fst: 0.00900000000000034 seconds
## Time to READ dataframe with .rds: 0.0739999999999981 seconds
## Time to READ dataframe with .fst: 0.0399999999999991 seconds
## Resulting FILE SIZE .rds: 5.333323 MBs
## Resulting FILE SIZE .fst: 3.568578 MBs
```

A diferença é gritante! O formato `fst` não somente lê e grava com mais rapidez mas o arquivo resultante também é menor. A mensagem aqui é clara, quando estiver trabalhando com dados volumosos, dê preferência ao formato *fst*.

******ebook converter DEMO Watermarks*******

******ebook converter DEMO Watermarks*******

4.5 Arquivos *SQLITE*

O uso de arquivos *csv*, *rds* e *fst* para armazenar conjuntos de dados tem seus limites a medida que o tamanho dos arquivos aumenta e os dados fragmentam-se em várias tabelas. Se você está esperando muito tempo para ler apenas uma tabela de um arquivo com várias tabelas, deves procurar alternativas mais eficientes. Da mesma forma, se você estiver trabalhando em uma rede de computadores e muitas pessoas estão usando os mesmos dados, faz sentido manter e distribuir as informações de um servidor central. Dessa forma, cada usuário pode ter acesso à mesma informação, simultaneamente.

Isso nos leva ao tópico de programas de armazenamento e distribuição de banco de dados. Esses programas específicos geralmente funcionam com uma linguagem de consulta chamada *SQL* (*Structured Query Language*), e permitem ao usuário ler partes dos dados e mesmo manipulá-lo de forma eficiente. Existem muitas opções de software de banco de dados que se integra muito bem com R. A lista inclui **mySQL**, **SQLite** e **MariaDB**. Aqui, forneceremos um tutorial rápido sobre esse tópico usando o SQLite, que é o mais fácil de usar, uma vez que não precisa de nenhuma configuração do servidor e todos dados estão contidos em um único arquivo.

Antes de irmos para os exemplos, precisamos entender como se usa o *software* de banco de dados. Primeiro, um banco de dados deve existir em seu computador ou rede. Segundo, o R se conectará ao banco de dados e retornará um objeto de conexão. Com base nessa conexão, enviaremos consultas para importar dados desse banco de dados usando a linguagem *SQL*. A principal vantagem é que podemos ter um grande banco de dados de, digamos, 10 GB e carregar apenas uma pequena porção dele na área de trabalho do R. Essa operação também é muito rápida, permitindo um acesso eficiente às tabelas disponíveis.

4.5.1 Importação de Dados

Assumindo a existência de um arquivo com formato SQLite, podemos importar suas tabelas com o pacote `RSQLite`:

```
library(RSQLite)

# set name of SQLITE file
f.sqlite <- 'data/MySQLiteDatabase.SQLITE'

# open connection
my.con <- dbConnect(drv = SQLite(), f.sqlite)

# read table
my.df <- dbReadTable(conn = my.con,
                     name = 'MyTable1')

# print with str
glimpse(my.df)
```

```
## Observations: 100,000
## Variables: 2
## $ x <dbl> 0.43625259, 0.69062294, 0.32039424, 0.3785276...
```

******ebook converter DEMO Watermarks*******

```
## $ G <chr> "B", "A", "A", "A", "B", "A", "B", "B", "B", ...
```

Outro exemplo do uso do SQLITE é com instruções de um comando *SQL*. Observe que, no código anterior, usamos função `dbReadTable` para obter o conteúdo de todas as linhas da tabela `MyTable1`. Agora, vamos usar o comando `dbGetQuery` para obter dados da tabela `myTable2` apenas quando a coluna `G` é igual a `A`:

```
# set sql statement
my.SQL <- "select * from myTable2 where G='A'"

# get query
my.df.A <- dbGetQuery(conn = my.con, statement = my.SQL)

# disconnect from db
dbDisconnect(my.con)

# print with str
glimpse(my.df.A)
```

```
## Observations: 50,135
## Variables: 2
## $ x <dbl> 0.2056976, 0.8606551, 0.9890443, 0.4443891, 0...
## $ G <chr> "A", "A", "A", "A", "A", "A", "A", "A", "A", ...
```

Funcionou, conforme esperado.

Nesse exemplo simples podemos ver como é fácil criar uma conexão com um banco de dados, recuperar tabelas e desconectar. Se você estiver trabalhando com dados volumosos, na minha opinião, qualquer banco de dados que ocupe mais de 50% da memória RAM do seu computador, vale a pena utilizar um software de banco de dados apropriado. Caso existir um servidor de banco de dados disponível em seu local de trabalho, eu recomendo fortemente aprender a conectar-se a ele e usar o idioma SQL para sua vantagem. Existem muitas outras maneiras de consultar e manipular dados usando SQL. Vários tutoriais estão disponíveis na internet.

4.5.2 Exportação de Dados

Como exemplo, vamos criar dois dataframes com dados aleatórios e salvar ambos em um arquivo SQLite usando pacote `RSQLite`.

```
library(RSQLite)

# set number of rows in df
N = 10^5

# create simulated dataframe
my.large.df.1 <- data.frame(x=runif(N),
                            G= sample(c('A','B'),
                                      size = N,
                                      replace = TRUE))

my.large.df.2 <- data.frame(x=runif(N),
                            G = sample(c('A','B'),
                                       size = N,
                                       replace = TRUE))
```

******ebook converter DEMO Watermarks*******

```
# set name of SQLITE file
f.sqlite <- 'data/MySQLiteDatabase.SQLITE'

# open connection
my.con <- dbConnect(drv = SQLite(), f.sqlite)

# write df to sqlite
dbWriteTable(conn = my.con, name = 'MyTable1', value = my.large.df.1)
dbWriteTable(conn = my.con, name = 'MyTable2', value = my.large.df.2)

# disconnect
dbDisconnect(my.con)
```

A saída `TRUE` de `dbWriteTable` indica que tudo ocorreu bem. Uma conexão foi aberta usando a função `dbConnect` e os dataframes foram escritos em um arquivo SQLITE chamado data/MySQLiteDatabase.SQLITE. É boa política de programação sempre se desconectar do banco de dados após a utilização. Fizemos isso com a função `dbDisconnect`.

******ebook converter DEMO Watermarks*******

4.6 Dados Não-Estruturados e Outros Formatos

O uso das funções de importação de arquivos nos formatos destacados anteriormente é suficiente na grande maioria das situações. Apesar disso, vale destacar que o R possui outras funções específicas para diferentes formatos. Isso inclui arquivos exportados de outros softwares, tal como SPSS, Matlab, entre vários outros. Se esse for o seu caso, sugiro um estudo aprofundado do pacote `foreign` (R Core Team 2015).

Em alguns casos nos deparamos com dados armazenados de uma forma não estruturada, tal como um texto qualquer. Pode-se importar o conteúdo de um arquivo de texto (linha por linha) através da função `readLines`. Veja a seguir:

```
# set ibov file
my.f <- 'data/Ibov.csv'

# read it
my.txt <- readLines(my.f)

# print contents
print(my.txt[1:5])
```

```
## [1] "\"ref.date\",\"price.close\""
## [2] "2010-01-04,70045"
## [3] "2010-01-05,70240"
## [4] "2010-01-06,70729"
## [5] "2010-01-07,70451"
```

Nesse caso importou-se todo o conteúdo do arquivo `Ibov.csv` como um texto. Observe que cada linha desse arquivo virou um elemento de um vetor atômico da classe de caracteres. Se fosse necessário, é possível escrever uma rotina que processasse esse arquivo linha por linha, separando as colunas e isolando os valores numéricos.

Em algumas situações, é necessário exportar algum tipo de texto para um arquivo. Por exemplo: quando se precisa salvar o registro de um procedimento em um arquivo de texto; ou quando se precisa gravar informações em um formato específico não suportado pelo R. Esse procedimento é bastante simples. Junto à função `cat`, basta indicar um arquivo de texto para a saída com o argumento `file`. Todo texto alimentado na função `cat` será escrito no arquivo de texto. Veja a seguir:

```
# set file
my.f <- 'data/temp.txt'

# set char to file
my.str <- paste('Today is', Sys.Date(), '\n')

# write it
cat(my.str, file = my.f, append = FALSE)
```

Criamos um objeto de texto com uma mensagem sobre a data atual. Executamos a função `cat` com os argumentos `file` e `append`. Esse último sinaliza a opção de adicionar novos textos ao final do arquivo ou não. Podemos checar o resultado com a função `readLines`:

******ebook converter DEMO Watermarks*******

```r
# print contents of my.f
print(readLines(my.f))
```

```
## [1] "Today is 2018-06-10 "
```

******ebook converter DEMO Watermarks*******

4.7 Selecionando o Formato

Após entendermos a forma de salvar e carregar dados de arquivos locais em diferentes formatos, é importante discutirmos sobre a escolha do formato. Caso possível, o usuário deve levar em conta três pontos nessa decisão:

- velocidade de importação e exportação;
- tamanho do arquivo resultante;
- compatibilidade com outros programas e sistemas.

Na grande maioria das situações, o uso de arquivos *csv* satisfaz esses quesitos. Ele nada mais é do que um arquivo de texto que pode ser aberto, visualizado e importado em qualquer programa. Desse modo, fica muito fácil compartilhar dados com outros usuários. Além disso, o tamanho de arquivos *csv* geralmente não é exagerado. Em caso negativo, pode-se compactá-lo utilizando a função *zip*. Por esses motivos, o uso de arquivos *csv* para importações e exportações é preferível na grande maioria das situações.

Em casos específicos, onde estiver trabalhando com dados volumosos, sugiro o formato `fst`. Esse permite um acesso extremamente rápido aos dados. O lado negativo é que os dados perdem a sua portabilidade. No caso do trabalho envolver diversas tabelas ou acesso a pequenas porções dos dados em cada estágio da pesquisa, sugiro o uso de arquivos SQLite.

******ebook converter DEMO Watermarks*******

4.8 Exercícios

1. Crie um dataframe com o código a seguir:

```
library(dplyr)

my.N <- 10000
my.df <- data_frame(x = 1:my.N,
                    y = runif(my.N))
```

Exporte o dataframe resultante para cada um dos cinco formatos destacados a seguir:

- csv
- rds
- xlsx
- fst
- SQLITE

Qual dos formatos ocupou maior espaço na memória do computador? Podes verificar o tamanho dos arquivos no Windows Explorer, por exemplo. No R, função `file.size` realiza o mesmo serviço.

2. Melhore o código anterior com a mensuração do tempo de execução necessário para gravar os dados nos diferentes formatos (veja exemplos no capítulo atual). Qual deles teve a gravação mais rápida?

3. Defina o valor de `my.N` no código anterior para `1000000`. Esta mudança modifica as respostas das duas últimas perguntas?

4. No material do livro existe um arquivo de dados chamado `IbovStocks_long.csv`. Baixe o arquivo da página do livro e utilize função `readr::read_csv` para carregar o seu conteúdo. Utilize função `glimpse` para verificar o conteúdo dos dados importados. Quantas colunas e qual o nome de cada coluna da tabela?

5. Na página da CVM é possível obter informações de todas as empresas atualmente listadas na bolsa em um arquivo disponível no link http://sistemas.cvm.gov.br/cadastro/SPW_CIA_ABERTA.ZIP. Utilizando o R, baixe o arquivo em seu computador, importe os dados diretamente usando `readr::read_csv` (a função importa diretamente de arquivo compactados, sem necessidade de descompactação explícita). Quantas empresas fazem parte da bolsa atualmente (verifique o número de linhas do dataframe com função `nrow`).

6. **DESAFIO** - Na página a seguir:

http://bit.ly/2L04RxX

é possível baixar dados de cotações de fechamento de todos os instrumentos financeiros

******ebook converter DEMO Watermarks*******

negociados na B3. Porém, os dados estão em um formato de coluna com tamanho fixo. Após ler o manual de importação disponível no endereço anterior, utilize as funções do `readr` para importar as informações do ano 2018 na sua sessão do R.

******ebook converter DEMO Watermarks*******

5 Importação de Dados via Pacotes

Uma das grandes vantagens de se utilizar o R para realizar pesquisas é a quantidade de dados que podem ser importados através da internet. Isso é especialmente prático pois pode-se atualizar uma base de dados através de um simples comando, evitando todo o trabalho de coleta manual dos dados. É nesse ponto que o conhecimento e uso de pacotes do R torna-se essencial. Neste capítulo iremos tratar dos seguintes pacotes especializados para dados financeiros e econômicos:

- BatchGetSymbols
- Quandl
- BETS
- rbcb
- GetDFPData
- GetHFData
- GetTDData

Todos esses pacotes são gratuitos e disponíveis no CRAN. Os mesmos foram selecionados por sua facilidade de uso e abrangência.

******ebook converter DEMO Watermarks*******

5.1 Conhecendo os Dados

Antes de aprendermos a utilizar o R para importar dados da Internet, é importante discutirmos as origens destes dados e suas particularidades. Esses são conjuntos de dados que você provavelmente encontrará no seu trabalho. Esta primeira seção irá fornecer contexto e ajudá-lo a entender de onde os dados estão vindo e o que eles representam. Iniciaremos analisando o conjunto de dados do mercados financeiro.

5.1.1 Contratos Financeiros

Sem dúvida, o conjunto de dados mais popular em finanças são os preços de contratos financeiros. No mercado de capitais, investidores podem comprar contratos que lhes dão direito a um fluxo de caixa futuro. Por exemplo, o governo Brasileiro financia suas atividades através da emissão de contratos de dívida. Quando você compra uma dívida do governo, o que você adquire é um acordo formal onde o governo promete pagar-lhe de volta o empréstimo em datas especificadas e dentro de um período de tempo. Do ponto de vista do investidor, dinheiro é gasto hoje para adquirir um dinheiro no futuro.

O que difere um contrato financeiro do outro é a incerteza subjacente, a estrutura de como os pagamentos são definidos e os direitos adquiridos pelo comprador. Quando uma empresa privada precisa de dinheiro, pode levantar capital externo através da venda de propriedade acionária. Vários direitos, incluindo uma parcela de lucros futuros, são concedidos a quem compra suas novas ações. Uma vez que é difícil prever os lucros futuros, o contrato de ação é geralmente mais arriscado do que um contrato de dívida.

A emissão de ações é um processo elaborado. Quando uma empresa abre seu capital, o primeiro passo é a venda no mercado primário, composto por investidores selecionados. É nessa etapa que ocorre a oferta pública inicial das ações (IPO - *Initial Public Offering*). A empresa geralmente mantém parte das novas ações em sua propriedade (tesouraria), para que possa vender mais tarde ou oferecer como remuneração variável para seus administradores. Na segunda etapa do IPO, o mercado secundário, as ações são negociadas no mercado comum, onde todos têm acesso. É do mercado secundário que a maior parte dos dados financeiros dos mercados financeiros é proveniente. Investidores ganham dinheiro vendendo ações de volta ao mercado com um preço mais alto, ou recebendo proventos (parcelas do lucro) das empresas.

Os preços nos mercados financeiros se movem de acordo com oferta e demanda. Por exemplo, se muitos investidores comprarem ações, seu preço provavelmente aumentará. Toda vez que um vendedor encontra um comprador, uma operação é registrada e o dinheiro troca de mãos. Novas informações e expectativas do mercado sobre a lucratividade futura da empresa desempenham um papel proeminente na dinâmica da demanda. Se o mercado entender que uma empresa irá ter muito lucro nos próximos anos, os investidores provavelmente comprarão a ação e seu preço aumentará. Os preços dos contratos financeiros são, portanto, uma indicação de como o mercado percebe o futuro.

Certos eventos também podem alterar o preço de uma ação, mesmo que nenhuma nova

******ebook converter DEMO Watermarks*******

informação ou mudança na demanda seja observada. Esse é o caso típico de uma operação *split*, em que uma única ação é dividida em várias. Isso geralmente ocorre quando o preço de uma ação está muito alto e, para fomentar liquidez, diminui-se o preço unitário de uma forma arbitrária, porém aumentando o número de ações. Para uma divisão de um para dois, um investidor com 100 ações cotadas a 10 reais agora terá 200 ações a 5 reais cada. O valor do portfólio permanece o mesmo. Ao olhar para a série de preços não-ajustados, uma operação *split* resulta em uma grande queda de valor entre um dia e outro. Um erro bastante comum entre investidores iniciantes é se deparar com um eventual *split* e, erroneamente, se desesperar ao ver que o preço de suas ações cairam mais de 50% entre um dia e outro.

Outro evento que afeta o preço é o pagamento de dividendos. Na data anterior ao pagamento, a ação baixará seu preço em relação ao valor do dividendo por ação. Isso ocorre porque um novo comprador perde o direito ao próximo pagamento de dividendos e, portanto, um desconto é esperado. É importante compreender e reconhecer esses eventos, pois você pode ter que lidar com dados de preços brutos. Em geral, os provedores de dados oferecem a opção de preços ajustados, ou seja, o preço já está ajustado a esses eventos. Os preços ajustados oferecem uma visão mais realista sobre o lucro efetivo que um investidor obteve ao comprar as ações no passado.

Dados de preços de contratos financeiros são registrados sempre que há uma operação efetivada, isto é, uma compra por parte de um investidor e venda por parte de outro. Sites como *Yahoo Finance* e *Google Finance* oferecem estatísticas diárias sobre os preços das ações: preço mais alto/baixo, volume negociado e preço de fechamento do dia. A variedade de conjuntos de dados é ampla, cobrindo quase todos contratos do mercado nacional e internacional. Qualquer pessoa pode baixar esses dados usando o site ou então via código, o que é mais comum. Logo veremos que existem pacotes no CRAN que facilitam o acesso ao conjunto de dados desses sites.

Dados do *Yahoo Finance* estão na frequência diária, isto é, agregam estatísticas de preço para um dia. A obtenção de informações sobre operações de compra e venda dentro do dia é muito mais difícil.[1] Isso geralmente é chamado de *tick-by-tick*, ou dados de alta freqüência. Para mercados líquidos, o tamanho desse banco de dados pode ser restritivo. Felizmente, a bolsa Brasileira oferece acesso gratuito aos dados tick-by-tick de ações e outros contratos derivativos. Logo iremos discutir um pacote para baixar este dados de forma fácil, `GetHFData`.

Do ponto de vista da análise de dados, os conjuntos de dados de preços geralmente têm três dimensões, um identificador da ação (*ticker*), uma referência de data/hora e um preço. Isso significa que, no formato longo, você pode representar esses dados no R como um dataframe de três colunas. Quando novas ações são adicionadas ao conjunto de dados, este dataframe aumentará em linhas. Quando novas variáveis, como o volume negociado, são inseridas, uma nova coluna pode ser adicionada. Como veremos no capítulo 6, o formato longo facilita tarefas de processamento de dados e é recomendado.

Embora a discussão anterior foi majoritariamente sobre o mercado de ações, outros tipos de mercados possuem conjuntos de dados semelhantes, mas com informações adicionais. Por exemplo, contratos financeiros de dívida e derivativos geralmente têm uma data de validade (prazo de vencimento). Contratos de opção têm um preço de exercício. Informações adicionais sobre contratos de dívida e derivativos podem ser facilmente adicionadas a um dataframe com a

******ebook converter DEMO Watermarks*******

adição de colunas.

5.1.2 Relatórios Financeiros de Empresas

Outro conjunto popular de dados em finanças relaciona-se com demonstrações financeiras de empresas. Empresas negociadas na bolsa são legalmente obrigadas a publicar o seu resultado financeiro trimestralmente na forma de relatórios tal como balanço patrimonial, demonstrativo de resultado entre outros. A grosso modo, estes representam bens e dívidas atuais da empresa e o seu desempenho financeiro no último período (Assaf Neto 2010, Matarazzo and Pestana (2010)).

Esse tipo de divulgação é importante para manter os mercados conscientes da atual saúde financeira da empresa. As informações devem, então, serem utilizadas na decisão de compra e venda das ações. Além disso, qualquer outro evento financeiro ou informação relevante também deve ser anunciado. Isso inclui novo levantamento de capital, estrutura de propriedade, composição de comitês, remuneração do diretores, entre diversas outras informações.

Para o caso Brasileiro, todos dados relativos a empresas estão disponíveis no site da B3 e CVM. Separa-se os mesmos entre quatro distintos sistemas:

- ITR: Informações trimestrais
- DFP: Demonstrativos financeiros padronizados
- FRE: Formulário de referência
- FCA: Formulário de cadastro

Os sistemas ITR e DFP contém todos os demostrativos e relatórios financeiros das empresas na frequência trimestral e anual, respectivamente. O FRE possui dados de eventos financeiros mais diversos. Essa é a base de dados mais volumosa e heterogênea. O FCA inclui informações básicas de cadastro para cada empresa. Logo veremos que todos esses dados podem ser importados utilizando o pacote `GetDFPData`.

Dada a quantidade de relatórios financeiros, é impossível explicar cada tipo neste livro. Maiores detalhes sobre informes financeiros dos sistemas DFP e ITR são encontrados na literatura de contabilidade. Nesse caso, indico a obra clássica de Matarazzo and Pestana (2010) e as normas brasileiras de contabilidade.

Vale, porém, destacar como poderíamos organizar dados de demostrativos financeiros no R. Geralmente lida-se com informações para várias empresas e vários períodos de tempo. A melhor forma de representar os dados é através de um dataframe no formato longo com cinco colunas:

- RefDate - data ou período de referência
- CompanyID - identificador exclusivo para empresas
- TypeOfFinStatement - tipo de demonstração financeira (balanços patrimoniais, fluxos de caixa, entre outros)
- idAccount - Número identificador da conta, oriundo das normas contábeis
- nameAccount - nome (texto) da conta: total de ativos, ativo circulante, lucro líquido, entre outros
- Value - o valor da conta

******ebook converter DEMO Watermarks*******

O formato anterior é flexível e pode conter um grande número de demonstrações financeiras diferentes. Novas informações são facilmente adicionadas a este tipo de formato.

5.1.3 Dados Econômicos

Dados econômicos são bastante variados e geralmente publicados por órgãos locais ou internacionais de pesquisa tal como IBGE, IPEA, Banco Mundial, Bancos centrais, ou associações de classe. Tais informações são publicadas e geradas para fins de pesquisa e transparência da situação econômica de um país ou região. A diversidade de informações é grande. Seria impossível discorrer sobre todas possibilidades. Alguns exemplos: poder de compra médio de uma população, distribuição de faixas de renda, inflação anual média de um país, índices de alfabetização, índices de pobreza, entre vários outros.

Aqueles curiosos em descobrir quais informações econômicas estão disponíveis para o Brasil, sugiro ir ao site de estatísticas econômicas do IBGE ou acessar o sistema de séries temporais do Banco Central do Brasil. Logo veremos que pacotes especializados, tal como `Quandl`, `BETS` e `rbcb`, permitem o acesso direto a esses bancos de dados.

No lado do R, os dados econômicos geralmente possuem duas dimensões, um índice de tempo tal como dia ou ano, e um valor. Portanto, a estrutura tabular de um dataframe facilmente acomoda os mais diversos indicadores econômicos. Quando se está trabalhando com diversas séries econômicas, algum tipo de agregação é necessária. No decorrer dos exemplos iremos mostrar como reformatar os dados para a estrutura desejada.

******ebook converter DEMO Watermarks*******

5.2 Pacotes do CRAN

Agora vamos apresentar os principais pacotes do CRAN para o acesso de bases de dados financeiras e econômicas no Brasil. Antes de prosseguir, vamos instalar todos eles de uma vez só. Deves rodar o seguinte código no *prompt* do R:

```
my.pkgs <- c('BatchGetSymbols', 'Quandl', 'BETS', 'rbcb',
             'GetDFPData', 'GetHFData', 'GetTDData', 'dplyr')

install.packages(my.pkgs)
```

Além dos pacotes individuais, também iremos utilizar a função `glimpse` do pacote `dplyr` para checar o conteúdo dos dataframes. Após essa operação, todos os pré-requisitos para rodar os códigos apresentados a seguir foram satisfeitos.

5.2.1 `BatchGetSymbols`

Esse pacote faz a comunicação do R com os dados financeiros disponíveis no *Yahoo Finance* e *Google Finance*. Essa gigantesca base de dados inclui valores agregados de preços e volumes negociados de ações na B3 e outras bolsas internacionais na frequência diária. Tudo que se precisa saber para acessar a base de dados são os identificadores das ações (*tickers*) e um período de tempo.

Antes de começarmos, é importante salientar que, atualmente, **os preços do Yahoo Finance não são ajustados a dividendos**. O ajuste realizado pelo sistema é apenas para desdobramentos das ações. Isso significa que, ao olhar séries de preços longas, existe um viés de retorno para baixo. Os preços ajustados não são realistas pois ignoram o pagamento de dividendos ao acionista. Caso estiver usando uma série longa de preços, deves procurar alternativas.

Voltando ao código, pacote `BatchGetSymbols` (Perlin 2016b) é construído com base nas funções do pacote `quantmod` (Ryan 2015). Os diferenciais do `BatchGetSymbols` são:

Limpeza e organização: todos os dados financeiros de diferentes *tickers* são mantidos no mesmo `dataframe`. O cálculo de retorno diário das ações é automaticamente retornado e não precisa ser realizado pelo usuário;

Controle de erros de importação: caso ocorra um erro no *download* dos dados, esse será registrado na saída;

Comparação de datas a um *benchmark*: os dados de ativos individuais são comparados com dados disponíveis para um ativo *benchmark*, geralmente um índice de mercado. Caso o número de datas faltantes seja maior que um determinado limite imposto pelo usuário, a ação é retirada do dataframe final.

Uso de sistema de cache: no momento de acesso aos dados, os mesmos são salvos localmente no computador do usuário, em uma pasta selecionada. Caso o usuário requisitar os mesmos dados, o sistema de cache será utilizado. Se os dados desejados não estão disponíveis no cache, a

******ebook converter DEMO Watermarks*******

função irá comparar e baixar apenas as informações que faltam. Isso aumenta significativamente a velocidade de acesso aos dados, ao mesmo tempo em que minimiza o uso da conexão a Internet;

Acesso a *tickers* em índices de mercado: O pacote inclui funções para baixar a composição do Ibovespa e SP500. Isso facilita a importação de dados para uma grande quantidade de ações. Podes, por exemplo, baixar cotações de todas as ações que fazem parte de certo índice.

Flexibilidade de formato: O pacote também oferece funções para modificar o formato dos dados. Caso o usuário deseje dataframes no formato largo, onde datas são colunas, existe uma função para transformar os dados.

Veja o exemplo a seguir, onde baixamos dados financeiros referentes a quatro ações nos 30 dias anteriores usando a função de mesmo nome do pacote. Os *tickers* de cada ação podem ser encontrados nos próprios sites do *Yahoo Finance*. Note que adicionamos texto `.SA` a cada um deles. Essa é uma notação específica do site e vale para qualquer ação Brasileira.

Na chamada a função `BatchGetSymbols`, utilizamos um valor de `0.95` para o *input* `thresh.bad.data` e `'^BVSP'` para `bench.ticker`. Isso faz com que a função compare as datas obtidas para cada ativo em relação ao nosso *benchmark*, o índice Ibovespa, cujo *ticker* no *Yahoo Finance* é `^BVSP`. A função `Sys.Date()` retorna a data atual do sistema.

```
library(BatchGetSymbols)
library(dplyr)

# set tickers
my.tickers <- c('PETR4.SA', 'CIEL3.SA',
                'GGBR4.SA', 'GOAU4.SA')

# set dates and other inputs
first.date <- Sys.Date()-30
last.date <- Sys.Date()
thresh.bad.data <- 0.95   # sets percent threshold for bad data
bench.ticker <- '^BVSP'   # set benchmark as ibovespa
cache.folder <- 'data/BGS_Cache' # set folder for cache

l.out <- BatchGetSymbols(tickers = my.tickers,
                         first.date = first.date,
                         last.date = last.date,
                         bench.ticker = bench.ticker,
                         thresh.bad.data = thresh.bad.data,
                         cache.folder = cache.folder)

##
## Running BatchGetSymbols for:
##    tickers = PETR4.SA, CIEL3.SA, GGBR4.SA, GOAU4.SA
##    Downloading data for benchmark ticker | Found cache file
## PETR4.SA | yahoo (1|4) | Found cache file - Youre doing good!
## CIEL3.SA | yahoo (2|4) | Found cache file - Youre doing good!
## GGBR4.SA | yahoo (3|4) | Found cache file - Looking good!
## GOAU4.SA | yahoo (4|4) | Found cache file - You got it!
```

A saída de `BatchGetSymbols` é um objeto do tipo lista, o qual ainda não foi explicado. Por enquanto, tudo que precisas saber é que uma lista é um objeto flexível que acomoda outros

******ebook converter DEMO Watermarks*******

objetos, tal como dataframes. O acesso a cada elemento de uma lista pode ser feito pelo operador `$`. No capítulo 6 iremos estudar melhor esta classe de objetos.

Voltando ao nosso exemplo, função `BatchGetSymbols` retorna uma lista com dois elementos: um dataframe com o resultado do processo de importação e outro dataframe com os dados das ações. Vamos checar o conteúdo do primeiro dataframe.

```
# print result of download process
print(l.out$df.control)
```

```
##     ticker   src download.status total.obs
## 1 PETR4.SA yahoo              OK        21
## 2 CIEL3.SA yahoo              OK        21
## 3 GGBR4.SA yahoo              OK        21
## 4 GOAU4.SA yahoo              OK        21
##   perc.benchmark.dates threshold.decision
## 1                    1               KEEP
## 2                    1               KEEP
## 3                    1               KEEP
## 4                    1               KEEP
```

Objeto `df.control` mostra que todos *tickers* foram válidos, com um total de 21 observações para cada ativo. Note que as datas batem 100% com o Ibovespa.

Quanto aos dados financeiros, esses estão contidos no elemento `df.tickers` de `l.out`:

```
# print df.tickers
glimpse(l.out$df.tickers)
```

```
## Observations: 84
## Variables: 10
## $ price.open          <dbl> 25.83, 25.85, 26.35, 26.87,...
## $ price.high          <dbl> 26.31, 26.48, 26.90, 27.53,...
## $ price.low           <dbl> 25.42, 25.81, 25.89, 26.84,...
## $ price.close         <dbl> 25.44, 26.24, 26.79, 27.39,...
## $ volume              <dbl> 72996400, 66938700, 7523020...
## $ price.adjusted      <dbl> 25.44000, 26.24000, 26.7365...
## $ ref.date            <date> 2018-05-11, 2018-05-14, 20...
## $ ticker              <chr> "PETR4.SA", "PETR4.SA", "PE...
## $ ret.adjusted.prices <dbl> NA, 0.03144650, 0.01892256,...
## $ ret.closing.prices  <dbl> NA, 0.03144650, 0.02096040,...
```

Como esperado, a informação sobre preços, retornos e volumes está lá, com as devidas classes de *dbl* (*double* - numérico) e *date*. Observe que uma coluna chamada `ticker` está incluída. Essa indica em que linhas da tabela os dados de uma ação começam e terminam. Mais tarde, no capítulo 9, usaremos essa coluna para fazer diversos cálculos para cada ação.

Outra função útil do pacote é `BatchGetSymbols::GetIbovStocks`, a qual importa a composição atual do índice Ibovespa diretamente do site da B3. Esse índice é um termômetro do mercado local e as ações que o compõem são selecionadas devido sua alta negociabilidade. Portanto, o conjunto de ações que o compõem é uma ótima amostra de *tickers* para pesquisa científica. Sequenciando o uso de `GetIbovStocks` e `BatchGetSymbols`, podemos facilmente baixar uma volumosa quantidade de dados de ações para o mercado Brasileiro. Considere o seguinte fragmento de código, onde realizamos essa operação:

******ebook converter DEMO Watermarks*******

```
library(BatchGetSymbols)

# set tickers
df.ibov <- GetIbovStocks()
my.tickers <- paste0(df.ibov$tickers,'.SA')


# set dates and other inputs
first.date <- Sys.Date()-30
last.date <- Sys.Date()
thresh.bad.data <- 0.95   # sets percent threshold for bad data
bench.ticker <- '^BVSP'   # set benchmark as ibovespa
cache.folder <- 'data/BGS_Cache' # set folder for cache


l.out <- BatchGetSymbols(tickers = my.tickers,
                         first.date = first.date,
                         last.date = last.date,
                         bench.ticker = bench.ticker,
                         thresh.bad.data = thresh.bad.data,
                         cache.folder = cache.folder)
```

Note que utilizamos a função `paste0` para adicionar o texto `'.SA'` para cada *ticker* em `df.ibov$tickers`. A saída do código anterior não foi mostrada para não encher páginas e páginas com as mensagens do processamento. Destaco que, caso necessário, poderíamos facilmente exportar os dados em `l.out` para um arquivo `.rds` e futuramente carregá-los localmente para realizar algum tipo de análise.

5.2.2 `Quandl`

Outra fonte importante de dados financeiros e econômicos é a plataforma *Quandl*, um sistema estabelecido e abrangente que fornece acesso a uma série de dados gratuitos e pagos. Toda a infraestrutura é patrocinada pela empresa de mesmo nome. O sistema fornece informações de várias economias e mercados financeiros. Diversos bancos centrais e instituições de pesquisa, incluindo o Banco Central do Brasil (BCB) e o IBGE, fornecem informações econômicas e financeiras nessa plataforma. Por exemplo, a vasta base de dados do BCB está disponível neste link.

O primeiro passo no uso do `Quandl` é registrar um novo usuário no site. Logo após, vá para *account settings* e clique em *API KEY*. Esta página deve mostrar um código tal como "Asv8Ac7zuZzJSCGxynfG". Copie este texto para a área de transferência (*control + c*) e, no R, defina um objeto de caractere que contenha o conteúdo copiado da seguinte maneira:

```
# set api key to quandl
my.api.key <- 'Asv8Ac7zuZzJSCGxynfG'
```

Essa chave de API é exclusiva para cada usuário. A chave apresentada aqui não funcionará no seu computador. Você precisa obter a sua própria chave para executar os exemplos a seguir. Cada tabela de dados do `Quandl` possui um símbolo. Por exemplo, `'FED/RXI_N_B_BZ'` representa as séries temporais de preços do dólar em reais, as quais estão contidas na base de dados do FED (*US Federal Reserve*). Uma maneira de procurar os símbolos é ir até o site do Quandl e utilizar o sistema de busca. Após encontrar a série de interesse, o seu respectivo símbolo deve estar

******ebook converter DEMO Watermarks*******

disponível no lado direito superior da tela.

Outra forma mais prática, e recomendada, é usar função `Quandl.search` para pesquisar um texto diretamente da linha de comando do R. Vamos dar um exemplo para dados de inflação no Brasil, o índice INPC, mantendo apenas os cinco primeiros resultados:

```
library(Quandl)
library(dplyr)

# set api key
Quandl.api_key(my.api.key)

# search string in quandl
df.search <- Quandl.search('INPC',
                           silent = TRUE,
                           per_page = 5)
```

Função `Quandl.search` retornou um dataframe com 5 linhas e 13 colunas. Vamos averiguar o seu conteúdo.

```
# check content
glimpse(df.search)
```

```
## Observations: 5
## Variables: 13
## $ id                     <int> 23997, 37897382, 37778793...
## $ dataset_code           <chr> "188", "INPC", "INPC", "I...
## $ database_code          <chr> "BCB", "XNAS", "OTCB", "X...
## $ name                   <chr> "National Consumer Price ...
## $ description            <chr> "National Consumer Price ...
## $ refreshed_at           <chr> "2018-06-10T00:05:11.655Z...
## $ newest_available_date <chr> "2018-05-31", "2007-11-30...
## $ oldest_available_date <chr> "1979-04-30", "2006-12-28...
## $ column_names           <list> [<"Date", "Value">, <"Da...
## $ frequency              <chr> "monthly", "daily", "dail...
## $ type                   <chr> "Time Series", "Time Seri...
## $ premium                <lgl> FALSE, TRUE, TRUE, TRUE, ...
## $ database_id            <int> 35, 14999, 15099, 14999, ...
```

O dataframe inclui diversas informações a respeito da pesquisa. O próximo passo é analisar a saída e decidir qual que melhor satisfaz a necessidade da nossa análise. Para fins de exemplo, vamos selecionar o terceiro resultado, a inflação anual do Brasil publicada pelo BCB (Banco Central Brasileiro). O símbolo final é dado pela junção da coluna `database_code` com `dataset_code`, resultando em `BCB/433` para a nossa escolha.

Com a informação do API, símbolo da série e intervalo de datas desejado, usamos a função `Quandl` para baixar os dados:

```
library(Quandl)
library(dplyr)

# register api key
Quandl.api_key(my.api.key)

# set symbol and dates
```

******ebook converter DEMO Watermarks*******

```
my.symbol <- 'BCB/433'
first.date <- as.Date('2000-01-01')
last.date <- Sys.Date()

# get data!
my.df <- Quandl(code = my.symbol,
                type='raw',
                start_date = first.date,
                end_date = last.date)

# check content
glimpse(my.df)
```

```
## Observations: 221
## Variables: 2
## $ Date  <date> 2018-05-31, 2018-04-30, 2018-03-31, 2018...
## $ Value <dbl> 0.40, 0.22, 0.09, 0.32, 0.29, 0.44, 0.28,...
```

Um gráfico mostra a série de inflação anual ao longo dos anos:

Observe como usamos `type = 'raw'` nas opções da função `Quandl`. Essa entrada força a função a gerar um dataframe nativo. Outras opções incluem a saída de objetos do tipo `ts`, `zoo`, `xts` e `timeSeries`. Se o uso de `Quandl` se tornar parte de sua rotina de análise de dados, recomendo ler o manual de ajuda do pacote. Vários parâmetros podem ser passados para a função `Quandl`, incluindo a escolha para aplicar transformações no conjunto de dados resultante. Você pode encontrar mais detalhes sobre o uso do `Quandl` no R a partir do seu site de ajuda.

Outro aspecto interessante é a possibilidade de baixar dados de mais de uma série no mesmo código. Como exemplo, vamos acessar a base de dados do BCB no Quandl para baixar dados da SELIC, a taxa de juros básica da economia do Brasil e do CDI, a taxa de empréstimos interbancários. Após consultar o Quandl, vemos que os seus respectivos símbolos são `BCB/4389` e `BCB/1178`

```
library(Quandl)
library(dplyr)

# register api key
Quandl.api_key(my.api.key)

# set symbol and dates
my.symbol <- c('BCB/4389', # SELIC by year
               'BCB/1178') # CDI by year

first.date <- as.Date('2005-01-01')
last.date <- Sys.Date()

# get data!
my.df <- Quandl(code = my.symbol,
                type='raw',
                start_date = first.date,
                end_date = last.date)

# check content
glimpse(my.df)
```

```
## Observations: 3,372
```

******ebook converter DEMO Watermarks*******

```
## Variables: 3
## $ Date               <date> 2005-01-03, 2005-01-04, 200...
## $ `BCB.4389 - Value` <dbl> 17.76, 17.72, 17.72, 17.72, ...
## $ `BCB.1178 - Value` <dbl> 17.74, 17.74, 17.74, 17.74, ...
```

Note que o dataframe resultante apresenta dois problemas: o formato é esparso (*wide*), com colunas representando observações e os nomes das colunas são pouco intuitivos. O dataframe deve ser reformatado para utilizarmos as funções de análise do `tidyverse`. A renomeação dos tipos de dados facilitará a análise futura. Apesar de ainda não termos visto como realizar estas mudanças com funções específicas do `tidyverse`, iremos apresentar aqui a solução:

```
# change from wide to long
my.df.long <- tidyr::gather(my.df, type, value, -Date)

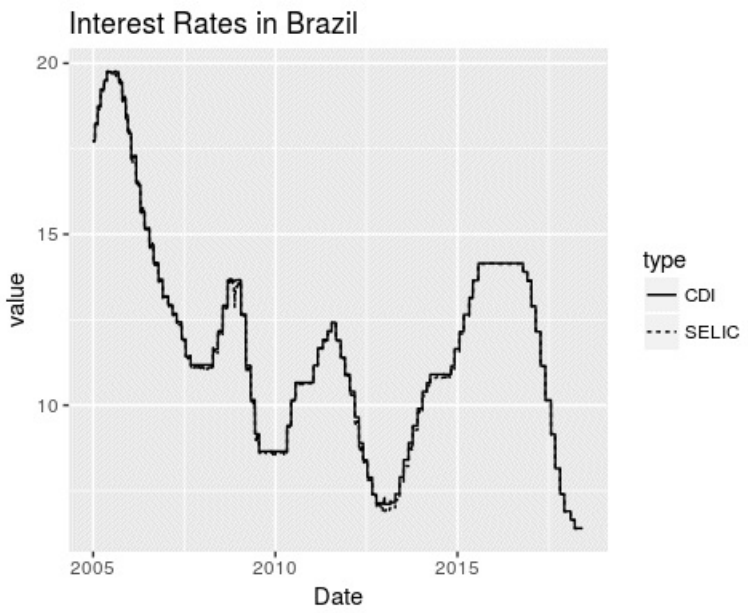
# rename values
my.df.long$type <- forcats::fct_recode(factor(my.df.long$type),
                                       'SELIC'= 'BCB.4389 - Value',
                                       'CDI' = 'BCB.1178 - Value')

# check content
glimpse(my.df.long)
```

```
## Observations: 6,744
## Variables: 3
## $ Date  <date> 2005-01-03, 2005-01-04, 2005-01-05, 2005...
## $ type  <fct> SELIC, SELIC, SELIC, SELIC, SELIC, SELIC,...
## $ value <dbl> 17.76, 17.72, 17.72, 17.72, 17.71, 17.71,...
```

Agora temos um dataframe aceitável, com o formato longo e informações intuitivas. Esse está pronto para ser analisado! Não se preocupe em entender o código anterior. Iremos tratar a operação de mudança de formato do dataframe no capítulo 9 e a recodificação de fatores no capítulo 7.

Para visualizar os dados importados, vamos mostrar uma figura dos seus valores:



******ebook converter DEMO Watermarks*******

Não é acidental valores da taxa CDI e SELIC serem tão próximos, com um leve viés positivo para o CDI. A SELIC é a taxa que bancos privados se financiam junto ao Banco Central. Ao emprestar dinheiro para outros bancos, os mesmos cobram um juros um pouco maior. Vale destacar que o ano de 2018 começou com a mais baixa taxa SELIC dos últimos tempos.

5.2.3 BETS

O pacote `BETS` permite acesso ao repositório de dados da Fundação Getúlio Vargas (FGV), o qual inclui bases de dados do BCB, IBGE e o Instituto Brasileiro de Economia da FGV (FGV/IBRE). Segundo sua própria descrição, o `BETS` permite o acesso a mais de 18,000 séries econômicas do Brasil.

O funcionamento do pacote `BETS` é semelhante ao `Quandl`. Seu diferencial é que não necessita de chave de API (ou registro em site) para o acesso e as tabelas disponíveis para o Brasil são mais variadas. Vamos dar um simples exemplo de uso do `BETS` para baixar dados de desemprego no Brasil. O primeiro passo é procurar o número de identificação da série com função `BETSsearch`:

```
library(BETS)
library(dplyr)

# search for id
df.res <- BETSsearch('unemployment', view = FALSE)

# chech res
glimpse(df.res)
```

```
## Observations: 30
## Variables: 7
## $ code        <chr> "10777", "10778", "10779", "10780",...
## $ description <chr> "Open unemployment rate - by metrop...
## $ unit        <chr> "%", "%", "%", "%", "%", "%", "%", ...
## $ periodicity <chr> "M", "M", "M", "M", "M", "M", "M", ...
## $ start       <chr> "31/10/2001", "31/10/2001", "31/10/...
## $ last_value  <chr> "feb/2016", "feb/2016", "feb/2016",...
## $ source      <chr> "IBGE", "IBGE", "IBGE", "IBGE", "IB...
```

Com o argumento `view = FALSE`, a função retorna um dataframe com os resultados. Caso contrário, uma nova janela aparece para o usuário. Encontramos 30 resultados para o nosso critério de busca. Temos várias séries sobre desemprego disponíveis, algumas até para diferentes países. Vamos acessar a série divulgada pelo PNAD/IBGE, cujo número de identificação é 24369.

```
# set id
my.id <- 24369

# set dates
first.date = '2010-01-01'
last.date  = as.character(Sys.Date())

# get data
df.bets <- BETSget(code = my.id,
                   data.frame = TRUE,
                   from = first.date,
```

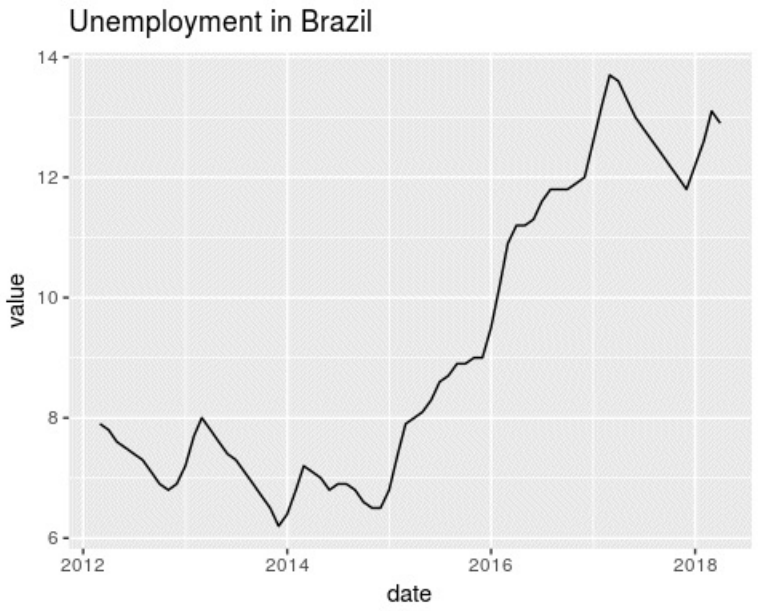
******ebook converter DEMO Watermarks*******

```
                  to = last.date)

# check data
glimpse(df.bets)

## Observations: 74
## Variables: 2
## $ date  <date> 2012-03-01, 2012-04-01, 2012-05-01, 2012...
## $ value <dbl> 7.9, 7.8, 7.6, 7.5, 7.4, 7.3, 7.1, 6.9, 6...
```

A saída de `BETSget` é um dataframe com duas colunas. Vamos verificar o comportamento do índice de desemprego importado com uma figura:



Como podemos ver, o desemprego atual está em um nível alto em relação aos demais anos, atigindo um pico no início de 2017.

Assim como o `Quandl`, o `BETS` também possibilita o acesso a mais de uma série. O objeto de saída, porém, é diferente. A função `BETSget` não agrega os dados e retorna uma lista onde cada elemento é uma série. No exemplo a seguir iremos baixar dados de desemprego de diferentes países e compará-los em uma figura. Os *ids* de cada país estão disponíveis no dataframe de busca de séries.

```
my.id <- 3785:3791

# set dates
first.date = '2010-01-01'
last.date  = as.character(Sys.Date())

# get data
l.out <- BETSget(code = my.id,
                 data.frame = TRUE,
                 from = first.date,
                 to = last.date)
```

******ebook converter DEMO Watermarks*******

```
# check data
dplyr::glimpse(l.out)
```

```
## List of 7
##  $ ts_3785:'data.frame': 79 obs. of  2 variables:
##   ..$ date : Date[1:79], format: "2010-01-01" ...
##   ..$ value: num [1:79] 7.4 7.4 7.3 7.2 7 6.9 6.8 6.8 6.7 6.7 ...
##  $ ts_3786:'data.frame': 80 obs. of  2 variables:
##   ..$ date : Date[1:80], format: "2010-01-01" ...
##   ..$ value: num [1:80] 8.3 8.2 8.2 8.1 8 7.9 8.1 8.1 8.1 7.8 ...
##  $ ts_3787:'data.frame': 80 obs. of  2 variables:
##   ..$ date : Date[1:80], format: "2010-01-01" ...
##   ..$ value: num [1:80] 9.8 9.8 9.9 9.9 9.6 9.4 9.4 9.5 9.5 9.4 ...
##  $ ts_3788:'data.frame': 79 obs. of  2 variables:
##   ..$ date : Date[1:79], format: "2010-01-01" ...
##   ..$ value: num [1:79] 9.4 9.4 9.3 9.3 9.3 9.2 9.2 9.2 9.2 9.2 ...
##  $ ts_3789:'data.frame': 79 obs. of  2 variables:
##   ..$ date : Date[1:79], format: "2010-01-01" ...
##   ..$ value: num [1:79] 8.5 8.5 8.4 8.6 8.5 8.3 8.2 8.1 8.1 8.5 ...
##  $ ts_3790:'data.frame': 79 obs. of  2 variables:
##   ..$ date : Date[1:79], format: "2010-01-01" ...
##   ..$ value: num [1:79] 5 5 5.1 5.1 5.1 5.2 5 5.1 5.1 5.1 ...
##  $ ts_3791:'data.frame': 77 obs. of  2 variables:
##   ..$ date : Date[1:77], format: "2010-01-01" ...
##   ..$ value: num [1:77] 7.9 8 7.9 7.8 7.8 7.7 7.7 7.6 7.7 7.8 ...
```

Cada série de desemprego está alocada em um elemento de `l.out`. Esses dados não estão no formato desejável. O ideal é agregarmos todos em um único dataframe, com uma coluna designando o nome da série. Note também que a informação sobre o país não está clara. Afinal, qual é o país identificado por `'ts_3785'`?

No próximo código iremos tratar estes problemas utilizando a linguagem funcional do R. Deixo claro que o código ficou mais complicado do que deveria. Novamente, não se preocupe se não entender. Iremos tratar do assunto *loops* e funções no capítulo 8.

```
clean.bets <- function(df.in, name.in) {
  # function for cleaning data from BETS
  #
  # ARGS: df.in - dataframe within a list
  #       name.in - name for dataframe
  #
  # VALUE: a new dataframe with new column type

  #set column
  df.in$type <- name.in

  # return df
  return(df.in)
}

# set names
name.in <- c("Germany", "Canada", "United States", "France",
             "Italy", "Japan", "United Kingdom"   )

# loop for changeing contents of l.out
for (i.l in 1:length(l.out )){
  l.out[[i.l]] <- clean.bets(l.out[[i.l]], name.in[i.l])
```

******ebook converter DEMO Watermarks*******

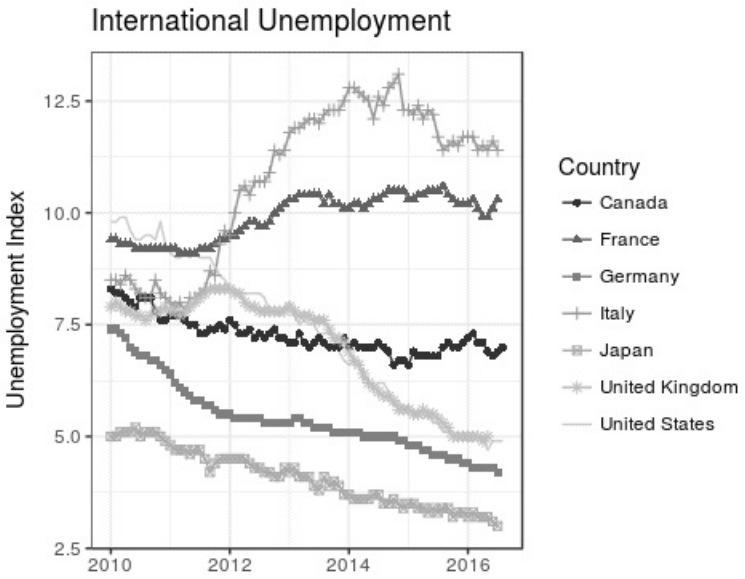
```
}

# bind all dfs together
my.df.long <- do.call(what = dplyr::bind_rows, args = l.out)

# check it
dplyr::glimpse(my.df.long)

## Observations: 553
## Variables: 3
## $ date  <date> 2010-01-01, 2010-02-01, 2010-03-01, 2010...
## $ value <dbl> 7.4, 7.4, 7.3, 7.2, 7.0, 6.9, 6.8, 6.8, 6...
## $ type  <chr> "Germany", "Germany", "Germany", "Germany...
```

Agora sim os dados estão no formato desejado. Vamos visualizar o desemprego nos diferentes países ao longo dos anos com uma figura.



Como podemos ver, a maioria dos países está em uma trajetória decrescente de desemprego, com excessão da França e Itália.

5.2.4 `rbcb`

Como vimos anteriormente, os pacotes `Quandl` e `BETS` permitem o acesso a base de dados de séries temporais do Banco Central Brasileiro (BCB). Nessa mesma função, um diferencial do pacote `rbcb` é o acesso direto ao API do BCB. Não é necessário cadastro, o pacote possui um baixo número de dependências para a instalação e é bastante estável. O formato de entrada dos símbolos é intuitivo e permite a definição do nome das colunas no objeto retornado.

Vamos usar o pacote para estudar a inadimplência de crédito no sistema financeiro Brasileiro. O primeiro passo no uso de `rbcb` é procurar o símbolo da série de interesse. Acessando o sistema de séries temporais do BCB, vemos que o símbolo para o percentual total de inandimplência é 21082.

******ebook converter DEMO Watermarks*******

```
library(rbcb)
library(dplyr)

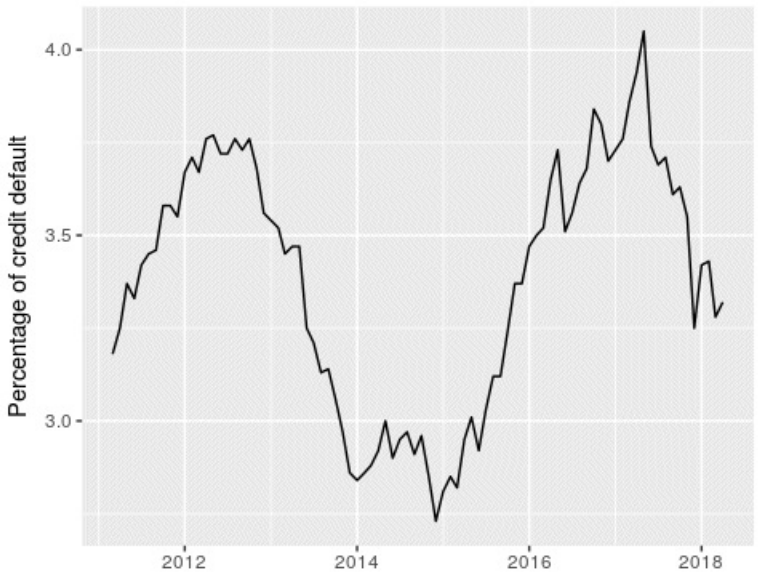
# set ids and dates
id.series <- c(perc.default = 21082)
first.date = '2010-01-01'

# get series from bcb
df.cred <- get_series(code = id.series,
                      start_date = first.date,
                      end_date = Sys.Date())

# check it
glimpse(df.cred)
```

```
## Observations: 86
## Variables: 2
## $ date         <date> 2011-03-01, 2011-04-01, 2011-05-0...
## $ perc.default <dbl> 3.18, 3.25, 3.37, 3.33, 3.42, 3.45...
```

Note que indicamos o nome da coluna na própria definição da entrada `code`. Vamos verificar os dados importados visualmente:



Como podemos ver, a percentagem de inadimplência aumentou a partir de 2015. Para ter uma idéia mais clara do problema, vamos incluir no gráfico a percentagem para pessoa física e pessoa jurídica. Olhando novamente o sistema do BCB, vemos que o símbolos de interesse são 21083 e 21084, respectivamente. O próximo código baixa os dados das duas séries.

```
# set ids
id.series <- c(people = 21083,
               companies = 21084)
first.date = '2010-01-01'

# get series from bcb
l.out <- get_series(code = id.series,
```

******ebook converter DEMO Watermarks*******

```
                    start_date = first.date ,
                    end_date = Sys.Date())

# check output
class(l.out) # or dplyr::glimpse

## [1] "list"
```

Note que a saída, nesse caso, é uma lista. Assim como para o caso do pacote `BETS`, teremos que reformatar os dados para um único dataframe no formato longo. Utilizaremos a mesma estratégia anterior, criaremos uma função para reestruturar os dados e depois chamaremos `do.call` para agregar os dados.

```
clean.rbcb <- function(df.in) {
  # function for cleaning data from rbcb
  #
  # ARGS: df.in - dataframe within a list
  #
  # VALUE: a new dataframe with new column type

  #set column
  df.in$type <- names(df.in)[2]
  names(df.in) <- c('date', 'value', 'type')

  # return df
  return(df.in)
}

# call clean.rbcb for each element of l.out
l.out <- lapply(X = l.out, FUN = clean.rbcb)

# bind all dataframes in l.out
df.cred <- do.call(rbind, l.out)

# check result
dplyr::glimpse(df.cred)

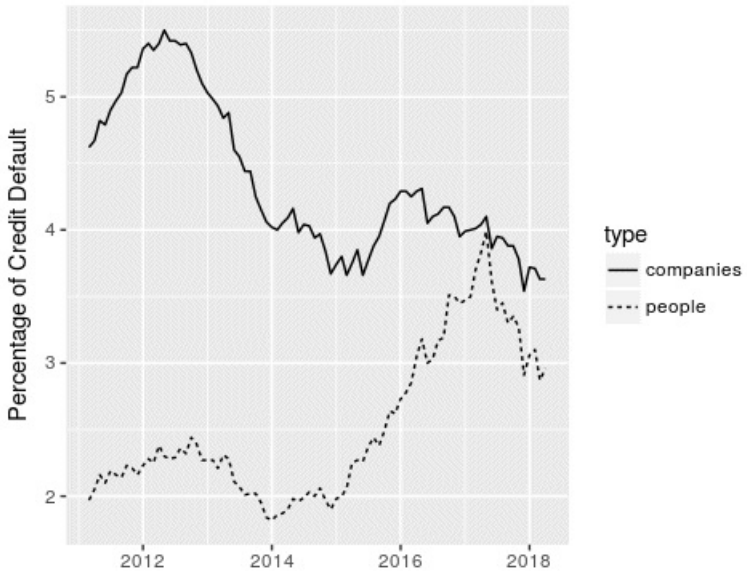
## Observations: 172
## Variables: 3
## $ date  <date> 2011-03-01, 2011-04-01, 2011-05-01, 2011...
## $ value <dbl> 1.97, 2.05, 2.16, 2.10, 2.19, 2.16, 2.14,...
## $ type  <chr> "people", "people", "people", "people", "...
```

Após estruturarmos os dados para o formato correto, partimos para a visualização das séries:

```
p <- ggplot(df.cred, aes(x=date, y = value, linetype = type)) +
  geom_line() + labs(y = 'Percentage of Credit Default',
                     x = '')

print(p)
```

******ebook converter DEMO Watermarks*******



Como podemos ver, a inadimplência de crédito para pessoa física aumentou muito mais do que a para pessoa jurídica (empresas) nos últimos anos. Poderiámos, facilmente, integrar o código anterior para uma análise mais completa dos dados em algum problema de pesquisa.

Sugiro o leitor se familiarizar com o sistema de séries temporais do BCB e o pacote `rbcb`. A quantidade e diversidade de series econômicas e financeiras é muito grande. Além disso, os dados estão sempre atualizados. O uso do `rbcb` permite uma sinergia entre a etapa de coleta de dados e o estudo em questão.

5.2.5 `GetDFPData`

Pacote `GetDFPData` (Perlin 2017) fornece uma interface aberta para todas as demonstrações financeiras distribuídas pela B3 e pela CVM nos sistemas DFP, FRE e FCA. Ele não só faz o *download* dos dados, mas também consolida as diferentes notações contábeis, ajusta à inflação e torna as tabelas prontas para pesquisa dentro de um formato tabular. Isso inclui a situação da empresa (ativa ou não-ativa), setor, código CVM[2], tickers relacionados, histórico de dividendos, estrutura de dívida, remuneração de diretores, e diversos outros eventos e informações corporativas. Os diferenciais do pacote em relação a outros distribuidores de dados comerciais são: livre acesso, facilidade para baixar dados em larga larga escala e a variedade de dados disponíveis.

Uma versão web de `GetDFPData` foi desenvolvida e publicada na internet como um aplicativo *shiny* em http://www.msperlin.com/shiny/GetDFPData/. Esse fornece uma interface gráfica direta e simples para as principais funcionalidade do pacote. Usuários podem selecionar as empresas disponíveis, o intervalo de datas e baixar os dados como uma planilha do Excel ou um arquivo compactado com vários arquivos *csv*.

O ponto de partida no uso de `GetDFPData` é baixar informações atuais sobre empresas disponíveis. Esta informação é coletada por um *script* externo que importa e mescla bases de

******ebook converter DEMO Watermarks*******

dados atualizadas da B3 e CVM. A tabela resultante é salva como um arquivo *csv* no Github. Os dados desse arquivo são atualizados periodicamente para acomodar mudanças, tal como a chegada de novas empresas e novos relatórios financeiros.

O acesso a tabela é possível com a função `gdfpd.get.info.companies`. Estabelecemos argumento `type.data = 'companies'` para que a função retorne apenas informações sobre empresas, e não arquivos.

```
library(GetDFPData)

df.info <- gdfpd.get.info.companies(type.data = 'companies')

## Found cache file. Loading data..

dplyr::glimpse(df.info)

## Observations: 694
## Variables: 16
## $ name.company         <chr> "521 PARTICIPAÇOES S.A. - ...
## $ id.company           <int> 16330, 16284, 108, 20940, ...
## $ cnpj                 <dbl> 1.547749e+12, 1.851771e+12...
## $ date.registration    <date> 1997-07-11, 1997-05-30, 1...
## $ date.constitution    <date> 1996-07-30, 1997-04-02, 1...
## $ city                 <chr> "RIO DE JANEIRO", "RIO DE ...
## $ estate               <chr> "RJ", "RJ", "RS", "SP", "S...
## $ situation            <chr> "ATIVO", "ATIVO", "CANCELA...
## $ situation.operations <chr> "LIQUIDAÇÃO EXTRAJUDICIAL"...
## $ listing.segment      <chr> NA, "Tradicional", "Tradic...
## $ main.sector          <chr> NA, "Financeiro e Outros",...
## $ sub.sector           <chr> NA, "Outros", "Siderurgia ...
## $ segment              <chr> NA, "Outros", "Siderurgia"...
## $ tickers              <chr> NA, "QVQP3B", NA, NA, "ADH...
## $ first.date           <date> 1998-12-31, 2001-12-31, 2...
## $ last.date            <date> 2017-12-31, 2017-12-31, 2...
```

Essa tabela disponibiliza os identificadores numéricos das empresas, setores de atividades, atual segmento de governança, *tickers* negociados na bolsa e situação atual (ativa ou não). O número atual de empresas ativas e inativas, a partir de 2018-06-10, está disponível na coluna `situation`. Observa-se 518 empresas ativas e 170 canceladas. Essa é uma excelente fonte de informação para um estudo exploratório. Pode-se facilmente filtrar empresas para datas, setores, *tickers* ou segmentos de governança corporativa.

Toda empresa no banco de dados pode ser identificada pelo seu nome oficial ou identificação numérica da CVM. Por uma questão de conveniência, escolhemos selecionar empresas pelo nome. A função `gdfpd.search.company` permite que o usuário procure o nome oficial de uma instituição. Dado um texto, a função procurará uma correspondência parcial com os nomes de todas as empresas disponíveis no banco de dados. Em seu uso, caracteres latinos e maiúsculas e minúsculas são ignorados. Vamos encontrar o nome oficial nome da Ambev, uma das maiores empresas do Brasil. Para isso, basta usar o comando `gdfpd.search.company('ambev')`. Após isso, vemos que o nome oficial da Ambev é `'AMBEV S.A.'`. Dados estão disponíveis de 2012 até final de 2017. Note que a situação da empresa também é apresentada. Isso ajuda a verificar a disponibilidade dos dados.

******ebook converter DEMO Watermarks*******

Usamos a função principal do pacote, `gdfpd.GetDFPData`, para baixar informações. Definimos o nome oficial da empresa como entrada `name.companies` e o período de tempo como entradas `first.date` e `last.date`. Vamos baixar os dados disponíveis para a Ambev:

```
library(GetDFPData)
library(dplyr)

# set options
name.companies <- 'AMBEV S.A.'
first.date <- '2015-12-31'
last.date  <- '2017-12-31'

# download data
df.reports <- gdfpd.GetDFPData(name.companies = name.companies,
                               first.date = first.date,
                               last.date = last.date)
```

```
## Found cache file. Loading data..
## Fetching new files from Bovespa.
##
## Downloading data for 1 companies
## First Date: 2015-12-31
## Laste Date: 2017-12-31
## Inflation index: dollar
##
## Downloading inflation data
##  Caching inflation RDATA into tempdir()  Done
##
## Inputs looking good! Starting download of files:
##
## AMBEV S.A.
##  Available periods: 2017-12-31   2016-12-31  2015-12-31
##
##
## Processing 23264 - AMBEV S.A.
##  Finding info from Bovespa
##      Found BOV cache file
##  Processing 23264 - AMBEV S.A. | date 2017-12-31
##      Acessing DFP data | Found DFP cache file
##      Acessing FRE data | Found FRE cache file
##      Acessing FCA data | Found FCA cache file
##  Processing 23264 - AMBEV S.A. | date 2016-12-31
##      Acessing DFP data | Found DFP cache file
##      Acessing FRE data | Found FRE cache file
##      Acessing FCA data | Found FCA cache file
##  Processing 23264 - AMBEV S.A. | date 2015-12-31
##      Acessing DFP data | Found DFP cache file
##      Acessing FRE data | Found FRE cache file
##      Acessing FCA data | Found FCA cache file
```

As mensagens de `gdfpd.GetDFPData` relatam os estágios do processo, desde a aquisição de dados da tabela de referência ao download e leitura dos arquivos da B3. Observe que os arquivos de três sistemas são acessados: DFP (*Demostrativos Financeiros Padronizados*), FRE (*Formulário de Referência*) e FCA (*Formulário Cadastral*). Observe também o uso de um sistema de cache, o qual acelera significativamente o uso do software ao salvar localmente as informações importadas.

******ebook converter DEMO Watermarks*******

O objeto resultante de `gdfpd.GetDFPData` é um `tibble`, um tipo de objeto tabular que permite colunas do tipo lista (Wickham 2016d). Isso possibilita salvar a informação financeira em uma estrutura tabular. Vamos dar uma olhada no conteúdo de `df.reports`:

```
glimpse(df.reports)
```

```
## Observations: 1
## Variables: 40
## $ company.name                    <chr> "AMBEV S.A."
## $ company.code                    <int> 23264
## $ cnpj                            <chr> "07526557000100"
## $ date.company.constitution       <date> 2005-07-08
## $ date.cvm.registration           <date> 2013-10-30
## $ company.tickers                 <chr> "ABEV3"
## $ min.date                        <date> 2015-12-31
## $ max.date                        <date> 2017-12-31
## $ n.periods                       <int> 3
## $ company.segment                 <chr> "Tradicional"
## $ current.stockholders            <list> [<c("AMBEV S....
## $ current.stock.composition       <list> [<c("AMBEV S....
## $ fr.assets                       <list> [<c("AMBEV S....
## $ fr.liabilities                  <list> [<c("AMBEV S....
## $ fr.income                       <list> [<c("AMBEV S....
## $ fr.cashflow                     <list> [<c("AMBEV S....
## $ fr.assets.consolidated          <list> [<c("AMBEV S....
## $ fr.liabilities.consolidated     <list> [<c("AMBEV S....
## $ fr.income.consolidated          <list> [<c("AMBEV S....
## $ fr.cashflow.consolidated        <list> [<c("AMBEV S....
## $ fr.auditing.report              <list> [<c("AMBEV S....
## $ history.dividends               <list> [<c("AMBEV S....
## $ history.stockholders            <list> [<c("AMBEV S....
## $ history.capital.issues          <list> [<c("AMBEV S....
## $ history.mkt.value               <list> [<c("AMBEV S....
## $ history.capital.increases       <list> [<c("AMBEV S....
## $ history.capital.reductions      <list> [<>]
## $ history.stock.repurchases       <list> [<c("AMBEV S....
## $ history.other.stock.events      <list> [<AMBEV S.A.,...
## $ history.compensation            <list> [<c("AMBEV S....
## $ history.compensation.summary    <list> [<c("AMBEV S....
## $ history.transactions.related    <list> [<c("AMBEV S....
## $ history.debt.composition        <list> [<c("AMBEV S....
## $ history.governance.listings     <list> [<c("AMBEV S....
## $ history.board.composition       <list> [<c("AMBEV S....
## $ history.committee.composition   <list> [<c("AMBEV S....
## $ history.family.relations        <list> [<c("AMBEV S....
## $ history.family.related.companies <list> [<c("AMBEV S....
## $ history.auditing                <list> [<c("AMBEV S....
## $ history.responsible.docs        <list> [<c("AMBEV S....
```

Como podemos ver, os dados retornados são vastos. Cada coluna em `df.reports` é um tabela indexada ao tempo. A explicação de cada coluna não cabe aqui mas vamos dar uma olhada no balanço patrimonial da empresa, disponível em `df.reports$fr.assets[[1]]`:

```
# save assets in df
fr.assets <- df.reports$fr.assets[[1]]

# check it
dplyr::glimpse(fr.assets)
```

******ebook converter DEMO Watermarks*******

```
## Observations: 48
## Variables: 6
## $ name.company       <chr> "AMBEV S.A.", "AMBEV S.A.", ...
## $ ref.date           <date> 2017-12-31, 2017-12-31, 201...
## $ acc.number         <chr> "1", "1.01", "1.01.01", "1.0...
## $ acc.desc           <chr> "Ativo Total", "Ativo Circul...
## $ acc.value          <dbl> 91568737, 11272513, 2017034,...
## $ acc.value.infl.adj <dbl> 27686018.32, 3408270.24, 609...
```

Para uma melhor visualização dos demostrativos financeiros, o pacote também inclui função `gdfpd.convert.to.wide` para converter os dados para formato *wide*, com as datas preenchendo colunas. Veja a seguir:

```
# convert assets to wide
fr.assets.wide <- gdfpd.convert.to.wide(fr.assets)

# display it
print(fr.assets.wide)
```

```
##    acc.number                       acc.desc name.company
## 1           1                    Ativo Total   AMBEV S.A.
## 2        1.01                Ativo Circulante   AMBEV S.A.
## 3     1.01.01  Caixa e Equivalentes de Caixa   AMBEV S.A.
## 4     1.01.02         Aplicações Financeiras   AMBEV S.A.
## 5     1.01.03               Contas a Receber   AMBEV S.A.
## 6     1.01.04                     Estoques   AMBEV S.A.
## 7     1.01.05              Ativos Biológicos   AMBEV S.A.
## 8     1.01.06           Tributos a Recuperar   AMBEV S.A.
## 9     1.01.07           Despesas Antecipadas   AMBEV S.A.
## 10    1.01.08      Outros Ativos Circulantes   AMBEV S.A.
## 11       1.02           Ativo Não Circulante   AMBEV S.A.
## 12    1.02.01 Ativo Realizável a Longo Prazo   AMBEV S.A.
## 13    1.02.02                   Investimentos   AMBEV S.A.
## 14    1.02.03                     Imobilizado   AMBEV S.A.
## 15    1.02.04                      Intangível   AMBEV S.A.
## 16    1.02.05                        Diferido   AMBEV S.A.
##    2015-12-31 2016-12-31 2017-12-31
## 1    94722744   91212661   91568737
## 2    14044725   13131543   11272513
## 3     1944872    1405387    2017034
## 4     2597469     648951     631637
## 5     3583444    2183510    3055953
## 6     1987853    2160921    2107274
## 7           0          0          0
## 8     2645342    4879093    2625960
## 9      551996     582460     572117
## 10     733749    1271221     262538
## 11   80678019   78081118   80296224
## 12    3764517    1830422    4148374
## 13   66073444   64358515   64593359
## 14    9911024   11144956   10806139
## 15     929034     747225     748352
## 16          0          0          0
```

A exportação dos dados também é fácil, basta selecionar um formato (`'csv'` ou `'xlsx'`) e usar `gdfpd.export.DFP.data`:

```
my.basename <- 'MyExcelData'
```

******ebook converter DEMO Watermarks*******

```
my.type.export <- 'xlsx'

gdfpd.export.DFP.data(df.reports = df.reports,
                      base.file.name = my.basename,
                      type.export = my.type.export)

## File  MyExcelData.xlsx  already exists. Deleting it..
## Start Copying Dataframes to MyExcelData.xlsx
##  Copying table metadata - 1 rows, 5 columns
##  Copying table metadata - 1 rows, 5 columns
##  Copying table current.stockholders - 6 rows, 6 columns
##  Copying table current.stock.composition - 3 rows, 4 columns
##  Copying table fr.assets - 48 rows, 6 columns
##  Copying table fr.liabilities - 75 rows, 6 columns
##  Copying table fr.income - 78 rows, 6 columns
##  Copying table fr.cashflow - 85 rows, 6 columns
##  Copying table fr.assets.consolidated - 45 rows, 6 columns
##  Copying table fr.liabilities.consolidated - 78 rows, 6 columns
##  Copying table fr.income.consolidated - 84 rows, 6 columns
##  Copying table fr.cashflow.consolidated - 81 rows, 6 columns
##  Copying table fr.auditing.report - 3 rows, 3 columns
##  Copying table history.dividends - 22 rows, 12 columns
##  Copying table history.stockholders - 18 rows, 14 columns
##  Copying table history.capital.issues - 6 rows, 5 columns
##  Copying table history.mkt.value - 3 rows, 5 columns
##  Copying table history.capital.increases - 35 rows, 8 columns
##  Copying table history.capital.reductions - 0 rows, 0 columns
##  Copying table history.stock.repurchases - 6 rows, 12 columns
##  Copying table history.other.stock.events - 1 rows, 8 columns
##  Copying table history.compensation - 9 rows, 18 columns
##  Copying table history.compensation.summary - 9 rows, 9 columns
##  Copying table history.transactions.related - 92 rows, 13 columns
##  Copying table history.debt.composition - 17 rows, 9 columns
##  Copying table history.governance.listings - 3 rows, 5 columns
##  Copying table history.board.composition - 88 rows, 18 columns
##  Copying table history.committee.composition - 23 rows, 18 columns
##  Copying table history.family.relations - 3 rows, 10 columns
##  Copying table history.family.related.companies - 69 rows, 11 columns
##  Copying table history.auditing - 12 rows, 10 columns
##  Copying table history.responsible.docs - 6 rows, 5 columns
##
## Export sucessful. Data available at file MyExcelData.xlsx
```

O arquivo Excel resultante conterá cada tabela de `df.reports` em uma aba diferente da planilha.

Apenas tocamos nos pontos principais do pacote `GetDFPData`. Aos interessados, sugiro a leitura do artigo em Perlin (2017), o qual inclui maiores detalhes sobre os dados disponíveis e o funcionamento das funções.

5.2.6 `GetHFData`

Pacote `GetHFData` (Perlin and Ramos 2016) foi projetado para facilitar a importação e análise de dados de negociação de alta frequência na B3. Esses dados são comumente utilizados para a realização de pesquisa na área de microestrutura de mercado. Os dados são coletados diretamente do site *ftp* da bolsa, no formato de arquivos zipados. Um grande diferencial do pacote é o acesso a dados não somente do mercado de ações, mas também de contratos derivativos tal como

******ebook converter DEMO Watermarks*******

opções e futuros, os quais são particularmente difíceis de encontrar. No momento de escrita do livro, `GetHFData` é o único pacote disponível no CRAN que permite o acesso direto e gratuito a dados de alta frequência.

Vamos apresentar um exemplo simples, baixando dados de transações em uma data atual para duas ações bastante negociadas no mercado financeiro brasileiro, Petrobras (PETR4) e Gerdau (GGBR4). Iremos agregar os dados em intervalos de 5 em 5 minutos.

```
library(GetHFData)

# set tickers and type of market
my.ticker <- c('PETR4', 'GGBR4')
my.type.market <- 'equity'

# set last date
last.date <-  '2018-04-05'

# get data!
my.df <- ghfd_get_HF_data(my.assets = my.ticker,
                          type.market = 'equity',
                          first.date = last.date,
                          last.date = last.date,
                          first.time = '10:00:00',
                          last.time = '17:00:00',
                          type.output = 'agg',
                          agg.diff = '5 min')

##
## Running ghfd_get_HF_Data for:
##    type.market = equity
##    type.data = trades
##    my.assets = PETR4 GGBR4
##    type.output = agg
##       agg.diff = 5 min
## Reading ftp contents for equity(trades) (attempt = 1|10)
##    Found  489  files in ftp
##    First available date in ftp:  2016-06-09
##    Last available date in ftp:   2018-06-07
##    First date to download:  2018-04-05
##    Last date to download:   2018-04-05
## Downloading ftp files/NEG_20180405.zip (1|1) Attempt 1
##    -> Reading files - Imported  1092115 lines, 489 unique tickers
##    -> Processing file - Found 65798 lines for 2 selected tickers
##    -> Aggregation resulted in dataframe with 164 rows

# print results
dplyr::glimpse(my.df)

## Observations: 164
## Variables: 13
## $ InstrumentSymbol <chr> "GGBR4", "GGBR4", "GGBR4", "GG...
## $ SessionDate      <date> 2018-04-05, 2018-04-05, 2018-...
## $ TradeDateTime    <dttm> 2018-04-05 10:05:00, 2018-04-...
## $ n.trades         <int> 182, 312, 118, 85, 87, 317, 23...
## $ last.price       <dbl> 15.88, 15.84, 15.77, 15.76, 15...
## $ weighted.price   <dbl> 15.88797, 15.86229, 15.79817, ...
## $ period.ret       <dbl> -0.0006293266, -0.0031466331, ...
## $ period.ret.volat <dbl> 0.0006628653, 0.0005021294, 0....
```

******ebook converter DEMO Watermarks*******

```
## $ sum.qtd          <dbl> 170800, 269000, 56800, 46100, ...
## $ sum.vol          <dbl> 2713665, 4266944, 897335, 7267...
## $ n.buys           <int> 27, 148, 21, 34, 16, 193, 86, ...
## $ n.sells          <int> 155, 164, 97, 51, 71, 124, 153...
## $ Tradetime        <chr> "10:05:00", "10:10:00", "10:15...
```

A saída de `ghfd_get_HF_data` é um dataframe com várias colunas: tempo, último preço, volatilidade, número de negociações iniciadas pelo comprador/vendedor, entre outras informações. Aqui não cabe explicar cada coluna. Essa informação pode ser encontrada no artigo dos autores, Perlin and Ramos (2016).

O usuário também pode baixar dados brutos, sem agregação, definindo a entrada `type.output` como `'raw'`. Vamos tentar esta opção:

```
library(GetHFData)

# set options
my.ticker <- c('PETR4','PETR3')
my.type.market <- 'equity'
my.type.data <- 'trades'
first.time <- '10:00:00'
last.time <- '17:00:00'
type.output <- 'raw'

# set last date
last.date <- '2018-04-05'

# get data!
my.df <- ghfd_get_HF_data(my.assets = my.ticker,
                          type.market = my.type.market,
                          type.data = my.type.data,
                          first.date = last.date,
                          last.date = last.date,
                          first.time = first.time ,
                          last.time = last.time ,
                          type.output = type.output)

##
## Running ghfd_get_HF_Data for:
##    type.market = equity
##    type.data = trades
##    my.assets = PETR4 PETR3
##    type.output = raw
## Reading ftp contents for equity(trades) (attempt = 1|10)
##    Found  259  files in ftp
##    First available date in ftp:  2017-05-22
##    Last available date in ftp:   2018-06-07
##    First date to download:  2018-04-05
##    Last date to download:   2018-04-05
## Downloading ftp files/NEG_20180405.zip (1|1) Attempt 1
##    -> Reading files - Imported  1092115 lines, 489 unique tickers
##    -> Processing file - Found 69732 lines for 2 selected tickers

# print results
dplyr::glimpse(my.df)

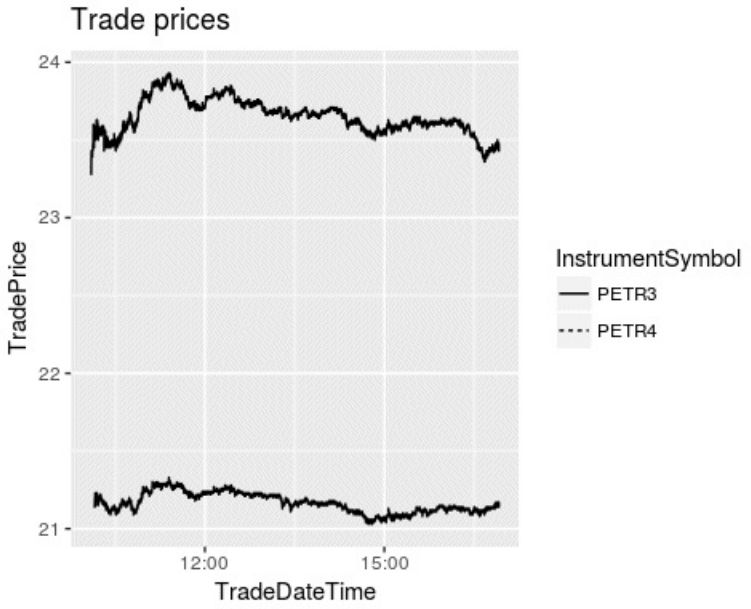
## Observations: 68,994
## Variables: 10
```

******ebook converter DEMO Watermarks*******

```
## $ SessionDate         <date> 2018-04-05, 2018-04-05, 20...
## $ InstrumentSymbol    <chr> "PETR3", "PETR3", "PETR3", ...
## $ TradePrice          <dbl> 23.31, 23.31, 23.31, 23.31,...
## $ TradedQuantity      <dbl> 300, 100, 200, 600, 800, 25...
## $ Tradetime           <chr> "10:06:00.003", "10:06:00.0...
## $ CrossTradeIndicator <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
## $ BuyMember           <dbl> 85, 3, 21, 72, 72, 85, 3, 2...
## $ SellMember          <dbl> 238, 21, 21, 3, 115, 3, 21,...
## $ TradeDateTime       <dttm> 2018-04-05 10:06:00, 2018-...
## $ TradeSign           <dbl> -1, -1, -1, -1, -1, -1, -1,...
```

A função retorna um dataframe com diversas colunas. Esses dados não tem agregação temporal e mostram as transações no formato cru, com um irregular espaço de tempo entre cada observação. Assim como para o exemplo anterior, não cabe neste livro explicar a informação disponível em cada coluna. Para os mais interessados, recomendo novamente a leitura do artigo dos autores.

Seguindo em frente, vamos visualizar os dados de preços:



Na última atualização do pacote foi implementado a possibilidade de baixar também dados de ordens financeiras enviadas a bolsa por investidores. Isso abre um novo leque de possibilidades de pesquisa, permitindo a construção de índices de liquidez mais fidedignos, tal como o *spread* de preços, diferença entre melhor preço de venda e preço de compra, resiliência do livro de negócios, entre outras possibilidades. O acesso aos dados é bastante fácil. Seguindo o exemplo anterior, basta trocar argumento `my.type.data` para `orders`. O tempo de processamento, porém, é significativamente maior devido ao tamanho dos arquivos a serem baixados e lidos. Veja o exemplo a seguir:

```
library(GetHFData)

# set options
my.ticker <- c('PETR4','VALE5')
my.type.market <- 'equity'
my.type.data <- 'orders'
```

******ebook converter DEMO Watermarks*******

```
first.time <- '10:00:00'
last.time <- '17:00:00'
type.output <- 'raw'

# get available dates from ftp
df.available.dates <- ghfd_get_ftp_contents(my.type.market)

##
## Reading ftp contents for equity(trades) (attempt = 1|10)

# set last date
last.date <- max(df.available.dates$dates)

# get data!
my.df <- ghfd_get_HF_data(my.assets = my.ticker,
                          type.market = my.type.market,
                          type.data = my.type.data,
                          first.date = last.date,
                          last.date = last.date,
                          first.time = first.time ,
                          last.time = last.time ,
                          type.output = type.output)

##
## Running ghfd_get_HF_Data for:
##    type.market = equity
##    type.data = orders
##    my.assets = PETR4 VALE5
##    type.output = raw
## Reading ftp contents for equity(orders) (attempt = 1|10)
##    Found  517  files in ftp
##    First available date in ftp:  2017-05-22
##    Last available date in ftp:   2018-06-07
##    First date to download:  2018-06-07
##    Last date to download:   2018-06-07
## Downloading ftp files/OFER_CPA_20180607.zip (1|2) Attempt 1
##    -> Reading files - Imported  11644775 lines, 818 unique tickers
##    -> Processing file - Found 299432 lines for 2 selected tickers
## Downloading ftp files/OFER_VDA_20180607.zip (2|2) Attempt 1
##    -> Reading files - Imported  11721025 lines, 814 unique tickers
##    -> Processing file - Found 263651 lines for 2 selected tickers

# print results
dplyr::glimpse(my.df)

## Observations: 546,362
## Variables: 16
## $ SessionDate           <date> 2018-06-07, 2018-06-07, ...
## $ InstrumentSymbol      <chr> "PETR4", "PETR4", "PETR4"...
## $ OrderSide             <fct> Buy, Buy, Buy, Buy, Buy, ...
## $ SequentialOrderNumber <dbl> 82120693514, 82120803465,...
## $ SecondaryOrderID      <int> 16525532, 18524265, 18547...
## $ ExecutionType         <fct> Update, Update, Update, U...
## $ PriorityTime          <chr> "13:50:59.077000", "15:50...
## $ PriorityIndicator     <chr> "2147483647", "2147483647...
## $ OrderPrice            <dbl> 14.89, 14.00, 15.40, 15.5...
## $ TotalQuantity         <dbl> 400, 1000, 1000, 1000, 10...
## $ TradedQuantity        <dbl> 0, 0, 0, 0, 0, 1000, 0, 0...
## $ OrderDatetime         <dttm> 2018-06-07 13:50:59, 201...
## $ OrderStatus           <fct> Replaced, Replaced, Repla...
```

******ebook converter DEMO Watermarks*******

```
## $ AgressorIndicator     <int> 0, 0, 0, 0, 0, 2, 0, 0, 2...
## $ Member                <int> 3, 3, 3, 3, 3, 3, 114, 3,...
## $ PriorityDateTime      <dttm> 2018-06-07 13:50:59.077,...
```

Não é incomum o computador travar com esse exemplo. O problema é que o R usa toda a memória da máquina para alocar os dados de ordens financeiras, não deixando memória suficiente para o sistema operacional. Caso isso acontecer, tente fechar o RStudio ou reiniciar a sessão. Em caso extremo, reinicialize o seu computador.

5.2.7 `GetTDData`

Arquivos com informações sobre preços e retornos de títulos emitidos pelo governo brasileiro podem ser baixados manualmente no site do Tesouro Nacional. O tesouro direto é um tipo especial de mercado onde o governo brasileiro é o credor e formador de mercado ao mesmo tempo (Perlin 2016a). Os títulos vendidos na plataforma são bastante populares devido a atratividade das taxas de retorno e alta liquidez oferecida ao investidor comum. Isso inclui títulos do tipo LFT, LTN, NTN-C, NTN-B, NTN-B Principal e NTN-F.

Pacote `GetTDData` importa os dados das planilhas em Excel e os organiza. O resultado é um dataframe com formato adequado que torna o uso destes dados muito mais fácil. O usuário apenas precisa indicar o nome dos ativos de renda fixa e as datas de maturidade desejadas.

Como exemplo, vamos baixar dados de um título prefixado do tipo LTN com vencimento em 2021-01-01. Esse é o tipo de contrato de dívida mais simples que o governo brasileiro emite, não pagando nenhum cupom durante sua validade e, na data de vencimento, retorna 1.000 R$ ao comprador. Para baixar os dados da internet, basta usar o código a seguir:

```
library(GetTDData)

asset.codes <- 'LTN'   # Identifier of assets
maturity <- '010121'  # Maturity date as string (ddmmyy)

# download
my.flag <- download.TD.data(asset.codes = asset.codes)

# read files
df.TD <- read.TD.files(asset.codes = asset.codes,
                       maturity = maturity)
```

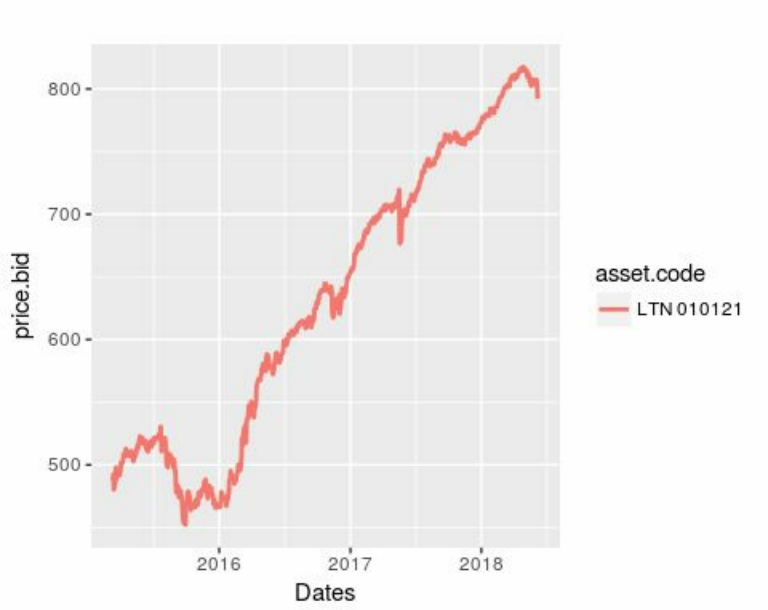
Vamos checar o conteúdo do dataframe:

```
# check content
dplyr::glimpse(df.TD)
```

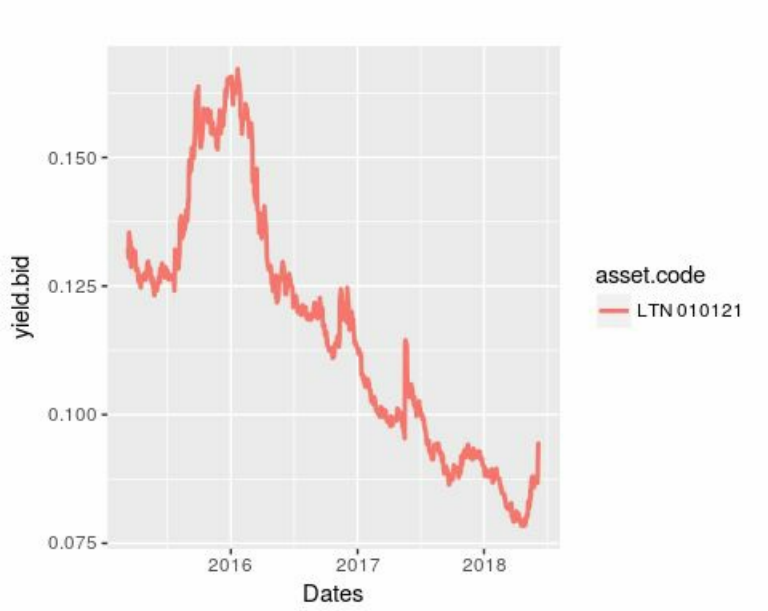
```
## Observations: 807
## Variables: 5
## $ ref.date   <date> 2015-03-10, 2015-03-11, 2015-03-12,...
## $ yield.bid  <dbl> 0.1324, 0.1310, 0.1304, 0.1354, 0.13...
## $ price.bid  <dbl> 487.04, 490.78, 492.53, 480.37, 485....
## $ asset.code <chr> "LTN 010121", "LTN 010121", "LTN 010...
## $ matur.date <date> 2021-01-01, 2021-01-01, 2021-01-01,...
```

Temos informações sobre data de referência, *yield* (retorno), preço e dia de maturidade. No ******ebook converter DEMO Watermarks*******

gráfico a seguir checamos os dados:



Como esperado de um título de dívida pré-fixado, os preços possuem uma tendência positiva. Essa dívida irá valer exatamente 1000 R$ na sua data de maturidade, 2021-01-01. Podemos também visualizar as mudanças do *yield* do título:



Os retornos do título estão em tendência de queda. Este resultado é esperado pois, como vimos anteriormente, a taxa SELIC está num nível muito baixo.

As funções do `GetTDData` também funcionam com vários argumentos como `asset.codes` e `maturity`. Suponhamos que desejamos visualizar todos os preços de todos os prazos disponíveis para títulos do tipo LTN a partir de 2010. Tudo o que precisamos fazer é adicionar o valor `NULL`

******ebook converter DEMO Watermarks*******

ao argumento `maturity` e filtrar as datas:

```
library(GetTDData)

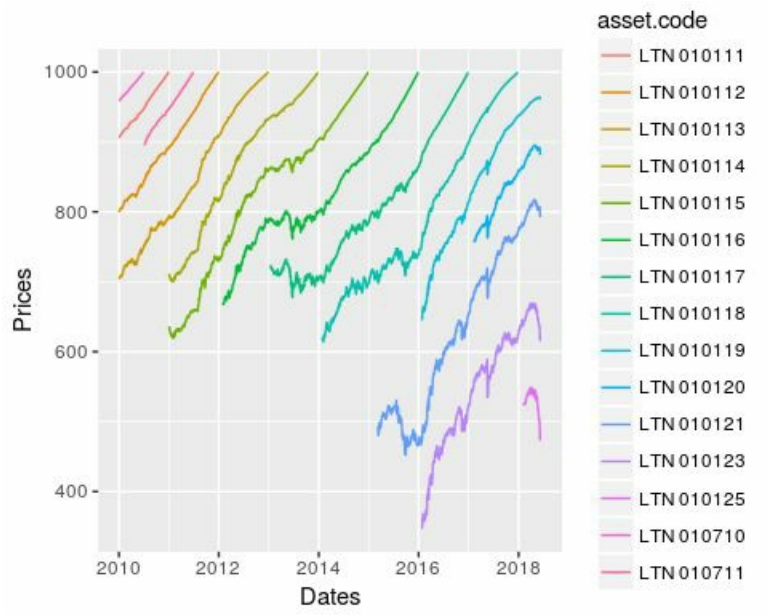
asset.codes <- 'LTN'   # Name of asset
maturity <- NULL      # = NULL, downloads all maturities

# download data
my.flag <- download.TD.data(asset.codes = asset.codes,
                            do.clean.up = F)

# reads data
df.TD <- read.TD.files(asset.codes = asset.codes,
                       maturity = maturity)

# remove data prior to 2010
df.TD <- dplyr::filter(df.TD,
                       ref.date >= as.Date('2010-01-01'))
```

Após a importação das informações, plotamos os preços dos diferentes ativos:



Note como todos contratos do tipo LTN terminam com valor 1000 R$ em sua data de expiração e possuem uma dinâmica linear de crescimento de preço ao longo do tempo.

Outra funcionalidade do pacote `GetTDData` é o acesso a curva de juros atual do sistema financeiro brasileiro diretamente do site da Anbima. Para isso, basta utilizar a função `get.yield.curve`:

```
library(GetTDData)

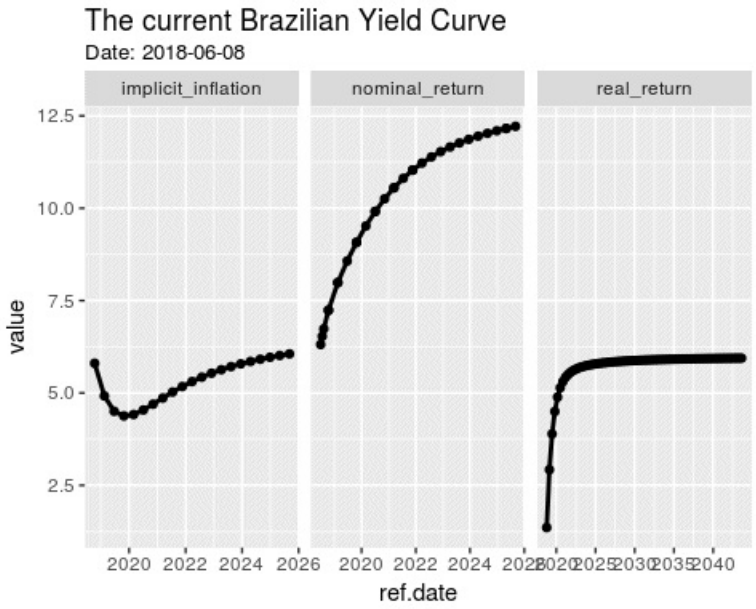
# get yield curve
df.yield <- get.yield.curve()

# check result
dplyr::glimpse(df.yield)
```

******ebook converter DEMO Watermarks*******

```
## Observations: 125
## Variables: 5
## $ n.biz.days   <dbl> 126, 252, 378, 504, 630, 756, 882,...
## $ type         <chr> "real_return", "real_return", "rea...
## $ value        <dbl> 1.3566, 2.9230, 3.8922, 4.4996, 4....
## $ ref.date     <date> 2018-10-12, 2019-02-15, 2019-06-2...
## $ current.date <date> 2018-06-08, 2018-06-08, 2018-06-0...
```

Os dados incluem a curva de juros nominal, juros real e da inflação. Para melhor visualizar as informações, vamos plotá-las em um gráfico:



Observamos um nível relativamente estável da taxa de juros real e uma expectativa de queda do juros nominal nos próximos meses. Esse código pode ser rodado a cada dia, mostrando a curva de juros atualizada.

5.2.8 `Rbitcoin`

Dada a recente popularidade das criptomoedas, outro pacote que vale a pena mencionar é `RBitcoin`. Esse permite o acesso a informações de transação do Bitcoin. Aqui, vamos mostrar um exemplo simples de importar dados da bolsa internacional `kraken`, usando o Euro como a moeda.

```
library(Rbitcoin)

# set mkt, currency pair and type of action
my.mkt <- "kraken"
my.currency <- c("BTC","EUR")
my.action <- 'trades'

# import data
my.l <- market.api.process(market = my.mkt,
                           currency_pair = my.currency,
                           action = my.action)
```

******ebook converter DEMO Watermarks*******

```
# check content
class(my.l)
```

```
## [1] "list"
```

A saída de `market.api.process` é um objeto do tipo lista com informações sobre Bitcoin no mercado `'kraken'`. As informações de negociações estão disponíveis no item `trade` de `my.l`. Vamos dar uma olhada no seu conteúdo:

```
dplyr::glimpse(my.l$trades)
```

```
## Observations: 1,000
## Variables: 5
## $ date   <dttm> 2018-06-10 17:49:04, 2018-06-10 17:49:0...
## $ price  <dbl> 5779.0, 5779.0, 5779.0, 5779.0, 5779.0, ...
## $ amount <dbl> 1.00000000, 0.03950122, 0.09959878, 0.28...
## $ tid    <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ...
## $ type   <chr> "ask", "ask", "ask", "ask", "ask", "ask"...
```

Essas incluem informações sobre preços e horários para os últimos 1000 negócios. O pacote também inclui funções para procurar o livro de ofertas de cada mercado e gerenciar carteiras Bitcoin. Mais detalhes sobre as funcionalidades do pacote são encontrados em seu site do Github.

5.2.9 Outros Pacotes

Nas seções anteriores destacamos os principais pacotes gratuitos para aquisição de dados financeiros e econômicos no Brasil. Esses representam uma pequena parcela da totalidade. No CRAN você encontrará muitos outros. A interface para fontes de dados comerciais também é possível. Várias empresas fornecem APIs para facilitar o envio de dados aos seus clientes. Pacotes como `Rblpapi` (Bloomberg), `IBrokers` (Interactive Brokers), `TFX` (TrueFX), `rdatastream` (Thomson Dataworks) fazem o R se comunicar com essas APIs e permitir a importação de conjuntos de dados. Se a empresa de fornecimento de dados que você usa no trabalho não for apresentada aqui, a lista de pacotes CRAN pode ajudá-lo a encontrar o pacote, caso existir.

******ebook converter DEMO Watermarks*******

5.3 Acessando Dados de Páginas na Internet (*Webscraping*)

Os pacotes destacados anteriormente são muito úteis pois facilitam a importação de dados específicos diretamente da internet. Em muitos casos, porém, os dados de interesse não estão disponíveis via API formal, mas sim em uma página na internet - geralmente no formato de uma tabela. O processo de extrair informações de páginas da internet chama-se *webscraping* (raspagem de dados). Dependendo da estrutura e da tecnologia da página da *web* acessada, importar essas informações diretamente para o R pode ser um procedimento trivial – mas também pode se tornar um processo extremamente trabalhoso.

Como exemplo, observe o caso da página do Tesouro Direto, figura **??**. Essa página mostra os preços e os retornos atuais dos diferentes títulos da dívida pública que os investidores do tipo pessoa física podem comprar. As informações da página são atualizadas diariamente. Portanto, como os valores refletem as marcações do mercado dos títulos, os dados dessa página podem ser utilizados para atualizar o valor de um portfólio com investimentos em títulos públicos.

A importação desses dados para o R é fácil. O pacote `XML` possui uma série de funções para ler e criar códigos em *xml* e *html*. Uma dessas funções é `readHTMLTable`, que lê tabelas de páginas da internet, retornando um objeto da classe de listas. Nesse objeto, cada elemento equivale a uma tabela encontrada na página. Veja o seu uso para o caso do Tesouro Direto a seguir:

```
library(XML)

# set url
my.url <-paste0('http://www.tesouro.fazenda.gov.br/',
                'tesouro-direto-precos-e-taxas-dos-titulos')

# read site
l.out <- readHTMLTable(my.url, encoding = 'UTF-8')

# find table in slot 2
df.out <- l.out[[2]]

# check contents
dplyr::glimpse(df.out)
```

```
## Observations: 14
## Variables: 5
## $ V1 <fct> Título, Indexados ao IPCA, Tesouro IPCA+ 202...
## $ V2 <fct> Vencimento, , 15/08/2024, 15/05/2035, 15/05/...
## $ V3 <fct> Taxa de Rendimento (% a.a.), NA, 5,73, 5,84,...
## $ V4 <fct> Valor Mínimo, NA, R$43,75, R$35,50, R$33,59,...
## $ V5 <fct> Preço Unitário, NA, R$2.187,53, R$1.183,49, ...
```

Observa-se que a tabela importada é exatamente igual à da página. Um lado negativo da importação, porém, é que as informações são importadas exatamente como são apresentadas no site. Por exemplo: os retornos (*yields*) dos títulos contêm o caractere *%* e são da classe fator. Além disso, os nomes das colunas da tabela importada devem ser adicionados manualmente. Uma etapa posterior à importação dos dados crus é a limpeza dos mesmos e a modificação das colunas para as classes corretas.

******ebook converter DEMO Watermarks*******

O uso de técnicas de *webscraping* se torna um forte aliado do pesquisador. Essas lhe dão acesso a uma imensa quantidade de informações disponíveis na web. No entanto, cada cenário de *webscraping* é particular. Não é sempre o caso que você pode importar dados de forma direta e fácil. Outro problema no *webscraping* é que o método de captura de dados não é estável. Qualquer mudança na estrutura da página irá quebrar o código. Na minha experiência, é bastante comum que códigos que funcionavam um ano atrás não funcionem mais pois a página de origem foi modificada. Isso requere mais tempo de desenvolvimento do programador. Àqueles leitores interessados nesse tópico, sugiro fortemente o estudo das funcionalidades dos pacotes `rvest` (Wickham 2016b), `XML` (Lang and CRAN Team 2016), `RSelenium` (Harrison 2016) e `splashr`, disponível no Github.

******ebook converter DEMO Watermarks*******

5.4 Exercícios

1. Utilizando pacote `BatchGetSymbols`, baixe os dados da ação MDIA3 no *Yahoo Finance* para os últimos 30 dias. Qual o preço ajustado mais baixo no período analisado? Qual a data com o preço mais baixo?

2. Caso não o tenha feito, crie um perfil no site do Quandl e baixe dados sobre preços de café arábica no banco de dados do CEPEA (Centro de Estudos Avançados em Economia Aplicada) para o último mês.

3. Utilize pacote `rbcb` para baixar dados do IPCA nos últimos 5 anos. É possível observar algum período com deflação (inflação negativa) mensal?

4. Visite o site de sistema de séries temporais do Banco Central do Brasil e escolha uma série para análise de acordo com seus próprios interesses de pesquisa. Importe a mesma usando pacote `bcb`.

5. Utilizando função `GetDFPData::gdfpd.get.info.companies`, baixe informações sobre as ações negociadas na B3. Quantas empresas estão atualmente disponíveis no banco de dados? Qual a proporção de empresas ativas? Qual a empresa mais antiga? Quantas empresas existem para o setor de Utilidade Pública?

6. Com pacote `GetDFPData`, baixe os demostrativos financeiros mais recentes da Petrobras. Qual foi o seu lucro líquido no exercício? Qual foi o pagamento de proventos para o acionista (dividendo ou JSCP) mais recente?

7. Com base no pacote `GetTDData`, baixe dados para títulos do tipo LTN (Letras Financeiras do Tesouro). Entre todas LTNs, qual o título com maior data de vencimento?

---

1. Empresas como kibot, pitrading e backtestmarket, vendem esses dados ao público.↩

2. O código CVM é o identificador único que a CVM (Comissão de Valores Mobiliários) define para cada empresa negociada em bolsa. Estas informações são usadas extensivamente por `GetDFPData` para identificar empresas em outros conjuntos de dados da B3 e da CVM.↩

******ebook converter DEMO Watermarks*******

6 Dataframes e outros Objetos

No R, tudo é um objeto. Os códigos que executamos nos capítulos anteriores, incluindo importação de dados, resultou em algum objeto criado, seja ele um texto, um dataframe ou lista. Cada tipo ou classe de objeto terá uma série de propriedades diferentes. Por exemplo, um dataframe pode ser incrementado com novas colunas ou linhas. Uma coluna numérica de um dataframe pode interagir com outros valores numéricos através de operações de multiplicação, divisão e soma. Para colunas com textos, porém, tal propriedade não é válida, uma vez que não faz sentido somar um valor numérico a um texto ou dividir um texto por outro. Entretanto, a classe de texto tem outras propriedades, como a que permite procurar uma determinada sequência textual dentro de um texto maior, a manipulação de partes do texto e a substituição de caracteres específicos, dentre tantas outras possibilidades. **Um dos aspectos mais importantes no trabalho com o R é o aprendizado das classes de objetos e as suas funcionalidades**.

As classes básicas de objetos no R inclui valores numéricos, caracteres (texto), fatores, datas, entre vários outros casos. Na prática, porém, as classes básicas são armazenadas em estruturas de dados mais complexas, tal como dataframes e listas. Isso organiza e facilita o trabalho. Imagine realizar um estudo sobre as *63* ações que compõem o índice Ibovespa, onde a base de dados é composta por preços e volumes negociados ao longo de um ano. Caso fôssemos criar um vetor numérico de preços e de volumes para cada ação, teríamos uma quantidade de *126* objetos para lidar no nosso *enviromnent*. Apesar de ser possível trabalhar dessa forma, o código resultante seria desorganizado, difícil de entender e passível de uma série de erros.

Uma maneira mais simples de organizar os nossos dados é criar um objeto com o nome `my.data` e alocar todos os preços e volumes ali. Todas as informações necessárias para executar a pesquisa estariam nesse objeto, facilitando a importação e exportação dos dados. Esses objetos que armazenam outros objetos de classe básica constituem a classe de estrutura de dados. Nessa classificação, estão incluídas listas, matrizes e dataframes.

******ebook converter DEMO Watermarks*******

6.1 `Dataframes`

O dataframe é o principal objeto utilizado no trabalho com o R e o mais importante de se estudar. Dados externos são importados para o R no formato de dataframes. É na manipulação desses que gastará maior parte do tempo realizando a sua análise. Traduzindo para o português, dataframe significa “estrutura ou organização de dados”. Grosso modo, um objeto da classe dataframe nada mais é do que uma tabela com linhas e colunas. Internamente, um dataframe é um tipo especial de lista, onde cada coluna é um vetor atômico com o mesmo número de elementos, porém com sua própria classe. Podemos organizar em um dataframe dados de texto juntamente com números, por exemplo.

Note que o formato tabular força a sincronização dos dados no sentido de linhas, isto é, cada caso de cada variável deve ser pareado com casos de outras variáveis. Apesar de simples, esse tipo de estruturação de dados é intuitiva e pode acomodar uma variedade de informações. Cada acréscimo de dados (informações) incrementa as linhas e cada novo tipo de informação incrementa as colunas da tabela.

Um dos pontos positivos na utilização do dataframe para a acomodação de dados é que diversas funções de diversos pacotes irão funcionar a partir dessa classe de objetos. Por exemplo, o pacote de manipulação de dados `dplyr`, assim como o pacote de criação de figuras `ggplot2`, funcionam a partir de um dataframe. Esse objeto, portanto, está no centro de uma série de funcionalidades do R e, sem dúvida, é uma classe de objeto extremamente importante para aprender a utilizar corretamente.

O objeto dataframe é uma das classes nativas do R e vem implementado no pacote `base`. Entretanto, o universe `tidyverse` oferece sua própria versão de um dataframe, chamada `tibble`, a qual é utilizada sistematicamente em todos pacotes do `tidyverse`. A conversão de um dataframe para `tibble` é interna e automática. O `tibble` possui propriedades mais flexíveis que dataframes nativos, facilitando de forma significativa o seu uso. Seguindo a nossa preferência para o `tidyverse`, a partir de agora iremos utilizar `tibbles` como representantes de dataframes.

6.1.1 Criando Dataframes

A criação de um dataframe do tipo `tibble` ocorre a partir da função `data_frame` ou `tibble`. Note que a criação de um dataframe nativo ocorre com a função `base::data.frame`, enquanto a criação do `tibble` parte da função `tibble::data_frame`, `tibble::tibble` ou `dplyr::data_frame`. Para manter o código mais limpo, iremos dar preferência a `dplyr::data_frame` e utilizar o nome *dataframe* para se referir a um `tibble`. Veja o exemplo a seguir, onde criamos um dataframe correspondente a dados financeiros de diferentes ações.

```
library(tibble)

# set tickers
ticker <- c(rep('ABEV3',4),
            rep('BBAS3', 4),
            rep('BBDC3', 4))
```

******ebook converter DEMO Watermarks*******

```
# set dates
ref.date <- as.Date(rep(c('2010-01-01', '2010-01-04',
                          '2010-01-05', '2010-01-06'),
                    3) )

# set prices
price <- c(736.67, 764.14, 768.63, 776.47,
           59.4  , 59.8  , 59.2  , 59.28,
           29.81 , 30.82 , 30.38 , 30.20)

# create tibble/dataframe
my.df <- data_frame(ticker, ref.date , price)

# print it
print(my.df)
```

```
## # A tibble: 12 x 3
##    ticker ref.date   price
##    <chr>  <date>     <dbl>
##  1 ABEV3  2010-01-01 737.
##  2 ABEV3  2010-01-04 764.
##  3 ABEV3  2010-01-05 769.
##  4 ABEV3  2010-01-06 776.
##  5 BBAS3  2010-01-01  59.4
##  6 BBAS3  2010-01-04  59.8
##  7 BBAS3  2010-01-05  59.2
##  8 BBAS3  2010-01-06  59.3
##  9 BBDC3  2010-01-01  29.8
## 10 BBDC3  2010-01-04  30.8
## 11 BBDC3  2010-01-05  30.4
## 12 BBDC3  2010-01-06  30.2
```

Observe que utilizamos a função `rep` para replicar e facilitar a criação dos dados do dataframe anterior. Assim, não é necessário repetir os valores múltiplas vezes. Destaca-se que, no uso dos dataframes, podemos salvar todos os nossos dados em um único objeto, facilitando o acesso e a organização do código resultante.

O uso da função `print` em um dataframe mostra a sua representação tabular na tela do R. O conteúdo do dataframe criado anteriormente também pode ser visualizado no RStudio. Para isso, basta clicar no nome do objeto na aba *environment*, canto superior direito da tela. Após isso, um visualizador aparecerá na tela principal do programa. Essa operação é nada mais que uma chamada a função `utils::View`. Portanto, poderíamos visualizar o dataframe anterior executando o comando `View(my.df)`.

6.1.2 Conhecendo um Dataframe

Após a criação do dataframe, o segundo passo é conhecer os seu conteúdo. Particularmente, é importante tomar conhecimento dos seguintes itens:

- Nomes e classes das Colunas;
- Número de linhas e colunas;
- Existência de dados omissos (`NA`).

É comum não termos controle sobre o nome original das colunas nos dados importados. No ******ebook converter DEMO Watermarks*******

momento de trabalhar com os dados, porém, é importante termos nomes de colunas que façam sentido e que sejam fáceis de acessar. Portanto, o primeiro passo em conhecer um dataframe é analisar os nomes das colunas e seus respectivos conteúdos. O número de linhas e colunas também deve ser consultado. Caso os valores forem diferentes do esperado, deve-se checar o arquivo de importação dos dados.

Nesse ponto, vale salientar que cada coluna de um dataframe tem sua própria classe. É de suma importância que as classes dos dados estejam corretamente especificadas. Caso contrário, operações futuras podem resultar em um erro. Por exemplo, caso um vetor de valores numéricos seja importado com a classe de texto (`character`), qualquer operação matemática nesse vetor irá resultar em um erro no R. **Identificar a classe das colunas logo após a importação é de suma importância**.

Uma das funções mais recomendadas para se familiarizar com um dataframe é `dplyr::glimpse`. Essa mostra na tela o nome e a classe das colunas, além do número de linhas/colunas. Abusamos dessa função nos capítulos anteriores. Veja um exemplo simples a seguir:

```
library(dplyr)

# check content of my.df
glimpse(my.df)
```

```
## Observations: 12
## Variables: 3
## $ ticker   <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3", "B...
## $ ref.date <date> 2010-01-01, 2010-01-04, 2010-01-05, 2...
## $ price    <dbl> 736.67, 764.14, 768.63, 776.47, 59.40,...
```

Em muitas situações, o uso de `glimpse` é suficiente para entender se o processo de importação de dados ocorreu de forma satisfatória. Porém, uma análise mais profunda é entender qual a variação de cada coluna nos dados importados. Aqui entra o papel da função `base::summary`:

```
# check variation my.df
summary(my.df)
```

```
##     ticker            ref.date             price
##  Length:12          Min.   :2010-01-01   Min.   : 29.81
##  Class :character   1st Qu.:2010-01-03   1st Qu.: 30.71
##  Mode  :character   Median :2010-01-04   Median : 59.34
##                     Mean   :2010-01-04   Mean   :283.73
##                     3rd Qu.:2010-01-05   3rd Qu.:743.54
##                     Max.   :2010-01-06   Max.   :776.47
```

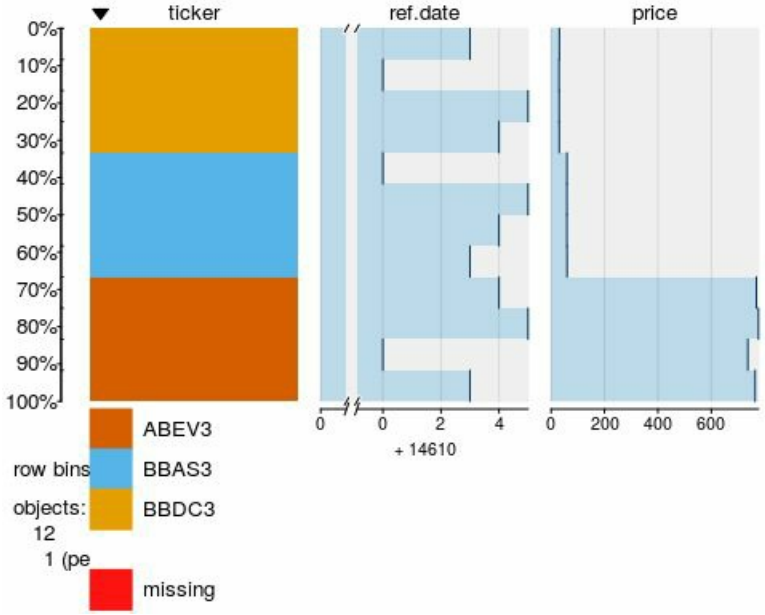
Note que `summary` interpreta cada coluna de forma diferente. Para o primeiro caso, coluna `ticker`, mostra apenas o tamanho do vetor. No caso de datas e valores numéricos, essa apresenta o máximo, mínimo, mediana e quartis. Por exemplo, uma observação extrema (*outlier*) poderia ser facilmente identificada na análise da saída de `summary`.

Outra forma interessante de visualizar o conteúdo de um dataframe é com a função `tabplot`, oferecida pelo pacote de mesmo nome. Esse não é parte do `tidyverse` e deve ser instalado antes do uso. Veja o exemplo a seguir:

******ebook converter DEMO Watermarks*******

```
library(tabplot)

# check df with tableplot
tableplot(my.df)
```



A saída na forma de uma figura mostra a distribuição dos dados ao longo do dataframe. É possível ver claramente como o dataframe está dividido entre três ativos. O nosso exemplo não possui casos de dados faltantes (`NA`), porém o uso de `tabplot` também acusaria esses na cor vermelha.

6.1.3 Operador de *pipeline* (`%>%`)

Uma característica importante do universo `tidyverse` é o uso extensivo do operador de *pipeline*, primeiro proposto por Bache and Wickham (2014) e definido pelo símbolo `%>%`. Esse "comando" permite que operações de dados sejam realizadas de forma sequencial e modular, como em uma *tubulação*, facilitando a otimização e legibilidade do código resultante.

Imagine uma situação onde temos três funções para aplicar nos dados salvos em um dataframe. Cada função depende da saída de outra função. Isso requer o encadeamento de suas chamadas. Usando o operador de *pipeline*, podemos escrever o procedimento de manipulação `dataframe` com o seguinte código:

```
my.tab <- my.df %>%
  fct1(arg1) %>%
  fct2(arg2) %>%
  fct3(arg3)
```

Usamos símbolo `%>%` no final de cada linha para vincular as operações. As funções `fct*` são operações realizadas em cada etapa. O resultado de cada linha é passado para a próxima função de forma sequencial. Assim, não há necessidade de criar objetos intermediários. Veja a seguir duas formas de realizar a mesma operação sem o operador de *pipeline*:

******ebook converter DEMO Watermarks*******

```
# version 1
my.tab <- fct3(fct2(fct1(my.df,
                         arg1),
                    arg2),
               arg1)

# version 2
temp1 <- fct1(my.df, arg1)
temp2 <- fct2(temp1, arg2)
my.tab <- fct3(temp1, arg3)
```

Observe como as alternativas formam um código com estrutura estranha e passível a erros. Provavelmente não deves ter notado, mas ambos os códigos possuem erros de digitação. Para o primeiro, o último `arg1` deveria ser `arg3` e, no segundo, a função `fct3` está usando o dataframe `temp1` e não `temp2`. Este exemplo deixa claro como o uso de *pipelines* torna o código mais elegante e legível. A partir de agora iremos utilizar o operador `%>%` de forma extensiva.

6.1.4 Acessando Colunas

Um objeto do tipo `dataframe` utiliza-se de diversos comandos e símbolos que também são usados em matrizes e listas. Para descobrir os nomes das colunas de um dataframe, temos duas funções: `names` ou `colnames`:

```
# check names of df
names(my.df)

## [1] "ticker"   "ref.date" "price"

colnames(my.df)

## [1] "ticker"   "ref.date" "price"
```

Ambas também podem ser usadas para modificar os nomes das colunas:

```
# set temp df
temp.df <- my.df

# check names
names(temp.df)

## [1] "ticker"   "ref.date" "price"

# change names
names(temp.df) <- paste('Col', 1:ncol(temp.df))

# check names
names(temp.df)

## [1] "Col 1" "Col 2" "Col 3"
```

Destaca-se que a forma de usar `names` é bastante distinta das demais funções do R. Nesse caso, utilizamos a função ao lado esquerdo do símbolo de `assign` (`<-`). Internamente, o que estamos fazendo é definindo um atributo do objeto `temp.df`, o nome de suas colunas.

Para acessar uma determinada coluna, podemos utilizar o nome da mesma com o operador `$`, ou

******ebook converter DEMO Watermarks*******

pela sua posição na tabela:

```
# isolate columns of df
my.ticker <- my.df$ticker
my.prices <- my.df['price']
my.date <- my.df[ ,2]

# print contents
print(my.ticker)
```

```
##  [1] "ABEV3" "ABEV3" "ABEV3" "ABEV3" "BBAS3" "BBAS3" "BBAS3"
##  [8] "BBAS3" "BBDC3" "BBDC3" "BBDC3" "BBDC3"
```

```
print(my.prices)
```

```
## # A tibble: 12 x 1
##    price
##    <dbl>
##  1 737.
##  2 764.
##  3 769.
##  4 776.
##  5  59.4
##  6  59.8
##  7  59.2
##  8  59.3
##  9  29.8
## 10  30.8
## 11  30.4
## 12  30.2
```

```
print(my.date)
```

```
## # A tibble: 12 x 1
##    ref.date
##    <date>
##  1 2010-01-01
##  2 2010-01-04
##  3 2010-01-05
##  4 2010-01-06
##  5 2010-01-01
##  6 2010-01-04
##  7 2010-01-05
##  8 2010-01-06
##  9 2010-01-01
## 10 2010-01-04
## 11 2010-01-05
## 12 2010-01-06
```

Vale apontar que, no R, **um objeto da classe dataframe é representado internamente como uma lista**, onde cada elemento é uma coluna. Isso é importante saber, pois alguns comandos de listas também funcionam para dataframes. Um exemplo é o uso de duplo colchetes (`[[]]`) para selecionar colunas:

```
print(my.df[[2]])
```

```
##  [1] "2010-01-01" "2010-01-04" "2010-01-05" "2010-01-06"
##  [5] "2010-01-01" "2010-01-04" "2010-01-05" "2010-01-06"
```

******ebook converter DEMO Watermarks*******

```
##  [9] "2010-01-01" "2010-01-04" "2010-01-05" "2010-01-06"
```

Para acessar linhas e colunas específicas de um dataframe, basta utilizar colchetes simples:

```
print(my.df[1:5,2])
```

```
## # A tibble: 5 x 1
##   ref.date
##   <date>
## 1 2010-01-01
## 2 2010-01-04
## 3 2010-01-05
## 4 2010-01-06
## 5 2010-01-01
```

```
print(my.df[1:5,c(1,2)])
```

```
## # A tibble: 5 x 2
##   ticker ref.date
##   <chr>  <date>
## 1 ABEV3  2010-01-01
## 2 ABEV3  2010-01-04
## 3 ABEV3  2010-01-05
## 4 ABEV3  2010-01-06
## 5 BBAS3  2010-01-01
```

```
print(my.df[1:5, ])
```

```
## # A tibble: 5 x 3
##   ticker ref.date   price
##   <chr>  <date>     <dbl>
## 1 ABEV3  2010-01-01 737.
## 2 ABEV3  2010-01-04 764.
## 3 ABEV3  2010-01-05 769.
## 4 ABEV3  2010-01-06 776.
## 5 BBAS3  2010-01-01  59.4
```

Essa seleção de colunas também pode ser realizada utilizando o nome das mesmas da seguinte forma:

```
print(my.df[1:3, c('ticker','price')])
```

```
## # A tibble: 3 x 2
##   ticker price
##   <chr>  <dbl>
## 1 ABEV3   737.
## 2 ABEV3   764.
## 3 ABEV3   769.
```

ou, pelo operador de *pipeline* e a função `dplyr::select`:

```
library(dplyr)

my.temp <- my.df %>%
  select(ticker, price) %>%
  glimpse()
```

```
## Observations: 12
## Variables: 2
```

******ebook converter DEMO Watermarks*******

```
## $ ticker <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3", "BBA...
## $ price  <dbl> 736.67, 764.14, 768.63, 776.47, 59.40, 5...
```

6.1.5 Modificando um Dataframe

Para criar novas colunas em um dataframe, basta utilizar a função `mutate`. Aqui iremos abusar do operador de *pipeline* (`%>%`). Veja a seguir:

```
library(dplyr)

# add columns with mutate
my.df <- my.df %>%
  mutate(ret = price/lag(price) -1,
         my.seq1 = 1:nrow(my.df),
         my.seq2 =  my.seq1 +9) %>%
  glimpse()
```

```
## Observations: 12
## Variables: 6
## $ ticker   <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3", "B...
## $ ref.date <date> 2010-01-01, 2010-01-04, 2010-01-05, 2...
## $ price    <dbl> 736.67, 764.14, 768.63, 776.47, 59.40,...
## $ ret      <dbl> NA, 0.037289424, 0.005875887, 0.010199...
## $ my.seq1  <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
## $ my.seq2  <dbl> 10, 11, 12, 13, 14, 15, 16, 17, 18, 19...
```

Note que precisamos indicar o dataframe de origem dos dados, nesse caso o objeto `my.df`, e as colunas são definidas como argumentos em `dplyr::mutate`. Observe também que usamos a coluna `price` na construção de `ret`, o retorno aritmético dos preços. Um caso especial é a construção de `my.seq2` com base em `my.seq1`, isto é, antes mesmo dela ser explicitamente calculada já é possível utilizar a nova coluna para criar outra.

A maneira mais tradicional, e comumente encontrada em código, para criar novas colunas é utilizar o símbolo `$`:

```
# add new column with base R
my.df$my.seq3 <- 1:nrow(my.df)

# check it
glimpse(my.df)
```

```
## Observations: 12
## Variables: 7
## $ ticker   <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3", "B...
## $ ref.date <date> 2010-01-01, 2010-01-04, 2010-01-05, 2...
## $ price    <dbl> 736.67, 764.14, 768.63, 776.47, 59.40,...
## $ ret      <dbl> NA, 0.037289424, 0.005875887, 0.010199...
## $ my.seq1  <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
## $ my.seq2  <dbl> 10, 11, 12, 13, 14, 15, 16, 17, 18, 19...
## $ my.seq3  <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

Portanto, o operador `$` vale tanto para acessar quanto para criar novas colunas.

Se adicionarmos uma coluna com número de elementos diferente do número de linhas, o R emitirá uma mensagem de erro:

******ebook converter DEMO Watermarks*******

```
my.df <- my.df %>%
  mutate(my.seq3 =  1:100) %>%
  glimpse()
```

```
## Error in mutate_impl(.data, dots): Column `my.seq3` must be length 12 (the numl
```

Para remover colunas de um dataframe, basta usar `dplyr::select` com operador negativo para o nome das colunas indesejadas:

```
# removing columns
 my.df.temp <- my.df %>%
  select(-my.seq1, -my.seq2, -my.seq3) %>%
  glimpse()
```

```
## Observations: 12
## Variables: 4
## $ ticker   <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3", "B...
## $ ref.date <date> 2010-01-01, 2010-01-04, 2010-01-05, 2...
## $ price    <dbl> 736.67, 764.14, 768.63, 776.47, 59.40,...
## $ ret      <dbl> NA, 0.037289424, 0.005875887, 0.010199...
```

No uso de funções nativas do R, a maneira tradicional de remover colunas é alocar o valor nulo `NULL`:

```
# set temp df
temp.df <- my.df

# remove cols
temp.df$price <- NULL
temp.df$ref.date  <- NULL

# check it
glimpse(temp.df)
```

```
## Observations: 12
## Variables: 5
## $ ticker  <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3", "BB...
## $ ret     <dbl> NA, 0.037289424, 0.005875887, 0.0101999...
## $ my.seq1 <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
## $ my.seq2 <dbl> 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,...
## $ my.seq3 <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

6.1.6 Filtrando um Dataframe

Uma operação bastante comum no R é filtrar linhas de um dataframe de acordo com uma ou mais condições. Por exemplo, caso quiséssemos apenas os dados da ação `ABEV3`, poderíamos utilizar a função `filter` para filtrar a tabela:

```
library(dplyr)

# filter df for single stock
my.df.temp <- my.df %>%
  filter(ticker == 'ABEV3') %>%
  glimpse()
```

```
## Observations: 4
```

******ebook converter DEMO Watermarks*******

```
## Variables: 7
## $ ticker   <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3"
## $ ref.date <date> 2010-01-01, 2010-01-04, 2010-01-05, 2...
## $ price    <dbl> 736.67, 764.14, 768.63, 776.47
## $ ret      <dbl> NA, 0.037289424, 0.005875887, 0.010199966
## $ my.seq1  <int> 1, 2, 3, 4
## $ my.seq2  <dbl> 10, 11, 12, 13
## $ my.seq3  <int> 1, 2, 3, 4
```

A função também aceita mais de uma condição. Veja a seguir onde filtramos os dados para `'ABEV3'` em datas após ou igual a `'2010-01-05'`:

```
library(dplyr)
# filter df for single stock and date
my.df.temp <- my.df %>%
  filter(ticker == 'ABEV3',
         ref.date >= as.Date('2010-01-05')) %>%
  glimpse()
```

```
## Observations: 2
## Variables: 7
## $ ticker   <chr> "ABEV3", "ABEV3"
## $ ref.date <date> 2010-01-05, 2010-01-06
## $ price    <dbl> 768.63, 776.47
## $ ret      <dbl> 0.005875887, 0.010199966
## $ my.seq1  <int> 3, 4
## $ my.seq2  <dbl> 12, 13
## $ my.seq3  <int> 3, 4
```

Aqui utilizamos o símbolo == para testar uma igualdade. Iremos estudar mais profundamente a classe de testes lógicos no capítulo 7.

6.1.7 Ordenando um Dataframe

Após a criação ou importação de um dataframe, pode-se ordenar seus componentes de acordo com os valores de alguma coluna. Um caso bastante comum em que é necessário realizar uma ordenação explícita é quando importamos dados financeiros em que as datas não estão em ordem crescente. Na grande maioria das situações, dados temporais devem estar ordenados de acordo com a antiguidade, isto é, dados mais recentes são alocados na última linha da tabela. Essa operação é realizada através do uso da função `base::order` ou `dplyr::arrange`.

Como exemplo, considere a criação de um dataframe com os valores a seguir:

```
# set df
my.df <- data_frame(col1 = c(4,1,2),
                    col2=c(1,1,3),
                    col3= c('a','b','c'))

# print it
print(my.df)
```

```
## # A tibble: 3 x 3
##    col1  col2 col3
##   <dbl> <dbl> <chr>
## 1     4     1 a
```

******ebook converter DEMO Watermarks*******

```
## 2     1     1 b
## 3     2     3 c
```

A função `order` retorna os índices relativos à ordenação dos valores dados como entrada. Para o caso da primeira coluna de `my.df`, os índices dos elementos formadores do novo vetor, com seus valores ordenados em forma crescente, são:

```
idx <- order(my.df$col1)
print(idx)
```

```
## [1] 2 3 1
```

Portanto, ao utilizar a saída da função `order` como indexador do dataframe, acaba-se ordenando o mesmo de acordo com os valores da coluna `col1`. Veja a seguir:

```
my.df.2 <- my.df[order(my.df$col1), ]
print(my.df.2)
```

```
## # A tibble: 3 x 3
##    col1  col2 col3
##   <dbl> <dbl> <chr>
## 1     1     1 b
## 2     2     3 c
## 3     4     1 a
```

Essa operação de ordenamento também pode ser realizada levando em conta mais de uma coluna. Veja o exemplo a seguir, onde se ordena o dataframe pelas colunas `col2` e `col1`.

```
idx <- order(my.df$col2, my.df$col1)
my.df.3 <- my.df[idx, ]
print(my.df.3)
```

```
## # A tibble: 3 x 3
##    col1  col2 col3
##   <dbl> <dbl> <chr>
## 1     1     1 b
## 2     4     1 a
## 3     2     3 c
```

No `tidyverse`, a forma de ordenar dataframes é pelo uso da função `arrange`. No caso de ordenamento crescente, encapsulamos o nome das colunas com `desc`:

```
# sort ascending, by col1 and col2
my.df <- my.df %>%
  arrange(col1, col2) %>%
  print()
```

```
## # A tibble: 3 x 3
##    col1  col2 col3
##   <dbl> <dbl> <chr>
## 1     1     1 b
## 2     2     3 c
## 3     4     1 a
```

```
# sort descending, col1 and col2
my.df <- my.df %>%
  arrange(desc(col1), desc(col2)) %>%
```

******ebook converter DEMO Watermarks*******

```
  print()
```

```
## # A tibble: 3 x 3
##    col1  col2 col3
##   <dbl> <dbl> <chr>
## 1     4     1 a
## 2     2     3 c
## 3     1     1 b
```

O resultado prático no uso de `arrange` é o mesmo de `order`. Um dos seus benefícios é a possibilidade de encadeamento de operações através do uso do *pipeline*.

6.1.8 Combinando e Agregando Dataframes

Em muitas situações de análise de dados será necessário juntar dataframes distintos em um único objeto. Tabelas diferentes são importadas no R e, antes de analisar os dados, precisamos combinar as informações em um único objeto. Nos casos mais simples, onde as tabelas a serem agregadas possuem o mesmo formato, nós as juntamos de acordo com as linhas, verticalmente, ou colunas, horizontalmente. Para esse fim, temos as funções `dplyr::bind_rows` e `dlyr::bind_cols` no tidyverse e `base::rbind` e `base::cbind` nas funções nativas do R. Observe o exemplo a seguir.

```
library(dplyr)

# set dfs
my.df.1 <- data_frame(col1 = 1:5,
                      col2 = rep('a', 5))

my.df.2 <- data_frame(col1 = 6:10,
                      col2 = rep('b', 5),
                      col3 = rep('c', 5))

# bind by row
my.df <- bind_rows(my.df.1, my.df.2) %>%
  glimpse()
```

```
## Observations: 10
## Variables: 3
## $ col1 <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
## $ col2 <chr> "a", "a", "a", "a", "a", "b", "b", "b", "b...
## $ col3 <chr> NA, NA, NA, NA, NA, "c", "c", "c", "c", "c"
```

Note que, no exemplo anterior, os nomes das colunas são os mesmos. De fato, a função `dplyr::bind_rows` procura os nomes iguais em ambos os objetos para fazer a junção dos dataframes corretamente. As colunas que não ocorrem em ambos objetos, tal como `col3` no exemplo, saem como `NA` no objeto final. Já para o caso de `bind_cols`, os nomes das colunas devem ser diferentes, porém o número de linhas deve ser o mesmo.

```
# set dfs
my.df.1 <- data_frame(col1 = 1:5, col2 = rep('a', 5))
my.df.2 <- data_frame(col3 = 6:10, col4 = rep('b', 5))

# bind by column
my.df <- bind_cols(my.df.1, my.df.2) %>%
```

******ebook converter DEMO Watermarks*******

```
  glimpse()
```

```
## Observations: 5
## Variables: 4
## $ col1 <int> 1, 2, 3, 4, 5
## $ col2 <chr> "a", "a", "a", "a", "a"
## $ col3 <int> 6, 7, 8, 9, 10
## $ col4 <chr> "b", "b", "b", "b", "b"
```

Para casos mais complexos, onde a junção deve ser realizada de acordo com algum índice tal como uma data, é possível juntar dataframes diferentes com o uso das funções da família `dplyr::join*` tal como `dplyr::inner_join` `dplyr::left_join`, `dplyr::full_join`, entre outras. A descrição de todas elas não cabe aqui. Iremos descrever apenas o caso mais provável, `inner_join`. Essa combina os dados, mantendo apenas os casos onde existe o índice em ambos.

```
# set df
my.df.1 <- data_frame(date = as.Date('2016-01-01')+0:10,
                      x = 1:11)

my.df.2 <- data_frame(date = as.Date('2016-01-05')+0:10,
                      y = seq(20,30, length.out = 11))
```

Note que os dataframes criados possuem uma coluna em comum, `date`. A partir desta coluna que agregamos as tabelas com `inner_join`:

```
# aggregate tables
my.df <- inner_join(my.df.1, my.df.2)

glimpse(my.df)
```

```
## Observations: 7
## Variables: 3
## $ date <date> 2016-01-05, 2016-01-06, 2016-01-07, 2016-...
## $ x    <int> 5, 6, 7, 8, 9, 10, 11
## $ y    <dbl> 20, 21, 22, 23, 24, 25, 26
```

O R automaticamente verifica a existência de colunas com mesmo nome nos dataframes e realiza a junção por essas. Caso quiséssemos juntar dataframes onde os nomes das colunas para utilizar o índice não são iguais, temos duas soluções: modificar os nomes das colunas ou então utilizar argumento `by` em `dplyr::inner_join`. Veja a seguir:

```
# set df
my.df.3 <- data_frame(ref.date = as.Date('2016-01-01')+0:10,
                      x = 1:11)

my.df.4 <- data_frame(my.date = as.Date('2016-01-05')+0:10,
                      y = seq(20,30, length.out = 11))

# join by my.df.3$ref.date and my.df.4$my.date
my.df <- inner_join(my.df.3, my.df.4,
                    by = c('ref.date' = 'my.date'))

glimpse(my.df)
```

```
## Observations: 7
## Variables: 3
```

******ebook converter DEMO Watermarks*******

```
## $ ref.date <date> 2016-01-05, 2016-01-06, 2016-01-07, 2...
## $ x        <int> 5, 6, 7, 8, 9, 10, 11
## $ y        <dbl> 20, 21, 22, 23, 24, 25, 26
```

Para o caso de uso da função nativa de agregação de dataframes, `base::merge`, temos que indicar explicitamente o nome da coluna com argumento `by`:

```
# aggregation with base R
my.df <- merge(my.df.1, my.df.2, by = 'date')

glimpse(my.df)
```

```
## Observations: 7
## Variables: 3
## $ date <date> 2016-01-05, 2016-01-06, 2016-01-07, 2016-...
## $ x    <int> 5, 6, 7, 8, 9, 10, 11
## $ y    <dbl> 20, 21, 22, 23, 24, 25, 26
```

Note que, nesse caso, o dataframe resultante manteve apenas as informações compartilhadas entre ambos os objetos, isto é, aquelas linhas onde as datas em `date` eram iguais. Esse é o mesmo resultado quando no uso do `dplyr::inner_join`.

6.1.9 Extensões ao Dataframe

Como já foi relatado nos capítulos anteriores, um dos grandes benefícios no uso do R é a existência de pacotes para lidar com os problemas específicos dos usuários. Esse também é o caso da estrutura de dados. Enquanto um objeto tabular do tipo dataframe é uma boa solução para a maioria dos casos, às vezes pode fazer mais sentido armazenar os dados em algum tipo específico de objeto customizado. Ao longo do tempo, diversas soluções foram disponibilizadas por desenvolvedores.

Por exemplo, é muito comum trabalharmos com dados exclusivamente numéricos que são indexados ao tempo. Isto é, situações onde cada informação pertence a um índice temporal - um objeto da classe data/tempo. As linhas dessa tabela representam um ponto no tempo, enquanto as colunas indicam variáveis numéricas de interesse. Nesse caso, faria sentido representarmos os nossos dados como objetos do tipo `xts` (Ryan and Ulrich 2014). O grande benefício dessa opção é que a agregação e a manipulação de variáveis em função do tempo é muito fácil. Por exemplo, podemos transformar dados de frequência diária para a frequência semanal com apenas uma linha de comando. Além disso, diversas outras funções reconhecem automaticamente que os dados são indexados ao tempo. Um exemplo é a criação de uma figura com esses dados. Nesse caso, o eixo horizontal da figura é automaticamente organizado com as datas.

Veja um caso a seguir, onde carregamos os dados anteriores como um objeto `xts`:

```
library(xts)

# set data
ticker <- c('ABEV3', 'BBAS3','BBDC3')

date <- as.Date(c('2010-01-01', '2010-01-04',
                  '2010-01-05', '2010-01-06'))
```

******ebook converter DEMO Watermarks*******

```
price.ABEV3 <- c(736.67, 764.14, 768.63, 776.47)
price.BBAS3 <- c(59.4, 59.8, 59.2, 59.28)
price.BBDC3 <- c(29.81, 30.82, 30.38, 30.20)

# build matrix
my.mat <- matrix(c(price.BBDC3, price.BBAS3, price.ABEV3),
                 nrow = length(date) )

# set xts object
my.xts <- xts(my.mat,
              order.by = date)

# set correct colnames
colnames(my.xts) <- ticker

# check it!
print(my.xts)
```

```
##            ABEV3 BBAS3  BBDC3
## 2010-01-01 29.81 59.40 736.67
## 2010-01-04 30.82 59.80 764.14
## 2010-01-05 30.38 59.20 768.63
## 2010-01-06 30.20 59.28 776.47
```

O código anterior pode dar a impressão de que o objeto `my.xts` é semelhante a um dataframe, porém, não se engane. Por estar indexado a um vetor de tempo, objeto `xts` pode ser utilizado para uma série de procedimentos temporais, tal como uma agregação por período temporal. Veja o exemplo a seguir, onde agregamos duas variáveis de tempo através do cálculo de uma média a cada semana.

```
N <- 500

my.mat <- matrix(c(seq(1, N), seq(N, 1)), nrow=N)

my.xts <- xts(my.mat, order.by = as.Date('2016-01-01')+1:N)

my.xts.weekly.mean <- apply.weekly(my.xts, mean)

print(head(my.xts.weekly.mean))
```

```
##             X.1   X.2
## 2016-01-03  1.5 499.5
## 2016-01-10  6.0 495.0
## 2016-01-17 13.0 488.0
## 2016-01-24 20.0 481.0
## 2016-01-31 27.0 474.0
## 2016-02-07 34.0 467.0
```

Em Finanças e Economia, as agregações com objetos `xts` são extremamente úteis quando se trabalha com dados em frequências de tempo diferentes. Por exemplo, é muito comum que se agregue dados de transação no mercado financeiro em alta frequência para intervalos maiores. Esse tipo de procedimento é facilmente realizado no R através da correta representação dos dados como objetos `xts`. Existem diversas outras funcionalidades desse pacote. Encorajo os usuários a ler o manual e aprender o que pode ser feito.

Existem diversos outros tipos de dataframes customizados. Por exemplo, o dataframe proposto

******ebook converter DEMO Watermarks*******

pelo pacote `data.table` (Dowle et al. 2015) prioriza o tempo de operação nos dados e o uso de uma notação compacta para acesso e processamento. O `tibbletime` (Vaughan and Dancho 2018) é uma versão orientada pelo tempo para `tibbles`. Caso o usuário esteja necessitando realizar operações de agregação de tempo, o uso deste pacote é fortemente recomendado.

6.1.10 Outras Funções Úteis

head - Retorna os primeiros `n` elementos de um dataframe.

```
my.df <- data_frame(col1 = 1:5000, col2 = rep('a', 5000))
head(my.df, 5)
```

```
## # A tibble: 5 x 2
##    col1 col2
##   <int> <chr>
## 1     1 a
## 2     2 a
## 3     3 a
## 4     4 a
## 5     5 a
```

tail - Retorna os últimos `n` elementos de um dataframe.

```
tail(my.df, 5)
```

```
## # A tibble: 5 x 2
##    col1 col2
##   <int> <chr>
## 1  4996 a
## 2  4997 a
## 3  4998 a
## 4  4999 a
## 5  5000 a
```

complete.cases - Retorna um vetor lógico que testa se as linhas contêm apenas valores existentes e nenhum `NA`.

```
my.x <- c(1:5, NA, 10)
my.y <- c(5:10, NA)
my.df <- data_frame(my.x, my.y)

print(my.df)
```

```
## # A tibble: 7 x 2
##    my.x  my.y
##   <dbl> <int>
## 1     1     5
## 2     2     6
## 3     3     7
## 4     4     8
## 5     5     9
## 6    NA    10
## 7    10    NA
```

```
print(complete.cases(my.df))
```

******ebook converter DEMO Watermarks*******

```
## [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
```

```
print(which(!complete.cases(my.df)))
```

```
## [1] 6 7
```

na.omit - Retorna um dataframe sem as linhas onde valores `NA` são encontrados.

```
print(na.omit(my.df))
```

```
## # A tibble: 5 x 2
##    my.x  my.y
##   <dbl> <int>
## 1     1     5
## 2     2     6
## 3     3     7
## 4     4     8
## 5     5     9
```

unique - Retorna um dataframe onde todas as linhas duplicadas são eliminadas e somente os casos únicos são mantidos.

```
my.df <- data_frame(col1 = c(1,1,2,3,3,4,5),
                    col2 = c('A','A','A','C','C','B','D'))
```

```
print(my.df)
```

```
## # A tibble: 7 x 2
##    col1 col2
##   <dbl> <chr>
## 1     1 A
## 2     1 A
## 3     2 A
## 4     3 C
## 5     3 C
## 6     4 B
## 7     5 D
```

```
print(unique(my.df))
```

```
## # A tibble: 5 x 2
##    col1 col2
##   <dbl> <chr>
## 1     1 A
## 2     2 A
## 3     3 C
## 4     4 B
## 5     5 D
```

******ebook converter DEMO Watermarks*******

6.2 Listas

Uma lista é uma classe de objeto extremamente flexível. Trata-se de um recipiente expansível onde outros objetos podem ser alocados. Ao contrário de vetores atômicos, a lista não apresenta restrição alguma em relação aos tipos de elementos nela contidos. Podemos agrupar valores numéricos com caracteres, fatores com datas e até mesmo listas dentro de listas. Quando agrupamos vetores, também não é necessário que os mesmos tenham um número igual de elementos. Além disso, podemos dar um nome a cada elemento. Essas propriedades fazem da lista o objeto mais flexível para o armazenamento e estruturação de dados no R. Não é acidental o fato de que grande parte das funções do R retorna um objeto do tipo lista.

6.2.1 Criando Listas

Uma lista pode ser criada através do comando `base::list`, seguido por seus elementos separados por vírgula:

```
library(dplyr)

# create list
my.l <- list(c(1, 2, 3),
             c('a', 'b'),
             factor('A', 'B', 'C'),
             data_frame(col1 = 1:5))

# use base::print
print(my.l)
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "a" "b"
##
## [[3]]
## [1] <NA>
## Levels: C
##
## [[4]]
## # A tibble: 5 x 1
##    col1
##   <int>
## 1     1
## 2     2
## 3     3
## 4     4
## 5     5
```

```
# use dplyr::glimpse
glimpse(my.l)
```

```
## List of 4
##  $ : num [1:3] 1 2 3
##  $ : chr [1:2] "a" "b"
```

******ebook converter DEMO Watermarks*******

```
##  $ : Factor w/ 1 level "C": NA
##  $ :Classes 'tbl_df', 'tbl' and 'data.frame':    5 obs. of  1 variable:
##   ..$ col1: int [1:5] 1 2 3 4 5
```

Note que juntamos no mesmo objeto um vetor atômico numérico, outro de texto, um fator e um dataframe. A apresentação de listas com o comando `print` é diferente dos casos anteriores. Os elementos são separados verticalmente e os seus índices aparecem com duplo colchete (`[[ ]]`). Conforme será explicado logo a seguir, é dessa forma que os elementos de uma lista são armazenados e acessados.

Assim como para os demais tipos de objeto, os elementos de uma lista também podem ter nomes, o que facilita o entendimento e a interpretação das informações do problema em análise. Por exemplo, considere o caso de uma base de dados com informações sobre determinada ação negociada na bolsa. Nesse caso, podemos definir uma lista como:

```
# set named list
my.named.l <- list(ticker = 'TICK4',
                   market = 'Bovespa',
                   df.prices = data.frame(P = c(1,1.5,2,2.3),
                                          ref.date = Sys.Date()+0:3))

# check content
glimpse(my.named.l)
```

```
## List of 3
##  $ ticker   : chr "TICK4"
##  $ market   : chr "Bovespa"
##  $ df.prices:'data.frame':   4 obs. of  2 variables:
##   ..$ P       : num [1:4] 1 1.5 2 2.3
##   ..$ ref.date: Date[1:4], format: "2018-06-10" ...
```

A definição de nomes para os elementos de uma lista facilita o acesso aos dados, evitando possíveis erros no código.

6.2.2 Acessando os Elementos de uma Lista

Os elementos de uma lista podem ser acessados através do uso de duplo colchete (`[[ ]]`), tal como em:

```
# accessing elements from list
print(my.named.l[[2]])
```

```
## [1] "Bovespa"
```

```
print(my.named.l[[3]])
```

```
##     P   ref.date
## 1 1.0 2018-06-10
## 2 1.5 2018-06-11
## 3 2.0 2018-06-12
## 4 2.3 2018-06-13
```

Também é possível acessar os elementos com um colchete simples (`[ ]`), porém, tome cuidado com essa operação, pois o resultado não vai ser o objeto em si, mas uma outra lista. Esse é um

******ebook converter DEMO Watermarks*******

equívoco muito fácil de passar despercebido, resultando em erros no código. Veja a seguir:

```
# set list
my.l <- list('a',
             c(1,2,3),
             factor('a','b'))

# check classes
class(my.l[[2]])

## [1] "numeric"

class(my.l[2])

## [1] "list"
```

Caso tentarmos somar um elemento a `my.l[2]`, teremos uma mensagem de erro:

```
my.l[2] + 1

## Error in my.l[2] + 1: non-numeric argument to binary operator
```

Esse erro ocorre devido ao fato de que uma lista não tem operador de soma. Para corrigir, basta utilizar o duplo colchete, tal como em `my.l[[2]]+1`. O acesso a elementos de uma lista com colchete simples somente é útil quando estamos procurando uma sublista dentro de uma lista maior. No exemplo anterior, caso quiséssemos obter o primeiro e o segundo elemento da lista `my.l`, usaríamos:

```
# set new list
my.new.l <- my.l[c(1,2)]

# check contents
print(my.new.l)

## [[1]]
## [1] "a"
##
## [[2]]
## [1] 1 2 3

class(my.new.l)

## [1] "list"
```

No caso de listas com elementos nomeados, os mesmos podem ser acessados por seu nome através do uso do símbolo `$` tal como em `my.named.l$df.prices` ou `[['nome']]`, tal como em `my.named.l[['df.prices']]`. Em geral, essa é uma forma mais eficiente e recomendada de interagir com os elementos de uma lista. Como regra geral no uso do R, sempre dê preferência ao acesso de elementos através de seus nomes, seja em listas, vetores ou dataframes. Isso evita erros, pois, ao modificar os dados e adicionar algum outro objeto na lista, é possível que o ordenamento interno mude e, portanto, a posição de determinado objeto pode acabar sendo modificada.

Veja os exemplos a seguir, onde são apresentadas as diferentes formas de se acessar uma lista.

******ebook converter DEMO Watermarks*******

```
# different ways to access a list
my.named.l$ticker
my.named.l$price
my.named.l[['ticker']]
my.named.l[['price']]
```

Vale salientar que também é possível acessar diretamente os elementos de um vetor que esteja dentro de uma lista através de colchetes encadeados. Veja a seguir:

```
# accessing elements of a vector in a list
my.l <- list(c(1,2,3),
             c('a', 'b'))

print(my.l[[1]][2])
```

```
## [1] 2
```

```
print(my.l[[2]][1])
```

```
## [1] "a"
```

Tal operação é bastante útil quando interessa apenas um elemento dentro de um objeto maior criado por alguma função.

6.2.3 Adicionando e Removendo Elementos de uma Lista

A remoção, adição e substituição de elementos de uma lista também são procedimentos fáceis. Para adicionar ou substituir, basta definir um novo objeto na posição desejada da lista:

```
# set list
my.l <- list('a', 1, 3)
glimpse(my.l)
```

```
## List of 3
##  $ : chr "a"
##  $ : num 1
##  $ : num 3
```

```
# add new elements to list
my.l[[4]] <- c(1:5)
my.l[[2]] <- c('b')

# print result
glimpse(my.l)
```

```
## List of 4
##  $ : chr "a"
##  $ : chr "b"
##  $ : num 3
##  $ : int [1:5] 1 2 3 4 5
```

A operação também é possível com o uso de nomes e operador `$`:

```
# set list
my.l <- list(elem1 = 'a', name1=5)

# set new element
```

******ebook converter DEMO Watermarks*******

```
my.l$name2 <- 10
glimpse(my.l)
```

```
## List of 3
##  $ elem1: chr "a"
##  $ name1: num 5
##  $ name2: num 10
```

Para remover elementos de uma lista, basta definir o elemento para o símbolo reservado `NULL` (nulo):

```
# set list
my.l <- list(text = 'b', num1 = 2, num2 = 4)
glimpse(my.l)
```

```
## List of 3
##  $ text: chr "b"
##  $ num1: num 2
##  $ num2: num 4
```

```
# remove elements
my.l[[3]] <- NULL
glimpse(my.l)
```

```
## List of 2
##  $ text: chr "b"
##  $ num1: num 2
```

```
my.l$num1 <- NULL
glimpse(my.l)
```

```
## List of 1
##  $ text: chr "b"
```

Outra maneira de retirar elementos de uma lista é utilizando um índice negativo para os elementos indesejados. Observe a seguir, onde eliminamos o segundo elemento de uma lista:

```
# set list
my.l <- list(a = 1, b = 'texto')

# remove second element
glimpse(my.l[[-2]])
```

```
##  num 1
```

Assim como no caso de vetores atômicos, essa remoção também pode ser realizada por condições lógicas. Veja a seguir:

```
# set list
my.l <- list(1, 2, 3, 4)

# remove elements by condition
my.l[my.l > 2] <- NULL
glimpse(my.l)
```

```
## List of 2
##  $ : num 1
##  $ : num 2
```

******ebook converter DEMO Watermarks*******

Porém, note que esse atalho só funciona porque todos os elementos de `my.l` são numéricos.

6.2.4 Processando os Elementos de uma Lista

Um ponto importante a ser destacado a respeito de listas é que os seus elementos podem ser processados e manipulados individualmente através de funções específicas. Imagine uma lista com vetores numéricos de diferentes tamanhos, tal como a seguir:

```
# set list
my.l.num <- list(c(1, 2, 3),
                 seq(1:50),
                 seq(-5, 5, by = 0.5))
```

Caso quiséssemos calcular a média de cada elemento de `my.l.num` e apresentar o resultado na tela como um vetor, poderíamos fazer isso através de um procedimento simples, processando cada elemento individualmente:

```
# calculate mean of vectors
mean.1 <- mean(my.l.num[[1]])
mean.2 <- mean(my.l.num[[2]])
mean.3 <- mean(my.l.num[[3]])

# print it
print(c(mean.1, mean.2, mean.3))
```

```
## [1]  2.0 25.5  0.0
```

Uma maneira mais fácil, elegante e inteligente seria utilizar a função `sapply`. Nela, basta indicar o nome do objeto de tipo lista e a função que queremos utilizar para processar cada elemento. Internamente, os cálculos são realizados automaticamente. Veja a seguir:

```
# using sapply
my.mean <- sapply(my.l.num, mean)

# print result
print(my.mean)
```

```
## [1]  2.0 25.5  0.0
```

O uso da função `sapply` é preferível por ser mais compacto e eficiente do que a alternativa – a criação de `mean.1`, `mean.2` e `mean.3`. Note que o primeiro código, com médias individuais, só funciona para uma lista com três elementos. Caso tivéssemos um quarto elemento e quiséssemos manter essa estrutura do código, teríamos que adicionar uma nova linha `mean.4 <- mean(my.l.num[[4]])` e modificar o comando de saída na tela para `print(c(mean.1, mean.2, mean.3, mean.4))`. A função `sapply`, ao contrário, funcionaria da mesma forma em listas de qualquer tamanho. Caso tivéssemos mais elementos, nenhuma modificação seria necessária no código anterior, o que o torna extensível a chegada de novos dados.

Essa visão e implementação de código voltado a procedimentos genéricos é um dos lemas para tornar o uso do R mais eficiente. **A regra é simples: sempre escreva códigos que sejam adaptáveis a chegada de novos dados**. Em inglês, isso é chamado de regra *DRY* (*don't repeat yourself*). Caso você esteja repetindo códigos, como no exemplo anterior, certamente existe uma

******ebook converter DEMO Watermarks*******

solução mais elegante e flexível que poderia ser utilizada. No R, existem diversas outras funções da família `apply` para esse objetivo. Essas funções serão explicadas com maiores detalhes no capítulo 8.

6.2.5 Outras Funções Úteis

unlist - Retorna os elementos de uma lista em um único vetor atômico.

```
my.named.l <- list(ticker = 'XXXX4',
          price = c(1,1.5,2,3),
          market = 'Bovespa')
my.unlisted <- unlist(my.named.l)
print(my.unlisted)
```

```
##    ticker    price1    price2    price3    price4    market
##   "XXXX4"       "1"     "1.5"       "2"       "3" "Bovespa"
```

```
class(my.unlisted)
```

```
## [1] "character"
```

as.list - Converte um objeto para uma lista, tornando cada elemento um elemento da lista.

```
my.x <- 10:13
my.x.as.list <- as.list(my.x)
print(my.x.as.list)
```

```
## [[1]]
## [1] 10
##
## [[2]]
## [1] 11
##
## [[3]]
## [1] 12
##
## [[4]]
## [1] 13
```

names - Retorna ou define os nomes dos elementos de uma lista. Assim como para o caso de nomear elementos de um vetor atômico, usa-se a função `names` alocada ao lado esquerdo do símbolo `<-`.

```
my.l <- list(value1 = 1, value2 = 2, value3 = 3)
print(names(my.l))
```

```
## [1] "value1" "value2" "value3"
```

```
my.l <- list(1,2,3)
names(my.l) <- c('num1', 'num2', 'num3')
print(my.l)
```

```
## $num1
## [1] 1
##
## $num2
## [1] 2
```

******ebook converter DEMO Watermarks*******

```
##
## $num3
## [1] 3
```

******ebook converter DEMO Watermarks*******

6.3 Matrizes

Como você deve lembrar das aulas de matemática, uma matriz é uma representação bidimensional de diversos valores, arranjados em linhas e colunas. O uso de matrizes é uma poderosa maneira de representar dados numéricos em duas dimensões e, em certos casos, funções matriciais podem simplificar operações matemáticas complexas.

No R, matrizes são objetos com duas dimensões, onde todos os elementos devem ser da mesma classe. Além disso, as linhas e colunas também podem ter nomes. Assim como para listas e dataframes, isso facilita a interpretação e contextualização dos dados.

Um claro exemplo do uso de matrizes em Finanças seria a representação dos preços de diferentes ações ao longo do tempo. Nesse caso, teríamos as linhas representando as diferentes datas e as colunas representando cada ativo, tal como a seguir:

Data	ABEV3	BBAS3	BBDC3
2010-01-01	736.67	59.4	29.81
2010-01-04	764.14	59.8	30.82
2010-01-05	768.63	59.2	30.38
2010-01-06	776.47	59.28	30.20

A matriz anterior poderia ser criada da seguinte forma no R:

```
# create matrix
data <- c(736.67, 764.14, 768.63, 776.47,
          59.4, 59.8, 59.2, 59.28,
          29.81, 30.82, 30.38, 30.20)

my.mat <- matrix(data, nrow = 4, ncol = 3)

# set names of cols and rows
colnames(my.mat) <- c('ABEV3', 'BBAS3', 'BBDC3')
rownames(my.mat) <- c('2010-01-01', '2010-01-04',
                      '2010-01-05', '2010-01-06')

print(my.mat)
```

```
##             ABEV3 BBAS3 BBDC3
## 2010-01-01 736.67 59.40 29.81
## 2010-01-04 764.14 59.80 30.82
## 2010-01-05 768.63 59.20 30.38
## 2010-01-06 776.47 59.28 30.20
```

Observe que, na construção da matriz, definimos o número de linhas e colunas explicitamente com os argumentos `nrow = 4` e `ncol = 3`. Já os nomes das linhas e colunas são definidos pelos comandos `colnames` e `rownames`. Destaca-se, novamente, que a forma de utilizá-los é bastante distinta das demais funções do R. Nesse caso, utilizamos a função ao lado esquerdo do símbolo de `assign` (`<-`). Poderíamos, porém, recuperar os nomes das linhas e colunas com as mesmas funções. Veja a seguir:

******ebook converter DEMO Watermarks*******

```
colnames(my.mat)
```

```
## [1] "ABEV3" "BBAS3" "BBDC3"
```

```
rownames(my.mat)
```

```
## [1] "2010-01-01" "2010-01-04" "2010-01-05" "2010-01-06"
```

No momento em que temos essa nossa matriz criada, podemos utilizar todas as suas propriedades numéricas. Um exemplo simples é o cálculo do valor de um portfólio de investimento ao longo do tempo. Caso um investidor tenha 200 ações da ABEV3, 300 da BBAS3 e 100 da BBDC3, o valor do seu portfólio ao longo do tempo poderá ser calculado assim:

$$V_t = \sum_{i=1}^{4} N_i P_{i,t}$$

Onde *Ni* é o número de ações compradas para ativo *i* e *Pit* é o preço da ação *i* na data *t*. Essa é uma operação bastante simples de ser efetuada com uma multiplicação matricial. Traduzindo o procedimento para o R, temos:

```
my.w <- as.matrix(c(200, 300, 100), nrow = 3)
my.port <- my.mat %*% my.w
print(my.port)
```

```
##              [,1]
## 2010-01-01 168135
## 2010-01-04 173850
## 2010-01-05 174524
## 2010-01-06 176098
```

Nesse último exemplo, utilizamos o símbolo `%*%`, o qual permite a multiplicação matricial entre os objetos. O objeto `my.port` indica o valor desse portfólio ao longo das datas, resultando em um leve lucro para o investidor.

Um ponto importante a ressaltar é que uma matriz no R não precisa necessariamente ser composta por valores. É possível, também, criar matrizes de caracteres (texto). Veja o exemplo a seguir:

```
my.mat.char <- matrix(rep(c('a','b','c'), 3) ,
                      nrow = 3,
                      ncol = 3)
print(my.mat.char)
```

```
##      [,1] [,2] [,3]
## [1,] "a"  "a"  "a"
## [2,] "b"  "b"  "b"
## [3,] "c"  "c"  "c"
```

Essa flexibilidade dos objetos matriciais possibilita a fácil representação e visualização de seus dados em casos específicos.

******ebook converter DEMO Watermarks*******

6.3.1 Selecionando Valores de uma Matriz

Tal como no caso dos vetores atômicos, também é possível selecionar pedaços de uma matriz através de indexação. Uma diferença aqui é que os objetos de matrizes possuem duas dimensões, enquanto vetores possuem apenas uma.[1] Essa dimensão extra requer a indexação não apenas de linhas mas também de colunas. Os elementos de uma matriz podem ser acessados pela notação `[i,j]`, onde *i* representa a linha e *j* a coluna. Veja o exemplo a seguir:

```
my.mat <- matrix(1:9, nrow = 3)
print(my.mat)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
print(my.mat[1,2])
```

```
## [1] 4
```

Para selecionar colunas ou linhas inteiras, basta deixar o índice vazio, tal como no exemplo a seguir:

```
print(my.mat[ , 2])
```

```
## [1] 4 5 6
```

```
print(my.mat[1, ])
```

```
## [1] 1 4 7
```

Observe que a indexação retorna um vetor atômico da classe dos dados. Caso quiséssemos que o pedaço da matriz mantivesse a sua classe e orientação vertical ou horizontal, poderíamos forçar essa conversão pelo uso de `matrix`:

```
print(as.matrix(my.mat[ ,2]))
```

```
##      [,1]
## [1,]    4
## [2,]    5
## [3,]    6
```

```
print(matrix(my.mat[1, ], nrow=1))
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
```

Pedaços da matriz também podem ser selecionados via indexadores. Caso quiséssemos uma matriz formada a partir da seleção dos elementos da segunda linha e primeira coluna até a terceira linha e segunda coluna, usaríamos o seguinte código:

```
print(my.mat[2:3,1:2])
```

```
##      [,1] [,2]
## [1,]    2    5
```

******ebook converter DEMO Watermarks*******

```
## [2,]    3    6
```

Por fim, o uso de testes lógicos para selecionar valores de matrizes também é possível. Veja a seguir:

```
my.mat <- matrix(1:9, nrow = 3)
print(my.mat >5)
```

```
##       [,1]  [,2] [,3]
## [1,] FALSE FALSE TRUE
## [2,] FALSE FALSE TRUE
## [3,] FALSE  TRUE TRUE
```

```
print(my.mat[my.mat >5])
```

```
## [1] 6 7 8 9
```

6.3.2 Outras Funções Úteis

as.matrix - Transforma dados para a classe `matrix`.

```
my.mat <- as.matrix(1:5)
print(my.mat)
```

```
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
## [4,]    4
## [5,]    5
```

t - Retorna a transposta da matriz.

```
my.mat <- matrix(seq(10,20, length.out = 6), nrow = 3)
print(my.mat)
```

```
##      [,1] [,2]
## [1,]   10   16
## [2,]   12   18
## [3,]   14   20
```

```
print(t(my.mat))
```

```
##      [,1] [,2] [,3]
## [1,]   10   12   14
## [2,]   16   18   20
```

rbind - Retorna a junção (cola) vertical de matrizes, orientando-se pelas linhas.

```
my.mat.1 <- matrix(1:5, nrow = 1)
print(my.mat.1)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
```

```
my.mat.2 <- matrix(10:14, nrow = 1)
print(my.mat.2)
```

******ebook converter DEMO Watermarks*******

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   10   11   12   13   14
```

```
my.rbind.mat <- rbind(my.mat.1, my.mat.2)
print(my.rbind.mat)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]   10   11   12   13   14
```

cbind - Retorna a junção (cola) horizontal de matrizes, orientando-se pelas colunas.

```
my.mat.1 <- matrix(1:4, nrow = 2)
print(my.mat.1)
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
my.mat.2 <- matrix(10:13, nrow = 2)
print(my.mat.2)
```

```
##      [,1] [,2]
## [1,]   10   12
## [2,]   11   13
```

```
my.cbind.mat <- cbind(my.mat.1, my.mat.2)
print(my.cbind.mat)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3   10   12
## [2,]    2    4   11   13
```

******ebook converter DEMO Watermarks*******

6.4 Exercícios

1. Utilizando `set.seed(1)`, crie um dataframe chamado `my.df` com uma coluna chamada `x` contendo uma sequência de 1 a 100 e outra coluna chamada `y` com o valor da coluna `x` adicionada de 5. Usando código, qual a quantidade de valores em `x` maiores que 10 e menores que 25?

2. Crie uma nova coluna em `my.df` chamada `cumsum_x`, contendo a soma cumulativa de `x` (função `cumsum`). Desta nova coluna, quantos valores são maiores que 50?

3. Filtre o dataframe `my.df` para manter apenas as linhas onde o valor da coluna `x` é maior que 0.

4. Repita os exercícios 1, 2 e 3 utilizando as funções do `tidyverse` e o operador de *pipeline*.

5. Utilize pacote `BatchGetSymbols` para baixar dados da ação do Facebook (FB) desde a data `'2010-01-01'`. Adicione ao dataframe com os dados de preços uma nova coluna com o retorno acumulado da ação.

6. Importe os dados disponíveis no arquivo `grunfeld.csv`, disponível no arquivo com os dados do livro. Utilize funções `dplyr::glimpse`, `summary` e `tabplot::tabplot` para conhecer os dados importados.

7. Crie um objeto do tipo lista com três dataframes em seu conteúdo, `df1`, `df2` e `df3`. O conteúdo e tamanho dos dataframes é livre. Utilize função `sapply` para descobrir o número de linhas e colunas em cada dataframe.

8. Utilizando o R, crie uma matrix identidade (valor 1 na diagonal, zero em qualquer outro) de tamanho 3X3.

---

1. Para evitar confusão, vale salientar que, no R, vetores atômicos não possuem dimensão no sentido estrito da função. Ao usar a função `dim` em um vetor atômico, tal como em `dim(c(1,2,4))`, o resultado é nulo (`NULL`). Fora do ambiente computacional, porém, vetores atômicos podem ser entendidos como unidimensionais pois crescem em apenas um sentido.↩

******ebook converter DEMO Watermarks*******

7 As Classes Básicas de Objetos

As classes básicas são os elementos mais primários na representação de dados no R. Nos capítulos anteriores utilizamos as classes básicas para importar dados como dataframes. Dados numéricos viraram uma coluna do tipo `numeric` em um dataframe, enquanto dados de texto viraram um objeto do tipo `character`.

Neste capítulo iremos estudar mais a fundo as classes básicas de objetos do R, incluindo a sua criação até a manipulação do seu conteúdo. Este capítulo é de suma importância pois mostrará quais operações são possíveis com cada classe de objeto e como podes manipular as informações de forma eficiente. Os tipos de objetos tratados aqui serão:

- Numéricos (`numeric`)
- Texto (`character`)
- Fatores (`factor`)
- Valores lógicos (`logical`)
- Datas e tempo (`Date` e `timedate`)
- Dados Omissos (`NA`)

******ebook converter DEMO Watermarks*******

7.1 Objetos Numéricos

Uma das classes mais utilizadas em pesquisa empírica. Os valores numéricos são representações de uma quantidade. Por exemplo: o preço de uma ação em determinada data, o volume negociado de um contrato financeiro em determinado dia, a inflação anual de um país, entre várias outras possibilidades.

7.1.1 Criando e Manipulando Vetores Numéricos

A criação e manipulação de valores numéricos é fácil e direta. Os símbolos de operações matemáticas seguem o esperado, tal como soma (+), diminuição (-), divisão (/) e multiplicação (*). Todas as operações matemáticas são efetuadas com a orientação de elemento para elemento e possuem notação vetorial. Isso significa, por exemplo, que podemos criar vetores inteiros em uma única linha de comando. Veja a seguir, onde se cria dois vetores e realiza-se diversas operações entre eles.

```
# create numeric vectors
x <- 1:5
y <- 2:6

# print sum
print(x+y)
```

```
## [1]  3  5  7  9 11
```

```
# print multiplication
print(x*y)
```

```
## [1]  2  6 12 20 30
```

```
# print division
print(x/y)
```

```
## [1] 0.5000000 0.6666667 0.7500000 0.8000000 0.8333333
```

```
# print exponentiation
print(x^y)
```

```
## [1]     1     8    81  1024 15625
```

Um diferencial do R em relação a outras linguagens é que, nele, são aceitas operações entre vetores diferentes. Por exemplo, podemos somar um vetor numérico de quatro elementos com outro de apenas dois. Nesse caso, aplica-se a chamada **regra de reciclagem** (*recycling rule*). Ela define que, se dois vetores de tamanho diferente estão interagindo, o vetor menor é repetido tantas vezes quantas forem necessárias para obter-se o mesmo número de elementos do vetor maior. Veja o exemplo a seguir:

```
# set x with 4 elements and y with 2
x <- 1:4
y <- 2:1

# print multiplication
```

******ebook converter DEMO Watermarks*******

```
print(x + y)
```

```
## [1] 3 3 5 5
```

O resultado de `x + y` é equivalente a `1:4 + c(2, 1, 2, 1)`. Caso interagirmos vetores em que o tamanho do maior não é múltiplo do menor, o R realiza o mesmo procedimento de reciclagem, porém emite uma mensagem de `warning`:

```
# set x = 4 elements and y with 3
x <- c(1, 2, 3, 4)
y <- c(1, 2, 3)

# print sum (recycling rule)
print(x +y)
```

```
## Warning in x + y: longer object length is not a multiple of
## shorter object length
```

```
## [1] 2 4 6 5
```

Os três primeiros elementos de `x` foram somados aos três primeiros elementos de `y`. O quarto elemento de `x` foi somado ao primeiro elemento de `y`. Uma vez que não havia um quarto elemento em `y`, o ciclo reinicia, resgatando o primeiro elemento de `y` e resultando em uma soma igual a 5.

Os elementos de um vetor numérico também podem ser nomeados quando na criação do vetor:

```
# create named vector
x <- c(item1 = 10,
       item2 = 14,
       item3 = 9,
       item4 = 2)

# print it
print(x)
```

```
## item1 item2 item3 item4
##    10    14     9     2
```

Para nomear os elementos após a criação, podemos utilizar a função `names`. Veja a seguir:

```
# create unnamed vector
x <- c(10, 14, 9, 2)

# set names of elements
names(x) <- c('item1', 'item2', 'item3', 'item4')

# print it
print(x)
```

```
## item1 item2 item3 item4
##    10    14     9     2
```

Vetores numéricos vazios também podem ser criados. Em algumas situações de desenvolvimento de código faz sentido pré-alocar o vetor antes de preenchê-lo com valores. Nesse caso, utilize a função `numeric`:

******ebook converter DEMO Watermarks*******

```r
# create empty numeric vector of length 10
my.x <- numeric(length = 10)

# print it
print(my.x)
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

Observe que, nesse caso, os valores de `my.x` são definidos como zero.

7.1.1.1 Criando Sequências de Valores

Existem duas maneiras de criar uma sequência de valores no R. A primeira, que já foi utilizada nos exemplos anteriores, é o uso do operador `:`. Por exemplo, `my.seq <- 1:10` ou `my.seq <- -5:5`. Esse método é bastante prático, pois a notação é clara e direta.

Porém, o uso do operador : limita as possibilidades. A diferença entre os valores é sempre *1* para sequências ascendentes e *-1* para sequências descendentes. Uma versão mais poderosa para a criação de sequências é o uso da função `seq`. Com ela, é possível definir os intervalos entre cada valor com o argumento `by`. Veja a seguir:

```r
# set sequence
my.seq <- seq(from = -10, to = 10, by = 2)

# print it
print(my.seq)
```

```
##  [1] -10  -8  -6  -4  -2   0   2   4   6   8  10
```

Outro atributo interessante da função `seq` é a possibilidade de criar vetores com um valor inicial, um valor final e o número de elementos desejado. Isso é realizado com o uso da opção `length.out`. Observe o código a seguir, onde cria-se um vetor de *0* até *10* com *20* elementos:

```r
# set sequence with fixed size
my.seq <- seq(from = 0, to = 10, length.out = 20)

# print it
print(my.seq)
```

```
##  [1]  0.0000000  0.5263158  1.0526316  1.5789474  2.1052632
##  [6]  2.6315789  3.1578947  3.6842105  4.2105263  4.7368421
## [11]  5.2631579  5.7894737  6.3157895  6.8421053  7.3684211
## [16]  7.8947368  8.4210526  8.9473684  9.4736842 10.0000000
```

No caso anterior, o tamanho final do vetor foi definido e a própria função se encarregou de descobrir qual a variação necessária entre cada valor de `my.seq`.

7.1.1.2 Criando Vetores com Elementos Repetidos

Outra função interessante é a que cria vetores com o uso de repetição. Por exemplo: imagine que estamos interessado em um vetor preenchido com o valor 1 dez vezes. Para isso, basta utilizar a função `rep`:

******ebook converter DEMO Watermarks*******

```
# repeat vector three times
my.x <- rep(x = 1, times = 10)

# print it
print(my.x)
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1
```

A função também funciona com vetores. Considere uma situação onde temos um vetor com os valores `c(1,2)` e gostaríamos de criar um vetor maior com os elementos `c(1, 2, 1, 2, 1, 2)` - isto é, repetindo o vetor menor três vezes. Veja o resultado a seguir:

```
# repeat vector three times
my.x <- rep(x = c(1, 2), times = 3)

# print it
print(my.x)
```

```
## [1] 1 2 1 2 1 2
```

7.1.1.3 Criando Vetores com Números Aleatórios

Em muitas situações será necessário a criação de números aleatórios. Esse procedimento numérico é bastante utilizado para simular modelos matemáticos em Finanças. Por exemplo, o método de simulação de preços de ativos de Monte Carlo parte da simulação de números aleatórios. No R, existem diversas funções que criam números aleatórios para diferentes distribuições estatísticas. As mais utilizadas, porém, são as funções `rnorm` e `runif`.

A função `rnorm` gera números aleatórios da distribuição Normal, com opções para a média e o desvio padrão. Veja o seu uso a seguir:

```
# generate 10 random numbers from a Normal distribution
my.rnd.vec <- rnorm(n = 10,
                    mean = 0,
                    sd = 1)

# print it
print(my.rnd.vec)
```

```
##  [1]  0.50030801 -1.21556058  0.02955926 -0.60149662
##  [5]  2.34311912  0.29308964 -1.52765475  1.42400406
##  [9]  0.54595678 -0.78161595
```

O código anterior gera dez números aleatórios de uma distribuição Normal com média zero e desvio padrão igual a um.

Já a função `runif` gera valores aleatórios da distribuição uniforme entre um valor máximo e um valor mínimo. Ela é geralmente utilizada para simular probabilidades. A função `runif` tem três parâmetros de entrada: o número de valores aleatórios desejado, o valor mínimo e o valor máximo. Veja exemplo a seguir:

```
# create a random vector with minimum and maximum
my.rnd.vec <- runif(n = 10,
```

******ebook converter DEMO Watermarks*******

```
                  min = -5,
                  max = 5)

# print it
print(my.rnd.vec)
```

```
##  [1] -1.2628859 -0.6069444 -1.4510819 -3.6340779 -2.9760311
##  [6]  3.4101217 -4.9245814 -4.5227119  2.5183084  3.2244318
```

Observe que ambas as funções anteriores são limitadas à sua respectiva distribuição. Uma maneira alternativa e flexível de gerar valores aleatórios é utilizar a função `sample`. Essa tem como entrada um vetor qualquer e retorna uma versão embaralhada de seus elementos. A sua flexibilidade reside no fato de que o vetor de entrada pode ser qualquer coisa. Por exemplo, caso quiséssemos criar um vetor aleatório com os números `c(0, 5, 15, 20, 25)` apenas, poderíamos fazê-lo da seguinte forma:

```
# create sequence
my.vec <- seq(from = 0, to = 25, by=5)

# sample sequence
my.rnd.vec <- sample(my.vec)

# print it
print(my.rnd.vec)
```

```
## [1] 20 10 15  5  0 25
```

A função `sample` também permite a seleção aleatória de um certo número de termos. Por exemplo, caso quiséssemos selecionar aleatoriamente apenas um elemento de `my.vec`, escreveríamos o código da seguinte maneira:

```
# sample one element of my.vec
my.rnd.vec <- sample(my.vec, size = 1)

# print it
print(my.rnd.vec)
```

```
## [1] 15
```

Caso quiséssemos dois elementos, escreveríamos:

```
# sample two elements of my.vec
my.rnd.vec <- sample(my.vec, size = 2)

# print it
print(my.rnd.vec)
```

```
## [1] 10 15
```

Também é possível selecionar valores de uma amostra menor para a criação de um vetor maior. Por exemplo, considere o caso em que se tem um vetor com os números `c(10, 15, 20)` e deseja-se criar um vetor aleatório com dez elementos retirados desse vetor menor, onde alguns seriam repetidos. Para isso, podemos utilizar a opção `replace = TRUE`.

```
# create vector
```

******ebook converter DEMO Watermarks*******

```
my.vec <- c(5, 10, 15)

# sample
my.rnd.vec <- sample(x = my.vec, size = 10, replace = TRUE)
print(my.rnd.vec)

##  [1]  5 15 15  5  5 10 15 15 10  5
```

Vale destacar que a função `sample` funciona para qualquer tipo ou objeto, não sendo, portanto, exclusiva para vetores numéricos. Poderíamos, também, escolher elementos aleatórios de um vetor de texto ou então uma lista:

```
# example of sample with characters
print(sample(c('elem 1','elem 2','elem 3'),
             size = 1))

## [1] "elem 3"

# example of sample with list
print(sample(list(x = c(1,1,1),
                  y = c('a', 'b')),
             size = 1))

## $y
## [1] "a" "b"
```

É importante ressaltar que a geração de valores aleatórios no R (ou qualquer outro programa) **não é totalmente aleatória!** De fato, o próprio computador escolhe os valores dentre uma fila de valores possíveis. Cada vez que funções tal como `rnorm`, `runif` e `sample` são utilizadas, o computador escolhe um lugar diferente dessa fila de acordo com vários parâmetros, incluindo a data e horário atual. Portanto, do ponto de vista do usuário, os valores são gerados de forma imprevisível. Para o computador, porém, essa seleção é completamente determinística e guiada por regras claras.

Uma propriedade interessante no R é a possibilidade de selecionar uma posição específica na fila de valores aleatórios utilizando função `set.seed`. É ela que **fixa a semente** para gerar os valores. Na prática, o resultado é que todos os números e seleções aleatórias realizadas pelo código serão iguais em cada execução. O uso de `set.seed` é bastante recomendado para manter a reprodutibilidade dos códigos envolvendo aleatoriedade. Veja o exemplo a seguir, onde utiliza-se essa função.

```
# fix seed
set.seed(seed = 10)

# set vec and print
my.rnd.vec.1 <- runif(5)
print(my.rnd.vec.1)

## [1] 0.50747820 0.30676851 0.42690767 0.69310208 0.08513597

# set vec and print
my.rnd.vec.2 <- runif(5)
print(my.rnd.vec.2)

## [1] 0.2254366 0.2745305 0.2723051 0.6158293 0.4296715
```

******ebook converter DEMO Watermarks*******

No código anterior, o valor de `set.seed` é um inteiro escolhido pelo usuário. Após a chamada de `set.seed`, todas as seleções e números aleatórios irão iniciar do mesmo ponto e, portanto, serão iguais. Motivo o leitor a executar o código anterior no R. Verás que os valores de `my.rnd.vec.1` e `my.rnd.vec.2` serão exatamente iguais aos valores colocados aqui.

O uso de `set.seed` também funciona para o caso de `sample`. Veja a seguir:

```
# fix seed
set.seed(seed = 15)

# print vectors
print(sample(1:10))
```

```
##  [1]  7  2  8  5  3 10  4  1  9  6
```

```
print(sample(10:20))
```

```
##  [1] 11 16 14 15 19 17 12 13 10 20 18
```

Novamente, execute os comandos anteriores no R e verás que o resultado na tela bate com o apresentado aqui.

7.1.2 Acessando Elementos de um Vetor Numérico

Todos os elementos de um vetor numérico podem ser acessados através do uso de colchetes (`[ ]`). Por exemplo, caso quiséssemos apenas o primeiro elemento de `x`, teríamos:

```
# set vector
x <- c(-1, 4, -9, 2)

# get first element
first.elem.x <- x[1]

# print it
print(first.elem.x)
```

```
## [1] -1
```

A mesma notação é válida para extrair porções de um vetor. Caso quiséssemos um subvetor de `x` com o primeiro e o segundo elemento, faríamos essa operação da seguinte forma:

```
# sub-vector of x
sub.x <- x[1:2]

# print it
print(sub.x)
```

```
## [1] -1  4
```

Para acessar elementos nomeados de um vetor numérico, basta utilizar seu nome junto aos colchetes.

```
# set named vector
x <- c(item1 = 10, item2 = 14, item3 = -9, item4 = -2)
```

******ebook converter DEMO Watermarks*******

```
# access elements by name
print(x['item2'])
```

```
## item2
##    14
```

```
print(x[c('item2','item4')])
```

```
## item2 item4
##    14    -2
```

O acesso aos elementos de um vetor numérico também é possível através de testes lógicos. Por exemplo, caso tivéssemos interesse em saber quais os valores de `x` que são maiores do que *0*, o código resultante seria da seguinte forma:

```
# find all values of x higher than zero
print(x[x > 0])
```

```
## item1 item2
##    10    14
```

Os usos de regras de segmentação dos dados de acordo com algum critério é chamado de indexação lógica. Os objetos do tipo `logical` serão tratados mais profundamente em seção futura deste capítulo.

7.1.3 Modificando e Removendo Elementos

A modificação de um vetor numérico é muito simples. Basta indicar a posição dos elementos e os novos valores com o símbolo de *assign* (`<-`):

```
# set vector
my.x <- 1:4

# modify first element to 5
my.x[1] <- 5

# print result
print(my.x)
```

```
## [1] 5 2 3 4
```

Essa modificação também pode ser realizada em bloco:

```
# set vector
my.x <- 0:5

# set the first three elements to 5
my.x[1:3] <- 5

# print result
print(my.x)
```

```
## [1] 5 5 5 3 4 5
```

O uso de condições para definir elementos é realizada pela indexação:

******ebook converter DEMO Watermarks*******

```
# set vector
my.x <- -5:5

# set any value lower than 2 to 0
my.x[my.x<2] <- 0

# print result
print(my.x)
```

```
##  [1] 0 0 0 0 0 0 0 2 3 4 5
```

A remoção de elementos é realizada com o uso de índices negativos:

```
# create vector
my.x <- -5:5

# remove first and second element of my.x
my.x <- my.x[-(1:2)]

# show result
print(my.x)
```

```
## [1] -3 -2 -1  0  1  2  3  4  5
```

Note como o uso do índice negativo em `my.x[-(1:2)]` retorna o vetor original sem o primeiro e segundo elemento.

7.1.4 Criando Grupos

Em algumas situações será necessário entender quantos casos da amostra estão localizados entre um determinado intervalo. Por exemplo, imagine o vetor dos retornos diários de uma ação, isto é, a variação percentual dos preços de fechamento entre um dia e outro. Uma possível análise de risco que pode ser realizada é dividir o intervalo de retornos em cinco partes e verificar o percentual de ocorrência dos valores em cada um dos intervalos. Essa análise numérica é bastante semelhante à construção e visualização de histogramas.

A função `cut` serve para criar grupos de intervalos a partir de um vetor numérico. Veja o exemplo a seguir, onde cria-se um vetor aleatório oriundo da distribuição Normal e cinco grupos a partir de intervalos definidos pelos dados.

```
# set rnd vec
my.x <- rnorm(10)

# "cut" it into 5 pieces
my.cut <- cut(x = my.x, breaks = 5)
print(my.cut)
```

```
##  [1] (0.34,0.82]    (-1.57,-1.09]  (0.34,0.82]
##  [4] (-1.09,-0.616] (-0.138,0.34]  (-1.09,-0.616]
##  [7] (-1.57,-1.09]  (0.34,0.82]    (-0.138,0.34]
## [10] (0.34,0.82]
## 5 Levels: (-1.57,-1.09] (-1.09,-0.616] ... (0.34,0.82]
```

Observe que os nomes dos elementos da variável `my.cut` são definidos pelos intervalos e o

******ebook converter DEMO Watermarks*******

resultado é um objeto do tipo fator. Em seções futuras, iremos explicar melhor esse tipo de objeto e as suas propriedades.

No exemplo anterior, os intervalos para cada grupo foram definidos automaticamente. No uso da função `cut`, também é possível definir quebras customizadas nos dados e nos nomes dos grupos. Veja a seguir:

```
# set random vector
my.x <- rnorm(10)

# create groups with 5 breaks
my.cut <- cut(x = my.x, breaks = 5)

# print it!
print(my.cut)
```

```
##  [1] (-0.749,0.0246] (-0.749,0.0246] (-0.749,0.0246]
##  [4] (0.0246,0.798]  (-2.3,-1.52]    (0.0246,0.798]
##  [7] (0.798,1.57]    (0.798,1.57]    (0.0246,0.798]
## [10] (-0.749,0.0246]
## 5 Levels: (-2.3,-1.52] (-1.52,-0.749] ... (0.798,1.57]
```

Note que os nomes dos elementos em `my.cut` foram definidos como intervalos e o resultado é um objeto do tipo fator. É possível também definir intervalos e nomes customizados para cada grupo com o uso dos argumentos `labels` e `breaks`:

```
# create random vector
my.x <- rnorm(10)

# define breaks manually
my.breaks <- c(min(my.x)-1, -1, 1, max(my.x)+1)

# define labels manually
my.labels <- c('Low','Normal', 'High')

# create group from numerical vector
my.cut <- cut(x = my.x, breaks = my.breaks, labels = my.labels)

# print both!
print(my.x)
```

```
##  [1]  2.6909832  0.5981759  1.6113647 -0.4373813  1.3526206
##  [6]  0.4705685  0.4702481  0.3963088 -0.7304926  0.6531176
```

```
print(my.cut)
```

```
##  [1] High   Normal High   Normal High   Normal Normal Normal
##  [9] Normal Normal
## Levels: Low Normal High
```

Como podemos ver, os nomes dos grupos estão mais amigáveis para uma futura análise.

7.1.5 Outras Funções Úteis

as.numeric - Converte determinado objeto para numérico.

******ebook converter DEMO Watermarks*******

```
my.text <- c('1', '2', '3')
class(my.text)

## [1] "character"

my.x <- as.numeric(my.text)
print(my.x)

## [1] 1 2 3

class(my.x)

## [1] "numeric"
```

sum - Soma os elementos de um vetor.

```
my.x <- 1:50
my.sum <- sum(my.x)
print(my.sum)

## [1] 1275
```

max - Retorna o máximo valor numérico do vetor.

```
x <- c(10, 14, 9, 2)
max.x <- max(x)
print(max.x)

## [1] 14
```

min - Retorna o mínimo valor numérico do vetor.

```
x <- c(12, 15, 9, 2)
min.x <- min(x)
print(min.x)

## [1] 2
```

which.max - Retorna a posição do máximo valor numérico do vetor.

```
x <- c(100, 141, 9, 2)
which.max.x <- which.max(x)
cat(paste('The position of the maximum value of x is ', which.max.x))

## The position of the maximum value of x is  2

cat(' and its value is ', x[which.max.x])

##  and its value is  141
```

which.min - Retorna a posição do mínimo valor numérico do vetor.

```
x <- c(10, 14, 9, 2)
which.min.x <- which.min(x)
cat(paste('The position of the minimum value of x is ',
          which.min.x, ' and its value is ', x[which.min.x]))

## The position of the minimum value of x is  4  and its value is  2
```

******ebook converter DEMO Watermarks*******

sort - Retorna uma versão ordenada de um vetor.

```
x <- runif(5)
print(sort(x, decreasing = FALSE))
```

```
## [1] 0.7646008 0.8347800 0.8902690 0.9099027 0.9935257
```

```
print(sort(x, decreasing = TRUE))
```

```
## [1] 0.9935257 0.9099027 0.8902690 0.8347800 0.7646008
```

cumsum - Soma os elementos de um vetor de forma cumulativa.

```
my.x <- 1:25
my.cumsum <- cumsum(my.x)
print(my.cumsum)
```

```
##  [1]   1   3   6  10  15  21  28  36  45  55  66  78  91 105
## [15] 120 136 153 171 190 210 231 253 276 300 325
```

prod - Realiza o produto de todos os elementos de um vetor.

```
my.x <- 1:10
my.prod <- prod(my.x)
print(my.prod)
```

```
## [1] 3628800
```

cumprod - Calcula o produto cumulativo de todos os elementos de um vetor.

```
my.x <- 1:10
my.prod <- cumprod(my.x)
print(my.prod)
```

```
##  [1]       1       2       6      24     120     720    5040
##  [8]   40320  362880 3628800
```

******ebook converter DEMO Watermarks*******

7.2 Classe de Caracteres (texto)

A classe de caracteres, ou texto, serve para armazenar informações textuais. Um exemplo prático em Finanças seria o reconhecimento de uma ação através dos seus símbolos de identificação (*tickers*) ou então por sua classe de ação: ordinária ou preferencial. Este tipo de dado tem sido utilizado cada vez mais em pesquisa empírica (Gentzkow, Kelly, and Taddy 2017), resultando em uma diversidade de pacotes.

O R possui vários recursos que facilitam a criação e manipulação de objetos de tipo texto. As funções básicas fornecidas com a instalação de R são abrangentes e adequadas para a maioria dos casos. No entanto, pacote `stringr` (Wickham 2015) do `tidyverse` fornece muitas funções que expandem a funcionalidade básica do R na manipulação de texto.

Um aspecto positivo de `stringr` é que as funções começam com o nome `str_` e possuem nomes informativos. Combinando isso com o recurso de preenchimento automático (*autocomplete*) pela tecla *tab*, fica fácil de localizar os nomes das funções do pacote. Seguindo a prioridade ao universo do `tidyverse`, esta seção irá dar preferência ao uso das funções do pacote `stringr`. As rotinas nativas de manipulação de texto serão apresentadas, porém de forma limitada.

7.2.1 Criando um Objeto Simples de Caracteres

Todo objeto de caracteres é criado através da encapsulação de um texto por aspas duplas (`" "`) ou simples (`' '`). Para criar um vetor de caracteres com *tickers* de ações, podemos fazê-lo com o seguinte código:

```
my.assets <- c('PETR3', 'VALE4', 'GGBR4')
print(my.assets)
```

```
## [1] "PETR3" "VALE4" "GGBR4"
```

Confirma-se a classe do objeto com a função `class`:

```
class(my.assets)
```

```
## [1] "character"
```

7.2.2 Criando Objetos Estruturados de Texto

Em muitos casos no uso do R, estaremos interessados em criar vetores de texto com algum tipo de estrutura própria. Por exemplo, o vetor `c("text 1", "text 2", ..., "text 20")` possui um lógica de criação clara. Computacionalmente, podemos definir a sua estrutura como sendo a junção do texto `text` e um vetor de sequência, de 1 até 20.

Para criar um vetor textual capaz de unir texto com número, utilizamos a função `stringr::str_c` ou `base::paste`. Veja o exemplo a seguir, onde replica-se o caso anterior com e sem espaço entre número e texto:

```
library(stringr)
```

******ebook converter DEMO Watermarks*******

```
# create sequence
my.seq <- 1:20

# create character
my.text <- 'text'

# paste objects together (without space)
my.char <- str_c(my.text, my.seq)
print(my.char)
```

```
##  [1] "text1"  "text2"  "text3"  "text4"  "text5"  "text6"
##  [7] "text7"  "text8"  "text9"  "text10" "text11" "text12"
## [13] "text13" "text14" "text15" "text16" "text17" "text18"
## [19] "text19" "text20"
```

```
# paste objects together (with space)
my.char <- str_c(my.text, my.seq, sep = ' ')
print(my.char)
```

```
##  [1] "text 1"  "text 2"  "text 3"  "text 4"  "text 5"
##  [6] "text 6"  "text 7"  "text 8"  "text 9"  "text 10"
## [11] "text 11" "text 12" "text 13" "text 14" "text 15"
## [16] "text 16" "text 17" "text 18" "text 19" "text 20"
```

```
# paste objects together (with space)
my.char <- paste(my.text, my.seq)
print(my.char)
```

```
##  [1] "text 1"  "text 2"  "text 3"  "text 4"  "text 5"
##  [6] "text 6"  "text 7"  "text 8"  "text 9"  "text 10"
## [11] "text 11" "text 12" "text 13" "text 14" "text 15"
## [16] "text 16" "text 17" "text 18" "text 19" "text 20"
```

O mesmo procedimento também pode ser realizado com vetores de texto. Veja a seguir:

```
# set character value
my.x <- 'My name is'

# set character vector
my.names <- c('Marcelo', 'Ricardo', 'Tarcizio')

# paste and print
print(str_c(my.x, my.names, sep = ' '))
```

```
## [1] "My name is Marcelo"  "My name is Ricardo"
## [3] "My name is Tarcizio"
```

Outra possibilidade de construção de textos estruturados é a repetição do conteúdo de um objeto do tipo caractere. No caso de texto, utiliza-se a função `stringr::str_dup`/`base::strrep` para esse fim. Observe o exemplo a seguir:

```
my.char <- str_dup(string = 'abc', times = 5)
print(my.char)
```

```
## [1] "abcabcabcabcabc"
```

7.2.3 Objetos Constantes de Texto

******ebook converter DEMO Watermarks*******

O R também possibilita o acesso direto a todas as letras do alfabeto. Esses estão guardadas nos objetos reservados chamados `letters` e `LETTERS`:

```
# print all letters in alphabet (no cap)
print(letters)
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
## [15] "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
# print all letters in alphabet (WITH CAP)
print(LETTERS)
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N"
## [15] "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

Observe que em ambos os casos não é necessário criar os objetos. Por serem constantes embutidas automaticamente na área de trabalho do R, elas já estão disponíveis para uso. Podemos sobrescrever o nome do objeto com outro conteúdo, porém isso não é aconselhável. Nunca se sabe onde esse objeto constante está sendo usado. Outros objetos de texto constantes no R incluem `month.abb` e `month.name`. Veja a seguir o seu conteúdo:

```
# print abreviation and full names of months
print(month.abb)
```

```
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep"
## [10] "Oct" "Nov" "Dec"
```

```
print(month.name)
```

```
##  [1] "January"   "February"  "March"     "April"
##  [5] "May"       "June"      "July"      "August"
##  [9] "September" "October"   "November"  "December"
```

7.2.4 Selecionando Pedaços de um Texto

Um erro comum praticado por iniciantes é tentar selecionar pedaços de um texto através do uso de colchetes. Observe o código abaixo:

```
# set char object
my.char <- 'ABCDE'

# print its second character: 'B' (WRONG - RESULT is NA)
print(my.char[2])
```

```
## [1] NA
```

O resultado `NA` indica que o segundo elemento de `my.char` não existe. Isso acontece porque o uso de colchetes refere-se ao acesso de **elementos** de um vetor atômico, e não de caracteres dentro de um texto maior. Observe o que acontece quando utilizamos `my.char[1]`:

```
print(my.char[1])
```

```
## [1] "ABCDE"
```

O resultado é simplesmente o texto *ABCDE*, que está localizado no primeiro item de `my.char`.

******ebook converter DEMO Watermarks*******

Para selecionar pedaços de um texto, devemos utilizar a função específica `stringr::str_sub`/`base::substr`:

```
# print third and fourth characters
my.substr <- str_sub(string = my.char,
                     start = 4,
                     end = 4)
print(my.substr)
```

```
## [1] "D"
```

Essa função também funciona para vetores atômicos. Vamos assumir que você importou dados de texto e o conjunto de dados bruto contém um identificador de 3 dígitos de uma empresa, sempre na mesma posição do texto. Vamos simular a situação no R:

```
# build char vec
my.char.vec <- paste0(c('123','231','321'),
                      ' - other ignorable text')
print(my.char.vec)
```

```
## [1] "123 - other ignorable text"
## [2] "231 - other ignorable text"
## [3] "321 - other ignorable text"
```

Só estamos interessados na informação das três primeiras letras de cada elemento em `my.char.vec`. Para selecioná-los, podemos usar as mesmas funções que antes.

```
# get ids with stringr::str_sub
ids.vec <- str_sub(my.char.vec, 1, 3)
print(ids.vec)
```

```
## [1] "123" "231" "321"
```

Operações vetorizadas são comuns no R. Quase tudo o que você pode fazer para um único elemento pode ser expandido para vetores. Isso facilita o desenvolvimento de rotinas pois pode-se facilmente realizar tarefas complicadas em uma série de elementos em uma única linha de código.

7.2.5 Localizando e Substituindo Pedaços de um Texto

Uma operação útil na manipulação de textos é a localização de letras e padrões específicos com funções `stringr::str_locate`/`base::regexpr` e `stringr::str_locate_all`/`base::gregexpr`. É importante destacar que, por *default*, essas funções utilizam de expressões do tipo *regex* - expressões regulares (Thompson 1968). Essa é uma linguagem específica para processar textos. Diversos símbolos são utilizados para estruturar, procurar e isolar padrões textuais. Quando utilizada corretamente, o *regex* é bastante útil e de extrema valia. Para a grande maioria dos casos, porém, o seu uso não é essencial.

Usualmente, o caso mais comum em pesquisa é verificar a posição ou a existência de um texto menor dentro de um texto maior. Isto é, um padrão explícito e fácil de entender. Por isso, a localização e substituição de caracteres no próximo exemplo será do tipo fixo, sem o uso de *regex*. Tal informação pode ser passada às funções do pacote `stringr` através de outra função

******ebook converter DEMO Watermarks*******

chamada `stringr::fixed`.

O exemplo a seguir mostra como encontrar o caractere *D* dentre uma série de caracteres.

```
library(stringr)

my.char <- 'ABCDEF-ABCDEF-ABC'
pos = str_locate(string = my.char, pattern = fixed('D') )
print(pos)
```

```
##      start end
## [1,]     4   4
```

Observe que a função `str_locate` retorna apenas a primeira ocorrência de *D*. Para resgatar todas as ocorrências, devemos utilizar a função `str_locate_all`:

```
# set object
my.char <- 'ABCDEF-ABCDEF-ABC'

# find position of ALL 'D' using str_locate_all
pos = str_locate_all(string = my.char, pattern = fixed('D'))
print(pos)
```

```
## [[1]]
##      start end
## [1,]     4   4
## [2,]    11  11
```

Observe também que as funções `regexp` e `grepexpr` retornam objetos com propriedades específicas, apresentando-as na tela.

Para substituir caracteres em um texto, basta utilizar a função `stringr::str_replace` ou `base::sub` e `str_replace_all` ou `base::gsub`. Vale salientar que `str_replace` substitui a primeira ocorrência do caractere, enquanto `str_replace_all` executa uma substituição global - isto é, aplica-se a todas as ocorrências. Veja a diferença a seguir:

```
# set char object
my.char <- 'ABCDEF-ABCDEF-ABC'

# substitute the FIRST 'ABC' for 'XXX' with sub
my.char <- sub(x = my.char,
               pattern = 'ABC',
               replacement = 'XXX')
print(my.char)
```

```
## [1] "XXXDEF-ABCDEF-ABC"
```

```
# substitute the FIRST 'ABC' for 'XXX' with str_replace
my.char <- 'ABCDEF-ABCDEF-ABC'
my.char <- str_replace(string = my.char,
                       pattern = fixed('ABC'),
                       replacement = 'XXX')
print(my.char)
```

```
## [1] "XXXDEF-ABCDEF-ABC"
```

******ebook converter DEMO Watermarks*******

E agora fazemos uma substituição global dos caracteres.

```
# set char object
my.char <- 'ABCDEF-ABCDEF-ABC'

# substitute the FIRST 'ABC' for 'XXX' with str_replace
my.char <- str_replace_all(string = my.char,
                           pattern = 'ABC',
                           replacement = 'XXX')
print(my.char)
```

```
## [1] "XXXDEF-XXXDEF-XXX"
```

Mais uma vez, vale ressaltar que as operações de substituição também funcionam em vetores. Dê uma olhada no próximo exemplo.

```
# set char object
my.char <- c('ABCDEF','DBCFE','ABC')

# create an example of vector
my.char.vec <- str_c(sample(my.char, 5, replace = T),
                     sample(my.char, 5, replace = T),
                     sep = ' - ')

# show it
print(my.char.vec)
```

```
## [1] "ABCDEF - ABCDEF" "ABC - ABCDEF"    "DBCFE - ABCDEF"
## [4] "ABC - ABC"       "ABC - ABCDEF"
```

```
# substitute all occurrences of 'ABC'
my.char.vec <- str_replace_all(string = my.char.vec,
                               pattern = 'ABC',
                               replacement = 'XXX')

# print result
print(my.char.vec)
```

```
## [1] "XXXDEF - XXXDEF" "XXX - XXXDEF"    "DBCFE - XXXDEF"
## [4] "XXX - XXX"       "XXX - XXXDEF"
```

7.2.6 Separando Textos

Em algumas situações, principalmente no processamento de textos, é possível que se esteja interessado em quebrar um texto de acordo com algum separador. Por exemplo, o texto `abc ; bcd ; adf` apresenta informações demarcadas pelo símbolo `;`. Para separar um texto em várias partes, utilizamos a função `stringr::str_split`/`base::strsplit`. Essas quebram o texto em diversas partes de acordo com algum caractere escolhido. Observe os exemplos a seguir:

```
# set char
my.char <- 'ABCXABCXBCD'

# split it based on 'X' and using stringr::str_split
split.char <- str_split(my.char, 'X')

# print result
```

******ebook converter DEMO Watermarks*******

```
print(split.char)
```

```
## [[1]]
## [1] "ABC" "ABC" "BCD"
```

A saída dessa função é um objeto do tipo lista. Para acessar os elementos de uma lista, deve-se utilizar o operador `[[ ]]`. Por exemplo, para acessar o texto `bcd` da lista `split.char`, executa-se o seguinte código:

```
print(split.char[[1]][2])
```

```
## [1] "ABC"
```

Para visualizar um exemplo de dividir textos em vetores, veja o próximo código.

```
# set char
my.char.vec <- c('ABCDEF','DBCFE','ABFC','ACD')

# split it based on 'B' and using stringr::strsplit
split.char <- str_split(my.char.vec, 'B')

# print result
print(split.char)
```

```
## [[1]]
## [1] "A"    "CDEF"
##
## [[2]]
## [1] "D"   "CFE"
##
## [[3]]
## [1] "A"  "FC"
##
## [[4]]
## [1] "ACD"
```

Observe como, novamente, um objeto do tipo `list` é retornado. Cada elemento é correspondente ao processo de quebra de texto em `my.char`.

7.2.7 Descobrindo o Número de Caracteres de um Texto

Para descobrir o número de caracteres de um texto, utilizamos a função `stringr::str_length`/`base::nchar`. Ela também funciona para vetores atômicos de texto. Veja os exemplos mostrados a seguir:

```
# set char
my.char <- 'abcdef'

# print number of characters using stringr::str_length
print(str_length(my.char))
```

```
## [1] 6
```

E agora um exemplo com vetores.

******ebook converter DEMO Watermarks*******

```
#set char
my.char <- c('a', 'ab', 'abc')

# print number of characters using stringr::str_length
print(str_length(my.char))
```

```
## [1] 1 2 3
```

7.2.8 Gerando Combinações de Texto

Um truque útil no R é usar as funções `base::outer` e `base::expand.grid` para criar todas as combinações possíveis de elementos em diferentes objetos. Isso é útil quando você quer criar um vetor de texto combinando todos os elementos possíveis de diferentes vetores. Por exemplo, se quisermos criar um vetor com todas as combinações entre `c('a' , 'b')` e `c('A', 'B')` como `c('a-A', 'a-B', ...)`, podemos escrever:

```
# set char vecs
my.vec.1 <- c('a','b')
my.vec.2 <- c('A','B')

# combine in matrix
comb.mat <- outer(my.vec.1,
                  my.vec.2,
                  paste,sep = '-')

# print it!
print(comb.mat)
```

```
##      [,1]  [,2]
## [1,] "a-A" "a-B"
## [2,] "b-A" "b-B"
```

A saída de `outer` é um objeto do tipo matriz. Se quisermos mudar `comb.mat` para um vetor atômico, podemos usar a função `as.character`:

```
print(as.character(comb.mat))
```

```
## [1] "a-A" "b-A" "a-B" "b-B"
```

Outra maneira de atingir o mesmo objetivo é usar a função `expand.grid`. Veja o próximo exemplo.

```
# create df with all combinations
my.df <- expand.grid(my.vec.1, my.vec.2)

# print df
print(my.df)
```

```
##   Var1 Var2
## 1    a    A
## 2    b    A
## 3    a    B
## 4    b    B
```

```
# paste columns together
my.comb.vec <- str_c(my.df$Var1, my.df$Var2, sep = '-')
```

******ebook converter DEMO Watermarks*******

```
# print result
print(my.comb.vec)
```

```
## [1] "a-A" "b-A" "a-B" "b-B"
```

Aqui, usamos a função `expand.grid` para criar um dataframe contendo todas as combinações possíveis de `my.vec.1` e `my.vec.2`. Posteriormente, colamos o conteúdo das colunas do dataframe usando `str_c`.

7.2.9 Codificação de Objetos `character`

Para o R, um *string* de texto é apenas uma sequência de *bytes*. A tradução de *bytes* para caracteres é realizada de acordo com alguma estrutura de codificação. Para a maioria dos casos de uso do R, especialmente em países de língua inglesa, a codificação de caracteres não é um problema pois os textos importados no R já possuem a codificação correta. Ao lidar com dados de texto em diferentes idiomas, tal como Português do Brasil, a codificação de caracteres é algo que você deve entender pois eventualmente precisará lidar com isso.

Vamos explorar um exemplo. Aqui, vamos importar dados de um arquivo de texto com a codificação `'ISO-8859-9'` e verificar o resultado.

```
# read text file
my.char <- readLines('data/FileWithLatinChar_ISO-8859-9.txt')

# print it
print(my.char)
```

```
## [1] "A casa \xe9 bonita e tem muito espa\xe7o"
```

O conteúdo original do arquivo é um texto em português. Como você pode ver, a saída de `readLines` mostra todos os caracteres latinos com símbolos estranhos. Isso ocorre pois a codificação foi manualmente trocada para `'ISO-8859-9'`, enquanto a função `readLines` utiliza `'UTF-8'` como *default*. A solução mais fácil e direta é modificar a codificação do arquivo antes de lê-lo no R. Essa mudança pode ser feita por programas externos no Windows, tal como o notepad++. Veja a seguir, onde importamos um arquivo com a codificação correta (`'UTF-8'`):

```
# read text file with utf-8
my.char <- readLines('data/FileWithLatinChar_UTF-8.txt')

# print it
print(my.char)
```

```
## [1] "A casa é bonita e tem muito espaço"
```

Os caracteres latinos agora estão corretos pois a codificação *default* em `readLines` é a mesma do arquivo, `'UTF-8'`. Uma boa política neste tópico é sempre verificar a codificação de arquivos de texto importados e combiná-lo em R. A maioria das funções de importação tem uma opção para fazê-lo. Quando possível, sempre dê preferência para `'UTF-8'`. Caso necessário, programas de edição de texto, tal como o notepad++, possuem ferramentas para verificar e trocar a codificação de um arquivo.

******ebook converter DEMO Watermarks*******

Quanto aos objetos de texto disponíveis no ambiente, você pode usar a função `Encoding` para verificar e configurar a codificação:

```
# read text file
my.char <- readLines('data/FileWithLatinChar_ISO-8859-9.txt')

# show its encoding
print(Encoding(my.char))
```

```
## [1] "unknown"
```

```
# change encoding
Encoding(my.char) <- 'UTF-8'

# show its encoding
print(Encoding(my.char))
```

```
## [1] "UTF-8"
```

Depois de ler o conteúdo de `data/FileWithLatinChar_ISO-8859-9`, mudamos a codificação do objeto de saída usando a função `Encoding`. Vale destacar que, assim como as demais, esta função também funciona em vetores.

7.2.10 Outras Funções Úteis

`stringr::str_to_lower/base::tolower` - Converte um objeto de texto para letras minúsculas.

```
print(stringr::str_to_lower('ABC'))
```

```
## [1] "abc"
```

`stringr::str_to_upper/base::toupper` - Convertem um texto em letras maiúsculas.

```
print(toupper('abc'))
```

```
## [1] "ABC"
```

```
print(stringr::str_to_upper('abc'))
```

```
## [1] "ABC"
```

******ebook converter DEMO Watermarks*******

7.3 Fatores

A classe de fatores (`factor`) é utilizada para representar grupos ou categorias dentro de uma base de dados no formato tabular. Por exemplo, imagine um banco de informações com os gastos de diferentes pessoas ao longo de um ano. Nessa base de dados existe um item que define o gênero do indivíduo: masculino ou feminino (M ou F). Essa respectiva coluna pode ser importada e representada como texto, porém, no R, a melhor maneira de representá-la é através do objeto fator, uma vez que a mesma representa uma categoria.

A classe de fatores oferece um significado especial para denotar grupos dentro dos dados. Essa organização é integrada aos pacotes e facilita muito a vida do usuário. Por exemplo, caso quiséssemos criar um gráfico para cada grupo dentro da nossa base de dados, poderíamos fazer o mesmo simplesmente indicando a variável de fator para a função de criação da figura. Outra possibilidade é determinar se as diferentes médias de uma variável numérica são estatisticamente diferentes para os grupos dos nossos dados. Podemos também estimar um determinado modelo estatístico para cada grupo. Quando os dados de categorias são representados apropriadamente, o uso das funções do R torna-se mais fácil e eficiente.

7.3.1 Criando Fatores

A criação de fatores dá-se através da função `factor`:

```
my.factor <- factor(c('M','F','M','M','F'))
print(my.factor)
```

```
## [1] M F M M F
## Levels: F M
```

Observe, no exemplo anterior, que a apresentação de fatores com a função `print` mostra os seus elementos e também o item chamado `Levels`. Esse último identifica os possíveis grupos que abrangem o vetor - nesse caso apenas `M` e `F`. Se tivéssemos um número maior de grupos, o item `Levels` aumentaria. Veja a seguir:

```
my.factor <- factor(c('M','F','M','M','F','ND'))
print(my.factor)
```

```
## [1] M  F  M  M  F  ND
## Levels: F M ND
```

Um ponto importante na criação de fatores é que os `Levels` são inferidos através dos dados criados, e isso pode não corresponder à realidade. Por exemplo, observe o seguinte exemplo:

```
my.status <- factor(c('Solteiro', 'Solteiro', 'Solteiro'))
print(my.status)
```

```
## [1] Solteiro Solteiro Solteiro
## Levels: Solteiro
```

Nota-se que, por ocasião, os dados mostram apenas uma categoria: `Solteiro`. Entretanto, sabe-se

******ebook converter DEMO Watermarks*******

que outra categoria do tipo `Casado` é esperada. No caso de utilizarmos o objeto `my.status` da maneira que foi definida anteriormente, omitiremos a informação de outros gêneros, e isso pode ocasionar problemas no futuro tal como a criação de gráficos incompletos. Nessa situação, o correto é definir os `Levels` manualmente da seguinte maneira:

```
my.status <- factor(c('Solteiro', 'Solteiro', 'Solteiro'),
                    levels = c('Solteiro', 'Casado'))
print(my.status)
```

```
## [1] Solteiro Solteiro Solteiro
## Levels: Solteiro Casado
```

7.3.2 Modificando Fatores

Um ponto importante sobre os objetos do tipo fator é que seus `Levels` são imutáveis e não atualizam-se com a entrada de novos dados. Em outras palavras, não é possível modificar os valores dos `Levels` após a criação do objeto. Toda nova informação que não for compatível com os `Levels` do objeto será transformada em `NA` (*Not available*) e uma mensagem de `warning` irá aparecer na tela. Essa limitação pode parecer estranha porém, na prática, ela evita possíveis erros no código. Veja o exemplo a seguir:

```
# set factor
my.factor <- factor(c('a', 'b', 'a', 'b'))

# change first element of a factor to 'c'
my.factor[1] <- 'c'
```

```
## Warning in `[<-.factor`(`*tmp*`, 1, value = "c"): invalid
## factor level, NA generated
```

```
# print result
print(my.factor)
```

```
## [1] <NA> b    a    b
## Levels: a b
```

Nesse caso, a maneira correta de proceder é primeiro transformar o objeto da classe fator para a classe caractere e depois realizar a conversão:

```
# set factor
my.factor <- factor(c('a', 'b', 'a', 'b'))

# change factor to character
my.char <- as.character(my.factor)

# change first element
my.char[1] <- 'c'

# mutate it back to class factor
my.factor <- factor(my.char)

# show result
print(my.factor)
```

```
## [1] c b a b
```

******ebook converter DEMO Watermarks*******

```
## Levels: a b c
```

Utilizando essas etapas temos o resultado desejado no vetor `my.factor`, com a definição de três `Levels`: `a`, `b` e `c`.

O universo `tidyverse` também possui um pacote próprio para manipular fatores, o `forcats`. Para o problema atual de modificação de fatores, podemos utilizar função `forcats::fct_recode`. Veja um exemplo a seguir, onde trocamos as siglas dos fatores:

```
library(forcats)

# set factor
my.fct <- factor(c('A', 'B', 'C', 'A', 'C', 'M', 'N'))

# modify factors
my.fct <- fct_recode(my.fct,
                     'D' = 'A',
                     'E' = 'B',
                     'F' = 'C')

# print result
print(my.fct)
```

```
## [1] D E F D F M N
## Levels: D E F M N
```

Observe como o uso da função `forcats::fct_recode` é intuitivo. Basta indicar o novo nome dos grupos com o operador de igualdade.

7.3.3 Convertendo Fatores para Outras Classes

Outro ponto importante no uso de fatores é a sua conversão para outras classes, especialmente a numérica. Quando convertemos um objeto de tipo fator para a classe caractere, o resultado é o esperado:

```
# create factor
my.char <-factor(c('a', 'b', 'c'))

# convert and print
print(as.character(my.char))
```

```
## [1] "a" "b" "c"
```

Porém, quando fazemos o mesmo procedimento para a classe numérica, o que o R retorna é longe do esperado:

```
# set factor
my.values <- factor(5:10)

# convert to numeric (WRONG)
print(as.numeric(my.values))
```

```
## [1] 1 2 3 4 5 6
```

******ebook converter DEMO Watermarks*******

Esse resultado pode ser explicado pelo fato de que, internamente, fatores são armazenados como índices, indo de 1 até o número total de `Levels`. Essa simplificação minimiza o uso da memória do computador. Quando pedimos ao R para transformar esses fatores em números, ele entende que buscamos o número do índice e não do valor. Para contornar, é fácil: basta transformar o objeto fator em caractere e, depois, em numérico, conforme mostrado a seguir:

```
# converting factors to character and then to numeric
print(as.numeric(as.character(my.values)))
```

```
## [1]  5  6  7  8  9 10
```

7.3.4 Criando Tabelas de Contingência

Após a criação de um fator, podemos calcular a ocorrência de cada fator com a função `table`. Essa também é chamada de tabela de contingência. Em um caso simples, com apenas um fator, a função `table` conta o número de ocorrências de cada categoria, como a seguir:

```
# create factor
my.factor <- factor(sample(c('Pref', 'Ord'),
                           size = 20,
                           replace = TRUE))

# print contingency table
print(table(my.factor))
```

```
## my.factor
##  Ord Pref
##   12    8
```

Um caso mais avançado do uso de `table` é utilizar mais de um fator para a criação da tabela. Veja o exemplo a seguir:

```
# set factors
my.factor.1 <- factor(sample(c('Pref', 'Ord'),
                           size = 20,
                           replace = TRUE))

my.factor.2 <- factor(sample(paste('Grupo', 1:3),
                           size = 20,
                           replace = TRUE))

# print contingency table with two factors
print(table(my.factor.1, my.factor.2))
```

```
##            my.factor.2
## my.factor.1 Grupo 1 Grupo 2 Grupo 3
##        Ord        2       3       3
##        Pref       4       5       3
```

A tabela criada anteriormente mostra o número de ocorrências para cada combinação de fator. Essa é uma ferramenta descritiva simples, mas bastante informativa para a análise de grupos de dados.

7.3.5 Outras Funções

******ebook converter DEMO Watermarks*******

levels - Retorna os `Levels` de um objeto da classe fator.

```
my.factor <- factor(c('A', 'A', 'B', 'C', 'B'))
print(levels(my.factor))
```

```
## [1] "A" "B" "C"
```

as.factor - Transforma um objeto para a classe fator.

```
my.y <- c('a','b', 'c', 'c', 'a')
my.factor <- as.factor(my.y)
print(my.factor)
```

```
## [1] a b c c a
## Levels: a b c
```

split - Com base em um objeto de fator, cria uma lista com valores de outro objeto. Esse comando é útil para separar dados de grupos diferentes e aplicar alguma função com `sapply` ou `lapply`.

```
my.factor <- factor(c('A','B','C','C','C','B'))
my.x <- 1:length(my.factor)

my.l <- split(x = my.x, f = my.factor)

print(my.l)
```

```
## $A
## [1] 1
##
## $B
## [1] 2 6
##
## $C
## [1] 3 4 5
```

******ebook converter DEMO Watermarks*******

7.4 Valores Lógicos

Testes lógicos em dados são centrais no uso do R. Em uma única linha de código podemos testar condições para uma grande quantidade de casos. Esse cálculo é muito utilizado para encontrar casos extremos nos dados (*outliers*) e também para separar diferentes amostras de acordo com algum critério.

7.4.1 Criando Valores Lógicos

Em uma sequência de 1 até 10, podemos verificar quais são os elementos maiores que 5 com o seguinte código:

```
# set numerical
my.x <- 1:10

# print a logical test
print(my.x > 5)
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
## [10]  TRUE
```

```
# print position of elements from logical test
print(which(my.x > 5))
```

```
## [1]  6  7  8  9 10
```

A função `which` do exemplo anterior retorna os índices onde a condição é verdadeira (`TRUE`). O uso do `which` é recomendado quando se quer saber a posição de elementos que satisfazem alguma condição.

Para realizar testes de igualdade, basta utilizar o símbolo de igualdade duas vezes (==).

```
# create char
my.char <- rep(c('abc','bcd'),5)

# print its contents
print(my.char)
```

```
##  [1] "abc" "bcd" "abc" "bcd" "abc" "bcd" "abc" "bcd" "abc"
## [10] "bcd"
```

```
# print logical test
print(my.char=='abc')
```

```
##  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
## [10] FALSE
```

Para o teste de inigualdades, utilizamos o símbolo `!=`:

```
# print inequality test
print(my.char!='abc')
```

```
##  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

******ebook converter DEMO Watermarks*******

```
## [10]  TRUE
```

Destaca-se que também é possível testar condições múltiplas, isto é, a ocorrência simultânea de eventos. Utilizamos o operador `&` para esse propósito. Por exemplo: se quiséssemos verificar quais são os valores de uma sequência de 1 a 10 que são maiores que 4 **e** menores que 7, escreveríamos:

```
my.x <- 1:10

# print logical for values higher than 4 and lower than 7
print((my.x > 4)&(my.x < 7) )
```

```
##  [1] FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE
## [10] FALSE
```

```
# print the actual values
idx <- which( (my.x > 4)&(my.x < 7) )
print(my.x[idx])
```

```
## [1] 5 6
```

Para testar condições não simultâneas, isto é, ocorrências de um ou outro evento, utilizamos o operador `|`. Por exemplo: considerando a sequência anterior, acharíamos os valores maiores que 7 **ou** menores que 4 escrevendo:

```
# location of elements higher than 7 or lower than 4
idx <- which( (my.x > 7)|(my.x < 4) )

# print elements from previous condition
print(my.x[idx])
```

```
## [1]  1  2  3  8  9 10
```

Observe que, em ambos os casos de uso de testes lógicos, utilizamos parênteses para encapsular as condições lógicas. Poderíamos ter escrito `idx <- which( my.x > 7|my.x < 4 )`, porém o uso do parênteses deixa o código mais claro ao isolar os testes de condições e sinalizar que o resultado da operação será um vetor lógico. Em alguns casos, porém, o uso do parênteses indica hierarquia na ordem das operações e portanto não pode ser ignorado.

Outro uso interessante de objetos lógicos é o teste para saber se um item ou mais pertence a um vetor ou não. Para isso utilizamos o operador `%in%`. Por exemplo, imagine que tens os *tickers* de duas ações, `c('ABC', 'DEF')` e queres saber se é possível encontrar esses tickers na coluna de outra base de dados. Essa é uma operação semelhante ao uso do teste de igualdade, porém em notação vetorial. Veja um exemplo a seguir:

```
library(dplyr)
# location of elements higher than 7 or lower than 4
my.tickers <- c('ABC', 'DEF')

# set df
n.obs <- 100
df.temp <- data_frame(tickers = sample(c('ABC', 'DEF', 'GHI', 'JKL'),
                                       size = n.obs,
                                       replace = TRUE),
```

******ebook converter DEMO Watermarks*******

```
                    ret = rnorm(n.obs, sd = 0.05) )

# find rows with selected tickers
idx <- df.temp$tickers %in% my.tickers

# print elements from previous condition
glimpse(df.temp[idx, ])

## Observations: 44
## Variables: 2
## $ tickers <chr> "ABC", "DEF", "DEF", "DEF", "DEF", "DEF...
## $ ret     <dbl> -0.0234343600, -0.0787086283, -0.005187...
```

O dataframe mostrado na tela possui dados apenas para ações em `my.tickers`.

******ebook converter DEMO Watermarks*******

7.5 Datas e Tempo

A representação e manipulação de datas é um importante aspecto das pesquisas em Finanças e Economia. Manipular datas e horários de forma correta, levando em conta mudanças decorridas de horário de verão, feriados locais, em diferentes zonas de tempo, não é uma tarefa fácil! Felizmente, o R possui um aparato completo para qualquer tipo de operação com datas e tempo.

Nesta seção estudaremos as funções e classes nativas que representam e manipulam o tempo em R. Aqui, daremos prioridade as funções do pacote `lubridate` (Grolemund and Wickham 2011). Existem, no entanto, muitos pacotes que podem ajudar o usuário a processar objetos do tipo data e tempo de forma mais avançada. Caso alguma operação com data e tempo não for encontrada aqui, sugiro o estudo dos pacotes `chron` (James and Hornik 2017), `timeDate` (Team et al. 2015) e `bizdays` (Freitas 2016).

Antes de começarmos, vale relembrar que toda data no R segue o formato ISO 8601. Datas seguem o padrão `YYYY-MM-DD`, onde `YYYY` é o ano em quatro números, `MM` é o mês e `DD` é o dia. Por exemplo, uma data em ISO 8601 é 2018-06-10. Deves familiarizar-se com esse formato pois toda importação de dados com formato de datas diferente desta notação exigirá conversão. Felizmente, essa operação é bastante simples de executar com o `lubridate`.

7.5.1 Criando Datas Simples

No R, existem diversas classes que podem representar datas. A escolha entre uma classe de datas e outra baseia-se na necessidade da pesquisa. Em muitas situações não é necessário saber o horário, enquanto que em outras isso é extremamente pertinente pois os dados são coletados ao longo de um dia.

A classe mais básica de datas é `Date`. Essa indica dia, mês e ano, apenas. No `lubridate`, criamos datas verificando o formato da data de entrada e as funções `ymd` (year-month-date), `dmy` (day-month-year) e `mdy` (month-day-year). Veja a seguir:

```
library(lubridate)

# set Date object
print(ymd('2016-06-24'))
```

```
## [1] "2016-06-24"
```

```
# set Date object
print(dmy('24-06-2016'))
```

```
## [1] "2016-06-24"
```

```
# set Date object
print(mdy('06-24-2016'))
```

```
## [1] "2016-06-24"
```

Note que as funções retornam exatamente o mesmo objeto. A diferença no uso é somente pela

******ebook converter DEMO Watermarks*******

forma que a data de entrada está estruturada com a posição do dia, mês e ano.

Um benefício no uso das funções do pacote `lubridate` é que as mesmas são inteligentes ao lidar com formatos diferentes. Observe no caso anterior que definimos os elementos das datas com o uso do separador - e valores numéricos. Outros formatos também são automaticamente reconhecidos:

```
# set Date object
print(ymd('2016/06/24'))
```

```
## [1] "2016-06-24"
```

```
# set Date object
print(ymd('2016&06&24'))
```

```
## [1] "2016-06-24"
```

```
# set Date object
print(ymd('2017 june 24'))
```

```
## [1] "2017-06-24"
```

```
# set Date object
print(dmy('24 of june 2017'))
```

```
## [1] "2017-06-24"
```

Isso é bastante útil pois o formato de datas no Brasil é `dia/mês/ano` (DD/MM/YYYY). Ao usar `dmy` para uma data brasileira, a conversão é correta:

```
# set Date from dd/mm/yyyy
my.date <- dmy('24/06/2016')

# print result
print(my.date)
```

```
## [1] "2016-06-24"
```

Já no pacote `base`, a função correspondente é `as.Date`. O formato da data, porém, deve ser explicitamente definido com argumento `format`, conforme mostrado a seguir:

```
# set Date from dd/mm/yyyy with the definition of format
my.date <- as.Date('24/06/2016', format = '%d/%m/%Y')

# print result
print(my.date)
```

```
## [1] "2016-06-24"
```

Os símbolos utilizados na entrada `format`, tal como `%d` e `%Y`, são indicadores de formato, os quais definem a forma em que a data a ser convertida está estruturada. Nesse caso, os símbolos `%Y`, `%m` e `%d` definem ano, mês e dia, respectivamente. Existem diversos outros símbolos que podem ser utilizados para processar datas em formatos específicos. Um panorama das principais codificações é apresentado a seguir:

******ebook converter DEMO Watermarks*******

Código	Valor	Exemplo
%d	dia do mês (decimal)	0
%m	mês (decimal)	12
%b	mês (abreviado)	Abr
%B	mês (nome completo)	Abril
%y	ano (2 dígitos)	16
%Y	ano (4 dígitos)	2016

Os símbolos anteriores permitem a criação de datas a partir de variados formatos. Observe como a utilização das funções do `lubridate`, em relação a `base`, são mais simples e fáceis de utilizar, justificando a nossa escolha.

7.5.2 Criando Sequências de Datas

Um aspecto interessante no uso de objetos do tipo `Date` é que eles interagem com operações de adição de valores numéricos e com testes lógicos de comparação de datas. Por exemplo: caso quiséssemos adicionar um dia à data `my.date` criada anteriormente, bastaria somar o valor 1 ao objeto:

```
# create date
my.date <- ymd('2016-06-24')

# find next day
my.date.2 <- my.date + 1

# print result
print(my.date.2)
```

```
## [1] "2016-06-25"
```

A propriedade também funciona com vetores, o que deixa a criação de sequências de datas muito fácil. Nesse caso, o próprio R encarrega-se de verificar o número de dias em cada mês.

```
# create a sequence of Dates
my.date.vec <- my.date + 0:15

# print it
print(my.date.vec)
```

```
##  [1] "2016-06-24" "2016-06-25" "2016-06-26" "2016-06-27"
##  [5] "2016-06-28" "2016-06-29" "2016-06-30" "2016-07-01"
##  [9] "2016-07-02" "2016-07-03" "2016-07-04" "2016-07-05"
## [13] "2016-07-06" "2016-07-07" "2016-07-08" "2016-07-09"
```

Uma maneira mais customizável de criar sequências de datas é utilizar a função `seq`. Com ela, é possível definir intervalos diferentes de tempo e até mesmo o tamanho do vetor de saída. Caso quiséssemos uma sequência de datas de dois em dois dias, poderíamos utilizar o seguinte código:

```
# set first and last Date
my.date.1 <- ymd('2017-03-07')
my.date.2 <- ymd('2017-03-20')
```

******ebook converter DEMO Watermarks*******

```
# set sequence
my.vec.date <- seq(from = my.date.1,
                   to = my.date.2,
                   by = '2 days')

# print result
print(my.vec.date)

## [1] "2017-03-07" "2017-03-09" "2017-03-11" "2017-03-13"
## [5] "2017-03-15" "2017-03-17" "2017-03-19"
```

Caso quiséssemos de duas em duas semanas, escreveríamos:

```
# set first and last Date
my.date.1 <- ymd('2017-03-07')
my.date.2 <- ymd('2017-04-20')

# set sequence
my.vec.date <- seq(from = my.date.1,
                   to = my.date.2,
                   by = '2 weeks')

# print result
print(my.vec.date)

## [1] "2017-03-07" "2017-03-21" "2017-04-04" "2017-04-18"
```

Outra forma de utilizar `seq` é definir o tamanho desejado do objeto de saída. Por exemplo, caso quiséssemos um vetor de datas com 10 elementos, usaríamos:

```
# set first and last Date
my.date.1 <- ymd('2017-03-07')
my.date.2 <- ymd('2017-10-20')

# set sequence
my.vec.date <- seq(from = my.date.1,
                   to = my.date.2,
                   length.out = 10)

# print result
print(my.vec.date)

##  [1] "2017-03-07" "2017-04-01" "2017-04-26" "2017-05-21"
##  [5] "2017-06-15" "2017-07-11" "2017-08-05" "2017-08-30"
##  [9] "2017-09-24" "2017-10-20"
```

O intervalo entre as datas em `my.vec.date` é definido automaticamente pelo R.

7.5.3 Operações com Datas

É possível descobrir a diferença de dias entre datas simplesmente diminuindo uma data da outra:

```
# set dates
my.date.1 <- ymd('2015-06-24')
my.date.2 <- ymd('2016-06-24')
```

******ebook converter DEMO Watermarks*******

```
# calculate difference
diff.date <- my.date.2 - my.date.1

# print result
print(diff.date)
```

```
## Time difference of 366 days
```

A saída da operação de subtração é um objeto da classe `diffdate`, o qual possui a classe de lista como sua estrutura básica. Destaca-se que a notação de acesso aos elementos da classe `diffdate` é a mesma utilizada para listas. O valor numérico do número de dias está contido no primeiro elemento de `diff.date`:

```
# print difference of days as numerical value
print(diff.date[[1]])
```

```
## [1] 366
```

Podemos testar se uma data é maior do que outra com o uso das operações de comparação:

```
# set date and vector
my.date.1 <- ymd('2016-06-20')
my.date.vec <- ymd('2016-06-20') + seq(-5,5)

# test which elements of my.date.vec are older than my.date.1
my.test <- (my.date.vec > my.date.1)

# print result
print(my.test)
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
## [10]  TRUE  TRUE
```

A operação anterior é bastante útil quando se está buscando filtrar um determinado período de tempo nos dados. Nesse caso, basta buscar nas datas o período específico em que estamos interessados e utilizar o objeto lógico da comparação para selecionar os elementos. Veja o exemplo a seguir:

```
library(dplyr)
library(lubridate)

# set first and last dates
first.date <- ymd('2016-06-01')
last.date <- ymd('2016-06-15')

# create dataframe and glimpse it
my.temp.df <- data_frame(date.vec = ymd('2016-05-25') + seq(0,30),
                         prices=seq(1,10,
                                    length.out = length(date.vec)))

glimpse(my.temp.df)
```

```
## Observations: 31
## Variables: 2
## $ date.vec <date> 2016-05-25, 2016-05-26, 2016-05-27, 2...
## $ prices   <dbl> 1.0, 1.3, 1.6, 1.9, 2.2, 2.5, 2.8, 3.1...
```

******ebook converter DEMO Watermarks*******

```
# find dates that are between the first and last date
my.idx <- (my.temp.df$date.vec >= first.date) &
    (my.temp.df$date.vec <= last.date)

# use index to filter dataframe
my.temp.df.filtered <- my.temp.df %>%
          filter(my.idx) %>%
          glimpse()
```

```
## Observations: 15
## Variables: 2
## $ date.vec <date> 2016-06-01, 2016-06-02, 2016-06-03, 2...
## $ prices   <dbl> 3.1, 3.4, 3.7, 4.0, 4.3, 4.6, 4.9, 5.2...
```

Nesse caso, o vetor final de preços `my.prices` contém apenas informações para o período entre `first.date` e `last.date`.

7.5.4 Lidando com Data e Tempo

O uso da classe `Date` é suficiente quando se está lidando apenas com datas. Em casos em que é necessário levar em consideração o horário, temos que utilizar um objeto do tipo `datetime`.

No pacote `base`, uma das classes utilizadas para esse fim é a `POSIXlt`, a qual armazena o conteúdo de uma data na forma de uma lista. Outra classe que também é possível utilizar é a `POSIXct`, que armazena as datas como segundos contados a partir de `1970-01-01`. Devido ao seu formato de armazenamento, a classe `POSIXct` ocupa menos memória do computador. Junto ao `lubridate`, a classe utilizada para representar data-tempo é `POSIXct` e portanto daremos prioridade a essa. Vale destacar que todos os exemplos apresentados aqui também podem ser replicados para objetos do tipo `POSIXlt`.

O formato tempo/data também segue a norma ISO 8601, sendo representado como `ano-mês-dia horas:minutos:segundos zonadetempo` (YYYY-MM-DD HH:mm:SS TMZ). Veja o exemplo a seguir:

```
# creating a POSIXct object
my.timedate <- as.POSIXct('2016-01-01 16:00:00')

# print result
print(my.timedate)
```

```
## [1] "2016-01-01 16:00:00 -02"
```

O pacote `lubridate` também oferece funções inteligentes para a criação de objetos do tipo data-tempo. Essas seguem a mesma linha de raciocínio que as funções de criar datas. Veja a seguir:

```
library(lubridate)

# creating a POSIXlt object
my.timedate <- ymd_hms('2016-01-01 16:00:00')

# print it
print(my.timedate)
```

```
## [1] "2016-01-01 16:00:00 UTC"
```

******ebook converter DEMO Watermarks*******

Destaca-se que essa classe adiciona automaticamente o fuso horário. Caso seja necessário representar um fuso diferente, é possível fazê-lo com o argumento `tz`:

```
# creating a POSIXlt object with custom timezone
my.timedate.tz <- ymd_hms('2016-01-01 16:00:00',
                          tz = 'GMT')

# print it
print(my.timedate.tz)
```

```
## [1] "2016-01-01 16:00:00 GMT"
```

É importante ressaltar que, para o caso de objetos do tipo `POSIXlt` e `POSIXct`, **as operações de soma e diminuição referem-se a segundos** e não dias, como no caso do objeto da classe `Date`.

```
# Adding values (seconds) to a POSIXlt object and printing it
print(my.timedate.tz + 30)
```

```
## [1] "2016-01-01 16:00:30 GMT"
```

Assim como para a classe `Date`, existem símbolos específicos para lidar com componentes de um objeto do tipo data/tempo. Isso permite a formatação customizada de datas. A seguir, apresentamos um quadro com os principais símbolos e os seus respectivos significados.

Código	Valor	Exemplo
%H	Hora (decimal, 24 horas)	23
%I	Hora (decimal, 12 horas)	11
%M	Minuto (decimal, 0-59)	12
%p	Indicador AM/PM	AM
%S	Segundos (decimal, 0-59)	50

A seguir veremos como utilizar essa tabela para customizar datas.

7.5.5 Personalizando o Formato de Datas

A notação básica para representar datas e data/tempo no R pode não ser a ideal em algumas situações. No Brasil, por exemplo, indicar datas no formato `YYYY-MM-DD` pode gerar bastante confusão. É recomendado, portanto, modificar a representação das datas para o formato esperado, isto é, `DD/MM/YYYY`.

Para formatar uma data, utilizamos a função `format`. Seu uso baseia-se nos símbolos de data e de horário apresentados anteriormente. A partir desses, pode-se criar qualquer customização. Veja o exemplo a seguir, onde apresenta-se a modificação de um vetor de datas para o formato brasileiro:

```
# create vector of dates
my.dates <- seq(from = ymd('2016-01-01'),
                to = ymd('2016-01-15'),
                by = '1 day')
```

******ebook converter DEMO Watermarks*******

```
# change format
my.dates.brformat <- format(my.dates, '%d/%m/%Y')

# print result
print(my.dates.brformat)
```

```
##  [1] "01/01/2016" "02/01/2016" "03/01/2016" "04/01/2016"
##  [5] "05/01/2016" "06/01/2016" "07/01/2016" "08/01/2016"
##  [9] "09/01/2016" "10/01/2016" "11/01/2016" "12/01/2016"
## [13] "13/01/2016" "14/01/2016" "15/01/2016"
```

O mesmo procedimento pode ser realizado para objetos do tipo data/tempo (`POSIXct`):

```
# create vector of date-time
my.datetime <- ymd_hms('2016-01-01 12:00:00') + seq(0,560,60)

# change to Brazilian format
my.dates.brformat <- format(my.datetime, '%d/%m/%Y %H:%M:%S')

# print result
print(my.dates.brformat)
```

```
##  [1] "01/01/2016 12:00:00" "01/01/2016 12:01:00"
##  [3] "01/01/2016 12:02:00" "01/01/2016 12:03:00"
##  [5] "01/01/2016 12:04:00" "01/01/2016 12:05:00"
##  [7] "01/01/2016 12:06:00" "01/01/2016 12:07:00"
##  [9] "01/01/2016 12:08:00" "01/01/2016 12:09:00"
```

Pode-se também customizar para formatos bem específicos. Veja a seguir:

```
# set custom format
my.dates.myformat <- format(my.dates,
                            'Year=%Y | Month=%m | Day=%d')

# print result
print(my.dates.myformat)
```

```
##  [1] "Year=2016 | Month=01 | Day=01"
##  [2] "Year=2016 | Month=01 | Day=02"
##  [3] "Year=2016 | Month=01 | Day=03"
##  [4] "Year=2016 | Month=01 | Day=04"
##  [5] "Year=2016 | Month=01 | Day=05"
##  [6] "Year=2016 | Month=01 | Day=06"
##  [7] "Year=2016 | Month=01 | Day=07"
##  [8] "Year=2016 | Month=01 | Day=08"
##  [9] "Year=2016 | Month=01 | Day=09"
## [10] "Year=2016 | Month=01 | Day=10"
## [11] "Year=2016 | Month=01 | Day=11"
## [12] "Year=2016 | Month=01 | Day=12"
## [13] "Year=2016 | Month=01 | Day=13"
## [14] "Year=2016 | Month=01 | Day=14"
## [15] "Year=2016 | Month=01 | Day=15"
```

7.5.6 Extraindo Elementos de uma Data

Para extrair elementos de datas tal como o ano, mês, dia, hora, minuto e segundo, uma alternativa é utilizar função `format`. Observe o próximo exemplo, onde recuperamos apenas as

******ebook converter DEMO Watermarks*******

horas de um objeto `POSIXct`:

```
library(lubridate)

# create vector of date-time
my.datetime <- seq(from = ymd_hms('2016-01-01 12:00:00'),
                   to = ymd_hms('2016-01-01 18:00:00'),
                   by = '1 hour')

# get hours from POSIXlt
my.hours <- format(my.datetime, '%H')

# print result
print(my.hours)
```

```
## [1] "12" "13" "14" "15" "16" "17" "18"
```

Da mesma forma, poderíamos utilizar os símbolos `%M` e `%S` para recuperar facilmente minutos e segundos de um vetor de objetos `POSIXct`.

```
# create vector of date-time
my.datetime <- seq(from = ymd_hms('2016-01-01 12:00:00'),
                   to = ymd_hms('2016-01-01 18:00:00'),
                   by = '15 min')

# get minutes from POSIXlt
my.minutes <- format(my.datetime, '%M')

# print result
print(my.minutes)
```

```
##  [1] "00" "15" "30" "45" "00" "15" "30" "45" "00" "15" "30"
## [12] "45" "00" "15" "30" "45" "00" "15" "30" "45" "00" "15"
## [23] "30" "45" "00"
```

Outra forma é utilizar as funções do `lubridate`, tal como `hour` e `minute`:

```
# get hours with lubridate
print(hour(my.datetime))
```

```
##  [1] 12 12 12 12 13 13 13 13 14 14 14 14 15 15 15 15 16 16
## [19] 16 16 17 17 17 17 18
```

```
# get minutes with lubridate
print(minute(my.datetime))
```

```
##  [1]  0 15 30 45  0 15 30 45  0 15 30 45  0 15 30 45  0 15
## [19] 30 45  0 15 30 45  0
```

Outras funções também estão disponíveis para os demais elementos de um objeto data-hora.

7.5.7 Conhecendo o Horário e a Data Atual

O R inclui várias funções que permitem o usuário utilizar no seu código o horário e data atual do sistema. Isso é bastante útil quando se está criando registros e é importante que a data e horário de execução do código seja conhecida futuramente.

******ebook converter DEMO Watermarks*******

Para conhecer o dia atual, basta utilizarmos a função `base::Sys.Date` ou `lubridate::today`:

```
library(lubridate)

# get today
print(Sys.Date())
```

```
## [1] "2018-06-10"
```

```
# print it
print(today())
```

```
## [1] "2018-06-10"
```

Para descobrir a data e horário, utilizamos a função `base::Sys.time` ou `lubridate::now`:

```
# get time!
print(Sys.time())
```

```
## [1] "2018-06-10 14:56:32 -03"
```

```
# get time!
print(now())
```

```
## [1] "2018-06-10 14:56:32 -03"
```

Com base nessas, podemos escrever:

```
library(stringr)

# example of log message
my.str <- str_c('This code was executed in ', now())

# print it
print(my.str)
```

```
## [1] "This code was executed in 2018-06-10 14:56:32"
```

7.5.8 Outras Funções Úteis

weekdays - Retorna o dia da semana de uma ou várias datas.

```
# set date vector
my.dates <- seq(from = ymd('2016-01-01'),
                to = ymd('2016-01-5'),
                by = '1 day')

# find corresponding weekdays
my.weekdays <- weekdays(my.dates)

# print it
print(my.weekdays)
```

```
## [1] "Friday"   "Saturday" "Sunday"   "Monday"   "Tuesday"
```

months - Retorna o mês de uma ou várias datas.

******ebook converter DEMO Watermarks*******

```
# create date vector
my.dates <- seq(from = ymd('2016-01-01'),
                to = ymd('2016-12-31'),
                by = '1 month')

# find months
my.months <- months(my.dates)

# print result
print(my.months)
```

```
##  [1] "January"   "February"  "March"     "April"
##  [5] "May"       "June"      "July"      "August"
##  [9] "September" "October"   "November"  "December"
```

quarters - Retorna a localização de uma ou mais datas dentro dos quartis do ano.

```
# get quartiles of the year
my.quarters <- quarters(my.dates)
print(my.quarters)
```

```
##  [1] "Q1" "Q1" "Q1" "Q2" "Q2" "Q2" "Q3" "Q3" "Q3" "Q4" "Q4"
## [12] "Q4"
```

OlsonNames - Retorna um vetor com as zonas de tempo disponíveis no R. No total, são mais de 500 itens. Aqui, apresentamos apenas os primeiros cinco elementos.

```
# get possible timezones
possible.tz <- OlsonNames()

# print it
print(possible.tz[1:5])
```

```
## [1] "Africa/Abidjan"     "Africa/Accra"
## [3] "Africa/Addis_Ababa" "Africa/Algiers"
## [5] "Africa/Asmara"
```

Sys.timezone - Retorna a zona de tempo do sistema.

```
# get current timezone
print(Sys.timezone())
```

```
## [1] "America/Sao_Paulo"
```

cut - Retorna um fator a partir da categorização de uma classe de data e tempo.

```
# set example date vector
my.dates <- seq(from = ymd('2016-01-01'),
                to = ymd('2016-03-01'),
                by = '5 days')

# group vector based on monthly breaks
my.month.cut <- cut(x = my.dates,
                    breaks = 'month',
                    labels = c('Jan', 'Fev', 'Mar'))

# print result
print(my.month.cut)
```

******ebook converter DEMO Watermarks*******

```
##  [1] Jan Jan Jan Jan Jan Jan Jan Fev Fev Fev Fev Fev Mar
## Levels: Jan Fev Mar
```

```r
# set example datetime vector
my.datetime <- as.POSIXlt('2016-01-01 12:00:00') + seq(0,250,15)

# set groups for each 30 seconds
my.cut <- cut(x = my.datetime, breaks = '30 secs')

# print result
print(my.cut)
```

```
##  [1] 2016-01-01 12:00:00 2016-01-01 12:00:00
##  [3] 2016-01-01 12:00:30 2016-01-01 12:00:30
##  [5] 2016-01-01 12:01:00 2016-01-01 12:01:00
##  [7] 2016-01-01 12:01:30 2016-01-01 12:01:30
##  [9] 2016-01-01 12:02:00 2016-01-01 12:02:00
## [11] 2016-01-01 12:02:30 2016-01-01 12:02:30
## [13] 2016-01-01 12:03:00 2016-01-01 12:03:00
## [15] 2016-01-01 12:03:30 2016-01-01 12:03:30
## [17] 2016-01-01 12:04:00
## 9 Levels: 2016-01-01 12:00:00 ... 2016-01-01 12:04:00
```

******ebook converter DEMO Watermarks*******

7.6 Dados Omissos - `NA` (*Not available*)

Uma das principais inovações do R em relação a outras linguagens de programação é a representação de dados omissos através de objetos da classe `NA` (*Not Available*). A falta de dados pode ter inúmeros motivos, tal como a falha na coleta de informações ou simplesmente a não existência dos mesmos. Esses casos são tratados por meio da remoção ou da substituição dos dados omissos antes realizar uma análise mais profunda. A identificação desses casos, portanto, é de extrema importância.

7.6.1 Definindo Valores `NA`

Para definirmos os casos omissos nos dados, basta utilizar o símbolo `NA`:

```
# a vector with NA
my.x <- c(1, 2, NA, 4, 5)

# print it
print(my.x)
```

```
## [1]  1  2 NA  4  5
```

Vale destacar que a operação de qualquer valor `NA` com outro sempre resultará em `NA`.

```
# example of NA interacting with other objects
print(my.x + 1)
```

```
## [1]  2  3 NA  5  6
```

Isso exige cuidado quando se está utilizando alguma função com cálculo recursivo, tal como `cumsum` e `cumprod`. Nesses casos, todo valor consecutivo ao `NA` será transformado em `NA`. Veja os exemplos a seguir com as duas funções:

```
# set vector with NA
my.x <- c(1:5, NA, 5:10)

# print cumsum (NA after sixth element)
print(cumsum(my.x))
```

```
##  [1]  1  3  6 10 15 NA NA NA NA NA NA NA
```

```
# print cumprod (NA after sixth element)
print(cumprod(my.x))
```

```
##  [1]   1   2   6  24 120  NA  NA  NA  NA  NA  NA  NA
```

Portanto, antes de utilizar as funções `cumsum` e `cumprod`, verifique se existe algum valor `NA` no vetor de interesse. Caso estiver encontrando valores `NA` não esperados, deves checar os seus dados de entrada.

7.6.2 Encontrando e Substituindo Valores `NA`

******ebook converter DEMO Watermarks*******

Para encontrar os valores `NA` em um vetor, basta utilizar a função `is.na`:

```
# set vector with NA
my.x <- c(1:2, NA, 4:10)

# find location of NA
idx.na <- is.na(my.x)
print(idx.na)
```

```
##  [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [10] FALSE
```

Para substituí-los, use indexação com a saída de `is.na`:

```
# set vector
my.x <- c(1, NA, 3:4, NA)

# replace NA for 2
my.x[is.na(my.x)] <- 2

# print result
print(my.x)
```

```
## [1] 1 2 3 4 2
```

Outra maneira de limpar o objeto é utilizar a função `na.omit`, que retorna o mesmo objeto mas sem os valores `NA`. Note, porém, que o tamanho do vetor irá mudar e o objeto será da classe `omit`, o que indica que o vetor resultante não inclui os `NA` e apresenta, também, a posição dos elementos `NA` encontrados.

```
# set vector
my.char <- c(letters[1:3], NA, letters[5:8])

# print it
print(my.char)
```

```
## [1] "a" "b" "c" NA  "e" "f" "g" "h"
```

```
# use na.omit to remove NA
my.char <- na.omit(my.char)

# print result
print(my.char)
```

```
## [1] "a" "b" "c" "e" "f" "g" "h"
## attr(,"na.action")
## [1] 4
## attr(,"class")
## [1] "omit"
```

Apesar de o tipo de objeto ter sido trocado, devido ao uso de `na.omit`, as propriedades básicas do vetor inicial se mantêm. Por exemplo: o uso de `nchar` no objeto resultante é possível.

```
# trying nchar on a na.omit object
print(nchar(my.char))
```

```
## [1] 1 1 1 1 1 1 1
```

******ebook converter DEMO Watermarks*******

Para outros objetos, porém, recomenda-se cautela quando no uso da função `na.omit`.

7.6.3 Outras Funções Úteis

complete.cases - Retorna um vetor lógico que indica se as linhas do objeto possuem apenas valores não omissos. Essa função é usada exclusivamente para dataframes e matrizes.

```
# create matrix
my.mat <- matrix(1:15, nrow = 5)

# set an NA value
my.mat[2,2] <- NA

# print index with rows without NA
print(complete.cases(my.mat))
```

```
## [1]  TRUE FALSE  TRUE  TRUE  TRUE
```

******ebook converter DEMO Watermarks*******

7.7 Exercícios

1. Considere os seguintes os vetores x e y:

```
## [1] 2 2 3 1 3
```

```
## [1] 2 1 2 1 1
```

Qual é a soma dos elementos do novo vetor resultante da multiplicação entre os elementos de x e y?

2. Utilizando o R, crie uma sequência em objeto chamado `seq.1` entre -10 e 10, onde o intervalo é igual a 2.

3. Defina outro objeto chamado `seq.2` contendo uma sequência de tamanho 1000, com valores entre 0 e 100.

4. Calcule a soma entre `seq.1` e `seq.2`. Esta operação funcionou apesar do tamanho diferente dos vetores? Explique sua resposta.

5. Caso realizássemos uma soma cumulativa em uma sequência entre 1 e 100, em qual elemento esta soma iria passar de 50?

6. Crie um vetor x de acordo com a fórmula a seguir, onde i=1...100i=1...100. Qual é o valor da soma dos elementos de x?

xi=−1i+12i−1 x_i=\frac{-1^{i+1}}{2i-1}

7. **DESAFIO** - Crie um vetor ziz_i de acordo com a fórmula a seguir onde xi=1...50x_i=1...50 e yi=50...1y_i=50...1. Qual é o valor da soma dos elementos de ziz_i?

zi=yi−xi−1yi−2 z_i=\frac{y_i-x_{i-1}}{y_{i-2}}

8. Crie um objeto chamado `x` com valores aleatórios da distribuição Normal com média igual a 10 e desvio padrão igual a 10. Usando função `cut`, crie outro objeto que defina dois grupos com base em valores de `x` maiores que 10 e menores que 10. Qual a quantidade de observações em cada grupo?

9. Vamos assumir que compraste 100 ações de uma empresa com o valor individual de 15 reais por ação. Após algum tempo, vendeste 30 ações por 18 reais cada uma e o restante das ações foram vendidas por 22 reais. Utilizando um *script* no R, estruture esse problema financeiro com a criação de objetos numéricos no R e, baseado nesses, calcule o lucro bruto desta operação no mercado de ações. Qual é o resultado?

10. Crie o seguinte objeto com o código a seguir:

```
set.seed(10)
my.char <- paste(sample(letters,5000,replace = T), collapse = '')
```

******ebook converter DEMO Watermarks*******

Qual a quantidade de vezes que a letra `'x'` é encontrada no objeto de texto resultante?

11. Baseado no objeto `my.char` criado anteriormente, caso dividíssemos o mesmo em diversos pedaços menores utilizando a letra `"b"`, qual é o número de caracteres no maior pedaço encontrado?

12. No endereço https://www.gutenberg.org/files/1342/1342-0.txt é possível acessar um arquivo .txt contendo o texto integral do livro *Pride and Prejudice* de Jane Austen. Utilize funções `download.file` e `readLines` para importar o livro inteiro como um vetor de caracteres chamado `my.imported.lines` no R. Quantas linhas o objeto resultante possui?

13. Junte o vetor de caracteres em `my.imported.lines` para um único valor (texto) em outro objeto chamado `my.full.text` usando função `paste0(my.imported.lines, collapse = '\n')`. Utilizando este último e pacote `stringr`, quantas vezes a palavra `'Bennet'` é repetida na totalidade do texto?

14. **DESAFIO** - Para o objeto `my.full.text`, utilize função `str_split` para *quebrar* o texto inteiro em função de espaços em branco. Com base nesse, crie uma tabela de frequência. Qual a palavra mais utilizada no texto?

15. Qual é a data localizada após 10 dias de 2018-06-10?

16. Utilize o R para construir uma sequência de datas entre `'2010-01-01'` e `'2015-07-11'` separadas a cada três dias. Qual é a quantidade de dias caindo em quartas-feiras no vetor resultante?

******ebook converter DEMO Watermarks*******

8 Programando com o R

Nos capítulos anteriores estudamos o ecossistema de objetos do R e os procedimentos para importar dados no programa, seja de arquivos locais ou pela Internet. Neste capítulo trataremos do tópico de programação com o R. Isso inclui a criação de funções customizadas, a repetição estruturada de códigos (*loops*) e execuções condicionais. Uma seção mostrando a capacidade de um dos mais populates pacotes do CRAN, `dplyr` (Wickham and Francois 2016), também está inclusa. Os conceitos apresentados aqui são de extrema importância. O R oferece um ambiente de programação completo, permiting a resolução de qualquer problema computacional em sua análise de dados.

******ebook converter DEMO Watermarks*******

8.1 Criando Funções

Como já foi destacado anteriormente, **o uso de funções está no centro do R**. Qualquer procedimento pode ser organizado para formar uma função. O uso de funções organiza o código e aumenta a aplicabilidade de procedimentos genéricos. Isso facilita também a correção de erros. Se você estiver usando uma função qualquer em vários locais de um *script*, basta modificar o código em um local, a origem da função. Por exemplo, imagine criar um procedimento que limpa uma base de dados de um dataframe removendo valores extremos e casos omissos (`NA`). Esse procedimento pode ser escrito como uma função e executado para diferentes dataframes. Caso o autor necessite utilizar esse procedimento em outros scripts, bastará carregar a função escrita anteriormente e utilizá-la outra vez.

Uma função pode ser desenvolvida ao longo do tempo, aumentando sua qualidade e aplicabilidade. Caso precisarmos usá-la de maneira ligeiramente diferente, podemos adicionar argumentos que controlam a execução. Ao usar funções no seu trabalho, irás investir menos tempo reescrevendo tarefas repetitivas e mais na escrita de novos procedimentos. Com tempo e experiência, uma coleção de funções personalizadas estará disponível, permitindo que você escreva rotinas para análises complexas em poucos minutos.

Uma função sempre possui três partes conexas: as entradas, o processamento e a saída. As entradas são as informações necessárias para a **solução** oferecida pela função. O processamento é a operação que irá gerar um resultado na saída da função. Por exemplo, uma função que calcula o tamanho de um vetor tem como entrada um vetor qualquer e saída como o número de elementos do vetor de entrada. No R, o esqueleto de definição de uma função é estruturado conforme o código a seguir:

```
my.fct <- function(arg1 = 1, arg2 = 'abc'){

  msg1 <- paste0('\nValue of arg1: ', arg1)
  cat(msg1)

  msg2 <- paste0('\nValue of arg2: ', arg2)
  cat(msg2)

  out <- c(msg1, msg2)
  return(out)

}
```

Após o registro da função no ambiente do R (instruções a seguir), podemos mudar as suas entradas conforme a necessidade. Um exemplo seria chamar `my.fct` com outros argumentos:

```
out1 <- my.fct(arg1 = 2, arg2 = 'bcd')

##
## Value of arg1: 2
## Value of arg2: bcd

print(out1)
```

******ebook converter DEMO Watermarks*******

```
## [1] "\nValue of arg1: 2"   "\nValue of arg2: bcd"
```

```
out2 <- my.fct(arg1 = 10, arg2 = 'dab')
```

```
##
## Value of arg1: 10
## Value of arg2: dab
```

```
print(out2)
```

```
## [1] "\nValue of arg1: 10"  "\nValue of arg2: dab"
```

Observa-se que a definição de uma função é muito próxima da definição de um objeto, porém, o seu conteúdo é encapsulado por chaves (`{ }`). Os itens *arg1* e *arg2* são as entradas da função, ou seja, as informações necessárias para realizar algum procedimento.

O uso de igualdade nessas definições, tal como em `arg1 = 1`, define o caso *default*, isto é, determina qual será o valor padrão de entrada caso o usuário não defina nenhum em sua chamada. O uso de valores *default* é útil para definir a escolha mais provável das entradas de uma função. Na prática, isso facilita o seu uso ao acelerar o processo de chamada por parte do usuário, que não precisa conhecer e inserir *inputs* para todas as possíveis opções de uso da função.

Toda função irá retornar algum objeto com o comando `return`. Esse geralmente encontra-se ao final da função. Não existe restrição sobre o tipo de objeto retornado: pode ser uma lista, um vetor numérico ou um objeto de qualquer outro tipo. Essa flexibilidade permite que o usuário retorne várias informações. Basta organizá-las em uma lista, vetor ou dataframe. O comando `return` define o fim da função e o retorno do objeto criado.

Quanto ao uso da função, primeiro você precisa registrá-la no ambiente do R, executando sua definição como qualquer outro código. Uma maneira simples de fazer isso é colocar o cursor na segunda chave (`}`), encontrada no término da função, e pressionar *control* + *enter*. O uso de *autocomplete*, com tecla *tab*, também funciona para os argumentos da função. Para testar, primeiro registre a função no R, digite seu nome e, entre os dois parênteses da função, aperte *tab*. Uma tela com os argumentos da função deve aparecer na tela do editor.

Os argumentos da função podem ser definidos por posição ou nome. Então, se você chamou a função `my.fct` como `my.fct(1,2)`, o R reconhecerá suas entradas como `arg1 = 1` e `arg2 = 2`. Quando você usa nomes, a posição dos argumentos é irrelevante, isto é, a chamada de função anterior é equivalente a `my.fct(arg2 = 2, arg1 = 1)`.

Agora, vamos criar uma função que faça algo mais útil: aceite como entrada um vetor numérico e tenha como saída a média desse vetor. Esta função já existe no pacote `base`, mas vamos criar a nossa própria versão como exemplo:

```
my.fct <- function(x = c(1, 1, 1, 1)){
  # Calculates the average of input x
  #
  # Args:
  #     x: a numerical vector
  #
  # Returns:
```

******ebook converter DEMO Watermarks*******

```
  #   The mean of x

  out <- sum(x)/length(x)

  return(out)

}
```

Observe como definimos uma seção de comentários após a primeira chave para descrever a função escrita, incluindo seus argumentos e o valor retornado. Lembre-se que a uma função pode ser compartilhada com outras pessoas. Este rápido resumo oferece ao usuário informações sobre como utilizá-la. Segundo o manual de estilo do Google:

> "Functions should contain a comments section immediately below the function definition line. These comments should consist of a one-sentence description of the function; a list of the function's arguments, denoted by Args:, with a description of each (including the data type); and a description of the return value, denoted by Returns:. The comments should be descriptive enough that a caller can use the function without reading any of the function's code."
>
> --- Google's R style manual

Depois de escrever e registrar a função, vamos testá-la:

```
# testing function my.fct
my.mean <- my.fct(x = 1:100)

# print result
print(my.mean)
```

```
## [1] 50.5
```

Funcionou! A média resultante está correta, conforme esperado.

Caso a função `my.mean` for chamada sem *input* algum, ela utilizará o valor *default* de `x = c(1, 1, 1, 1)`, cuja média é exatamente igual a *1*. Esse valor *default* foi definido na definição da função. Veja, a seguir, o seu uso:

```
# calling my.fct without input
my.mean <- my.fct()

# print result
print(my.mean)
```

```
## [1] 1
```

Novamente, como esperado, o valor retornado está correto.

Uma estratégia para o uso de valores *default* de funções é sempre colocar o valor mais óbvio ou mais simples. O usuário não precisa entender toda a complexidade de uma função e as consequências na escolha de diferentes valores. Ao fazer seleções para ele, o uso da função fica mais fácil e intuitivo.

******ebook converter DEMO Watermarks*******

Apesar de simples, o exemplo anterior nos permite discorrer sobre uma prática muito útil na criação de funções: o teste das entradas. Observe que a função `my.fct` aceita qualquer tipo de objeto de entrada. Caso definirmos a entrada como um vetor de texto, a função o aceitaria e tentaria executar a linha `out <- sum(x)/length(x)`. O problema é que a função `sum` não aceita objetos de texto e iria retornar um erro. **Ao não testar as entradas de uma função, facilitamos erros que podem ser difíceis de identificar**. Isso é particularmente verdade para funções complexas com centenas de linhas de código.

Evitar esse erro é bastante simples, basta utilizar um teste lógico para a classe dos dados de entrada e, no caso de ser identificado algo diferente de `numeric` ou `integer`, emitir um erro com a função `stop`. Veja a seguir:

```
my.fct <- function(x = c(1, 1, 1, 1)){
  # Calculates the average of input x
  #
  # Args:
  #   x - a numerical vector
  #
  # Returns:
  #   The mean of x, as numeric

  if (!(class(x) %in% c('numeric','integer'))){
    stop('ERROR: x is not numeric or integer')
  }

  out <- sum(x)/length(x)

  return(out)
}
```

No código anterior, utilizamos a função `class` para testar a classe do objeto e a função `stop` para emitir um erro. Caso tentássemos utilizar um vetor diferente de numérico, um erro com a mensagem `ERROR: Class of x is not numeric or integer` apareceria na tela. Isso ajuda o usuário a entender a razão pela qual uma função criada não esteja funcionando da maneira esperada. Veja a seguir:

```
# using wrong inputs (ERROR)
my.fct(x = c('a','b'))
```

```
## Error in my.fct(x = c("a", "b")): ERROR: x is not numeric or integer
```

Outro ponto importante da função criada anteriormente é que ela não lida com valores `NA`. Caso o vetor numérico contenha algum `NA`, a média resultante também será do tipo `NA`, conforme foi explicado no capítulo anterior. Veja a seguir:

```
# sum with NA
print(sum(c(1, 2, 3, NA, 4)))
```

```
## [1] NA
```

Lembre que objetos do tipo `NA` são contagiosos! Ao usar `my.fct` em um vetor contendo `NA`, o resultado será outro `NA`. Dependendo dos próximos passos da rotina, esse pode ser um erro difícil de identificar.

******ebook converter DEMO Watermarks*******

Para resolver é bastante simples: basta informar o usuário da existência do `NA` com a função `warning`, eliminá-lo e continuar com a função normalmente. Veja, a seguir, uma nova definição:

```
my.fct <- function(x = c(1, 1, 1, 1)){
  # Calculates the average of input x
  #
  # Args:
  #   x: a numerical vector
  #
  # Returns:
  #   The mean of x, as numeric

  if (!(class(x) %in% c('numeric','integer'))){
    stop('ERROR: x is not numeric or integer')
  }

  if (any(is.na(x))){
    warning('Warning: Found NA in x. Removing it.')
    x <- na.omit(x)
  }

  out <- sum(x)/length(x)

  return(out)
}
```

Para o código anterior, usamos função `warning` para emitir uma mensagem no *prompt*, comando `any(is.na(x))` para testar se qualquer elemento de `x` tem valor `NA` e `na.omit` para removê-lo do vetor atômico. Vamos testar a nova função:

```
# set vector with NA
y <- c(1,2,3, NA,1)

# test function
print(my.fct(y))
```

```
## Warning in my.fct(y): Warning: Found NA in x. Removing it.
```

```
## [1] 1.75
```

Como podemos ver, a função reconheceu e alertou a existência do `NA` em `y`.

Na escrita de uma função, o teste de aderência das entradas é boa política de programação. **Escrever boas funções dá trabalho e exige bastante conhecimento sobre o R e a operação subjacente**. O grande benefício é que você só precisa fazer isso uma vez! A função pode ser usada repetidamente em diferentes cenários. Com o tempo, você criará um conjunto de funções que o ajudarão a fazer o seu trabalho, e essa coleção de código será seu maior ativo. Se você escreveu algo que possa interessar outras pessoas, empacote o código e envie-o para o CRAN. A comunidade certamente apreciará o gesto.

Agora, vamos passar para um exemplo mais completo de uso de funções. Uma tarefa muito comum na pesquisa financeira é o cálculo do retorno de ações. Esse é nada mais do que a percentagem de elevação ou diminuição do preço de uma ação entre um período e outro. Formalmente, define-se o retorno aritmético da ação *i* como:

******ebook converter DEMO Watermarks*******

$$R_{i,t} = \frac{P_{i,t}}{P_{i,t-1}} - 1$$

No R, esse procedimento toma como entrada um vetor de preços e oferece como saída um vetor de retornos. Observe que sempre perdemos a primeira observação de retorno uma vez que, para a primeira data, não existe data anterior. Imagine agora um dataframe no formato longo com preços para várias ações. Com base nesses preços, queremos calcular um vetor de retornos e adicioná-lo como uma nova coluna.

Para começar, vamos criar uma função que toma como entrada um vetor de preços e tem como saída um vetor de retornos:

```
calc.ret <- function(P) {
  # calculates arithmetic returns from a vector of prices
  # ret_t = p_{t}/p_{t-1} - 1
  #
  # Args:
  #   P - vector of prices (numeric)
  #
  # Returns:
  #   A vector of returns

  my.length <- length(P)
  ret <- c(NA, P[2:my.length]/P[1:(my.length - 1)] - 1)
  return(ret)
}
```

Observe como mantivemos a função simples e direta, deixando de lado o código para verificação de erros. Definimos um valor `NA` no primeiro elemento da série de retorno, `ret`. Faz-se isso porque é importante que o objeto de retorno tenha o mesmo comprimento que a entrada, e sempre perdemos a primeira observação ao calcular os retornos. Poderíamos simplesmente definir um valor 0 ou a média dos retornos, mas um valor `NA` parece mais apropriado pois é único e fácil de identificar.

Apesar de intuitiva, note que a função `calc.ret` não funcionaria para o nosso caso com dados empilhados. O problema é que a mesma foi criada para processar um ativo apenas. Lembre-se que em um dataframe com vários ativos, a coluna de preços é única. Portanto, caso utilizássemos `calc.ret`, estaríamos misturando preços de diferentes ações no cálculo. A primeira observação de retornos dos ativos estaria errada!

Para resolver é bastante fácil, basta inserirmos na função o vetor de *tickers* e, para as linhas onde muda a ação, inserimos um `NA` no cálculo de retorno. Aproveitamos também para inserir os testes de entrada. Veja a seguir:

```
calc.ret <- function(P, tickers = rep('ticker', length(P))) {
  # calculates arithmetic returns from a vector of prices
  #
  # Args:
  #   P - vector of prices (numeric)
  #   tickers - vector of tickers (optional)
  #
```

******ebook converter DEMO Watermarks*******

```
  # Returns:
  #   A vector of returns

  # ret_t = p_{t}/p_{t-1} - 1

  # error checking
  if ( !(class(P) %in% c('numeric', 'integer'))) {
    stop('ERROR: P should be a numeric object.')
  }

  if ( !(class(tickers) %in% c('character', 'factor'))) {
    stop('ERROR: tickers should be a character or factor object.')
  }

  if (length(P) != length(tickers)) {
    stop('ERROR: The length of P and tickers does not match.')
  }

  if ( length(P) < 2) {
    stop('ERROR: input P should have at least 2 elements.')
  }

  my.length <- length(P)
  ret <- c(NA, P[2:my.length]/P[1:(my.length - 1)] - 1)

  idx <- (tickers != c(NA, tickers[1:(my.length-1)]))
  ret[idx] <- NA

  return(ret)
}
```

Esse é um código extenso! Mas lembre-se, só é necessário escrever uma vez. O código possui várias checagens das entradas para garantir que está sendo usada corretamente pelo usuário. O tempo investido em sua escrita será facilmente justificado pelo seu uso futuro.

Agora, vamos aplicar a função com os dados disponíveis no arquivo com dados das ações do índice SP500.

```
library(readr)
library(dplyr)

my.f <- 'data/SP500-Stocks_long.csv'

# set columns types
my.cols <- cols(
  price.adjusted = col_double(),
  ref.date = col_date(format = ""),
  ticker = col_character()
)

# import data
my.df <- read_csv(my.f, col_types = my.cols)

# calculate return column
my.df <- my.df %>%
  mutate(ret = calc.ret(P = price.adjusted,
                        tickers = ticker))
```

******ebook converter DEMO Watermarks*******

Vamos verificar o resultado:

```
glimpse(my.df)
```

```
## Observations: 829,710
## Variables: 4
## $ price.adjusted <dbl> 69.15826, 68.72509, 69.69973, 69...
## $ ref.date       <date> 2010-01-04, 2010-01-05, 2010-01...
## $ ticker         <chr> "MMM", "MMM", "MMM", "MMM", "MMM...
## $ ret            <dbl> NA, -0.006263503, 0.014181794, 0...
```

```
summary(my.df)
```

```
##  price.adjusted       ref.date             ticker
##  Min.   :   1.59   Min.   :2010-01-04   Length:829710
##  1st Qu.:  28.35   1st Qu.:2011-09-30   Class :character
##  Median :  44.89   Median :2013-07-05   Mode  :character
##  Mean   :  62.00   Mean   :2013-07-03
##  3rd Qu.:  71.20   3rd Qu.:2015-04-06
##  Max.   :1578.13   Max.   :2016-12-30
##
##       ret
##  Min.   :-0.3919
##  1st Qu.:-0.0077
##  Median : 0.0006
##  Mean   : 0.0007
##  3rd Qu.: 0.0091
##  Max.   : 0.6191
##  NA's   :471
```

Perfeito! O novo dataframe possui a coluna `ret`, como desejado. Note na saída de `summary(my.df)` que essa coluna possui exatamente 471 `NAs`. Esse é o número de ações na base de dados.

Indo mais além, vamos remover todos `NAs` com função `complete.cases`. Essa retorna todas as linhas com dados completos:

```
library(dplyr)

my.df <- my.df %>%
  filter(complete.cases(.))

# check result
glimpse(my.df)
```

```
## Observations: 829,239
## Variables: 4
## $ price.adjusted <dbl> 68.72509, 69.69973, 69.74972, 70...
## $ ref.date       <date> 2010-01-05, 2010-01-06, 2010-01...
## $ ticker         <chr> "MMM", "MMM", "MMM", "MMM", "MMM...
## $ ret            <dbl> -0.0062635034, 0.0141817936, 0.0...
```

```
summary(my.df)
```

```
##  price.adjusted       ref.date             ticker
##  Min.   :   1.59   Min.   :2010-01-05   Length:829239
##  1st Qu.:  28.36   1st Qu.:2011-10-03   Class :character
##  Median :  44.90   Median :2013-07-05   Mode  :character
```

******ebook converter DEMO Watermarks*******

```
##  Mean   :  62.02   Mean   :2013-07-03
##  3rd Qu.:  71.21   3rd Qu.:2015-04-06
##  Max.   :1578.13   Max.   :2016-12-30
##       ret
##  Min.   :-0.3918766
##  1st Qu.:-0.0077069
##  Median : 0.0006451
##  Mean   : 0.0006771
##  3rd Qu.: 0.0090999
##  Max.   : 0.6190656
```

Por fim, salvamos o dataframe resultante em um arquivo *.rds*. Iremos utilizar esses dados nos capítulos seguintes.

```
saveRDS(object = my.df,
        file = 'data/SP500-Stocks-WithRet.rds')
```

******ebook converter DEMO Watermarks*******

8.2 Utilizando Loops (comando *for*)

Os *loops* são os comandos mais básicos dentro de qualquer linguagem de programação, permitindo a repetição estruturada de códigos e possibilitando o processamento de itens individualmente de uma forma fácil. OS `loops` são utilizados de forma ostensiva na análise de dados.

Considere, por exemplo, um cenário envolvendo uma base de dados composta por 1.000 arquivos em formato *.csv*. Nesse caso, poderíamos criar um *loop* para carregar os dados individualmente, processar o *dataframe* resultante de cada arquivo e salvar as variáveis de interesse em algum outro lugar, ou então agregar todos os dados dos arquivos em um único objeto.

A grande *sacada* no uso de *loops* é que o seu tamanho, o número de repetições, pode ser definido dinamicamente. Usando o exemplo anterior, se tivéssemos 5.000 arquivos, o *loop* processaria todos os 5.000 arquivos. Se tivéssemos apenas 500, o código do *loop* seria executado 500 vezes. Isso significa que podemos encapsular um procedimento genérico para processar todos os arquivos encontrados em uma determinada pasta. Com isso, você tem ao seu alcance uma ferramenta para a execução de qualquer tipo de processo sequencial. A profundidade do procedimento genérico é limitado apenas por sua imaginação.

A estrutura de um *loop* segue o seguinte esqueleto de código:

```
for (i in i.vec){

  # codes goes here

}
```

Para o código anterior, o comando `for` indica o início de um comando de *loop*. O objeto `i` é o iterador do mesmo. Esse iterador irá mudar de valor em cada iteração, tomando os valores individuais contidos em `i.vec`. Observe que o *loop* é encapsulado pelas chaves (`{ }`). Essas são de extrema importância, pois definem onde começa e onde termina o procedimento de iteração. A indentação (uso de margens maiores ou menores no código) também é importante, pois organiza visualmente o *loop*. Veja o exemplo prático a seguir:

```
# set seq
my.seq <- seq(-5,5)

# do loop
for (i in my.seq){
  cat(paste('\nThe value of i is',i))
}
```

```
##
## The value of i is -5
## The value of i is -4
## The value of i is -3
## The value of i is -2
## The value of i is -1
```

******ebook converter DEMO Watermarks*******

```
## The value of i is 0
## The value of i is 1
## The value of i is 2
## The value of i is 3
## The value of i is 4
## The value of i is 5
```

No exemplo anterior, criamos uma sequência de -5 até 5 e apresentamos um texto para cada elemento desse vetor na tela com o comando `cat`. Note que quebramos a linha com `\n`. O *loop* começa com `i=-5`, executa o comando `cat(paste('\nThe value of i is', -5))`, redefine `i` como `i=-4`, chama `cat` novamente e prossegue até a última iteração, quando `i=5`.

A sequência do *loop* não é exclusiva para vetores numéricos. De fato, qualquer tipo de vetor ou lista pode ser utilizado. Veja a seguir:

```
# set char vec
my.char.vec <- letters[1:5]

# loop it!
for (i.char in my.char.vec){
  cat(paste('\nThe value of i.char is', i.char))
}
```

```
##
## The value of i.char is a
## The value of i.char is b
## The value of i.char is c
## The value of i.char is d
## The value of i.char is e
```

O mesmo é válido para listas.

```
# set list
my.l <- list(x = 1:5,
             y = c('abc','dfg'),
             z = factor('A','B','C','D'))

# loop list
for (i.l in my.l){

  cat(paste0('\nThe class of i.l is ', class(i.l), '. '))
  cat(paste0('The number of elements is ', length(i.l), '.'))

}
```

```
##
## The class of i.l is integer. The number of elements is 5.
## The class of i.l is character. The number of elements is 2.
## The class of i.l is factor. The number of elements is 1.
```

Destaca-se que o iterador não precisa necessariamente ser o único objeto a ser incrementado em cada iteração do *loop*. Podemos criar outros objetos e incrementá-los manualmente. Veja a seguir:

```
# set vec and iterators
my.vec <- seq(1:5)
```

******ebook converter DEMO Watermarks*******

```
my.x <- 5
my.z <- 10

for (i in my.vec){
  # iterate "manually"
  my.x <- my.x + 1
  my.z <- my.z + 2

  cat('\nValue of i = ', i,
      ' | Value of my.x = ', my.x,
      ' | Value of my.z = ', my.z)
}
```

```
##
## Value of i =  1  | Value of my.x =  6  | Value of my.z =  12
## Value of i =  2  | Value of my.x =  7  | Value of my.z =  14
## Value of i =  3  | Value of my.x =  8  | Value of my.z =  16
## Value of i =  4  | Value of my.x =  9  | Value of my.z =  18
## Value of i =  5  | Value of my.x =  10  | Value of my.z =  20
```

O uso de *loops* encadeados, isto é, um *loop* dentro de outro *loop*, também é possível. Veja o exemplo a seguir, onde apresentamos todos os elementos de uma matriz:

```
# set matrix
my.mat <- matrix(1:9, nrow = 3)

# loop all values of matrix
for (i in seq(1,nrow(my.mat))){
  for (j in seq(1,ncol(my.mat))){
    cat(paste0('\nElement [', i, ', ', j, '] = ', my.mat[i,j]))
  }
}
```

```
##
## Element [1, 1] = 1
## Element [1, 2] = 4
## Element [1, 3] = 7
## Element [2, 1] = 2
## Element [2, 2] = 5
## Element [2, 3] = 8
## Element [3, 1] = 3
## Element [3, 2] = 6
## Element [3, 3] = 9
```

Vamos partir agora para um exemplo mais complexo. Iremos criar uma quantidade de arquivos com dados aleatórios no nosso computador e salvá-los em uma pasta chamada `many_datafiles`. Observe o *script* a seguir:

```
library(tibble)
library(readr)

# set number of files to create
n.files <- 10

# set first part of saved files
pattern.name <- 'myfiles_'

# set dir
```

******ebook converter DEMO Watermarks*******

```
out.dir <- 'many_datafiles/'

# test if out.dir exists, if not, creates it
if (!dir.exists(out.dir)){
  dir.create(out.dir)
}

# clean up folder before creating new files
file.remove(list.files(out.dir, full.names = TRUE))

## logical(0)

# set vec with filenames
file.names <- paste0(out.dir, pattern.name, seq(1,n.files), '.csv')

# loop it!
for (i.file in file.names){
  # create temp df
  temp.df <- data_frame(x = runif(100))

  # write it!
  write_csv(x = temp.df, path = i.file)
}
```

No exemplo anterior, usamos a função `if` em `if(!dir.exists(out.dir))` para testar se a pasta `many_datafiles` já existe. Em caso negativo, ela é criada no diretório atual de trabalho. Antes de executar o *loop*, removemos todos os arquivos em `out.dir` com o comando `file.remove(list.files(out.dir, full.names = TRUE))`.

No *loop*, usamos função `runif` para criar 100 números aleatórios entre *0* e *1*, salvando em um dataframe. Observe como o tamanho do *loop* é definido pelo objeto `n.files`. Se quiséssemos criar *10.000* arquivos, tudo o que precisamos fazer é definir `n.files = 10000` e o resto do código já está ajustado. Esse exemplo deixa claro que o *loop* é um procedimento poderoso e flexível que pode ser usado em muitas situações.

Podemos checar os arquivos criados com a função `list.files`:

```
# check files
print(list.files(out.dir))
```

```
##  [1] "myfiles_10.csv" "myfiles_1.csv"  "myfiles_2.csv"
##  [4] "myfiles_3.csv"  "myfiles_4.csv"  "myfiles_5.csv"
##  [7] "myfiles_6.csv"  "myfiles_7.csv"  "myfiles_8.csv"
## [10] "myfiles_9.csv"
```

Como esperado, os arquivos criados estão na respectiva pasta. Para finalizar o exemplo, vamos importar o conteúdo do todos os arquivos em um único dataframe usando um *loop* e funções `readr::read_csv` e `readr::bind_rows`:

```
library(dplyr)
library(readr)

# set empty df
df.agg <- data_frame()
for (i.file in file.names){
  # read file
```

******ebook converter DEMO Watermarks*******

```
  temp.df <- read_csv(i.file, col_types = cols(x = col_double()))

  # row bind
  df.agg <- bind_rows(df.agg, temp.df)
}

glimpse(df.agg)
```

```
## Observations: 1,000
## Variables: 1
## $ x <dbl> 0.25924564, 0.27192091, 0.37529416, 0.8344452...
```

```
summary(df.agg)
```

```
##        x
##  Min.   :0.0009245
##  1st Qu.:0.2838213
##  Median :0.5024252
##  Mean   :0.5112161
##  3rd Qu.:0.7513058
##  Max.   :0.9996456
```

No código anterior, observe como agregamos os dataframes com `df.agg <- bind_rows(df.agg, temp.df)`. O tamanho de `df.agg` aumenta a cada iteração do *loop*. Olhando para o conteúdo de `df.agg`, podemos ver que o código teve o resultado esperado, coluna `x` contém valores numéricos entre *0* e *1*.

Outro exemplo prático do uso de um *loop* é o processamento de dados de acordo com grupos. Por exemplo, para dados de preços de diferentes ações da Bovespa, caso quiséssemos calcular o preço médio de cada ação, construiríamos o seguinte código para importar os dados e processá-los:

```
library(readr)
library(dplyr)

# read data
my.f <- 'data/IbovStocks_long.csv'
my.df <- read_csv(my.f, col_types = cols(price.adjusted = col_double(),
                                         date = col_date(format = ""),
                                         ticker = col_character()))

# find unique tickers in column ticker
unique.tickers <- unique(my.df$ticker)

# create empty df
tab.out <- data_frame()

# loop tickers
for (i.ticker in unique.tickers){

  # create temp df with ticker i.ticker
  temp <- filter(my.df, ticker == i.ticker)

  # row bind i.ticker and mean.price
  temp.mean.price <- mean(temp$price.adjusted)
  tab.out <- bind_rows(tab.out,
                       data.frame(ticker = i.ticker,
```

******ebook converter DEMO Watermarks*******

```
                                mean.price = temp.mean.price))
}

# print result
print(tab.out[1:10, ])
```

```
## # A tibble: 10 x 2
##    ticker mean.price
##    <chr>       <dbl>
##  1 ABEV3        12.6
##  2 BBAS3        20.4
##  3 BBDC3        19.3
##  4 BBDC4        19.5
##  5 BRAP4        19.1
##  6 BRFS3        41.8
##  7 BRKM5        17.0
##  8 BRML3        11.1
##  9 BVSP      58373.
## 10 CCRO3        13.1
```

Note que, no código anterior, utilizamos a função `unique` para descobrir os diferentes tickers disponíveis no arquivo *.csv*. Logo após, criamos um *dataframe* vazio para guardar os resultados e um *loop* para, sequencialmente, filtrar os dados de cada ação e calcular a média de preços. No final do *loop*, utilizamos a função `bind_rows` para colar os resultados de cada ação com os resultados da tabela principal. Como podemos ver, podemos utilizar *loops* para realizar cálculos em grupo no R. Existem, porém, maneiras muito mais fáceis de fazer isso. Iremos mostrar estas funcionalidade logo em seguida.

******ebook converter DEMO Watermarks*******

8.3 Execuções Condicionais (`if`, `else`, `switch`)

Em algumas situaçõse de de pesquisa, torna-se necessário tomar algum tipo de decisão no código. A estrutura de decisão segue a lógica **se, então** (*if*, *then*), isto é, se uma condição for verdadeira, executa-se um código específico e, se ela for falsa, outras linhas de comando são executadas. Essas decisões definem as chamadas **execuções condicionais**. O esqueleto de código para criar execuções condicionais no R é:

```
if (cond){

  CodeIfTRUE...

} else {

  CodeIfFALSE...

}
```

Onde `cond` é a condição a ser avaliada, tomando dois valores apenas: `TRUE` ou `FALSE`. No caso de ser encontrado um valor verdadeiro para `cond`, o código em `CodeIfTRUE` será executado. Caso contrário, em que se encontre um valor falso para `cond`, o código em `CodeIfFALSE` será lido e executado pelo R. Veja, a seguir, um exemplo prático com base em um *loop*.

```
my.x <- 1:10
my.thresh <- 5

for (i in my.x){
  if (i > my.thresh){
    cat('\nValue of i', i, '- Higher than ', my.thresh)
  } else {
    cat('\nValue of i', i, '- Lower or equal than ', my.thresh)
  }
}
```

```
##
## Value of i 1 - Lower or equal than  5
## Value of i 2 - Lower or equal than  5
## Value of i 3 - Lower or equal than  5
## Value of i 4 - Lower or equal than  5
## Value of i 5 - Lower or equal than  5
## Value of i 6 - Higher than  5
## Value of i 7 - Higher than  5
## Value of i 8 - Higher than  5
## Value of i 9 - Higher than  5
## Value of i 10 - Higher than  5
```

Podemos, também, aplicar mais de uma condição lógica com o comando `else` e `else if`. Veja a seguir:

```
my.x <- 1:10
my.thresh <- 5

for (i in my.x){
```

******ebook converter DEMO Watermarks*******

```
  if (i > my.thresh){
    cat('\nValue of i', i, '- Higher than ', my.thresh)
  } else if (i==my.thresh) {
    cat('\nValue of i', i, '- equal to ', my.thresh)
  } else {
    cat('\nValue of i', i, '- lower than ', my.thresh)
  }
}
```

```
##
## Value of i 1 - lower than  5
## Value of i 2 - lower than  5
## Value of i 3 - lower than  5
## Value of i 4 - lower than  5
## Value of i 5 - equal to  5
## Value of i 6 - Higher than  5
## Value of i 7 - Higher than  5
## Value of i 8 - Higher than  5
## Value of i 9 - Higher than  5
## Value of i 10 - Higher than  5
```

Outra possibilidade para o uso de execuções condicionais é a função `switch`. Essa é projetada para fornecer uma melhor estrutura de código para decisões baseadas em mais de duas opções. Como exemplo, digamos que você deseja uma execução condicional baseada em cinco condições: A, B, C e D. Para cada condição, você deseja que o código mostre uma mensagem diferente na tela. Usando a função `if`, você poderia fazer isso usando o próximo código:

```
# set vec
my.vec <- c('A', 'D', 'B', 'A', 'C', 'B')

for (i.vec in my.vec){
  if (i.vec == 'A'){
    cat('\nGot an A!')
  } else if (i.vec == 'B') {
    cat('\nGot a B!')
  } else if (i.vec == 'C') {
    cat('\nGot a C!')
  } else if (i.vec == 'D') {
    cat('\nGot a D!')
  }
}
```

```
##
## Got an A!
## Got a D!
## Got a B!
## Got an A!
## Got a C!
## Got a B!
```

Enquanto o código anterior faria o que precisamos, o uso de várias condições `else if` não resulta em um código visualmente elegante. Uma maneira melhor de fazê-lo é usar a função `switch`. Veja em seguida:

```
# set vec
my.vec <- c('A', 'D', 'B', 'A', 'C', 'B')
```

******ebook converter DEMO Watermarks*******

```r
# using switch
for (i.vec in my.vec){
  msg.out <- switch(i.vec,
                    'A' = '\nGot an A!',
                    'B' = '\nGot a B!',
                    'C' = '\nGot a C!',
                    'D' = '\nGot a D!')

  cat(msg.out)

}
```

```
##
## Got an A!
## Got a D!
## Got a B!
## Got an A!
## Got a C!
## Got a B!
```

O grande benefício no uso de `switch` é a clareza no código.

******ebook converter DEMO Watermarks*******

8.4 Utilizando as Funções da Família `apply`

O R oferece uma maneira alternativa e funcional ao uso do comando de *loop*. Essa maneira é representada pelas funções nativas da família *apply*: `sapply`, `lapply`, `tapply`, `mapply`, `apply` e `by`. No `tidyverse`, o pacote `purr` oferece funções da família `map`, com funcionalidades semelhantes as anteriores. .

Vale salientar que todo procedimento criado via *loop* pode ser reestruturado com o uso das funções da família `base::apply`/`purrr::map` e vice-versa. A diferença de velocidade de processamento entre um e outro é desprezível. Recomenda-se, sempre que possível, utilizar as funções da família `apply` e pacote `purrr`. Essas forçam a criação de funções, criando um ambiente modular de trabalho. Ao criar diversas funções, podes reutilizar as mesmas no futuro.

8.4.1 Utilizando a Função `lapply`

A função `lapply` toma como entrada uma lista e um objeto do tipo função. Cada elemento dessa lista é passado para a função indicada. Os resultados desse processamento são sempre retornados como um objeto do tipo lista. Veja o exemplo a seguir, onde calculamos a média de uma série de vetores com tamanhos diferentes:

```
# set list
my.l <- list(vec1 = 1:10,
             vec2 = 2:5,
             vec3 = 10:-20)

# use lapply with mean
my.mean.vec <- lapply(X = my.l, FUN = mean)

# print result
print(my.mean.vec)
```

```
## $vec1
## [1] 5.5
##
## $vec2
## [1] 3.5
##
## $vec3
## [1] -5
```

O resultado mostra as médias de cada vetor em `my.l`. Caso necessário, poderíamos passar mais argumentos para função `mean` pela chamada a `lapply`. Veja o caso a seguir, onde utilizamos `na.rm=TRUE`:

```
# set list
my.l <- list(c(1,NA,2), c(2:5,NA), 10:-20)

# use lapply with mean
my.mean.vec <- lapply(X = my.l,
                      FUN = mean,
                      na.rm=TRUE)
```

******ebook converter DEMO Watermarks*******

```
# print result
print(my.mean.vec)

## [[1]]
## [1] 1.5
##
## [[2]]
## [1] 3.5
##
## [[3]]
## [1] -5
```

Como podemos ver, os argumentos extras em `lapply` são repassados para a função definida na entrada `FUN`.

O uso de `lapply` é particularmente útil quando utiliza-se alguma função customizada. Para isso, vamos utilizar o exemplo anterior de criação de arquivos. O primeiro passo é criar uma função que gere esses arquivos da mesma forma que fizemos no *loop*. A função toma como entrada o nome do arquivo gerado e o número de linhas do dataframe exportado.

```
# function to generate files
create.rnd.file <- function(name.file, N=100){
  # Generates a csv file with random content
  #
  # Args:
  #     name.file - name of csv file (character)
  #   N - number of rows in random dataframe (integer)
  #
  # Returns:
  #     TRUE, if successful

  if (class(name.file)!='character'){
    stop('ERROR: input name.file is not a character')
  }

  if ( !(class(N) %in% c('numeric','integer')) ){
    stop('ERROR: input N is not an integer or numeric!')
  }

  # create random df
  temp.df <- data.frame(x = runif(N))

  # write it!
  write.csv(x = temp.df, file = name.file)

  # return TRUE
  return(TRUE)
}
```

E utilizá-la juntamente com a função `lapply`:

```
# set options
n.files <- 5
pattern.name <- 'myfiles_with_lapply_'
out.dir <- 'many_datafiles/'
```

******ebook converter DEMO Watermarks*******

```
# set file names
file.names <- paste0(out.dir,pattern.name, seq(1,n.files), '.csv')

# test if out.dir exists, if not, create it
if (!dir.exists(out.dir)){
  dir.create(out.dir)
}

# clean up folder before creating new files
file.remove(list.files(out.dir, full.names = TRUE))
```

```
##  [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

```
# use lapply
out.l <- lapply(X = file.names, FUN = create.rnd.file, N=100)

# print result
print(out.l)
```

```
## [[1]]
## [1] TRUE
##
## [[2]]
## [1] TRUE
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] TRUE
##
## [[5]]
## [1] TRUE
```

O código funcionou corretamente, conforme esperado.

Uma observação importante aqui é que o retorno de `lapply` é sempre uma lista. Isso é bastante útil sempre que for necessário aplicar uma função que retorne objetos complexos, tal como o resultado da estimação de algum modelo. No capítulo 11 veremos um exemplo desse uso.

8.4.2 Utilizando a Função `sapply`

Função `sapply` é semelhante a `lapply`. A grande diferença está no tipo de *output*. Enquanto `lapply` retorna uma lista, função `sapply` retorna um vetor atômico ou matriz. Veja o exemplo a seguir:

```
# create list
my.l <- list(1:10, 2:5, 10:-20)

# use sapply
my.mean.vec <- sapply(my.l, mean)

# print result
print(my.mean.vec)
```

```
## [1]  5.5  3.5 -5.0
```

******ebook converter DEMO Watermarks*******

O uso de `sapply` é recomendado quando buscamos efetuar operações com saídas simples, tal como valores únicos. Nesses casos, não é necessário retornar um objeto flexível tal como uma lista.

Um aspecto importante da função `sapply` é que os retornos também podem envolver mais de um valor numérico. Veja o exemplo a seguir, onde criamos uma função que retorna a média e o desvio padrão de um vetor numérico:

```
# set list
my.l <- list(runif(10), runif(15), rnorm(1000))

my.fct <- function(x){
  # Returns mean and standard deviation of a vector
  #
  # Args:
  #   x - numerical vector
  #
  # Returns:
  #   Vector as c(mean(x), sd(x))

  if (!(class(x) %in% c('numeric','integer'))){
    stop('ERROR: Class of x is not numeric or integer.')
  }

  x <- na.omit(x)

  out <- c(mean(x), sd(x))
  return(out)

}

# use sapply
my.vec <- sapply(my.l, my.fct)

# check result
print(my.vec)

##           [,1]      [,2]       [,3]
## [1,] 0.5458824 0.5977961 0.02359729
## [2,] 0.3128141 0.3182017 0.98539676
```

Havendo mais de uma saída da função utilizada com `sapply`, o retorno será uma matriz: cada linha representa uma das diferentes saídas (nesse caso, a média e o desvio padrão de `x`) e as colunas indicarão os diferentes itens da lista `my.l`.

Um uso prático de `sapply` em pesquisa é a criação de tabelas descritivas. Por exemplo: vamos realizar uma análise descritiva dos dados de preços de diferentes ações da Bovespa. Iremos verificar o preço médio, o preço máximo e o preço mínimo para cada um dos ativos. O primeiro passo é escrever a função que irá analisar o vetor de preços de cada ativo, retornando a média, preço máximo e preço mínimo:

```
describe.vec <- function(x){
  # Describe numerical vector with mean and other stats
  #
  # Args:
```

******ebook converter DEMO Watermarks*******

```
  #   x - numerical vector
  #
  # Returns:
  #   A vector with mean, maximum and minimum

  # error checking
  if (!(class(x) %in% c('numeric','integer'))){
    stop('ERROR: Class of x is not numeric or integer.')
  }

  x <- na.omit(x)

  # calc vec
  out <- c(mean.price = mean(x),
           max.price = max(x),
           min.price = min(x))

  return(out)
}
```

Agora, vamos carregar dados do índice Ibovespa e aplicar a fórmula para cada ativo:

```
# set file and read it
library(dplyr)
library(readr)

my.f <- 'data/IbovStocks_long.csv'
my.df <- read_csv(my.f,
                  col_types = cols(price.adjusted = col_double(),
                                   ref.date = col_date(format = ""),
                                   ticker = col_character()))

# use split to split prices by ticker
my.l <- split(x = my.df$price.adjusted, my.df$ticker)

# use sapply
my.tab <- sapply(X = my.l, FUN = describe.vec)

# check result
print(head(t(my.tab)))
```

```
##       mean.price max.price min.price
## ABEV3   12.60977  22.03000  2.568153
## BBAS3   20.36675  37.20670 11.766141
## BBDC3   19.28285  35.44183 10.558298
## BBDC4   19.50249  36.89000 12.570860
## BRAP4   19.07545  34.72096  2.807214
## BRFS3   41.76806  70.11082 19.706953
```

Utilizamos a função `split` para separar os dados de `my.df$price` com base nos *tickers* disponíveis em `my.df$ticker`. Optou-se também por transpor a matriz resultante com o comando `t`, resultando nos ativos como linhas e nas diferentes medidas como colunas. O código anterior é muito útil para conhecer uma base de dados e identificar possíveis problemas nos mesmos. Vale observar que o usuário poderia estender as informações retornadas por `my.fct` para diversas outras estatísticas de interesse.

8.4.3 Utilizando a Função `tapply`

******ebook converter DEMO Watermarks*******

Função `tapply` é específica para cálculos em grupos de dados. Veja o exemplo a seguir, onde cria-se um vetor numérico com uma sequência de 1 até 150 e um vetor de fator com grupos A, B e C. O uso de `tapply` possibilita que as médias do objeto numérico sejam calculadas para cada grupo individualmente:

```
# set numeric vec and factor
my.x <- 1:150
my.factor <- factor(c(rep('C',50), rep('B',50), rep('A',50)))

# use tapply
my.mean.vec <- tapply(X = my.x, INDEX = my.factor, FUN = mean)

# print result
print(my.mean.vec)
```

```
##     A     B     C
## 125.5  75.5  25.5
```

Observa-se que o resultado é muito próximo do que obteríamos com o uso das funções `split` e `sapply` utilizadas anteriormente. Um ponto importante a ser ressaltado sobre o uso de `tapply` é que, com essa função, **a ordem dos fatores é diferente**. Note que os fatores foram ordenados alfabeticamente e não seguem a ordem encontrada em `my.factor`. Caso manter a ordem original de grupos for importante, o resultado deve ser reordenado.

Voltando ao exemplo anterior com dados reais, caso quiséssemos utilizar a função `tapply` para o cálculo dos preços médios, máximos e mínimos, poderíamos fazê-lo da seguinte maneira, utilizando a função `describe.vec` definida anteriormente:

```
# use tapply for descriptive stats
my.l.out <- tapply(X = my.df$price.adjusted,
                   INDEX = my.df$ticker,
                   FUN = describe.vec)

# print result
print(my.l.out[1:5])
```

```
## $ABEV3
## mean.price  max.price  min.price
##  12.609768  22.030001   2.568153
##
## $BBAS3
## mean.price  max.price  min.price
##   20.36675   37.20670   11.76614
##
## $BBDC3
## mean.price  max.price  min.price
##   19.28285   35.44183   10.55830
##
## $BBDC4
## mean.price  max.price  min.price
##   19.50249   36.89000   12.57086
##
## $BRAP4
## mean.price  max.price  min.price
##  19.075452  34.720959   2.807214
```

******ebook converter DEMO Watermarks*******

A saída de `tapply` é uma lista contendo os valores. Apesar de mostrar os mesmos resultados que vimos no caso anterior, uma lista não é o tipo de objeto mais recomendado para exportar dados na construção de tabelas. O ideal é converter a lista para um dataframe. Para obter um dataframe, podemos utilizar a função `do.call` juntamente com `rbind`.

```
# convert list to dataframe
my.tab <- do.call(what = rbind, args = my.l.out)

# print result
print(head(my.tab))
```

```
##       mean.price max.price min.price
## ABEV3   12.60977  22.03000  2.568153
## BBAS3   20.36675  37.20670 11.766141
## BBDC3   19.28285  35.44183 10.558298
## BBDC4   19.50249  36.89000 12.570860
## BRAP4   19.07545  34.72096  2.807214
## BRFS3   41.76806  70.11082 19.706953
```

Explicando melhor, a função `do.call` irá, recursivamente, usar pares de elementos em `args` para alimentar uma função definida na entrada `what`. Para o nosso caso, `do.call` usará o primeiro e segundo elemento de `my.l.out` como entradas no `rbind`. Isso equivale a `rbind(my.l.out[[1]], my.l.out[[2]]`. O resultado é então utilizado em outra chamada a `rbind`, dessa vez com o terceiro elemento de `my.l.out`. A função `do.call` continuará esta operação até o último elemento de `my.l.out` ser alcançado. Como você pode ver, é um cálculo recursivo, utilizando resultados anteriores para criar o resultado atual. O resultado final é um dataframe com todos os elementos de `my.l.out`. Mais detalhes sobre essa função podem ser obtidos com o comando `help(do.call)`.

8.4.4 Using `mapply`

Função `mapply` é uma versão multivariada de `lapply`, permitindo a iteração de múltiplos argumentos de uma função. Parece confuso? Não se preocupe, um exemplo deixará a ideia clara.

Suponha que estamos interessados em criar uma lista com o conteúdo como em `my.l <- (1, 1:2, 1:3, 1:4, .., 1:10)`. Uma possível solução é usar um *loop*:

```
# set size
N <- 10

# prealocate list
my.l <- list()

for (i in seq(1,N)){
  my.l[[i]] <- seq(1,i)
}

# print result
print(my.l)
```

```
## [[1]]
## [1] 1
##
## [[2]]
```

******ebook converter DEMO Watermarks*******

```
## [1] 1 2
##
## [[3]]
## [1] 1 2 3
##
## [[4]]
## [1] 1 2 3 4
##
## [[5]]
## [1] 1 2 3 4 5
##
## [[6]]
## [1] 1 2 3 4 5 6
##
## [[7]]
## [1] 1 2 3 4 5 6 7
##
## [[8]]
## [1] 1 2 3 4 5 6 7 8
##
## [[9]]
## [1] 1 2 3 4 5 6 7 8 9
##
## [[10]]
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Outra solução, menos verbosa, é usar `mapply`:

```
# use mapply for creating list
my.l <- mapply(FUN = seq, rep(1,N), seq(1,N))

print(my.l)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 1 2
##
## [[3]]
## [1] 1 2 3
##
## [[4]]
## [1] 1 2 3 4
##
## [[5]]
## [1] 1 2 3 4 5
##
## [[6]]
## [1] 1 2 3 4 5 6
##
## [[7]]
## [1] 1 2 3 4 5 6 7
##
## [[8]]
## [1] 1 2 3 4 5 6 7 8
##
## [[9]]
## [1] 1 2 3 4 5 6 7 8 9
```

******ebook converter DEMO Watermarks*******

```
##
## [[10]]
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Explicando o resultado, função `mapply` está chamando `seq` para cada par de elementos em `rep(1, N)` e `seq(1, N)`. Então, o primeiro elemento de `my.l` é simplesmente `seq(1,1)`. O segundo elemento é `seq(1,2)`, e o elemento final é `seq (1,10)`. Como você pode ver, função `mapply` tem um uso mais elaborado que `lapply` e `sapply`, e é útil em diversas situações de análise de dados onde é necessário iterar sequencialmente argumentos de funções.

8.4.5 Utilizando `apply`

Função `apply` é bastante parecida com `sapply` e `lapply`. Sua originalidade está na possibilidade de escolher a dimensão de processamento em objetos com mais de uma dimensão, tal como matrizes. Nesse caso, podes escolher a orientação do processamento, por linhas ou por colunas. Veja o exemplo a seguir, onde, com base em um objeto da classe matriz, calculamos a soma dos valores orientados por linha e a soma orientada por colunas.

```
# set matrix and print it
my.mat <- matrix(1:15, nrow = 5)
print(my.mat)
```

```
##      [,1] [,2] [,3]
## [1,]    1    6   11
## [2,]    2    7   12
## [3,]    3    8   13
## [4,]    4    9   14
## [5,]    5   10   15
```

```
# sum rows with apply and print it
sum.rows <- apply(X = my.mat, MARGIN = 1, FUN = sum)
print(sum.rows)
```

```
## [1] 18 21 24 27 30
```

```
# sum columns with apply and print it
sum.cols <- apply(X = my.mat, MARGIN = 2, FUN = sum)
print(sum.cols)
```

```
## [1] 15 40 65
```

No exemplo anterior, argumento `MARGIN` definiu a orientação do cálculo. Com `MARGIN = 1`, orienta-se a função a calcular a soma das linhas individualmente. Com `MARGIN = 2`, a orientação é calcular a soma das colunas. Poderíamos utilizar essa mesma notação para encontrar os valores máximos de `my.mat`, usando a orientação por linha ou por coluna.

```
# print max by row
print(apply(X = my.mat, MARGIN = 1, FUN = max))
```

```
## [1] 11 12 13 14 15
```

```
# print max by column
print(apply(X = my.mat, MARGIN = 2, FUN = max))
```

```
## [1]  5 10 15
```

******ebook converter DEMO Watermarks*******

******ebook converter DEMO Watermarks*******

8.5 Utilizando Pacote `purrr`

O universo `tidyverse` oferece não somente funções para manipulações básicas de objetos, mas também funcionalidades voltadas à iterações nos dados com linguagem funcional. Essas estão contidas no pacote `purrr` (Henry and Wickham 2017). As principais funções deste pacote são `map`, `map_dbl`, `map_chr`, `map_int`, `map_lgl`. O uso dessas é bastante semelhante ao que aprendemos com as funções da família `apply`. A vantagem reside no fato de que a sintaxe das funções do `purrr` são consistentes e permitem o uso do operador de `pipeline`. Outro ponto importante é o maior controle sobre o objeto retornado. Por exemplo, `map_dbl` sempre retorna valores numéricos, enquanto `map_chr` sempre retorna caracteres. Veja um exemplo simples a seguir:

```
library(purrr)

# set list
my.l <- list(vec1 = 1:10,
             vec2 = 1:50,
             vec3 = 1:5,
             char1 = letters[1:10])

# get length of objects
res.out <- my.l %>%
  map_int(length)

print(res.out)

##  vec1  vec2  vec3 char1
##    10    50     5    10

# find character objets
res.out <- my.l %>%
  map_lgl(is.character)

print(res.out)

##  vec1  vec2  vec3 char1
## FALSE FALSE FALSE  TRUE
```

Outro ponto interessante nas funções do `purrr` é que permitem, de uma forma simples, o acesso a elementos de uma lista. Para isso, basta indicar uma entrada de posição ou nome na função `map`:

```
# set list
my.l <- list(vec1 = c(elem1 = 10, elem2 = 20, elem3 = 5),
             char1 = c(elem1 = 40, elem2 = 50, elem3 = 15))

# get second element of each element in list, by position
res.out <- my.l %>% map(2)
print(res.out)

## $vec1
## [1] 20
##
## $char1
## [1] 50
```

******ebook converter DEMO Watermarks*******

```
# get third element of each element in list, by name
res.out <- my.l %>% map('elem3')
print(res.out)
```

```
## $vec1
## [1] 5
##
## $char1
## [1] 15
```

Essa funcionalidade é bastante útil pois em muitas situações de análise de dados estamos apenas interessados em um dos elementos de cada objeto de uma extensa lista.

Indo além, a grande inovação do `purrr` em relação ao `base` é a forma de tratar erros no código. As funções da família `apply` não oferecem maneiras de observar e tratar erros de execução. Isso é bastante frustante pois a informação sobre a qual elemento resultou em erro tem que ser investigada manualmente, adicionando tempo no desenvolvimento do código. Com este objetivo, funções `purrr::safely` e `purrr::possibly` oferecem uma maneira inteligente de observar e administrar erros de execução junto as funções da família `purrr::map`.

Função `safely` encapsula outra função e sempre retorna dois elementos, o resultado da chamada e a mensagem de erro, caso existirem. Veja um exemplo simples a seguir, onde forçamos o erro no R com uma chamada direta da função:

```
library(purrr)

my.fct <- function(x) {
  return(x+1)
}

# ERROR
print(my.fct('a'))
```

```
## Error in x + 1: non-numeric argument to binary operator
```

Agora vamos utilizar `safely` para encapsular `my.fct` em outra função chamada `my.fct.safely`:

```
# with safely
my.fct.safely <- safely(my.fct)

print(my.fct.safely('a'))
```

```
## $result
## NULL
##
## $error
## <simpleError in x + 1: non-numeric argument to binary operator>
```

Note que o código `print(my.fct.safely('a'))` resultou em uma lista, e não um erro. Portanto, ao utilizar `safely` juntamente com `map`, o retorno é o resultado da chamada da função para todos os casos. Veja o exemplo a seguir:

```
my.l <- list(1:5,
             'a',
             1:4)
```

******ebook converter DEMO Watermarks*******

```
res.out <- my.l %>%
  map(safely(my.fct))

print(res.out)
```

```
## [[1]]
## [[1]]$result
## [1] 2 3 4 5 6
##
## [[1]]$error
## NULL
##
##
## [[2]]
## [[2]]$result
## NULL
##
## [[2]]$error
## <simpleError in x + 1: non-numeric argument to binary operator>
##
##
## [[3]]
## [[3]]$result
## [1] 2 3 4 5
##
## [[3]]$error
## NULL
```

Podemos facilmente ver que a função teve erro no segundo elemento de `my.l`. Indo além, caso quiséssemos apenas os resultados, podemos escrever:

```
print(res.out %>% map('result'))
```

```
## [[1]]
## [1] 2 3 4 5 6
##
## [[2]]
## NULL
##
## [[3]]
## [1] 2 3 4 5
```

Ou apenas as mensagens de erro:

```
print(res.out %>% map('error'))
```

```
## [[1]]
## NULL
##
## [[2]]
## <simpleError in x + 1: non-numeric argument to binary operator>
##
## [[3]]
## NULL
```

A saída padrão para o caso de erros também pode ser modificada. Veja o próximo caso, onde inserimos o valor `NA` toda vez que a função resultar em erro (argumento `otherwise`), e

******ebook converter DEMO Watermarks*******

retornamos um vetor:

```
my.l <- list(1,
             'a',
             4)

# NA for errors
res.out <- my.l %>%
  map(safely(my.fct,
             otherwise = NA)) %>%
  map_dbl('result')

# print result
print(res.out)
```

```
## [1]  2 NA  5
```

Outras funções para controle de erros no `purrr` são `possibly` e `quietly`. Essas possuem comportamento bastante semelhante a `safely` e não serão demonstradas aqui. Maiores detalhes podem ser encontrados no sistema de busca do R.

******ebook converter DEMO Watermarks*******

8.6 Manipulação de Dados com `dplyr`

No capítulo 6 aprendemos sobre o uso de funções do pacote `dplyr` para manipular dataframes. A funcionalidade desse pacote, porém, é muito maior. Podemos utilizar o operador de *pipeline* junto a algumas de suas funções para realizar análises complexas com uma sintaxe simples e intuitiva. O código não somente fica fácil de ler, mas o tempo de processamento também diminui.

8.6.1 Operações de Grupo com `dplyr`

A primeira funcionalidade do `dplyr` é no processamento de dados em grupos. Essa operação é comumente chamada de *split-apply-combine*, isto é, separamos a amostra em grupos, aplicamos alguma função e, por fim, combinamos todos os resultados em uma nova tabela. Note que é exatamente esse procedimento que fizemos com *loops* na seção 8.2: separamos os dados por ação, calculamos média, máximo e mínimo de preço e, por fim, agregamos todos resultados em um único dataframe.

Para ilustrar o uso das funções `dply::group_by` e `dplyr::summarise`, replicaremos o exemplo de descrever os preços das ações do SP500. Desta vez, no entanto, usaremos a estrutura do pacote `dplyr` e o operador de *pipeline* para concretizar a operação:

```
library(readr)
library(dplyr)

# load data
my.f <- 'data/SP500-Stocks-WithRet.rds'
my.df <- readRDS(my.f)


# group data and calculate stats
my.tab <- my.df %>%
  group_by(ticker) %>%
  summarise(mean.price = mean(price.adjusted),
            max.price = max(price.adjusted),
            min.price = min(price.adjusted),
            max.ret = max(ret),
            min.ret = min(ret))

# check result
print(my.tab)
```

```
## # A tibble: 471 x 6
##    ticker mean.price max.price min.price max.ret min.ret
##    <chr>       <dbl>     <dbl>     <dbl>   <dbl>   <dbl>
##  1 A            33.1      48.2     18.3   0.118  -0.110
##  2 AAL          23.4      54.7      3.89  0.173  -0.158
##  3 AAP         103.      200.      38.4   0.166  -0.170
##  4 AAPL         75.0     128.      24.9   0.0887 -0.124
##  5 ABC          58.4     112.      23.3   0.0953 -0.130
##  6 ABT          32.1      49.3     18.4   0.0650 -0.0929
##  7 ACN          70.6     125.      31.6   0.0799 -0.104
##  8 ADBE         55.3     111.      22.7   0.128  -0.190
```

******ebook converter DEMO Watermarks*******

```
##  9 ADI          42.4      73.9     21.6   0.101  -0.0781
## 10 ADM          33.8      50.5     20.9   0.0730 -0.0874
## # ... with 461 more rows
```

Explicando o código, o primeiro passo no uso de `dplyr` é agrupar os dados com função `group_by`. Aqui, estamos agrupando com base na coluna `ticker`. Isso significa que pretendemos realizar um determinado cálculo para cada ação em nosso banco de dados. Depois de agrupar os dados com `group_by`, seguimos no *pipeline* com função `summarise`. Essa processa os dados em blocos definidos pelos grupos. Observe que cada argumento em `summarise` se transforma em uma coluna na saída: `mean.price`, `max.price`, `min.price`, `max.ret` e `min.ret`. Cada um dos argumentos anteriores está fazendo um cálculo diferente com base nas colunas de `my.df`. Caso fosse necessário, poderíamos adicionar vários outros cálculos em diferentes colunas simplesmente expandindo os argumentos de entrada em `summarise`.

Utilizar o `dplyr` é altamente recomendado quando você está agrupando com base em mais de uma coluna. Considere o caso onde estamos interessados nas estatísticas de preço e retorno não somente para cada ação, mas também por dia de semana. Para realizar isso, basta criar uma nova coluna com os dias da semana e adicionar o nome dela na chamada a `group_by`:

```
# set new col week.day
my.df$week.day <- weekdays(my.df$ref.date)

# print it
print(head(my.df$week.day))
```

```
## [1] "Tuesday"   "Wednesday" "Thursday"  "Friday"
## [5] "Monday"    "Tuesday"
```

```
my.tab <- my.df %>%
  mutate(week.day = lubridate::wday(ref.date, label = TRUE)) %>%
  group_by(ticker, week.day) %>%
  summarise(mean.price = mean(price.adjusted),
            max.price = max(price.adjusted),
            min.price = min(price.adjusted),
            max.ret = max(ret),
            min.ret = min(ret))

print(my.tab)
```

```
## # A tibble: 2,355 x 7
## # Groups:   ticker [?]
##    ticker week.day mean.price max.price min.price max.ret
##    <chr>  <ord>         <dbl>     <dbl>     <dbl>   <dbl>
##  1 A      Mon            33.1      48.2     18.5   0.0728
##  2 A      Tue            33.1      48.0     18.3   0.0781
##  3 A      Wed            33.0      48.2     18.7   0.0708
##  4 A      Thu            33.0      47.9     18.7   0.118
##  5 A      Fri            33.0      47.9     18.4   0.0918
##  6 AAL    Mon            23.6      54.2      4.14  0.0909
##  7 AAL    Tue            23.5      52.9      3.94  0.165
##  8 AAL    Wed            23.4      53.1      3.89  0.173
##  9 AAL    Thu            23.3      54.4      4.28  0.160
## 10 AAL    Fri            23.3      54.7      3.91  0.0841
## # ... with 2,345 more rows, and 1 more variable:
## #   min.ret <dbl>
```

******ebook converter DEMO Watermarks*******

E pronto! Agora temos as mesmas estatísticas de interesse separadas por ação e dia da semana. Caso fosse necessário, poderíamos segregar ainda mais a amostra. Utilizar `dplyr` para realizar cálculos em grupos é muito fácil e prático! O código resultante é eficiente e elegante.

8.6.2 Operações de Grupo Complexas com `dplyr`

O exemplo anterior mostra um caso simples de cálculos de grupo. Podemos dizer que foi simples porque a operação sempre retornou um valor único para cada caso: tivemos uma média para o ticker X no dia da semana Z, outra média para o ticker Y no dia da semana L, e assim por diante.

Pacote `dplyr` também suporta operações mais complexas, onde a saída não é um valor único, mas um objeto complexo. Isso é bastante útil nas tarefas de processamento de dados pois você pode manipular os dados na mesma estrutura tabular em que os mesmos estão representados no R.

Vejamos o exemplo a seguir, onde usamos os retornos das ações para calcular o retorno acumulado de cada ação ao longo do tempo. Note que este não é um valor único, mas sim um vetor de mesmo tamanho do vetor de retornos. Um exemplo:

```
# vector of returns
ret <- c(0, rnorm(4, sd= 0.05))
print(ret)
```

```
## [1]  0.000000000 -0.013935301 -0.045051653  0.008262416
## [5]  0.096010333
```

```
# vector of accumulated returns
acum.ret <- cumprod(1+ret)
print(acum.ret)
```

```
## [1] 1.0000000 0.9860647 0.9416409 0.9494211 1.0405753
```

Agora vamos realizar este cálculo para todas as ações do SP500, salvando também o máximo e mínimo acumulado:

```
library(dplyr)

# get acum ret of stoks
my.tab <- my.df %>%
  group_by(ticker) %>%
  do(acum.ret = cumprod(1+.$ret)) %>%
  mutate(last.cumret = acum.ret[length(acum.ret)],
         min.cumret = min(acum.ret))

print(head(my.tab))
```

```
## # A tibble: 6 x 4
##   ticker acum.ret      last.cumret min.cumret
##   <chr>  <list>              <dbl>      <dbl>
## 1 A      <dbl [1,761]>        2.14      0.862
## 2 AAL    <dbl [1,761]>       10.0       0.839
## 3 AAP    <dbl [1,761]>        4.27      0.970
## 4 AAPL   <dbl [1,761]>        4.16      0.897
## 5 ABC    <dbl [1,761]>        3.23      0.968
```

******ebook converter DEMO Watermarks*******

```
## 6 ABT    <dbl [1,761]>        1.77      0.855
```

Note que os elementos da coluna `acum.ret` não são valores únicos, mas sim vetores atômicos. Na verdade, `acum.ret` é uma coluna do tipo lista (*list-columns*), o que significa que pode armazenar qualquer tipo de objeto. Atingimos esse resultado usando função `dplyr::do` na sequência do *pipeline*. Quanto às outras colunas, `last.cumret` e `min.cumret`, essas são manipulações simples de `acum.ret` e portanto foram criadas com `mutate`. Uma observação importante é que usamos o símbolo `.$` para acessar as colunas de `my.df` na chamada a função `do`. Sempre que for utilizar colunas do tipo lista com `do`, você deve usar essa notação.

A maior vantagem de usar operações complexas de agrupamento com `dplyr` é que você pode manter a mesma representação tabular dos dados. Podemos "ler" objeto `my.tab` como "para cada ação, defina um vetor de retornos acumulados". O uso de `dplyr::do` é particularmente interessante na estimação de vários modelos econométricos a partir dos dados. Vamos abordar este tópico no capítulo 11.

Resumindo, pacote `dplyr` foi uma inovação muito significativa para a comunidade do R. Não é a toa que este é um dos pacotes mais instalados atualmente, segundo estatísticas do CRAN. Sempre que possível, tente utilizar suas funções para processar os dados.

******ebook converter DEMO Watermarks*******

8.7 Exercícios

1. Crie uma função chamada `say.my.name` que tome como entrada um nome de pessoa e mostre na tela o texto *Your name is* .... Dentro do escopo da função, utilize comentários para descrever o propósito da função, suas entradas e saídas.

2. Implemente um teste para os objetos de entrada, de forma que, quando o nome de entrada nao for da classe caractere, um erro é retornado ao usuário. Teste sua nova função e verifique se a mesma está funcionando conforme esperado.

3. Crie um vetor com cinco nomes quaisquer, chamado `my.names`. Utilizando um *loop*, aplique função `say.my.name` para cada elemento de `my.names`.

4. Refaça o exercício anterior utilizando comandos `sapply` e `purrr::map`.

5. Utilize pacote `BatchGetSymbols` para baixar os valores dos índices Ibovespa (`'^BVSP'`) e SP500 (`^GSPC`) de `'2010-01-01'` até a data atual. Com base na função `calc.ret`, utilizada no capítulo atual, adicione uma nova coluna chamada `ret2` com os retornos diários de ambos os índices. Essa coluna apresenta os mesmos valores que a coluna `ret` já existente?

6. Para os dados importados no exercício anterior, utilize um *loop* para calcular retorno médio, máximo e mínimo de cada um dos índices no período analisado.

7. Refaça o exercício anterior utilizando as funções do pacote `dplyr`: `group_by` e `summarise`.

******ebook converter DEMO Watermarks*******

9 Estruturando e Limpando Dados

Após entender o processo de importação de dados e as ferramentas de programação no R, é importante discutirmos o formato com que os dados serão trabalhados. Em muitos casos, a etapa após a importação dos dados é a manipulação do `dataframe` resultante para um formato mais fácil de lidar, chamado formato longo. Além disso, neste capítulo iremos tratar também de operações básicas de limpeza de dados financeiros e econômicos. Isto inclui identificação e tratamento de valores extremos (*outliers*), desinflação de dados de preços e mudança de frequência de dados temporais.

******ebook converter DEMO Watermarks*******

9.1 O Formato do Dataframe

O formato do dataframe é uma discussão relativamente nova que surgiu com a introdução de pacotes específicos de manipulação de dados e criação de figuras no `tidyverse`. No centro do debate, discute-se o formato de alocação, o modo de incremento das variáveis e se os dados devem ser orientados por colunas (formato *wide*) ou por linhas (formato *long*). Veja os conceitos destacados a seguir.

No **formato *wide* (largo)**, as linhas da tabela são geralmente indexadas por algum fator único, tal como uma data, e as colunas referenciam os casos trabalhados. À medida que novas informações (itens) são adicionadas à base de dados, a mesma geralmente cresce em número de colunas. Exemplo:

refdate	PETR4	GGBR4	VALE5
2015-01-01	10	3	6
2015-01-02	11	3.1	7
2015-01-03	10.5	3.2	7.5
2015-01-04	12	3.5	6
...	...	...	...

Observe que a tabela anterior possui três informações, apenas: *ticker*, preço e data. Caso fosse adicionado mais uma ação, a tabela seria incrementada em uma coluna.

No **formato *long* (longo)**, cada linha da tabela é uma nova informação de dados e cada coluna é uma variável coletada. As linhas da tabela podem ou não ser casos únicos (ou seja, que não se repetem). À medida que novos dados são adicionados, a tabela geralmente cresce em número de linhas. Exemplo:

refdate	asset.code	Price
2015-01-01	PETR4	10
2015-01-01	GGBR4	3
2015-01-01	VALE5	6
2015-01-02	PETR4	11
2015-01-02	GGBR4	3.1
2015-01-02	VALE5	7
2015-01-03	PETR4	10.5
2015-01-03	GGBR4	3.2
2015-01-03	VALE5	7.5
...	...	...

Em comparação, o formato largo (`wide`) é mais intuitivo pois os dados de cada ação são facilmente separáveis pelas colunas. O problema, nesse caso, é a dificuldade de expansão da base

******ebook converter DEMO Watermarks*******

de dados, visto que cada variável fica contida em uma única tabela. Por exemplo, se estivéssemos interessado, também, no volume negociado de cada ação em cada data, teríamos que criar outra tabela para alocar essa informação. A medida que o número de variáveis cresce, também cresce o número de tabelas. A tabela em formato longo não possui esse problema pois a entrada de novas variáveis incrementa as colunas, apenas.

À primeira vista, essa discussão pode parecer banal, dado que as informações são exatamente as mesmas em ambos os formatos. Porém, não se engane: o formato de apresentação dos dados é de extrema importância no dia-a-dia. Pacotes especializados, tal como `dplyr` (Wickham and Francois 2016) e `ggplot2` (Wickham 2009), esperam um `dataframe` no formato longo e, portanto, essa forma deve ser priorizada. Para Finanças, o formato de dados no estilo *wide* é geralmente utilizado na criação e manipulação de portfólios de investimento em diferentes ativos. Isso acontece porque os cálculos são de notação matricial e esperam uma matriz de retornos como *input*. Essa matriz deve apresentar os diferentes ativos nas colunas e, nas linhas, os diferentes períodos de tempo. Destaco, porém, que esses mesmos cálculos poderiam ser realizados com o formato longo. A conversão entre os formatos também é possível, conforme veremos a seguir.

9.1.1 Convertendo um Dataframe entre Formatos *long* e *wide*

A conversão entre um tipo de formato e outro é possível através do pacote `tidyr` (Wickham 2016c). Veja o exemplo a seguir, onde mudamos o formato largo da tabela anterior para longo com o uso da função `gather`. Para isso, é necessário saber qual é a variável de identidade que irá indexar as linhas (nesse caso, a de datas) e quais os nomes das novas colunas.

```
library(tidyr)
library(dplyr)

# set data
refdate <- as.Date('2015-01-01') + 0:3
PETR4 <- c(10, 11, 10.5, 12)
GGBR4 <- c(3, 3.1, 3.2, 3.5)
VALE5 <- c(6, 7, 7.5, 6)

# set dataframe
my.df.wide <- data_frame(refdate, PETR4, GGBR4, VALE5)

# from wide to long
my.df.long <- gather(data = my.df.wide,
                     key = 'ticker',
                     value = 'price', - refdate)

# print it
print(my.df.long)
```

```
## # A tibble: 12 x 3
##    refdate    ticker price
##    <date>     <chr>  <dbl>
##  1 2015-01-01 PETR4   10
##  2 2015-01-02 PETR4   11
##  3 2015-01-03 PETR4   10.5
##  4 2015-01-04 PETR4   12
##  5 2015-01-01 GGBR4    3
```

******ebook converter DEMO Watermarks*******

```
##  6 2015-01-02 GGBR4    3.1
##  7 2015-01-03 GGBR4    3.2
##  8 2015-01-04 GGBR4    3.5
##  9 2015-01-01 VALE5    6
## 10 2015-01-02 VALE5    7
## 11 2015-01-03 VALE5    7.5
## 12 2015-01-04 VALE5    6
```

Para realizar a conversão inversa, de longo para largo, podemos utilizar a função `tidyr::spread`:

```
my.df.wide.converted <- spread(data = my.df.long,
                               key = 'ticker',
                               value = 'price')

print(my.df.wide.converted)
```

```
## # A tibble: 4 x 4
##   refdate    GGBR4 PETR4 VALE5
##   <date>     <dbl> <dbl> <dbl>
## 1 2015-01-01   3    10     6
## 2 2015-01-02   3.1  11     7
## 3 2015-01-03   3.2  10.5   7.5
## 4 2015-01-04   3.5  12     6
```

No caso de conversões mais complexas, onde é necessário agregar algumas das variáveis, recomenda-se o uso do pacote `reshape2` (Wickham 2007), o qual oferece mais funções que o `tidyr`. A sintaxe, porém, é diferente. Veja a seguir, onde utilizamos as funções do pacote `reshape2` para o mesmo procedimento realizado anteriormente.

```
library(reshape2)
my.df.long <- melt(data = my.df.wide,
                   id.vars = 'refdate',
                   variable.name = 'ticker',
                   value.name = 'price')

print(my.df.long)
```

```
##       refdate ticker price
## 1  2015-01-01  PETR4  10.0
## 2  2015-01-02  PETR4  11.0
## 3  2015-01-03  PETR4  10.5
## 4  2015-01-04  PETR4  12.0
## 5  2015-01-01  GGBR4   3.0
## 6  2015-01-02  GGBR4   3.1
## 7  2015-01-03  GGBR4   3.2
## 8  2015-01-04  GGBR4   3.5
## 9  2015-01-01  VALE5   6.0
## 10 2015-01-02  VALE5   7.0
## 11 2015-01-03  VALE5   7.5
## 12 2015-01-04  VALE5   6.0
```

```
my.df.wide.converted <- dcast(data = my.df.long,
                              formula = refdate ~ ticker,
                              value.var = 'price')
print(my.df.wide.converted)
```

```
##      refdate PETR4 GGBR4 VALE5
```

******ebook converter DEMO Watermarks*******

```
## 1 2015-01-01  10.0   3.0   6.0
## 2 2015-01-02  11.0   3.1   7.0
## 3 2015-01-03  10.5   3.2   7.5
## 4 2015-01-04  12.0   3.5   6.0
```

É importante conhecer essas funções ao se trabalhar com o R, pois, muitas vezes, o pesquisador não tem controle sobre o formato dos dados importados. Como regra geral, sempre converta os dados para o formato longo após a importação. Isso facilitará o futuro processamento dos dados.

******ebook converter DEMO Watermarks*******

9.2 Convertendo Listas em Dataframes

Outro caso importante a se destacar na estruturação de dados é a situação onde vários dataframes estão alocados em uma única lista. Isto é bastante comum em dois casos, quando estamos importando dados de diferentes arquivos e também no uso de alguns pacotes de importação de dados da Internet.

Vamos resgatar um caso de estruturação de dados do capítulo 5. Lembre-se que no exemplo de importação de dados do pacote `BETS` observou-se que quando mais de uma série é requisitada, o retorno da função é uma lista. Veja a seguir a replicação do exemplo:

```
library(BETS)
my.id <- 3785:3791

# set dates
first.date = '2010-01-01'
last.date  = as.character(Sys.Date())

# get data
l.out <- BETSget(code = my.id, data.frame = TRUE,
                 from = first.date, to = last.date)

# check data
dplyr::glimpse(l.out)
```

```
## List of 7
##  $ ts_3785:'data.frame': 79 obs. of  2 variables:
##   ..$ date : Date[1:79], format: "2010-01-01" ...
##   ..$ value: num [1:79] 7.4 7.4 7.3 7.2 7 6.9 6.8 6.8 6.7 6.7 ...
##  $ ts_3786:'data.frame': 80 obs. of  2 variables:
##   ..$ date : Date[1:80], format: "2010-01-01" ...
##   ..$ value: num [1:80] 8.3 8.2 8.2 8.1 8 7.9 8.1 8.1 8.1 7.8 ...
##  $ ts_3787:'data.frame': 80 obs. of  2 variables:
##   ..$ date : Date[1:80], format: "2010-01-01" ...
##   ..$ value: num [1:80] 9.8 9.8 9.9 9.9 9.6 9.4 9.4 9.5 9.5 9.4 ...
##  $ ts_3788:'data.frame': 79 obs. of  2 variables:
##   ..$ date : Date[1:79], format: "2010-01-01" ...
##   ..$ value: num [1:79] 9.4 9.4 9.3 9.3 9.3 9.2 9.2 9.2 9.2 9.2 ...
##  $ ts_3789:'data.frame': 79 obs. of  2 variables:
##   ..$ date : Date[1:79], format: "2010-01-01" ...
##   ..$ value: num [1:79] 8.5 8.5 8.4 8.6 8.5 8.3 8.2 8.1 8.1 8.5 ...
##  $ ts_3790:'data.frame': 79 obs. of  2 variables:
##   ..$ date : Date[1:79], format: "2010-01-01" ...
##   ..$ value: num [1:79] 5 5 5.1 5.1 5.1 5.2 5 5.1 5.1 5.1 ...
##  $ ts_3791:'data.frame': 77 obs. of  2 variables:
##   ..$ date : Date[1:77], format: "2010-01-01" ...
##   ..$ value: num [1:77] 7.9 8 7.9 7.8 7.8 7.7 7.7 7.6 7.7 7.8 ...
```

Como usuário, o nosso objetivo é transformar a lista em um único dataframe com três colunas: `date`, `value` e `country`. Esse última coluna não existe e teremos que criá-la.

O primeiro passo para estruturar esses dados em um único dataframe é registrar os nomes dos países:

******ebook converter DEMO Watermarks*******

```
my.countries <- c("Germany", "Canada", "United States", "France",
                  "Italy", "Japan", "United Kingdom")
```

Agora, vamos criar uma função que aceite duas entradas, um dataframe a ser restruturado e um nome do país. Veja a seguir:

```
clean.bets <- function(df.in, country.in) {
  # function for cleaning data from BETS
  #
  # ARGS: df.in - dataframe within a list
  #       name.in - name of country
  #
  # VALUE: a new dataframe with new column type

  #set column
  df.in$country <- country.in

  # return df
  return(df.in)
}
```

Claramente, caso fosse necessário, poderíamos adicionar diversas outras informações aos dataframes. O próximo passo é utilizar a função anterior para criar outra lista com dataframes formatados. Com esse objetivo, usamos a função `purrr::map2` para iterar os elementos da lista em `l.out`. Veja a seguir:

```
library(purrr)

# format dfs
l.out.formatted <- map2(l.out, my.countries, .f = clean.bets)

# check first element (all are the same structure)
glimpse(l.out.formatted[[1]])
```

```
## Observations: 79
## Variables: 3
## $ date    <date> 2010-01-01, 2010-02-01, 2010-03-01, 20...
## $ value   <dbl> 7.4, 7.4, 7.3, 7.2, 7.0, 6.9, 6.8, 6.8,...
## $ country <chr> "Germany", "Germany", "Germany", "Germa...
```

Note que a coluna `country` foi adicionada a cada dataframe. Agora, o próximo passo é mesclar todos elementos da lista em um único dataframe. Para isso, usamos função `do.call`, vista no capítulo 8, ou então `purrr::map_df`. Veja a seguir:

```
# bind all rows of dataframes in list
df.unemp <- map_df(l.out.formatted, bind_rows)

# check it
glimpse(df.unemp)
```

```
## Observations: 553
## Variables: 3
## $ date    <date> 2010-01-01, 2010-02-01, 2010-03-01, 20...
## $ value   <dbl> 7.4, 7.4, 7.3, 7.2, 7.0, 6.9, 6.8, 6.8,...
## $ country <chr> "Germany", "Germany", "Germany", "Germa...
```

******ebook converter DEMO Watermarks*******

E pronto! O dataframe resultante está no formato adequado, pronto para a análise.

******ebook converter DEMO Watermarks*******

9.3 Eliminando Outliers

Uma parte comum a todas pesquisas com dados é a identificação e tratamento de casos extremos (*outliers*). Esses tem potencial de ameaçar toda uma pesquisa. Uma observação extrema pode, por exemplo, resultar na estimação de coeficientes viesados.

Para visualizar o efeito destrutivo de um *outlier* em uma pesquisa, vamos usar o R para simular um processo linear e adicionar um caso extremo na variável independente. O nosso interesse é entender como essa variável pode, de forma isolada, modificar completamente os coeficientes de uma estimação. Veja a seguir:

```
# set options
nT <- 50
sim.x <- rnorm(nT)
my.beta <- 0.5

# simulate x and y
sim.y <- sim.x*my.beta + rnorm(nT)
sim.y.with.outlier <- sim.y

# simulate y with outlier
sim.y.with.outlier[10] <- 50
```

Note que objetos `sim.y` e `sim.y.with.outlier` são exatamente iguais, com exceção da décima observação. Vamos verificar o efeito disso em um modelo de regressão:

```
library(texreg)

# estimate models
model.no.outlier <- lm(formula = sim.y ~ sim.x)
model.with.outlier <- lm(formula = sim.y.with.outlier ~ sim.x)

# report them
screenreg(list(model.no.outlier, model.with.outlier),
          custom.model.names = c('No Outlier', 'With Outlier'))
```

```
##
## ====================================
##              No Outlier  With Outlier
## ------------------------------------
## (Intercept)   0.23        1.34
##              (0.16)      (0.98)
## sim.x         0.85 ***    2.26 *
##              (0.17)      (1.05)
## ------------------------------------
## R^2           0.35        0.09
## Adj. R^2      0.34        0.07
## Num. obs.    50          50
## RMSE          1.10        6.93
## ====================================
## *** p < 0.001, ** p < 0.01, * p < 0.05
```

Aqui reportamos os modelos com o pacote `texreg`. Não se preocupe com o seu uso agora, iremos estudar o reporte de modelos no capítulo 12. Observa-se na tabela de saída que a ******ebook converter DEMO Watermarks*******

inclinação (beta) do modelo modificou-se de 0.847 para 2.255 no caso com *outlier*. Note que uma única observação modificou completamente o modelo. A razão é simples, o modelo priorizou minimizar o grande erro na observação extrema, puxando o coeficiente beta para cima. No caso de estarmos interessados em criar um modelo que explique, de forma razoável, a grande maioria dos dados e não somente uma observação, devemos identificar e tratar os casos *outliers* antes de uma análise.

Uma das maneiras de realizar isso é identificar potenciais valores extremos com os quartis da função. Esses retornam as observações da variável de acordo com uma probabilidade cumulativa. Por exemplo, caso quiséssemos saber qual observação de `sim.y.with.outlier` define o limite de distribuição acumulada em 95%, basta rodar o seguinte código:

```
quantile90 <- quantile(x = abs(sim.y.with.outlier),
                       probs = 0.95)
```

Portanto, as observações a seguir são maiores que 95% da amostra em valor absoluto:

```
idx <- which(sim.y.with.outlier > quantile90)
print(sim.y.with.outlier[idx])
```

```
## [1] 50.000000  2.735142
```

Observe que capturamos o *outlier* definido arbitrariamente, assim como dois outros casos. Caso fôssemos mais rigorosos, poderíamos aumentar a probabilidade para um intervalo maior. Por fim, podemos definir o *outlier* como `NA` e/ou removê-lo da série:

```
# copy content
sim.y.without.outlier <- sim.y.with.outlier

# set NA in outlier
sim.y.without.outlier[idx] <- NA

# remove it
sim.y.without.outlier <- sim.y.without.outlier[-idx]
```

Outra alternativa para a identificação de valores extremos é o uso de testes estatísticos. Particularmente, o pacote `outlier` fornece funções para identificar e remover esses casos com base na distância da média. Veja um exemplo a seguir.

```
library(outliers)

# find outlier
my.outlier <- outlier(sim.y.with.outlier)

# print it
print(my.outlier)
```

```
## [1] 50
```

O resultado, porém, é o mesmo que no uso dos quartis da distribuição.

Outro ponto importante aqui é a possibilidade de procurar e eliminar *outliers* em um dataframe. Imagine um caso em que se está trabalhando com um dataframe com 20 ou mais colunas

******ebook converter DEMO Watermarks*******

numéricas. Definir uma forma de encontrar e substituir *outliers* em cada coluna é desejável. Para isso, podemos utilizar as ferramentas de programação apresentadas no capítulo 8. Lembre-se que todo dataframe é representado como uma lista, onde cada coluna é um elemento. Portanto, podemos processar cada coluna separadamente e depois combiná-las novamente para formar um novo dataframe.

O primeiro passo é definir uma função que aceite uma coluna numérica e uma probabilidade como entrada, retornando o vetor original com os casos extremos substituídos por `NA`. A probabilidade, nesse caso, serve como parâmetro para identificar valores extremos na distribuição cumulativa dos dados. Também é necessário cuidar os casos de colunas que não são numéricas. Por exemplo, para uma coluna do tipo texto, não faz sentido procurar e eliminar *outliers*. Assim, devemos testar a classe e retornar o objeto original caso o mesmo não for da classe `numeric` ou `integer`. Veja a seguir a definição da função:

```
replace.outliers <- function(col.in, my.prob = 0.05) {
  # Replaces outliers from a vector
  #
  # INPUTS: col.in The vector
  #         my.prob Probability of quantiles (p and 1-p)
  #
  # OUTPUT: A vector

  # return if class is other than numeric
  if (!(class(col.in) %in% c('numeric', 'integer'))) return(col.in)

  my.outliers <- stats::quantile(x = col.in,
                                 probs = c(my.prob, 1-my.prob))

  idx <- (col.in <= my.outliers[1])|(col.in >= my.outliers[2])
  col.in[idx] <- NA

  return(col.in)

}
```

Vamos testá-la:

```
# set test vector
my.x <- runif(25)

# find and replace outliers
print(replace.outliers(my.x, my.prob = 0.05))
```

```
##  [1]        NA 0.6495414 0.4684044 0.7038278 0.3067673
##  [6] 0.5286257        NA 0.9242349 0.5955724 0.8180684
## [11] 0.1587992 0.6947190 0.2066385 0.1274163 0.6281963
## [16] 0.2696850 0.4727286 0.5417147 0.1085894        NA
## [21]        NA 0.3611833 0.9352485 0.2785125 0.2784321
```

Como podemos ver, a função funcionou corretamente. O vetor de saída possui alguns elementos `NA`, os quais eram *outliers* no vetor original. Agora, vamos utilizar essa função em um dataframe. Primeiro vamos criar um dataframe com colunas de diferentes classes:

```
# options
my.N <- 100
```

******ebook converter DEMO Watermarks*******

```
# set df
my.df <- data_frame(char1 = sample(LETTERS, my.N, replace = T),
                    factor1 = factor(sample(LETTERS,
                                            my.N,
                                            replace = T)),
                    x1 = runif(my.N),
                    x2 = rnorm(my.N),
                    x3 = rnorm(my.N))

# check it
glimpse(my.df)
```

```
## Observations: 100
## Variables: 5
## $ char1   <chr> "K", "A", "J", "B", "G", "P", "H", "A",...
## $ factor1 <fct> Z, E, A, F, D, H, O, E, L, D, B, Q, M, ...
## $ x1      <dbl> 0.5835005559, 0.3475242094, 0.510483423...
## $ x2      <dbl> 0.08872732, -0.87714352, -1.04845623, 0...
## $ x3      <dbl> -0.43232580, -0.04256174, 1.79623460, -...
```

Agora, utilizamos a função `purrr::map` para iterar todos os seus elementos, construindo novos vetores sem outliers. Veja a seguir:

```
library(purrr)

# remove outlivers from vectors
l.out <- map(my.df, replace.outliers)
```

Em seguida reagrupamos todos os vetores em um dataframe/tibble com função `as.tibble`:

```
# rebuild dataframe
my.df <- as_tibble(l.out)

# check it
glimpse(my.df)
```

```
## Observations: 100
## Variables: 5
## $ char1   <chr> "K", "A", "J", "B", "G", "P", "H", "A",...
## $ factor1 <fct> Z, E, A, F, D, H, O, E, L, D, B, Q, M, ...
## $ x1      <dbl> 0.58350056, 0.34752421, 0.51048342, 0.2...
## $ x2      <dbl> 0.08872732, -0.87714352, -1.04845623, 0...
## $ x3      <dbl> -0.43232580, -0.04256174, NA, -0.980037...
```

```
# summary of df
summary(my.df)
```

```
##     char1             factor1         x1
##  Length:100         E      : 9   Min.   :0.02314
##  Class :character   V      : 7   1st Qu.:0.24995
##  Mode  :character   I      : 6   Median :0.51425
##                     B      : 5   Mean   :0.50051
##                     C      : 5   3rd Qu.:0.74848
##                     D      : 5   Max.   :0.96062
##                     (Other):63   NA's   :10
##        x2                 x3
##  Min.   :-1.3179   Min.   :-1.58155
##  1st Qu.:-0.3435   1st Qu.:-0.67148
```

******ebook converter DEMO Watermarks*******

```
##  Median : 0.1187   Median :-0.08832
##  Mean   : 0.1614   Mean   :-0.03676
##  3rd Qu.: 0.7171   3rd Qu.: 0.51029
##  Max.   : 1.7478   Max.   : 1.71820
##  NA's   :10        NA's   :10
```

Note que os valores `NA` foram encontrados nas três colunas numéricas.

Por fim, eliminamos todos as linhas com outliers usando `na.omit`:

```
my.df <- na.omit(my.df)

glimpse(my.df)
```

```
## Observations: 72
## Variables: 5
## $ char1   <chr> "K", "A", "B", "P", "H", "A", "V", "B",...
## $ factor1 <fct> Z, E, F, H, O, E, L, Q, M, Z, W, H, S, ...
## $ x1      <dbl> 0.58350056, 0.34752421, 0.25691206, 0.0...
## $ x2      <dbl> 0.08872732, -0.87714352, 0.10188010, -1...
## $ x3      <dbl> -0.43232580, -0.04256174, -0.98003734, ...
```

Observe que algumas observações foram perdidas. O dataframe final possui apenas 72 linhas, e não 100 como estipulado inicialmente.

******ebook converter DEMO Watermarks*******

9.4 Desinflando Dados de Preços

Uma operação muito comum em dados econômicos e financeiros é a desinflação. Como sabemos, em uma economia como a Brasileira, a inflação histórica é alta e pode contaminar dados de preços. Esse efeito é maior quanto maior for o período de tempo analisado. De fato, na prática de pesquisa, qualquer preço não ajustado antes de 1994 é inutilizável devido a explosiva inflação encontrada naquela época.

Para tirar o efeito da inflação em dados de preços, o primeiro passo é baixar uma série de inflação. Aqui, iremos utilizar o índice mais popular, IPCA (índice nacional de preços ao consumidor). Utilizaremos o `Quandl` para baixar os dados de inflação anual desde 2010:

```
library(Quandl)
library(dplyr)

# register api key
Quandl.api_key(my.api.key)

# set symbol and dates
my.symbol <- 'BCB/433'
first.date <- as.Date('2010-01-01')
last.date <- Sys.Date()

# get data!
df.inflation <- Quandl(code = my.symbol,
                type='raw',
                collapse = 'annual',
                start_date = first.date,
                order = 'asc',
                end_date = last.date)

# check content
glimpse(df.inflation)

## Observations: 9
## Variables: 2
## $ Date  <date> 2010-12-31, 2011-12-31, 2012-12-31, 2013...
## $ Value <dbl> 0.63, 0.50, 0.79, 0.92, 0.78, 0.96, 0.30,...
```

Agora, vamos construir um dataframe com preços aleatórios a serem desinflacionados:

```
n.T <- nrow(df.inflation)

# create df with prices
my.df <- data_frame(Date = df.inflation$Date,
                    x = 100 + cumsum(rnorm(n.T)),
                    y = 100 + cumsum(rnorm(n.T)))

# check it
glimpse(my.df)

## Observations: 9
## Variables: 3
## $ Date <date> 2010-12-31, 2011-12-31, 2012-12-31, 2013-...
```

******ebook converter DEMO Watermarks*******

```
## $ x    <dbl> 99.86201, 100.36411, 101.10978, 99.59140, ...
## $ y    <dbl> 98.47942, 98.17991, 97.59847, 97.24126, 97...
```

O primeiro passo para desinflacionar é criar um índice deflator, com data base no último período. Como os dados do IPCA já estão em percentagem, basta acumulá-los assumindo um valor inicial de 1 e dividir o resultado pelo valor encontrado na data base:

```
# accumulate: R_a = cumprod(r_t + 1)
my.df$infl.idx <- cumprod(df.inflation$Value/100 +1)

# set inflation index
my.df$infl.idx <- my.df$infl.idx/my.df$infl.idx[nrow(my.df)]
```

E, com isso, multiplicar as colunas para criar novas variáveis:

```
my.df$x.desinflated <- my.df$x*my.df$infl.idx
my.df$y.desinflated <- my.df$y*my.df$infl.idx

glimpse(my.df)
```

```
## Observations: 9
## Variables: 6
## $ Date          <date> 2010-12-31, 2011-12-31, 2012-12-...
## $ x             <dbl> 99.86201, 100.36411, 101.10978, 9...
## $ y             <dbl> 98.47942, 98.17991, 97.59847, 97....
## $ infl.idx      <dbl> 0.9505483, 0.9553011, 0.9628480, ...
## $ x.desinflated <dbl> 94.92367, 95.87794, 97.35335, 96....
## $ y.desinflated <dbl> 93.60945, 93.79137, 93.97249, 94....
```

Pronto. Os dados estão sem o efeito da inflação. Seguindo o exemplo do tópico anterior, seria fácil aplicar este método para todas colunas numéricas com as funções `purrr:map` e `dplyr::as.tibble`.

******ebook converter DEMO Watermarks*******

9.5 Modificando a Frequência Temporal dos Dados

A última estruturação de dados descrita aqui é a modificação da frequência temporal: dados diários para semanais, mensais para anuais ou o contrário. Uma escolha deve ser feita a respeito da forma de agregar os dados. Como transformar um vetor de dados diários para uma valor único mensal? Podemos usar a média, mediana, último valor disponível, entre outras medidas.

Vamos partir para um exemplo com o índice Ibovespa, o qual é disponível na frequência diária e será transformado para frequência anual. O primeiro passo é baixar os dados com `BatchGetSymbols`.

```
library(BatchGetSymbols)

df.ibov <- BatchGetSymbols(tickers = '^BVSP',
                           first.date = '2010-01-01',
                           last.date = '2018-01-01')[[2]]
```

```
##
## Running BatchGetSymbols for:
##    tickers = ^BVSP
##    Downloading data for benchmark ticker | Found cache file
## ^BVSP | yahoo (1|1) | Found cache file - Nice!
```

Uma das maneiras de realizar a operação de mudança de frequência é utilizar o pacote `dplyr` e funções `group_by` e `summarise`. Veja os passos a seguir:

```
library(lubridate)

# from daily to annual
df.ibov.annual <- na.omit(df.ibov) %>%
  mutate(ref.year = year(ref.date)) %>%
  group_by(ref.year) %>%
  summarise(Value = last(price.adjusted)) %>%
  print()
```

```
## # A tibble: 8 x 2
##   ref.year Value
##      <dbl> <dbl>
## 1     2010 69305
## 2     2011 56754
## 3     2012 60952
## 4     2013 51507
## 5     2014 50007
## 6     2015 43350
## 7     2016 60227
## 8     2017 76402
```

Criamos uma nova coluna com os anos, agrupamos de acordo com essa coluna e, por fim, buscamos o último valor do Ibovespa disponível para cada ano. Um dos grande benefícios em utilizar o `dplyr` para operações no tempo é o controle total do processo de agregação. Lendo o código, fica fácil de entender como os dados foram agregados. Caso quiséssemos usar uma medida de tendência tal como média ou mediana, bastaria adicionar novas colunas com o uso das devidas funções.

******ebook converter DEMO Watermarks*******

Usuários interessados em conhecer uma forma mais completa de agregação no tempo, sugiro fortemente o estudo das funções do pacote `xts`. Esse foi feito especialmente para esse tipo de operação. Aqui, a preferência para o uso do `dplyr` é manter e incentivar a intimidade ao material estudado nos capítulos anteriores.

******ebook converter DEMO Watermarks*******

9.6 Exercícios

1. Considere o dataframe criado com o código a seguir:

```
library(dplyr)

my.N <- 100
df <- data_frame(ticker = rep('STOCK1', my.N),
                 ref.date = Sys.Date() + 1:my.N,
                 price = 100 + cumsum(rnorm(my.N)))
```

O formato do dataframe é longo ou largo? Explique sua resposta.

2. Modifique o formato do dataframe anterior, de longo para largo ou vice-versa.

3. Considere a seguinte lista:

```
my.l <- list(df1 = data_frame(x = 1:100, y = runif(100)),
             df2 = data_frame(x = 1:100, y = runif(100)),
             df3 = data_frame(x = 1:100, y = runif(100)) )
```

Agrege todos dataframes para um objeto único usando funções `do.call` e `dplyr::bind_rows`.

4. Utilize pacote `BatchGetSymbols` para baixar os dados do índice SP500 (`'^GSPC'`) desde 1950-01-01 até hoje. Quais são os 5 retornos absolutos (positivo ou negativo) do índice? Crie e apresente na tela um dataframe com os valores e datas desses retornos extremos.

5. Com base na função `BatchGetSymbols::BatchGetSymbols`, baixe os preços do índice Ibovespa desde `'2010-01-01'` até hoje. Em outro dataframe, baixe os dados do IPCA utilizando pacote `rbcb`. Crie uma nova coluna nos dados do Ibovespa com o valor desinflacionado do índice ao longo do tempo.

6. Utilize pacote `GetHFData` para baixar dados de transações financeiras para ação `GGBR3` para o período dos últimos 30 dias. Com pacote `dplyr`, transforme os dados intradiários para diário, salvando o preço e volume médio das transações.

7. **DESAFIO** - Para os dados do SP500 baixados anteriormente, verifique as datas e os preços das 20 maiores quedas de preço. Caso um investidor comprar e manter o índice por 30 dias após cada uma dessas quedas, qual seria o retorno médio dele?

8. **DESAFIO** - A função `rbcb::get_all_currencies` permite baixar cotações de moedas estrangeiras para uma determinada data. Utilizando pacote `purrr`, baixe os valores para uma sequência de datas de `2018-01-01` até 2018-06-10. Organize e estruture os dados em um único dataframe.

******ebook converter DEMO Watermarks*******

******ebook converter DEMO Watermarks*******

10 Criando e Salvando Figuras

O uso de recursos gráficos em relatórios técnicos e artigos acadêmicos é bastante disseminado. Em muitas situações, isso simplesmente é o que sua audiência espera. O R possui algumas funções nativas para a criação de figuras, tal como `plot` e `hist`. A forma de utilizar essas e outras funções nativas, porém, é restritiva e não recomendada. A customização dos gráficos não é intuitiva e o resultado final pode não ser visualmente atrativo. De fato, no passado, uma das críticas ao R era a limitada capacidade do programa na criação de figuras a serem utilizadas em publicações científicas.

Assim como para outros casos, essa deficiência foi sanada por Hadley Hickman. Ele propôs uma forma de estruturar e criar figuras no R com o pacote `ggplot2` (Wickham 2009), oferecendo funções para produzir gráficos de alta qualidade, estruturando o processo através de uma notação acessível e modular. Na prática, isso significa que gráficos podem ser criados e customizados facilmente.

Neste livro, não pretendemos entrar a fundo na forma de trabalhar com o `ggplot2` e em todos os detalhes da sua capacidade. Iremos estudar as principais funcionalidades do pacote nos casos mais comuns da área de Finanças e Economia. Usuários avançados que desejam conhecer mais sobre as capacidades do `ggplot2`, a minha orientação é consultar o livro do próprio autor, Wickham (2009).

Para todos os exemplos dados aqui, iremos trabalhar com o dataframe do arquivo `IbovStocks_long.csv`. Esse contém dados de preços de fechamento diários de diversas ações negociadas na B3. Conforme já indicado anteriormente, esse arquivo está disponibilizado no site do livro.

Primeiro, vamos carregar os dados e conhecer as colunas do dataframe importado:

```
library(dplyr)
library(readr)

# set files and column types
my.f <- 'data/IbovStocks_long.csv'
my.col.types <- cols(price.adjusted =col_double(),
                     ref.date = col_date(format = ""),
                     ticker = col_character())

# read file
my.df <- read_csv(my.f, col_types = my.col.types)

glimpse(my.df)
```

```
## Observations: 105,947
## Variables: 3
## $ ref.date       <date> 2010-01-04, 2010-01-05, 2010-01...
## $ ticker         <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV...
## $ price.adjusted <dbl> 2.736836, 2.752935, 2.781019, 2....
```

******ebook converter DEMO Watermarks*******

Antes de processar, vamos criar uma nova coluna com os retornos aritméticos dos ativos. Vamos reutilizar uma versão mais simples da função `calc.ret` desenvolvida no capítulo 8:

```
calc.ret <- function(P, tickers = rep('ticker', length(P))) {
  # calculates arithmetic returns from a vector of prices
  #
  # Args:
  #   P - vector of prices (numeric)
  #   tickers - vector of tickers (optional)
  #
  # Returns:
  #   A vector of returns

  # ret = p_{t}/p_{t-1} - 1

  my.length <- length(P)
  ret <- c(NA, P[2:my.length]/P[1:(my.length - 1)] - 1)

  idx <- (tickers != dplyr::lag(tickers))
  ret[idx] <- NA

  return(ret)
}
```

E adicionamos ao dataframe:

```
my.df <- my.df %>%
  mutate(ret = calc.ret(price.adjusted, ticker)) %>%
  glimpse()
```

```
## Observations: 105,947
## Variables: 4
## $ ref.date       <date> 2010-01-04, 2010-01-05, 2010-01...
## $ ticker         <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV...
## $ price.adjusted <dbl> 2.736836, 2.752935, 2.781019, 2....
## $ ret            <dbl> NA, 0.0058823400, 0.0102014759, ...
```

Pronto. Com base nos dados em `my.df`, partimos para a criação de figuras.

******ebook converter DEMO Watermarks*******

10.1 Criando Janelas de Gráficos com `x11`

Antes de apresentar os códigos para a criação de figuras no R, é necessário entender como essas imagens são administradas na plataforma e no RStudio. Caso uma figura seja criada tal como em `plot(1:20)`, a mesma aparecerá no painel *Plots* do RStudio (no canto inferior direito). O problema dessa abordagem é que o painel é pequeno, dificultando a análise visual do gráfico. É possível aumentar a área do painel manualmente, porém, isso gera um trabalho desnecessário: provavelmente terás que redimensionar logo após a análise da figura para dar espaço para os demais painéis.

Uma abordagem mais inteligente para a criação de figuras é criar uma janela externa ao RStudio, de forma que o gráfico possa ser visualizado fora da interface principal do programa. Para criar uma janela, basta utilizar o comando `x11()` antes da criação do gráfico, tal como em:

```
x11()
plot(1:10)
```

O resultado na tela do RStudio deve ser algo próximo à figura 10.1.

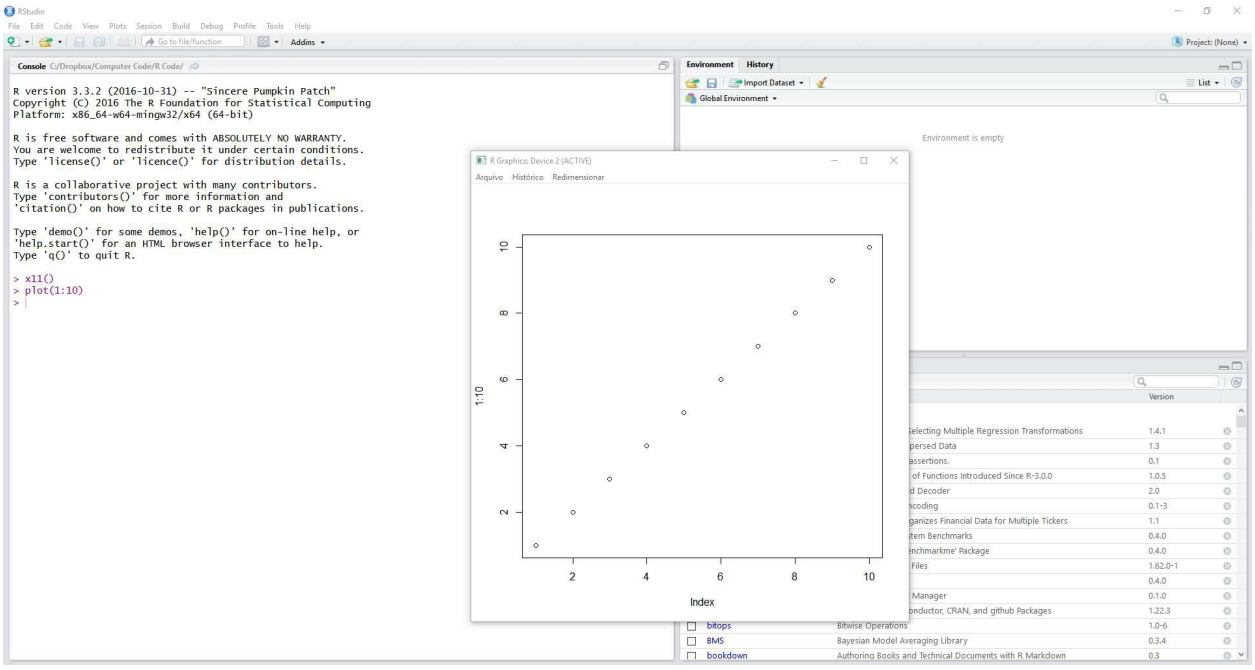


Figura 10.1 Tela de exemplo do uso do comando x11()

Destaca-se que cada chamada de `x11()` criará uma nova janela. Portanto, várias figuras poderiam ser alocadas em janelas diferentes, facilitando a visualização.

Depois de criar tantos janelas no R, é importante conhecer o procedimento para fechá-las! A função que faz isso é `graphics.off`. Quando chamada sem argumento, tal como em `graphics.off()`, todas janelas abertas do R serão fechadas. É prática comum usar essa função

******ebook converter DEMO Watermarks*******

no início de uma rotina de pesquisa, certificando-se que todas as janelas sejam recriadas, e não readicionadas a área de trabalho.

******ebook converter DEMO Watermarks*******

10.2 Criando Figuras com `qplot`

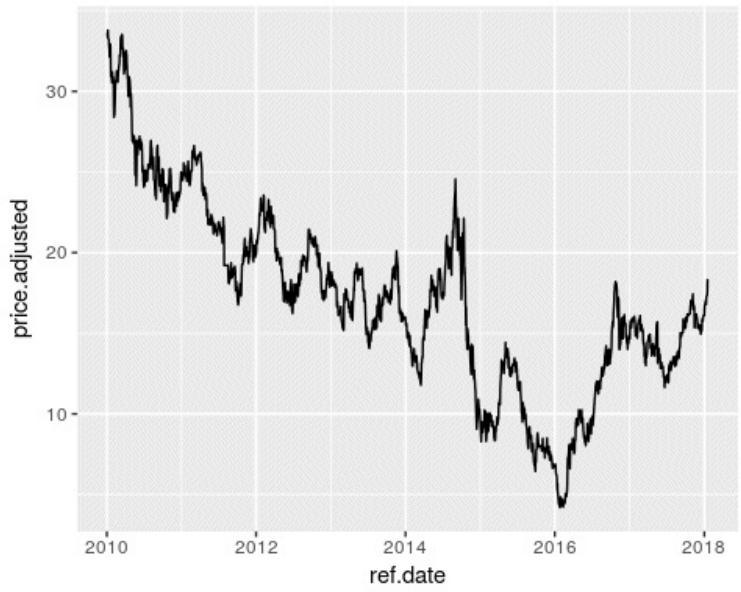
O pacote `ggplot2` possui uma função que imita o comportamento da função nativa `plot` do R. Essa função é chamada de `qplot` (*quick plot*). Para utilizá-la, basta saber os pontos que definem o eixo horizontal (x), os pontos do eixo vertical (y) e o tipo de objeto geométrico que irá definir o gráfico.

Por exemplo, caso formos construir um gráfico de linha dos preços da ação `PETR4` em relação ao tempo, basta utilizar o código a seguir:

```
library(ggplot2)
library(dplyr)

# filter stock data
temp.df <- filter(my.df, ticker == 'PETR4')

# plot its prices
qplot(data = temp.df,
      x = ref.date,
      y = price.adjusted,
      geom = 'line')
```

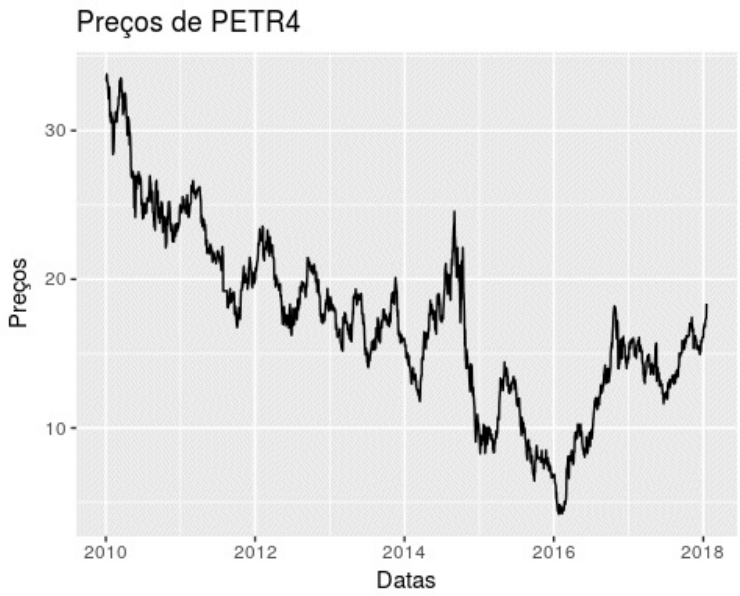


Observe que, no exemplo anterior, os nomes dos eixos correspondem aos nomes das colunas do dataframe. Caso quiséssemos customizá-los para um texto qualquer, bastaria utilizar os argumentos `xlab` e `ylab`.

```
qplot(data = temp.df,
      x = ref.date,
      y = price.adjusted,
      geom = 'line',
      main = 'Preços de PETR4',
      xlab = 'Datas',
```

******ebook converter DEMO Watermarks*******

```
       ylab = 'Preços')
```



Destaca-se que o eixo horizontal de datas na figura anterior já está formatado para apresentar os anos. A própria função interpreta a classe de datas e, automaticamente, formata o eixo. Caso não tivéssemos importado a coluna `ref.date` como um objeto da classe `Date`, isso não aconteceria.

******ebook converter DEMO Watermarks*******

10.3 Criando Figuras com `ggplot`

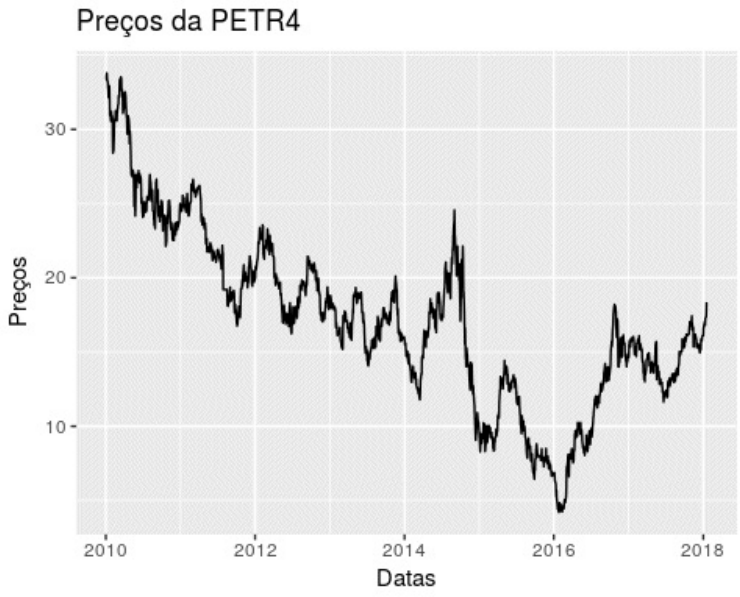
O uso do `qplot` é recomendado para a criação de figuras de forma rápida, para alguma visualização preliminar e imediata dos dados. Na prática, recomenda-se o uso da função `ggplot`. Essa permite uma série de construções mais complexas ao possibilitar o uso de camadas na figura.

Antes de apresentarmos exemplos de uso de `ggplot2::ggplot`, vamos discutir a filosofia por trás da criação de figuras. Em primeiro lugar, toda figura tem pontos de coordenadas horizontais e verticais. São esses pontos que definem onde os símbolos ou linhas serão mostrados. Em segundo lugar, temos o formato do objeto que irá ser posto nas coordenadas. Ele pode ser um círculo, um ponto ou mesmo uma linha que interliga os pontos, como nas figuras anteriores. Por fim, temos o tamanho e a cor desses objetos que irão formar a figura. Ao combinar essas seleções entre si é que formamos, afinal, a figura completa.

Essa distinção entre as etapas de formação de figuras é importante, pois é exatamente dessa maneira que o uso da função `ggplot2::ggplot` está estruturado. Efetivamente, os códigos a serem utilizados irão definir cada uma das opções anteriores. Quando tomadas juntas, essas escolhas irão compor uma figura.

Veja a sintaxe do código no exemplo a seguir, onde se recria a figura anteriormente gerada com a função `qplot`.

```
p <- ggplot(data = temp.df, aes(x = ref.date, y = price.adjusted))
p <- p + geom_line()
p <- p + labs(x = 'Datas', y = 'Preços', title  = 'Preços da PETR4')
print(p)
```



No uso da função `ggplot`, o argumento `data` é o dataframe com os dados. Vale destacar que essa

******ebook converter DEMO Watermarks*******

função somente funciona com esse tipo de objeto. Caso o usuário queira criar figuras a partir de vetores atômicos, terá que salvar os vetores como colunas em um dataframe e oferecer o objeto resultante como entrada em `ggplot2`.

Após a definição dos dados, temos o uso da função `aes`. Ela define a estética (*aesthetics*) do gráfico, assim como as coordenadas *x* e *y*. Nesse caso, observamos que o eixo horizontal é dado pela coluna `ref.date` e o eixo vertical pelos preços da ação (coluna `price.adjusted`). Em alguns casos específicos, também é possível colocar outras informações, tal como a cor e o formato dos pontos. Logo veremos exemplos disso.

Observa-se que, na primeira linha do código anterior, o resultado é salvo no objeto `p`. Esse objeto **salva as informações da figura e pode ser incrementado com o sinal +**, conforme é visto na segunda linha. Nela, define-se o tipo geométrico que irá criar o gráfico com linhas pela função `geom_lines`. Todas as funções que definem o tipo geométrico começam com *geom*. Como exercício de exploração, vamos verificar quais funções de `ggplot2` satisfazem essa condição:

```
library(ggplot2)
library(stringr)

# get names of functions in ggplot2
fcts <- ls('package:ggplot2')

# select those that starts with geom_
idx <- str_sub(fcts, 1, 5) == 'geom_'
fcts <- fcts[idx]

# print result
print(fcts)
```

```
##  [1] "geom_abline"     "geom_area"       "geom_bar"
##  [4] "geom_bin2d"      "geom_blank"      "geom_boxplot"
##  [7] "geom_col"        "geom_contour"    "geom_count"
## [10] "geom_crossbar"   "geom_curve"      "geom_density"
## [13] "geom_density2d"  "geom_density_2d" "geom_dotplot"
## [16] "geom_errorbar"   "geom_errorbarh"  "geom_freqpoly"
## [19] "geom_hex"        "geom_histogram"  "geom_hline"
## [22] "geom_jitter"     "geom_label"      "geom_line"
## [25] "geom_linerange"  "geom_map"        "geom_path"
## [28] "geom_point"      "geom_pointrange" "geom_polygon"
## [31] "geom_qq"         "geom_quantile"   "geom_raster"
## [34] "geom_rect"       "geom_ribbon"     "geom_rug"
## [37] "geom_segment"    "geom_smooth"     "geom_spoke"
## [40] "geom_step"       "geom_text"       "geom_tile"
## [43] "geom_violin"     "geom_vline"
```

Como visto, temos muitas opções. Iremos explorar algumas destas possibilidades nas próximas seções.

Voltando ao exemplo, a terceira linha do código define o nome dos eixos *x* e *y* com o comando `labs`. Por fim, apresenta-se a figura com o comando `print`. É importante destacar esse caráter modular no uso da função `ggplot`: cada camada da figura foi criada em uma linha do código. Por exemplo, caso não quiséssemos definir os eixos como *Datas* e *Preços*, bastaria comentar essas linhas do *script*.

******ebook converter DEMO Watermarks*******

O uso do *pipeline* também é possível. Para exemplificar, veja o mesmo código anterior em formato de *pipeline*:

```
p <- temp.df %>%
  ggplot(aes(x = ref.date, y = price.adjusted)) +
  geom_line() +
  labs(x = 'Datas', y = 'Preços')

print(p)
```

Note que usamos `+` e não `%>%` para interligar cada camada do gráfico após a chamada de `ggplot`. A partir de agora iremos usar e abusar da notação de *pipeline* para construir as figuras.

O uso do comando `ggplot` mostra as suas vantagens quando na criação de figuras com dados de diferentes grupos. Por exemplo: vamos criar uma figura que mostre, no mesmo eixo, os preços de quatro ações selecionadas aleatoriamente. O primeiro passo é criar um dataframe temporário que contenha apenas essas ações.

```
# fix seed
set.seed(10)

# select 4 stocks randomly
my.tickers <- sample(unique(my.df$ticker), 4)

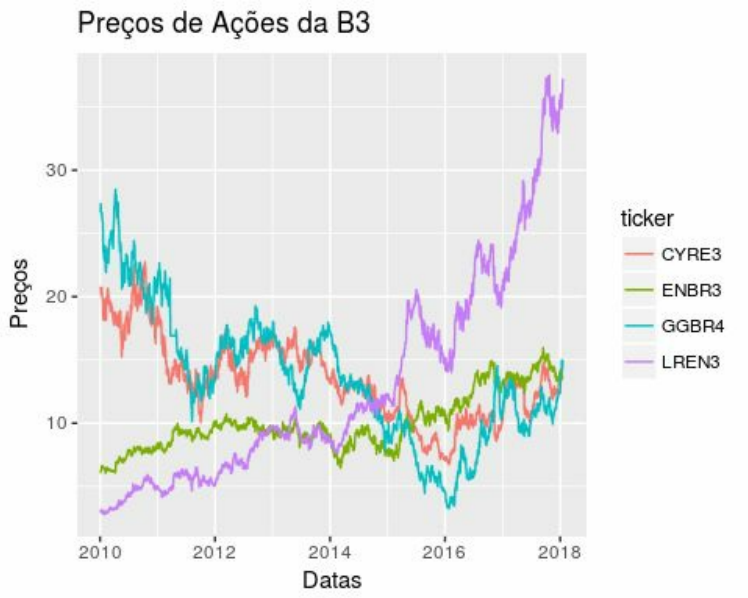
# filter df for all rows that contain the stocks
temp.df <- filter(my.df, ticker %in% my.tickers)
```

Nesse código, utilizamos o operador `%in%` e `filter` para verificar quais linhas em `my.df` contêm dados dos referidos *tickers*. A partir disso, cria-se a figura com o seguinte código:

```
p <- temp.df %>%
  ggplot(aes(x = ref.date, y = price.adjusted, color=ticker)) +
  geom_line() +
  labs(x = 'Datas', y = 'Preços', title = 'Preços de Ações da B3')

print(p)
```

******ebook converter DEMO Watermarks*******



A diferença para o exemplo anterior é que usamos argumento `color` em `aes`. As cores das linhas serão definida pelos elementos da coluna `ticker`. A escolha exata da cor, isto é, vermelho, azul, etc, é automaticamente realizada por `ggplot`. Note como foi fácil e rápido ajustar o código do `ggplot` para apresentar a nova figura.

10.3.1 A Curva de Juros Brasileira

Agora, vamos usar o que aprendemos até aqui para criar a curva de juros atual da economia Brasileira. O primeiro passo é baixar os dados com `GetTDData`:

```
library(GetTDData)
library(dplyr)

# get BR yield curve
df.yield.curve <- get.yield.curve()

# check it
glimpse(df.yield.curve)
```

```
## Observations: 125
## Variables: 5
## $ n.biz.days   <dbl> 126, 252, 378, 504, 630, 756, 882,...
## $ type         <chr> "real_return", "real_return", "rea...
## $ value        <dbl> 1.3566, 2.9230, 3.8922, 4.4996, 4....
## $ ref.date     <date> 2018-10-12, 2019-02-15, 2019-06-2...
## $ current.date <date> 2018-06-08, 2018-06-08, 2018-06-0...
```

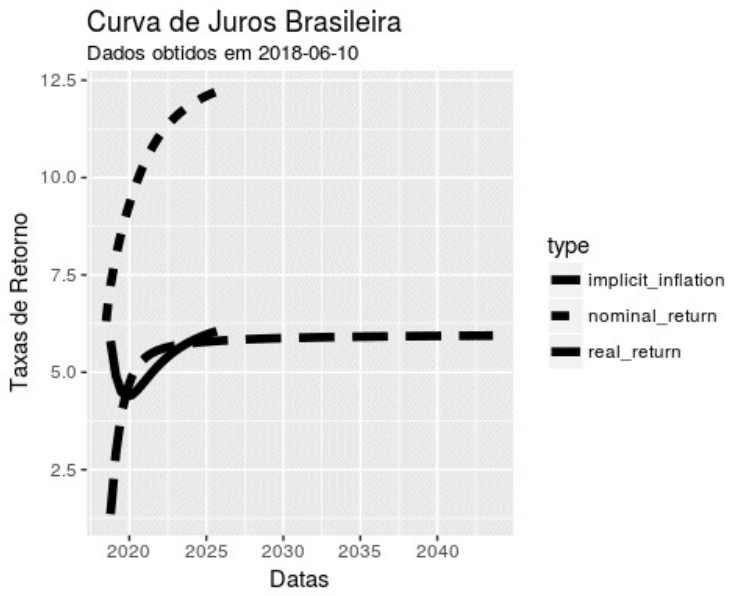
Felizmente, os dados já estão organizados e limpos. Para a construção da curva, temos `ref.date` no eixo `x`, `value` no eixo `y` e coluna `type` para os tipos de linhas. Traduzindo para o R:

```
p <- df.yield.curve %>%
  ggplot(aes(x = ref.date, y = value, linetype = type)) +
  geom_line(size =2) +
  labs(x = 'Datas', y = 'Taxas de Retorno',
```

******ebook converter DEMO Watermarks*******

```
       title = 'Curva de Juros Brasileira',
       subtitle = paste0('Dados obtidos em ', Sys.Date()) )

print(p)
```



O gráfico resultante não está ideal, a curva de juros real tem mais dados que a curva de juros nominal e inflação. Na seção 10.3.3 veremos uma maneira mais intuitiva de plotar esse gráfico com o uso de painéis.

10.3.2 Usando Temas

Caso estejas pouco satisfeito com as cores escolhidas automaticamente por `ggplot` na sua figura, uma forma de personalizar gráficos é usar função `themes`. Um tema é uma coleção de opções que define a organização da figura, seus pontos e cores de linha, notação de eixo, cor de fundo e vários outros recursos. Assim como para outros casos, o pacote `ggplot` possui uma coleção de funções para definir temas, e seu nome começa com o texto *theme*. Veja a lista a seguir:

```
library(ggplot2)
library(stringr)

# get all functions
fcts <- ls('package:ggplot2')

# find out those that start with theme_
idx <- str_sub(fcts, 1, 6) == 'theme_'
fcts <- fcts[idx]

# print result
print(fcts)
```

```
##  [1] "theme_bw"       "theme_classic"  "theme_dark"
##  [4] "theme_get"      "theme_gray"     "theme_grey"
##  [7] "theme_light"    "theme_linedraw" "theme_minimal"
## [10] "theme_replace"  "theme_set"      "theme_update"
```

******ebook converter DEMO Watermarks*******

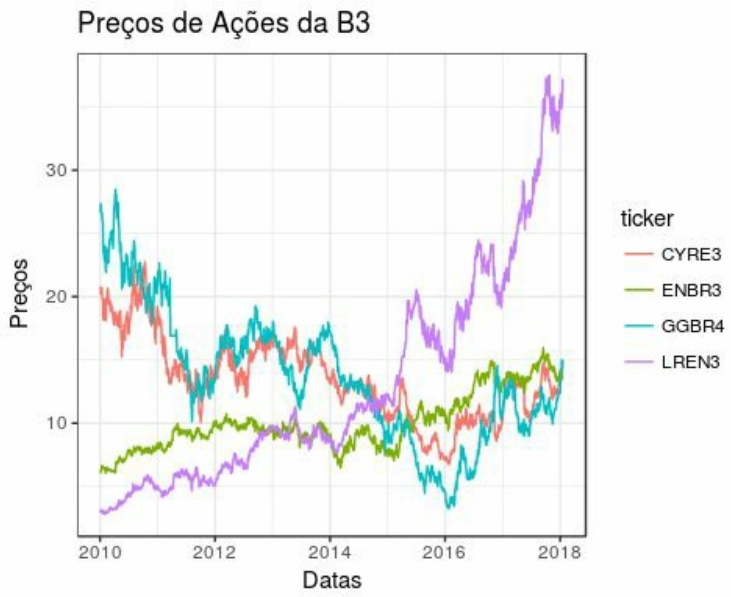
```
## [13] "theme_void"
```

Vamos dar um exemplo de uso com `theme_bw`. Do próprio manual: "theme_bw sets the classic dark-on-light ggplot2 theme. May work better for presentations displayed with a projector". Para usar no gráfico, basta adicionar uma nova linha ao *pipeline*:

```
set.seed(15)

# get assets
p <- temp.df %>%
  ggplot(aes(x = ref.date, y = price.adjusted, color = ticker)) +
  geom_line() +
  labs(x = 'Datas', y = 'Preços', title = 'Preços de Ações da B3') +
  theme_bw()

print(p)
```



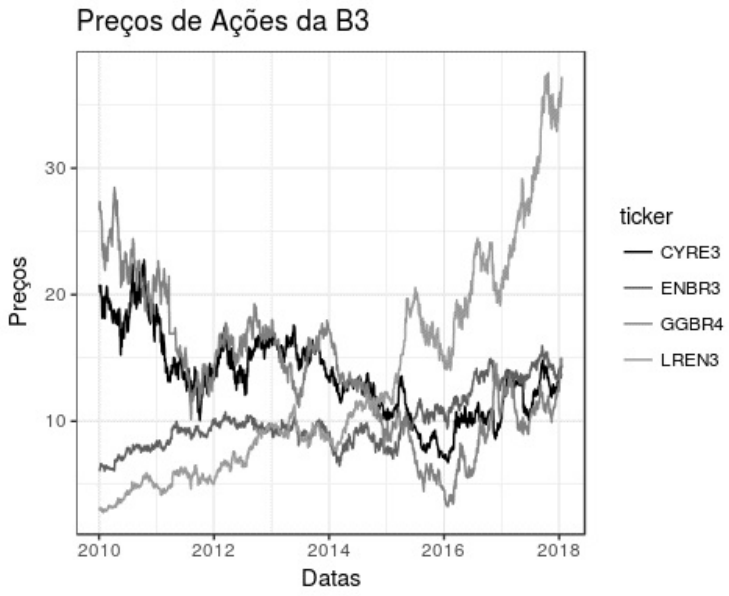
Como você pode ver, o novo tema define um fundo branco e uma caixa de quadro. Você pode testar outros temas em seu computador e ver qual deles mais gosta. Você também pode criar o seu próprio tema e aplicar a qualquer figura. Dê uma olhada em Wickham (2009) para obter instruções sobre essa tarefa específica.

No exemplo anterior, observe como a estrutura da figura mudou, mas não as cores das linhas. No `ggplot`, a seleção de cores segue um ciclo próprio. Quando a sequência completa das cores termina, ela reinicia. Às vezes, especialmente na submissão de artigos a revistas científicas, espera-se que todas as figuras sejam criadas com tons de cinza. Para isto, podemos usar a função `scale_colour_grey` para definir um ciclo de cores entre branco e preto em nossa figura anterior:

```
p <- temp.df %>%
  ggplot(aes(x = ref.date, y = price.adjusted, color = ticker)) +
  geom_line() +
  labs(x = 'Datas', y = 'Preços', title = 'Preços de Ações da B3') +
  theme_bw() +
  scale_colour_grey(start = 0.0, end = 0.6)
```

******ebook converter DEMO Watermarks*******

```
print(p)
```



A figura agora está no formato preto e branco, conforme desejado. As entradas `start` e `end` em `scale_color_grey` definem o máximo e mínimo de *brancura* das linhas. Portanto, nunca use `end=1` com um plano de fundo branco. Caso contrário, as linhas não serão visíveis.

10.3.3 Criando Painéis com `facet_wrap`

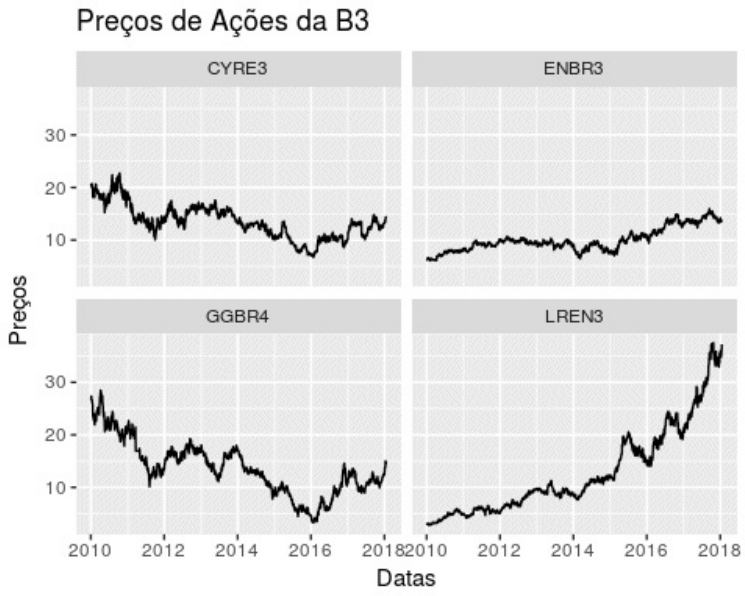
Outra possibilidade para produzir figuras de grupos de dados diferentes é a criação de painéis. Esses separam cada grupo em uma figura isolada. Ao serem colocadas lado a lado, com o mesmo eixo, dispomos de uma comparação mais clara entre os casos.

Essa operação é executada com o uso da função `facet_wrap`, a qual toma como entrada uma fórmula contendo a coluna com os grupos que irão definir cada painel. Veja, no exemplo a seguir, como se utiliza a função `facet_wrap` com a opção `facets = ~ticker` para criar um painel de gráficos de preços:

```
p <- temp.df %>%
  ggplot(aes(x = ref.date, y = price.adjusted)) +
  geom_line() +
  labs(x = 'Datas', y = 'Preços', title = 'Preços de Ações da B3') +
  facet_wrap(~ticker)

print(p)
```

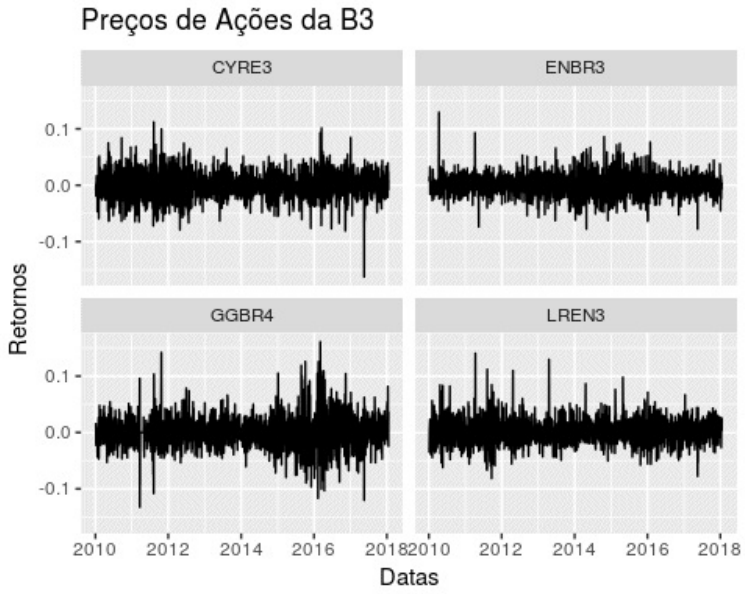
******ebook converter DEMO Watermarks*******



O uso de painéis é recomendado quando os dados dos grupos são semelhantes e tendem a se aglomerar. Isso dificulta a visualização em um gráfico único. Esse é o caso, por exemplo, dos retornos de ações. Veja o exemplo a seguir, onde se cria um painel para os retornos de cada ação.

```
p <- temp.df %>%
  ggplot(aes(x = ref.date, y = ret)) +
  geom_line() +
  labs(x = 'Datas', y = 'Retornos', title = 'Preços de Ações da B3') +
  facet_wrap(~ticker)

print(p)
```



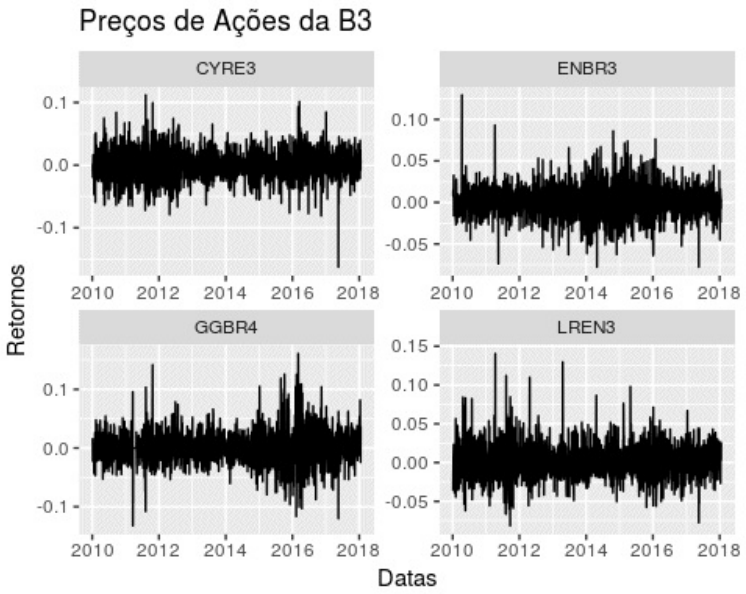
Vale destacar que o eixo vertical dos painéis é fixo para todos os elementos. Isso facilita a comparação entre os dados. Caso faça mais sentido não fixar os eixos, utilize o argumento

******ebook converter DEMO Watermarks*******

`scales='free'` em `facet_wrap`:

```
p <- temp.df %>%
  ggplot(aes(x = ref.date, y = ret)) +
  geom_line() +
  labs(x = 'Datas', y = 'Retornos',
       title = 'Preços de Ações da B3') +
  facet_wrap(~ticker, scales = 'free')

print(p)
```



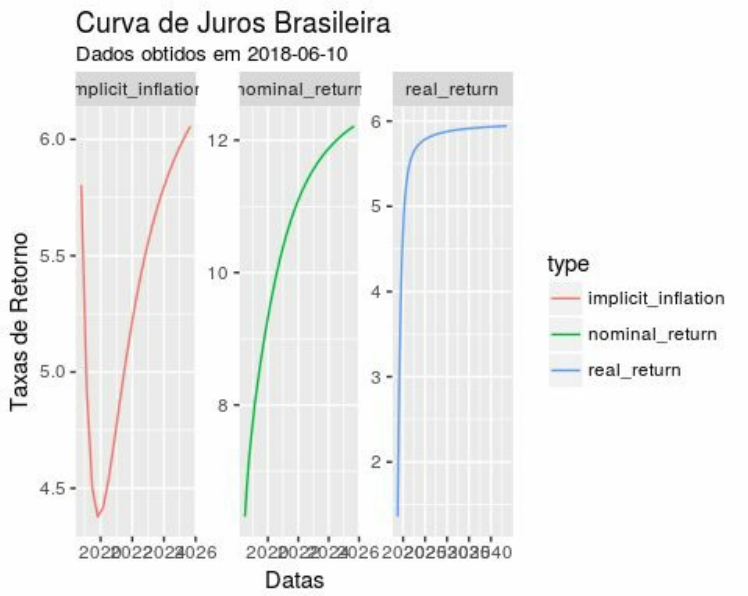
Note que agora cada grupo de dados possui seus próprios eixos verticais e horizontais na figura.

Voltando ao exemplo da curva de juros Brasileira, vamos utilizar o `facet_wrap` com `scales=free` para deixar a interpretação da curva mais fácil:

```
p <- df.yield.curve %>%
  ggplot(aes(x = ref.date, y = value, color = type)) +
  geom_line() +
  labs(x = 'Datas', y = 'Taxas de Retorno',
       title = 'Curva de Juros Brasileira',
       subtitle = paste0('Dados obtidos em ', Sys.Date())) +
  facet_wrap(~type, scale = 'free')

print(p)
```

******ebook converter DEMO Watermarks*******



Como podemos ver, é mais fácil de analisar o formato de cada tipo de juros usando painéis próprios.

******ebook converter DEMO Watermarks*******

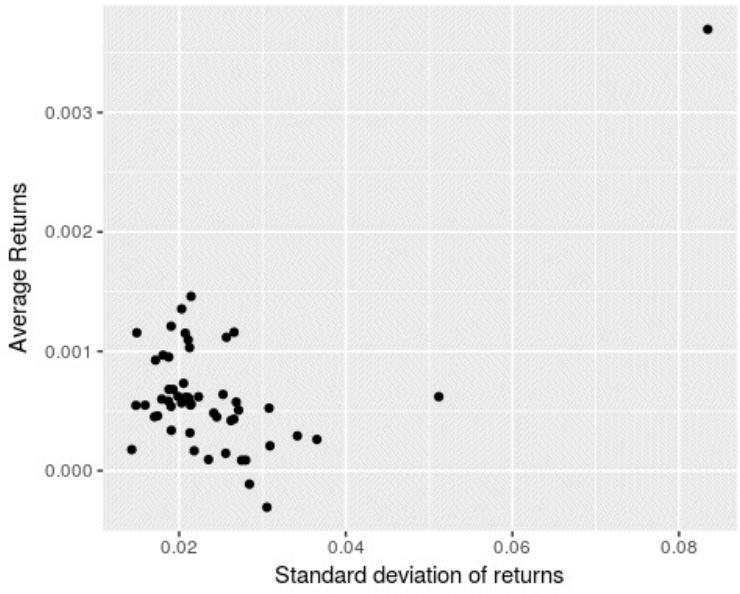
10.4 Uso do Operador *pipeline*

Como já vimos anteriormente, a função `ggplot` é amiga do operador de *pipeline*. Aqui iremos mais além para mostrar como escrever um código onde o processamento de dados e criação de figura são integrados no mesmo bloco de código. Veja o próximo exemplo, onde calculamos o retorno médio e o desvio padrão de todas as ações do conjunto de dados e construímos um gráfico:

```
library(dplyr)
library(readr)
library(ggplot2)

# import data, calculate mean and sd of returns, plot result
p <- read_csv(file = 'data/IbovStocks_long.csv',
              col_types = cols(ref.date = col_date(format = ""),
                               ticker = col_character(),
                               price.adjusted = col_double())) %>%
  mutate(ret = calc.ret(price.adjusted, ticker)) %>%
  na.omit(.) %>%
  group_by(ticker) %>%
  summarise(mean.ret = mean(ret),
            std.ret = sd(ret)) %>%
  ggplot(aes(x = std.ret, y = mean.ret)) +
  geom_point() +
  labs(x = 'Standard deviation of returns',
       y = 'Average Returns')


print(p)
```



O código anterior é autocontido, fácil de ler e elegante. Note que, em todo o processo, nenhum objeto intermediário foi criado. Dados brutos de retornos são importados diretamente do arquivo, uma nova coluna é criada e, por fim, médias e desvios padrões são calculadas para cada ação, os

******ebook converter DEMO Watermarks*******

quais são repassados para a criação do gráfico. Podemos ver claramente as etapas tomadas para construir a figura.

******ebook converter DEMO Watermarks*******

10.5 Criando Figuras Estatísticas

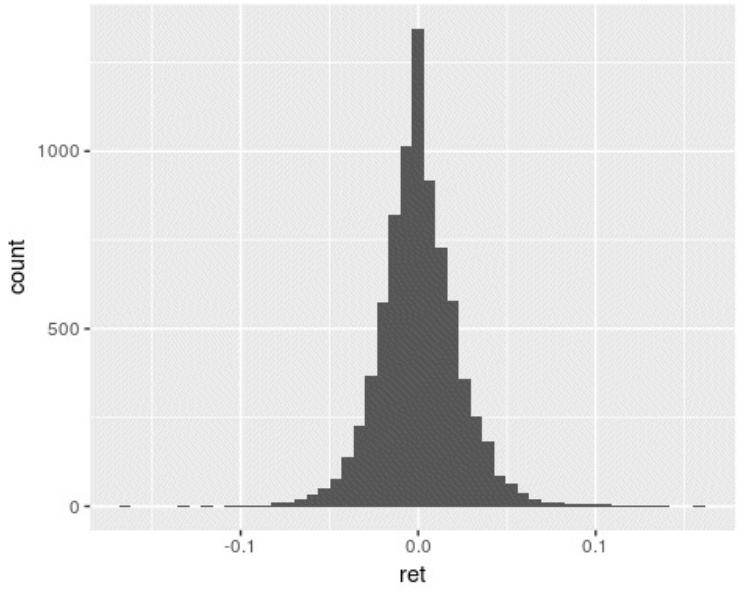
Pacote `ggplot2` inclui diversas opções para criar gráficos com interpretação estatística. Isso inclui histogramas, *boxplots*, gráficos *QQ*, entre outros. Vamos explorar cada caso com dados financeiros.

10.5.1 Criando Histogramas

Um histograma mostra a distribuição empírica dos dados. Esse pode ser facilmente construído com funções `ggplot` e `geom_histogram`. No código a seguir, geramos o histograma dos retornos de todas as ações, sem a segmentação por *ticker*.

```
p <- temp.df %>%
  na.omit(.) %>%
  ggplot(aes(x = ret)) +
  geom_histogram(bins = 50)

print(p)
```



Observa-se que, nesse caso, define-se apenas um eixo *x*, uma vez que essa é a única informação necessária para criar um histograma. O detalhamento da figura, isto é, o tamanho de cada intervalo no eixo *x* para a contagem dos pontos, é definido pela entrada `bins`. Quanto maior o seu valor, maior o nível de detalhamento do histograma.

Para o caso de diferentes grupos de dados, o uso de painéis é realizado da mesma forma que no exemplo anterior. Basta adicionar a chamada à função `facet_wrap` no código.

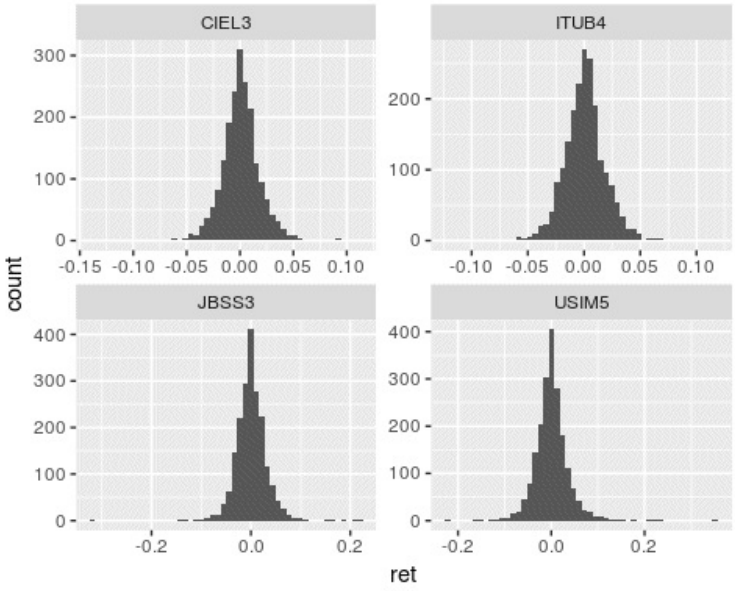
```
# fix seed
set.seed(15)
```

******ebook converter DEMO Watermarks*******

```
# select 4 stocks randomly
my.tickers <- sample(unique(my.df$ticker), 4)

p <- my.df %>%
  na.omit(.) %>%
  filter(ticker %in% my.tickers) %>%
  ggplot(aes(x = ret)) +
  geom_histogram(bins = 50) +
  facet_wrap(facets = ~ticker, scales = 'free')

print(p)
```



Também é possível a criação de densidades a partir dos dados com a função `geom_density`. Densidades nada mais são do que interpolações dos valores encontrados no gráfico principal, formando uma linha que define as frequências relativas. Veja a seguir:

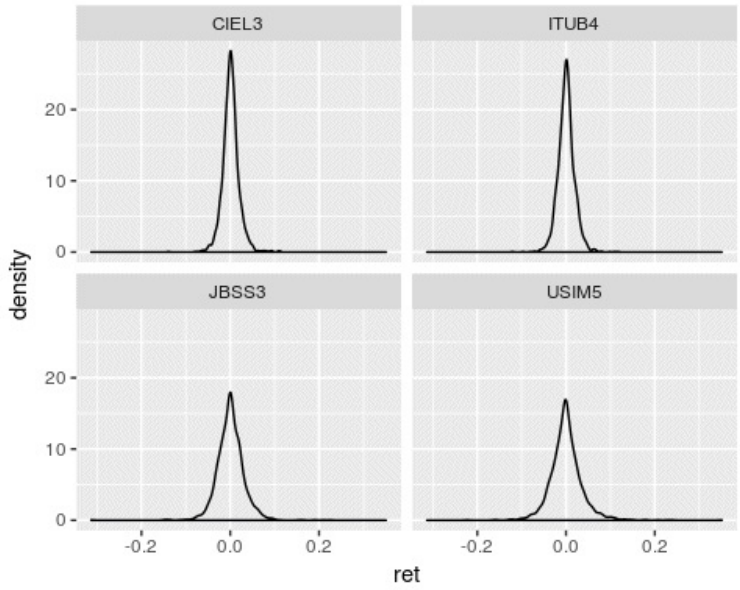
```
# fix seed
set.seed(15)

# select 4 stocks randomly
my.tickers <- sample(unique(my.df$ticker), 4)

p <- my.df %>%
  na.omit(.) %>%
  filter(ticker %in% my.tickers) %>%
  ggplot(aes(x = ret)) +
  geom_density() +
  facet_wrap(facets = ~ticker)

print(p)
```

******ebook converter DEMO Watermarks*******



A figura anterior permite uma comparação visual mais clara das diferenças entre as distribuições de retornos das diferentes ações.

10.5.2 Criando Figuras *Boxplot*

As figuras do tipo *boxplots* (ou diagrama *box and whisker*) servem para mostrar a distribuição de variáveis separadas por alguma categoria. Usando a mediana, máximo, mínimo e quartis dos dados, uma representação visual salienta as diferenças de distribuição entre cada grupo. Veja o exemplo a seguir, onde mostramos a diferença de distribuição de preços de cada ativo na base de dados analisada:

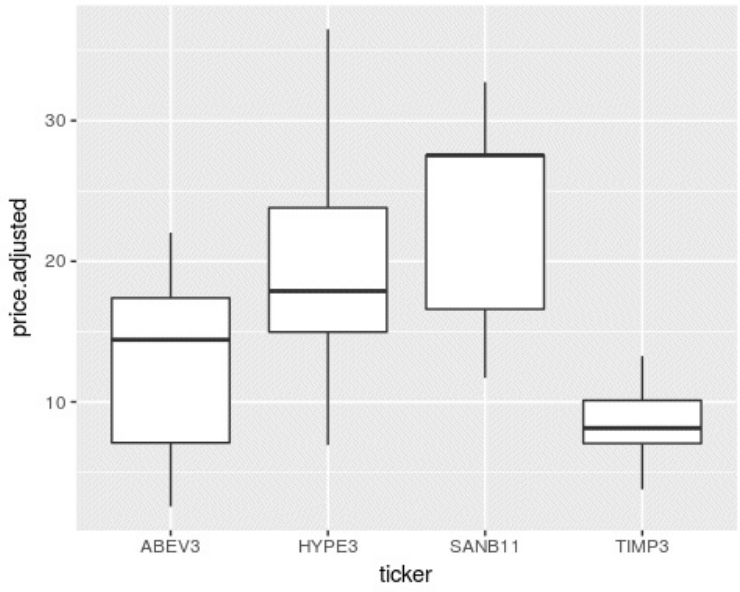
```
# fix seed
set.seed(35)

# select 4 stocks randomly
my.tickers <- sample(unique(my.df$ticker), 4)

p <- my.df %>%
  filter(ticker %in% my.tickers) %>%
  ggplot(aes(x = ticker, y = price.adjusted)) +
  geom_boxplot()

print(p)
```

******ebook converter DEMO Watermarks*******



No exemplo anterior, bastou definir o eixo horizontal como o *ticker* das ações e o eixo vertical como os preços ajustados. O gráfico resultante mostra a mediana e a dispersão dos preços de cada ação em formato geométrico. O gráfico deixa claro que as ações possuem distribuições de preços diferentes nos dados analisados.

10.5.3 Criando Figuras *QQ*

Figuras QQ permitem a comparação visual entre a distribuição de uma variável e uma distribuição teórica, como a Normal. É um gráfico de dispersão entre distribuições cumulativas. Quanto mais próximo de uma linha reta, mais semelhante é a distribuição empírica em relação a distribuição teórica.

Veja um exemplo com dados simulados.

```
# fix seed
set.seed(40)

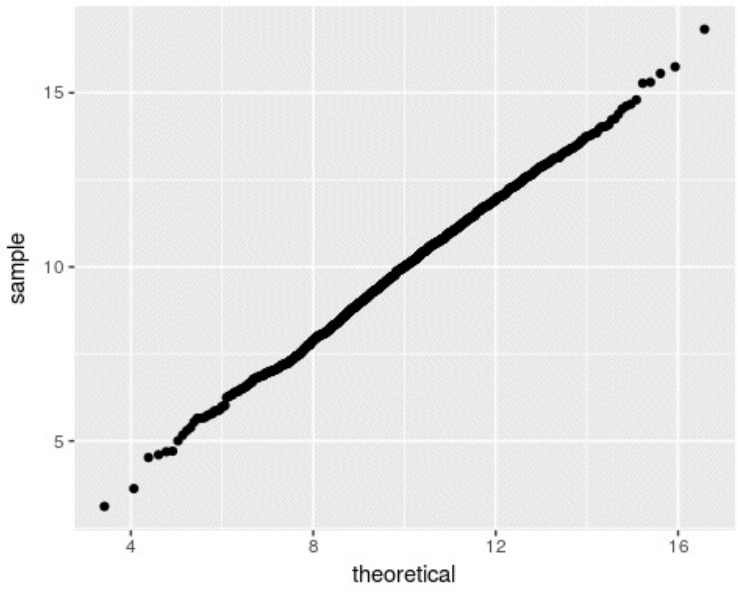
N=1000
my.mean <- 10
my.sd <- 2

# build sim data
temp.df <- data.frame(y=rnorm(n = N, mean = my.mean, sd = my.sd))

# create qq plot
p <- temp.df %>%
  ggplot(aes(sample = y)) +
  geom_qq(distribution = qnorm,
                dparams = c(mean=my.mean, sd=my.sd))

print(p)
```

******ebook converter DEMO Watermarks*******



Simulamos variáveis aleatórias normais com média igual a 10 e desvio padrão igual a 2. Como você pode ver, o gráfico QQ está perto de uma linha reta, indicando que a distribuição empírica dos dados simulados é próxima de uma distribuição Normal. Como usamos dados artificiais da distribuição acima mencionada, esse resultado não surpreende!

Agora, vamos tentar essa análise para o nosso conjunto de dados de retornos das ações. Selecionaremos 4 ações aleatoriamente, criaremos uma nova coluna, chamada `norm.ret`, com os valores normalizados dos retornos. A normalização funciona da seguinte forma; subtraímos a média para cada retorno e dividimos o resultado pelos desvios-padrão. Esse procedimento deve ser executado individualmente para cada ação. Depois disso, comparamos a distribuição resultante com uma Normal de média zero e desvio padrão igual a 1. O código a seguir realiza essa operação.

```
# fix seed
set.seed(45)

# select 4 stock randomly and filter from my.df
my.tickers <- sample(unique(my.df$ticker), 4)
temp.df <- filter(my.df, ticker %in% my.tickers)

# set function for normalization
norm.vec <- function(ret.in){
  # Normalizes a vector by subtracting mean and dividing
  # by the standard deviation
  #
  # Args:
  #   ret.in - numerical vector
  #
  # Returns:
  #   A normalized vector

  mean.ret <- mean(ret.in, na.rm = TRUE)
  sd.ret <- sd(ret.in, na.rm = TRUE)
```

******ebook converter DEMO Watermarks*******

```
  ret.norm <- (ret.in-mean.ret)/sd.ret
  return(ret.norm)
}

ret.in = temp.df$ret
ticker.in = temp.df$ticker

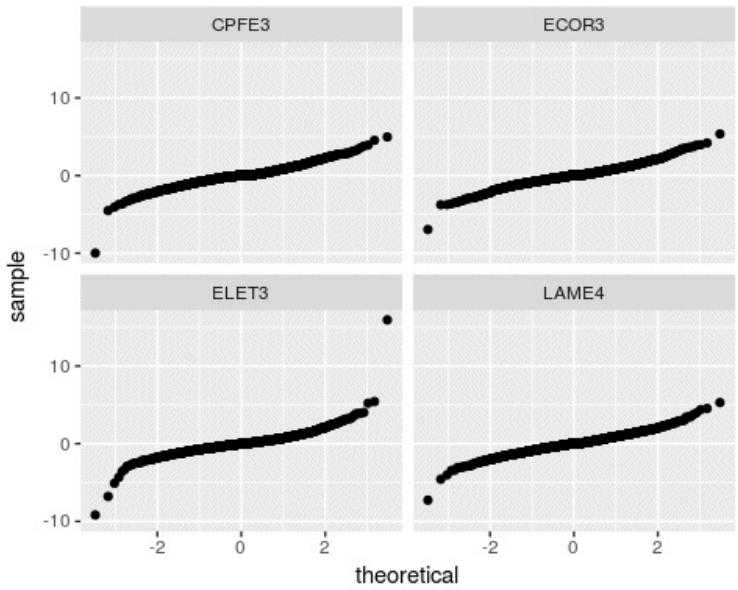
# apply function
my.l <- tapply(X = temp.df$ret,
               INDEX = factor(temp.df$ticker),
               FUN = norm.vec)

# reorder list (tapply sorts alphabetically)
my.l <- my.l[as.character(unique(temp.df$ticker))]

# save new column norm.ret
temp.df$norm.ret <- unlist(my.l)

# plot it!
p <- temp.df %>%
  ggplot(aes(sample = norm.ret)) +
  geom_qq() + facet_wrap(~ticker)

print(p)
```



Como podemos ver, o resultado não é visualmente semelhante ao resultado encontrado para a distribuição simulada. Observa-se, nas caudas das distribuições, uma maior proporção de casos extremos. Tal resultado é bastante conhecido na literatura, sendo geralmente chamado de efeito caudas gordas (*fat tails*). Isso significa que a distribuição Normal é um péssimo previsora da distribuição de retornos de ações, especialmente nas caudas. Essa questão é particularmente importante no cálculo de estimativas de risco, onde o uso da distribuição Normal provavelmente subestimará a probabilidade de uma perda extrema.

******ebook converter DEMO Watermarks*******

******ebook converter DEMO Watermarks*******

10.6 Salvando Figuras para Arquivos

Utilizamos a função `ggsave` para salvar figuras criadas com o comando `ggplot`. Essa toma como entrada o nome do arquivo em que será salvo o gráfico, incluindo a sua extensão (*jpg*, *png*) e o próprio objeto do gráfico. Caso o objeto não seja definido, o último gráfico gerado será salvo no arquivo. Uma sugestão é dar preferência ao formato *png*, que é mais aceito para publicações científicas devido à sua maior qualidade. Veja o exemplo a seguir:

```
p <- temp.df %>%
  ggplot(aes(x = ref.date, y = price.adjusted, color=ticker)) +
  geom_line() +
  labs(x = 'Datas', y = 'Preços')

my.fig.file <- 'fig_ggplot/MyPrices.png'
ggsave(filename = my.fig.file,
       plot = p,
       dpi = 600)
```

Pode-se verificar a criação do arquivo com o comando `list.files`:

```
print(list.files('fig_ggplot'))
```

```
## [1] "MyPrices.png"
```

Como esperado, o arquivo criado está lá, pronto para ser inserido em um relatório técnico ou artigo científico.

******ebook converter DEMO Watermarks*******

10.7 Exercícios

1. Baixe dados da ação CIEL3 com o pacote `BatchGetSymbols` para os últimos 500 dias. Apresente o gŕafico de linha do preço ajustado ao longo do tempo utilizando função `ggplot2::ggplot`.

2. Baixe dados das ações PETR3, VALE5, GGBR4 com `BatchGetSymbols` para os últimos 1500 dias. Apresente, no mesmo gráfico, os preços das ações com diferentes cores de linhas.

3. Para o gráfico anterior, adicione pontos nas linhas.

4. Para o mesmo gráfico, separe os preços das ações em painéis diferentes com a função `ggplot::facet_wrap`.

5. Modifique o tema do gráfico anterior para uma escala em preto e branco, tanto para a área do gráfico quanto para as cores das linhas.

6. Para os dados anteriores, apresente o histograma dos retornos das diferentes ações em painéis diferentes e salve o resultado em um arquivo chamado `'histograms.png'`.

7. **DESAFIO** - Utilizando pacote `gganimate`, crie uma animação para os preços das três ações do exercício 2. Em um gráfico de ponto (`geom_point`), podes utilizar o eixo `x` para os *tickers* e os preços no eixo `y`. A animação deve iterar sobre os diferentes dias da base de dados.

******ebook converter DEMO Watermarks*******

11 Modelos Econométricos no R

As ferramentas da Econometria permitem ao pesquisador estimar e simular processos estocásticos, fazer previsões e testar hipóteses específicas sobre os dados analisados. O maior benefício é a escala. Um modelo pode ser facilmente aplicado a volumosas bases de dados, com inferências automáticas de análise. O número de possíveis aplicações empíricas é enorme. Resumidamente, estimamos um modelo para aprender algo sobre os dados. Esse modelo fornece informações quantitativas que ajudarão uma possível tomada de decisão.

A variedade e quantidade de modelos utilizados em Econometria é enorme. Seria impossível cobrir todos possíveis modelos e suas particularidades. No entanto, alguns tipos são usados com maior frequência que outros. Neste capítulo lidaremos com os seguintes tipos de modelos e suas aplicações com o R:

- Modelos lineares (OLS)
- Modelos lineares generalizados (GLS)
- Modelos de dados do painel
- Modelos ARIMA (médias móveis autoregressivas integradas)
- Modelos GARCH (heterocedasticidade condicional autorregressiva)
- Modelos de mudança de regimes markovianos

Não apresentaremos a completa descrição da teoria subjacente à representação, estimação e possíveis testes relacionados a cada tipo de modelo. O foco deste capítulo é apresentar os detalhes computacionais, incluindo pacotes necessários, para trabalhar com esses modelos no R. Enquanto daremos um contexto quantitativo, fornecendo exemplos com dados simulados e reais, estes não serão profundos. Este capítulo deve ser estudado junto a literatura principal de Econometria (Gujarati and Porter 2011, Brooks (2014), J. D. Hamilton (1994), Greene (2003)). Como sugestão, um material mais completo sobre o uso de R para Econometria aplicada é encontrado em Kleiber and Zeileis (2008) e Farnsworth (2008).

******ebook converter DEMO Watermarks*******

11.1 Modelos Lineares (OLS)

Um modelo linear é, sem dúvida, o tipo de especificação mais utilizada em Economia e Finanças. Sempre que você precisa estimar uma relação linear a partir dos dados, você provavelmente usará um modelo linear do tipo OLS (*ordinary least squares* - mínimos quadrados ordinários). Sua principal vantagem é simplicidade e rápida estimação. Os coeficientes são encontrados a partir de uma fórmula fechada. Esse processo, mesmo para uma grande quantidades de dados, é bastante rápido e fácil de implementar.

Particularmente em Finanças, o uso mais direto e popular de modelos lineares encontra-se na estimativa dos fatores de risco em investimentos. Em uma versão mais simples, chamada modelo de mercado, o beta é uma medida de risco sistemático, onde os retornos das ações são a variável dependente e os retornos de um índice de mercado, como Ibovespa, representam a variável explicativa. O beta mede quão forte é a relação de uma ação com o mercado. Caso a ação tiver um alto valor de beta, ela segue fortemente o mercado e tem um alto risco sistemático.

Um modelo linear com *N* variáveis explicativas pode ser representado da seguinte forma:

$$y_t = \alpha + \beta_1 x_{1,t} + \beta_2 x_{2,t} + ... + \beta_N x_{N,t} + \epsilon_t$$

O lado esquerdo da equação, (*y*), é a variável dependente (ou explicativa). Esse é o vetor de informação que estamos tentando explicar e criar previsões. Variáveis *y* e *x* são os vetores com dados utilizados como entradas na função de estimação do R. Internamente, a função encontra os valores de *alpha* e *beta* que minimizam a soma de erros quadráticos. Em outras palavras, encontramos os parâmetros que dão a máxima precisão histórica para o modelo.

11.1.1 Simulando um Modelo Linear

Considere a seguinte equação:

$$y_t = 0.5 + 2x_t + \epsilon_t$$

Podemos usar o R para simular *1000* observações de *y*. Primeiro, definimos *x* e o erro do modelo, *epsilon*, como variáveis aleatórias da distribuição Normal com média zero e variância igual a um. A simulação completa de *y* é executada com o seguinte código:

```
set.seed(50)

# number of obs
nT <- 1000

# set x as Normal (0, 1)
x <- rnorm(nT)

# set coefficients
my.alpha <- 0.5
my.beta <- 2
```

******ebook converter DEMO Watermarks*******

```
# build y
y <- my.alpha + my.beta*x + rnorm(nT)
```

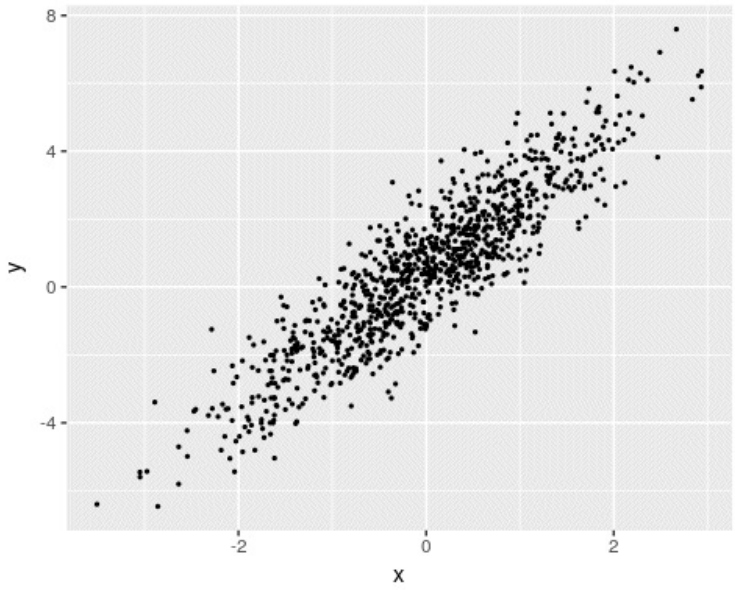
Usando `ggplot2:ggplot`, podemos criar um gráfico de dispersão para visualizar a correlação entre objetos `x` e `y`.

```
library(ggplot2)

# set temp df
temp.df <- data.frame(x, y)

# plot it
p <- ggplot(temp.df, aes(x = x, y = y)) +
  geom_point(size=0.5)

print(p)
```



Claramente há uma correlação linear positiva; uma linha direta ascendente seria uma boa aproximação para a relação entre essas variáveis. Podemos verificar esse resultado com o cálculo do coeficiente de correlação com comando `cor(temp.df$x, temp.df$y)`. Aqui, a correlação entre `y` e `x` é bastante alta, 0.901, sendo que o máximo possível de correlação é 1.

11.1.2 Estimando um Modelo Linear

A principal função para estimar um modelo linear no R é `lm`. Vamos usá-la para estimar um modelo a partir dos dados simulados anteriormente.

```
# set df
lm.df <- data.frame(x, y)

# estimate linear model
my.lm <- lm(data = lm.df, formula = y ~ x)
print(my.lm)
```

******ebook converter DEMO Watermarks*******

```
##
## Call:
## lm(formula = y ~ x, data = lm.df)
##
## Coefficients:
## (Intercept)            x
##      0.5083       1.9891
```

O argumento `formula` define o formato do modelo linear. Se tivéssemos outra coluna chamada `x2` e fôssemos incluí-la no modelo, poderíamos escrever `formula = y ~ x1 + x2`. Observe que o intercepto (alpha) está, por padrão, incluído na estimativa. Para omiti-lo, escrevemos `formula = y ~ 0 + x1` ou `formula = y ~ -1 + x1`.

Argumento `formula` permite outras opções personalizadas, incluindo combinações entre as variáveis explicativas. Vamos criar outro conjunto de dados artificiais e analisar algumas dessas opções:

```
set.seed(15)

# set simulated dataset
N <- 100
df <- data.frame(x = runif(N),
                 y = runif(N),
                 z = runif(N),
                 group = sample(LETTERS[1:3],
                                N,
                                replace = TRUE ))

# Vanilla formula
#
# example: y ~ x + z
# model: y(t) = alpha + beta(1)*x(t) + beta(2)*z(t) + error(t)
my.formula <- y ~ x + z
print(lm(data = df,
         formula = my.formula))

##
## Call:
## lm(formula = my.formula, data = df)
##
## Coefficients:
## (Intercept)            x            z
##     0.44971      0.14223     -0.03781

# Vannila formula with dummies
#
# example: y ~ group + x + z
# model: y(t) = alpha + beta(1)*D_1(t)+beta(2)*D_2(t) +
#               beta(3)*x(t) + beta(4)*z(t) + error(t)
# D_i(t) - dummy for group i
my.formula <- y ~ group + x + z
print(lm(data = df,
         formula = my.formula))

##
## Call:
## lm(formula = my.formula, data = df)
##
```

******ebook converter DEMO Watermarks*******

```
## Coefficients:
## (Intercept)       groupB       groupC            x
##     0.48309     -0.06397      0.01144      0.13511
##           z
##    -0.05156

# Without intercept
#
# example: y ~ -1 + x + z
# model: y(t) = beta(1)*x(t) + beta(2)*z(t) + error(t)
my.formula <- y ~ -1 + x + z
print(lm(data = df,
         formula = my.formula))

##
## Call:
## lm(formula = my.formula, data = df)
##
## Coefficients:
##      x       z
## 0.5183  0.3133

# Using combinations of variables
# example: y ~ x*z
# model: y(t) = alpha + beta(1)*x(t) + beta(2)*z(t) +
#               beta(3)*x(t)*z(t) + error(t)
my.formula <- y ~ x*z
print(lm(data = df,
         formula = my.formula))

##
## Call:
## lm(formula = my.formula, data = df)
##
## Coefficients:
## (Intercept)            x            z          x:z
##     0.39827      0.22970      0.05129     -0.15464

# Interacting variables
# example: y ~ x:group + z
# model: y(t) = alpha + beta(1)*z(t) + beta(2)*x(t)*D_1(t) +
#               beta(3)*x(t)*D_2(t) + beta(4)*x(t)*D_3(t) +
#               error(t)
# D_i(t) - dummy for group i
my.formula <- y ~ x:group + z
print(lm(data = df,
         formula = my.formula))

##
## Call:
## lm(formula = my.formula, data = df)
##
## Coefficients:
## (Intercept)            z     x:groupA     x:groupB
##     0.45995     -0.05105      0.16573      0.06271
##    x:groupC
##     0.20025
```

As diferentes opções na entrada `formula` permitem uma gama diversificada de modelos lineares. O uso de operações matemáticas, como `log(x)`, também é possível. Mais detalhes sobre os usos

******ebook converter DEMO Watermarks*******

avançados da entrada `formula` estão disponíveis no manual do R.

A saída da função `lm` é um objeto semelhante a uma lista. Portanto, seus elementos podem ser acessados com o operador `$`. Vamos imprimir todos os nomes disponíveis:

```
# print names in model
print(names(my.lm))
```

```
##  [1] "coefficients"  "residuals"     "effects"
##  [4] "rank"          "fitted.values" "assign"
##  [7] "qr"            "df.residual"   "xlevels"
## [10] "call"          "terms"         "model"
```

Como você pode ver, existe um elemento chamado `coefficients`:

```
print(my.lm$coefficients)
```

```
## (Intercept)           x
##   0.5083045   1.9890616
```

Todos os coeficientes estimados são armazenados em `my.lm$coeficientes`. Seu conteúdo é um vetor atômico que aumenta de comprimento de acordo com o número de variáveis explicativas no modelo.

No nosso exemplo de usar `lm` com dados simulados, os coeficientes estimados estão próximos aos valores reais. Lembre-se que, no código anterior, definimos esses valores como `my.alpha = 0.5` e `my.beta = 2`.

Pesquisadores experientes provavelmente devem ter observado que, do ponto de vista econométrico, função `print` exibe informações superficiais sobre a estimação de um modelo linear. Além dos valores dos coeficientes, muitos outros aspectos de um modelo linear devem ser analisados. No R, usamos função `summary` para obter mais informações sobre o modelo:

```
print(summary(my.lm))
```

```
##
## Call:
## lm(formula = y ~ x, data = lm.df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.0444 -0.6906 -0.0244  0.6807  3.2892
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.50830    0.03107   16.36   <2e-16 ***
## x            1.98906    0.03031   65.61   <2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.9824 on 998 degrees of freedom
## Multiple R-squared:  0.8118, Adjusted R-squared:  0.8116
## F-statistic:  4305 on 1 and 998 DF,  p-value: < 2.2e-16
```

******ebook converter DEMO Watermarks*******

Os coeficientes estimados têm elevados valores da estatística *T*, e o modelo possui um excelente ajuste ao dados, com um *R2* no valor de 0.8116. Esse resultado positivo não surpreende. Os dados foram simulados em um processo linear, ou seja, a correlação entre os vetores foi introduzida artificialmente.

Informações adicionais estão disponíveis no objeto resultante de `summary`. Vejamos os nomes da saída:

```
my.summary <- summary(my.lm)
print(names(my.summary))
```

```
##  [1] "call"          "terms"         "residuals"
##  [4] "coefficients"  "aliased"       "sigma"
##  [7] "df"            "r.squared"     "adj.r.squared"
## [10] "fstatistic"    "cov.unscaled"
```

Cada um desses elementos contém informações que podem ser relatadas em uma tabela com os resultados da estimação. Podemos exportar os valores de coeficientes, estatísticas *T* e outros para uma ferramenta de planilha e criar uma tabela personalizada para reportar os resultados. Isso, no entanto, não é recomendado ou necessário. No capítulo 12 discutiremos as melhores maneiras de reportar modelos econométricos usando pacotes especializados.

Agora, vamos passar para um exemplo com dados reais. Para isso, estimaremos o coeficiente beta de uma ação selecionada aleatoriamente. O modelo de mercado é dado pela seguinte equação:

$$R_t = \alpha + \beta R_{M,t} + \epsilon_t$$

Primeiro, vamos carregar e ajustar os dados.

```
# load stock data
library(dplyr)
library(readr)

my.f <- 'data/IbovStocks_long.csv'
my.col.types <- cols(price.adjusted = col_double(),
                     ref.date = col_date(format = ""),
                     ticker = col_character())

my.df <- read_csv(my.f, col_types = my.col.types)

# select rnd asset and filter data
set.seed(10)
my.asset <- sample(my.df$ticker,1)
my.df.asset <- my.df[my.df$ticker == my.asset, ]

# load Ibov data
df.ibov <- read_csv(file = 'data/Ibov.csv',
                    col_types = cols(ref.date = col_date(format = ""),
                                     price.close = col_integer()))

# calculate return for
df.ibov$ret <- calc.ret(df.ibov$price.close)
my.df.asset$ret <- calc.ret(my.df.asset$price.adjusted,
```

******ebook converter DEMO Watermarks*******

```
                                  my.df.asset$ticker)

# print number of rows in datasets
print(nrow(my.df.asset))

## [1] 1999

print(nrow(df.ibov))

## [1] 2102
```

Destaca-se que o número de linhas do conjunto de dados para ação GGBR4 não coincide com o número de linhas do índice Ibovespa. As datas dos diferentes `dataframes` não estão sincronizadas. Então, o primeiro passo é adicionar uma coluna em `my.df.asset` com os retornos do índice de mercado. Para isso, usamos a função `match` para encontrar os índices que sincronizam as datas.

```
# find location of dates
idx <- match(my.df.asset$ref.date, df.ibov$ref.date)

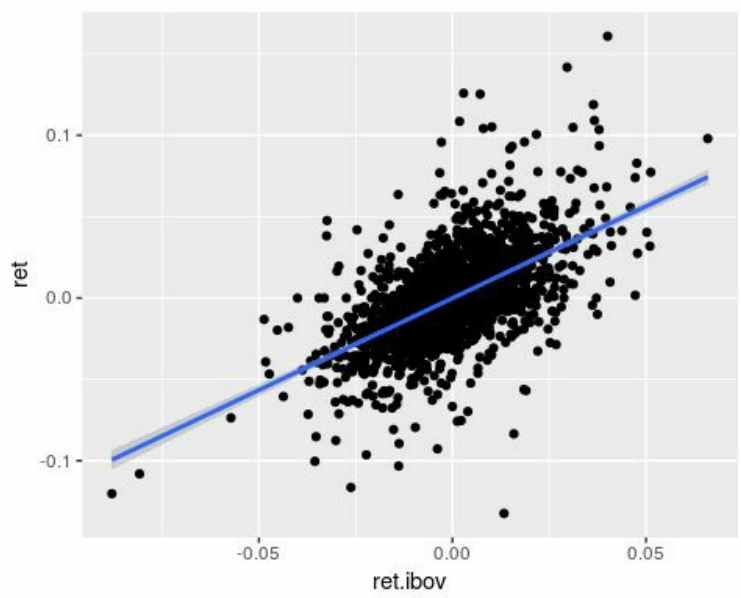
# create column in my.df
my.df.asset$ret.ibov <- df.ibov$ret[idx]
```

Como um começo, vamos criar o gráfico de dispersão com os retornos da ação e o índice de mercado, adicionando também uma tendência linear com função `geom_smooth`.

```
library(ggplot2)

p <- my.df.asset %>%
  na.omit(.) %>%
  ggplot(aes(x=ret.ibov, y= ret)) +
  geom_point() +
  geom_smooth(method = 'lm')

print(p)
```



******ebook converter DEMO Watermarks*******

A figura mostra uma clara tendência linear. O retorno do índice de mercado é um bom preditor do retorno da ação. Considerando esse resultado visual, partimos para a estimação do modelo.

```
# estimate beta model
my.beta.model <- lm(data = my.df.asset,
                    formula = ret ~ ret.ibov)

# print it
print(summary(my.beta.model))
```

```
##
## Call:
## lm(formula = ret ~ ret.ibov, data = my.df.asset)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.147328 -0.012032 -0.001173  0.010888  0.122687
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -9.777e-05  4.995e-04  -0.196    0.845
## ret.ibov     1.129e+00  3.483e-02  32.421   <2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.02225 on 1982 degrees of freedom
##   (15 observations deleted due to missingness)
## Multiple R-squared:  0.3465, Adjusted R-squared:  0.3462
## F-statistic:  1051 on 1 and 1982 DF,  p-value: < 2.2e-16
```

O resultado anterior mostra que a ação GGBR4 possui um beta igual a 1.13. Isso significa que esta é uma ação agressiva, com maior sensibilidade aos movimentos do índice de mercado.

11.1.3 Inferência Estatística em Modelos Lineares

Depois de estimar um modelo com a função `lm`, o próximo passo é testar hipóteses a respeito dos coeficientes. O teste F verifica a condição mais básica de um modelo para justificar sua existência - a suposição de que todos os coeficientes, excluindo a intercepto, são iguais a zero. Quando função `summary` é aplicada a uma saída `lm`, O teste F é fornecido na última linha da saída de texto. A hipótese nula do teste é que todos coeficientes atrelados a outras variáveis são iguais a zero. Veja a seguir:

```
nT <- 100
df <- data.frame(y = runif(nT),
                 x.1 = runif(nT),
                 x.2 = runif(nT))

my.lm <- lm(data = df,
            formula = y ~ x.1 + x.2)

print(summary(my.lm))
```

```
##
## Call:
## lm(formula = y ~ x.1 + x.2, data = df)
```

******ebook converter DEMO Watermarks*******

```
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.43125 -0.20631 -0.01184  0.18659  0.51744
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.449997   0.075048   5.996 3.46e-08 ***
## x.1         -0.004976   0.092259  -0.054    0.957
## x.2         -0.006775   0.088308  -0.077    0.939
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2621 on 97 degrees of freedom
## Multiple R-squared:  8.477e-05,  Adjusted R-squared:  -0.02053
## F-statistic: 0.004112 on 2 and 97 DF,  p-value: 0.9959
```

Neste exemplo, a estatística F é igual a 0.0041119. O valor *p* associado ao teste é superior a 10%, indicando uma forte evidência estatística para a hipótese nula. Não conseguimos rejeitar a hipótese de que os parâmetros relativos a `x.1` e `x.2` sejam iguais a zero. A associação entre a variável explicada e esses vetores é quase nula. Para um exemplo com a rejeição da hipótese nula do teste F, veja a estimativa do modelo com dados artificiais na seção 11.1.2.

Outro tipo de teste executado automaticamente pela função `lm` e `summary` é o teste *T*. Enquanto as estatísticas F testam a hipótese conjunta de que todos os coeficientes são zero, a estatística *T* realiza o teste para parâmetros individuais. Essa verifica a hipótese de que um coeficiente específico é igual a zero. Não é acidental que testes *T* sejam muito comuns em pesquisa. Ele permitem testar a evidência estatística para um efeito particular nos dados.

Na prática da pesquisa, é provável que ambos os testes, T e F, sejam suficientes para a análise. No entanto, você também pode testar hipóteses personalizadas, como a soma ou o produto de parâmetros serem iguais a um valor específico. As hipóteses testadas geralmente surgem de análises preliminares e possuem algum tipo de justificativa teórica.

Como um exemplo simples, vamos testar uma hipótese linear para um modelo estimado em dados simulados. Aqui, criaremos dados artificiais e testaremos a hipótese formal de que os coeficientes estimados são iguais aos valores reais fornecidos na simulação.

```
set.seed(10)

# number of time periods
nT <- 1000

# set parameters
my.intercept <- 0.5
my.beta <- 1.5

# simulate
x <- rnorm(nT)
y <- my.intercept + my.beta*x + rnorm(nT)

# set df
df <- data.frame(y, x)
```

******ebook converter DEMO Watermarks*******

```
# estimate model
my.lm <- lm(data = df,
            formula = y ~ x )
```

Após a estimação do modelo, usamos função `LinearHypothesis` do pacote `car` (Fox and Weisberg 2011) para implementar nosso teste formal. Antes de usá-lo, precisamos entender suas entradas. A primeira entrada, `model`, é o objeto resultante da etapa anterior. As entradas `hypothesis.matrix` e `rhs` determinam a hipótese linear do teste em um formato de matriz. Objeto `hypothesis.matrix` será multiplicado, em notação matricial, por um vetor vertical dos coeficientes do modelo. A entrada `rhs` (*right hand side*) determina o resultado hipotético desse cálculo. No nosso caso, a operação resultante é:

$$\underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}}_{hypothesis.matrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \underbrace{\begin{bmatrix} 0.5 \\ 1.5 \end{bmatrix}}_{rhs}$$

Com essa operação de matriz, testamos a hipótese conjunta de que a intercepto é igual a 0.5 e a inclinação é equivalente a 1.5. Observe que o uso de matrizes dá flexibilidade ao usuário. Poderíamos testar muitas outras hipóteses lineares através da alteração de `hypothesis.matrix` e `rhs`. Veja o código a seguir:

```
library(car)

# set test matrix
test.matrix <- matrix(c(my.intercept,  # alpha test value
                        my.beta))  # beta test value

# hypothesis matrix
hyp.mat <- matrix(c(1,0,
                    0,1),nrow = 2)

# do test
my.waldtest <- linearHypothesis(my.lm,
                                hypothesis.matrix = hyp.mat,
                                rhs = test.matrix)

# print result
print(my.waldtest)
```

```
## Linear hypothesis test
##
## Hypothesis:
## (Intercept) = 0.5
## x = 1.5
##
## Model 1: restricted model
## Model 2: y ~ x
##
##   Res.Df    RSS Df Sum of Sq      F Pr(>F)
## 1   1000 1089.1
## 2    998 1086.8  2    2.3766 1.0912 0.3362
```

Como podemos ver, o teste falha na rejeição da hipótese nula. Isso significa que nossa simulação funcionou, e os parâmetros foram corretamente identificados na estimação, como esperado.

******ebook converter DEMO Watermarks*******

Outra família de testes comumente aplicada a modelos lineares está relacionada às suas premissas. Todo modelo linear do tipo OLS assume várias condições para seu resíduo (erro), incluindo: 1) independência, 2) homoscesdasticidade (variância constante) e 3) adesão à distribuição normal. Se essas premissas não forem verdadeiras, o modelo pode ser ineficiente ou tendencioso (viciado), o que significa que é necessária alguma modificação ou uso de estimativas robustas. Mais detalhes sobre por que essa suposição deve ser verdadeira e possíveis soluções são encontradas em qualquer livro texto sobre Econometria, tal como Greene (2003) e Maddala (2001).

No R, podemos usar o pacote `lmtest` (Zeileis and Hothorn 2002) para testar a independência dos resíduos com o teste *Brecht-Godfrey* e *Durbin Watson*. O teste de *Shapiro-Wilk* para a normalidade está disponível no pacote `stats`. A seguir apresenta-se um exemplo de uso dessas funções para o modelo previamente estimado.

```
library(lmtest)

# Breush Pagan test 1 - Serial correlation
# Null Hypothesis: No serial correlation in residual
print(bgtest(my.lm, order = 5))

# Breush Pagan test 2 - Homocesdasticity of residuals
# Null Hypothesis: homocesdasticity
#                  (constant variance of residuals)
print(ncvTest(my.lm))

# Durbin Watson test - Serial correlation
# Null Hypothesis: No serial correlation in residual
print(dwtest(my.lm))

# Shapiro test  - Normality
# Null Hypothesis: Data is normally distributed
print(shapiro.test(my.lm$residuals))
```

```
##
##  Breusch-Godfrey test for serial correlation of order
##  up to 5
##
## data:  my.lm
## LM test = 4.2628, df = 5, p-value = 0.5122
##
## Non-constant Variance Score Test
## Variance formula: ~ fitted.values
## Chisquare = 1.54328    Df = 1     p = 0.2141302
##
##  Durbin-Watson test
##
## data:  my.lm
## DW = 2.092, p-value = 0.9271
## alternative hypothesis: true autocorrelation is greater than 0
##
##
##  Shapiro-Wilk normality test
##
## data:  my.lm$residuals
## W = 0.99803, p-value = 0.2964
```

******ebook converter DEMO Watermarks*******

Como esperado, o modelo com dados artificiais passou em todos os testes.

Outra abordagem interessante para validar modelos lineares é oferecida pelo pacote `gvlma` (Pena and Slate 2014). Esse fornece uma função que executa uma validação global do modelo, isto é, realiza diversos tipos de testes, incluindo aqueles descritos anteriormente. A principal vantagem é que a função produz todos os resultados em uma única chamada. Dê uma olhada:

```
library(gvlma)

# global validation of model
gvmodel <- gvlma(my.lm)

# print result
summary(gvmodel)
```

```
##
## Call:
## lm(formula = y ~ x, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.2703 -0.6898  0.0063  0.7346  3.8266
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.51510    0.03300   15.61   <2e-16 ***
## x            1.54658    0.03329   46.46   <2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.044 on 998 degrees of freedom
## Multiple R-squared:  0.6838, Adjusted R-squared:  0.6835
## F-statistic:  2159 on 1 and 998 DF,  p-value: < 2.2e-16
##
##
## ASSESSMENT OF THE LINEAR MODEL ASSUMPTIONS
## USING THE GLOBAL TEST ON 4 DEGREES-OF-FREEDOM:
## Level of Significance =  0.05
##
## Call:
##  gvlma(x = my.lm)
##
##                      Value p-value                Decision
## Global Stat        3.6404  0.4569 Assumptions acceptable.
## Skewness           1.7814  0.1820 Assumptions acceptable.
## Kurtosis           0.1738  0.6767 Assumptions acceptable.
## Link Function      0.6628  0.4156 Assumptions acceptable.
## Heteroscedasticity 1.0224  0.3119 Assumptions acceptable.
```

A saída de `gvlma` mostra vários testes realizados no modelo. O resultado também é positivo, uma vez que a decisão do modelo é que os pressupostos da OLS são aceitáveis. Caso o modelo não fosse aprovado, uma solução é usar estimativas robustas de erros padrão. O pacote `sandwich` (Zeileis 2004) oferece função `NeweyWest` para esse fim.

******ebook converter DEMO Watermarks*******

******ebook converter DEMO Watermarks*******

11.2 Modelos Lineares Generalizados (GLM)

O modelo linear generalizado (GLM) é uma alternativa flexível aos modelos lineares. Esse permite ao usuário alterar a distribuição do erro e a função de ligação (*link*) dos dados, uma sistemática maneira de quantificar como a variável explicada será afetada pela variável explicativa. Os modelos GLM são os mais adequados quando os pressupostos da OLS, como a normalidade dos resíduos, não são válidos. Por exemplo, quando você tem uma variável binária que toma apenas dois valores, o resíduo de um modelo linear não é normalmente distribuído, dados os limites da variável explicada. Nesse caso, portanto, o uso de modelos GLM é recomendado.

Podemos escrever uma especificação GLM univariada como:

$$E\left(y_t\right) = g\left(\alpha + \sum_{i=1}^{N} \beta_i x_{i,t}\right)$$

A principal diferença de um modelo GLM a um modelo OLS é o uso de uma função de ligação *g()* e uma suposição de distribuição personalizada para o termo de erro. Função *g()* pode ter várias formas. Por exemplo, se estamos modelando uma variável binária (0 e 1), podemos usar *g()* como a função *logit*:

$$g(x) = \frac{\exp(x)}{1 + \exp(x)}$$

Observe que, nesse caso, função *g()* garante que qualquer valor de *x* resultará em um número entre 0 e 1. Em consequência, a resposta da variável explicada à variável explicativa será não linear.

11.2.1 Simulando um Modelo GLM

Como exemplo, vamos simular o seguinte modelo GLM onde o vetor de resposta é uma variável de *Bernoulli* que toma valor 1 com probabilidade *p*. As probabilidades são calculadas a partir da transformação não-linear de *xt*:

******ebook converter DEMO Watermarks*******

$$p_t = \frac{\exp(2 + 5x_t)}{1 + \exp(2 + 5x_t)}$$

Voltando ao R, usamos o seguinte código para construir o vetor de resposta:

```
set.seed(15)

# set number of obs
nT <- 500

# set x
x = rnorm(nT)

my.alpha <- 2
my.beta <- 5

# set probabilities
z = my.alpha + my.beta*x
p = exp(z)/(1+exp(z))

# set response variable
y = rbinom(n = nT,size = 1, prob = p)
```

Função `rbinom` cria um vetor de 1s e 0s, com base nas probabilidades de entrada `prob`. Uma chamada a `summary` já mostra claramente a distribuição da variável:

```
# check distribution of y
summary(y)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00    0.00    1.00    0.67    1.00    1.00
```

Objeto `y` contém apenas 0s e 1s, como esperado.

11.2.2 Estimando um Modelo GLM

No R, a estimativa de modelos GLM é realizada com função `glm`. Essa funciona de forma semelhante a `lm`, mas contém vários argumentos extras que controlam os detalhes dos modelos, como a função de ligação (*link*) e a distribuição dos resíduos.

Primeiro, vamos usar os dados previamente simulados para estimar um modelo *logit*:

```
# create df
df <- data_frame(y = y, x = x)

# estimate GLM
my.glm <- glm(data=df,
              formula = y~x ,
              family= binomial(link = "logit"))

# print it with summary
```

******ebook converter DEMO Watermarks*******

```
print(summary(my.glm))
```

```
##
## Call:
## glm(formula = y ~ x, family = binomial(link = "logit"), data = df)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -2.99392  -0.13689   0.04087   0.23250   2.91383
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)   2.1488     0.2622   8.197 2.47e-16 ***
## x             4.9050     0.5110   9.598  < 2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 634.18  on 499  degrees of freedom
## Residual deviance: 214.14  on 498  degrees of freedom
## AIC: 218.14
##
## Number of Fisher Scoring iterations: 7
```

Os coeficientes estimados são próximos do que definimos em `my.alpha` e `my.beta`. Como esperado, o modelo tem um bom ajuste dos dados, sendo ambos os parâmetros estatisticamente significativos a 1%.

A função `glm` oferece muitas opções para configurar um modelo personalizado. A partir dos arquivos de ajuda, temos as seguintes alternativas para a função de distribuição e função *link*:

Family	**Formula**
binomial	link = "logit"
gaussian	link = "identity"
Gamma	link = "inverse"
inverse.gaussian	link = "1/mu^2"
poisson	link = "log"
quasi	link = "identity", variance = "constant"
quasibinomial	link = "logit"
quasipoisson	link = "log"

O primeiro passo no uso de um modelo GLM é identificar a função de distribuição e de ligamento que melhor adequa-se aos seus dados. Depois disso, você pode usar a tabela anterior para configurar as entradas da função `glm`.

Como exemplo com dados reais dos mercados financeiros, vamos selecionar aleatoriamente uma ação e estimar um modelo para a probabilidade de um retorno positivo com o modelo probit (`link = 'probit'`). Usaremos os retornos do Ibovespa como a variável independente, lado direito da equação. Explicando melhor, queremos testar se o índice de mercado influencia a

******ebook converter DEMO Watermarks*******

chance de uma ação ter um retorno positivo. Essa é uma versão alternativa ao modelo de mercado destacado na seção 11.1.2.

```
set.seed(15)

# select stock
my.stock <- sample(unique(my.df$ticker), 1)
#my.df.asset <- my.df[my.df$ticker == my.stock, ]

# find location of dates
idx <- match(my.df.asset$ref.date, df.ibov$ref.date)

# create column in my.df
my.df.asset$ret.ibov <- df.ibov$ret[idx]

# set column with dummy variable
my.df.asset$D_ret <- my.df.asset$ret > 0

# estimate model
my.glm <- glm(data=my.df.asset,
              formula = D_ret~ret.ibov ,
              family= binomial(link = "probit"))

print(summary(my.glm))
```

```
##
## Call:
## glm(formula = D_ret ~ ret.ibov, family = binomial(link = "probit"),
##     data = my.df.asset)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.5870  -0.9257  -0.4498   1.0161   2.6820
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.18727    0.03086  -6.069 1.29e-09 ***
## ret.ibov    53.28952    2.73847  19.460  < 2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2724.6  on 1983  degrees of freedom
## Residual deviance: 2227.2  on 1982  degrees of freedom
##   (15 observations deleted due to missingness)
## AIC: 2231.2
##
## Number of Fisher Scoring iterations: 4
```

Como esperado, o parâmetro para o índice de mercado é positivo e significativo. Esse resultado implica que a probabilidade da ação ITUB4 ter um retorno positivo é afetada pelas mudanças no índice Ibovespa. Quando o índice aumenta seu valor, um retorno positivo para a ação é mais provável.

******ebook converter DEMO Watermarks*******

******ebook converter DEMO Watermarks*******

11.3 Modelos para Dados em Painel

Os modelos de dados em painel são aconselhados quando as informações a serem analisadas são multidimensionais, tal como dados sobre diferentes indivíduos ou empresas ao longo do tempo. Um conjunto de informações financeiras sobre várias empresas é um caso clássico de dados de painel. Temos uma coluna que identifica a empresa, outra coluna para o tempo, e uma ou mais colunas que identificam os indicadores financeiros. Em uma seção transversal de tempo, temos várias empresas e vários índices financeiros. O conjunto de dados pode ser categorizado como balanceado, onde todas as empresas têm informações em todas as datas ou não balanceado, onde nem todas as empresas possuem dados para todas as datas.

A principal motivação para usar essa classe de modelos é permitir efeitos comuns dentro dos grupos de dados. A estimação de um modelo OLS diferente para cada grupo assume, implicitamente, que os modelos são independentes. Se a suposição de independência não é verdadeira, nossa análise econométrica é comprometida por um possível viés. O uso de modelos de dados em painel permite representações mais flexíveis. Alguns parâmetros podem ser individuais para cada grupo, enquanto outros são compartilhados. Usar modelos de dados de painel requer um procedimento cuidadoso sobre como o modelo é identificado. Muitos testes estatísticos estão disponíveis para esse fim.

Podemos representar o caso mais simples de um modelo de dados de painel como:

$$y_{i,t} = \alpha_i + \beta x_{i,t} + \epsilon_{i,t}$$

Observe que agora usamos índice *i* nas variáveis dependentes e independentes. Esse índice controla os grupos, como empresas diferentes. Em nosso modelo específico, todos os casos têm interceptos diferentes, mas compartilham a mesma versão beta. Dependendo dos pressupostos sobre o intercepto, a equação anterior pode representar um modelo de dados de painel de tipo fixo (*fixed*) ou efeitos aleatórios (*random effects*). Existem outras maneiras de personalizar um modelo de dados de painel e definir efeitos dinâmicos, como termos autoregressivos. Você pode encontrar mais detalhes em Hsiao (2014).

11.3.1 Simulando Dados em Painel

Vamos simular um painel balanceado com efeitos fixos para doze empresas diferentes e cinco períodos de tempo. Esse é um caso clássico de dados de painel, com valor alto de *N* (número de casos) e baixo de *T* (número de datas). Cada empresa terá uma variável explicativa, chamada `x`, que varia em diferentes datas. O código a seguir abusa de operações matriciais para simular todos os casos. Observe os muitos usos da função `sapply`. Depois de criar os dados multivariados, os empilhamos em vetores individuais e salvamos o resultado em um dataframe.

```
library(dplyr)

set.seed(25)
```

******ebook converter DEMO Watermarks*******

```
# number of obs for each case
nT <- 5

# set number of groups
N <- 12

# set possible cases
possible.cases <- LETTERS[1:N]

# set parameters
my.alphas <- seq(-10,10,length.out = N)
my.beta <- 1.5

# set indep var (x) and dates
indep.var <- sapply(rep(nT,N), rnorm)
my.dates <- Sys.Date() + 1:nT

# create response matrix (y)
response.matrix <- matrix(rep(my.alphas,nT),
                          nrow = nT,
                          byrow = TRUE) +
  indep.var*my.beta + sapply(rep(nT,N),rnorm, sd = 0.25)

# set df
sim.df <- data_frame(G = as.character(sapply(possible.cases,
                                             rep,
                                             times=nT )),
                     dates = rep(my.dates, times=N),
                     y = as.numeric(response.matrix),
                     x = as.numeric(indep.var))

# check result
glimpse(sim.df)

## Observations: 60
## Variables: 4
## $ G     <chr> "A", "A", "A", "A", "A", "B", "B", "B", "...
## $ dates <date> 2018-06-11, 2018-06-12, 2018-06-13, 2018...
## $ y     <dbl> -10.676541, -11.550739, -11.742509, -9.83...
## $ x     <dbl> -0.21183360, -1.04159113, -1.15330756, 0....
```

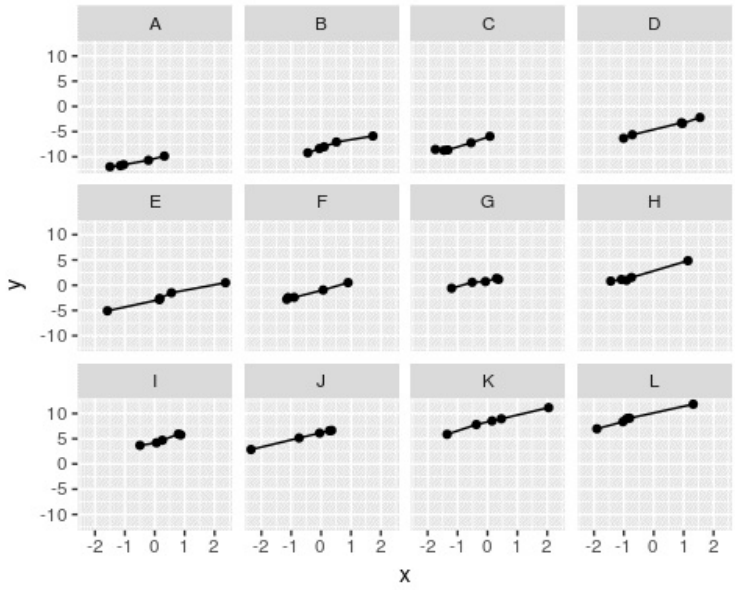
O resultado é um objeto dataframe com 60 linhas e 4 colunas. Podemos observar o gráfico de dispersão de `x` e `y` para cada empresa usando `ggplot2`:

```
library(ggplot2)

p <- ggplot(sim.df, aes(x=x, y=y)) +
  geom_point() + geom_line()+ facet_wrap(~G)

print(p)
```

******ebook converter DEMO Watermarks*******



A figura mostra o forte relacionamento linear entre `x` e `y` nos diferentes grupos. Se estimássemos um modelo linear a partir desses dados, teríamos que permitir um intercepto diferente para cada grupo que encontramos na coluna `G`.

11.3.2 Estimando Modelos de Dados em Painel

Com os dados artificiais simulados no passo anterior, vamos estimar o modelo usando pacote `plm` (Croissant and Millo 2008). Esse é um excelente pacote que oferece um conjunto abrangente de ferramentas para testar e estimar modelos de dados de painel. A interface da função `plm` é semelhante a `lm`. No entanto, precisamos definir o modelo no argumento `model` e os nomes das colunas que definem os grupos e a referência do tempo na entrada `index`.

```
library(plm)

# estimate panel data model with fixed effects
my.pdm <- plm(data = sim.df,
              formula = y ~ x,
              model = 'within',
              index = c('G','dates'))

# print result
print(summary(my.pdm))
```

```
## Oneway (individual) effect Within Model
##
## Call:
## plm(formula = y ~ x, data = sim.df, model = "within", index = c("G",
##     "dates"))
##
## Balanced Panel: n = 12, T = 5, N = 60
##
## Residuals:
##      Min.   1st Qu.    Median   3rd Qu.      Max.
## -0.440310 -0.147656 -0.032982  0.153713  0.479001
```

******ebook converter DEMO Watermarks*******

```
##
## Coefficients:
##   Estimate Std. Error t-value  Pr(>|t|)
## x 1.479366   0.035854  41.261 < 2.2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Total Sum of Squares:    106.87
## Residual Sum of Squares: 2.871
## R-Squared:      0.97313
## Adj. R-Squared: 0.96627
## F-statistic: 1702.44 on 1 and 47 DF, p-value: < 2.22e-16
```

Como esperado, os parâmetros foram recuperados corretamente dos dados, com uma pequena diferença do valor real definido em `my.beta`. Observe que os diferentes interceptos não foram impressos na saída. Podemos recuperá-los usando a função `fixef`:

```
print(fixef(my.pdm))
```

```
##           A           B           C           D           E
## -10.0934047  -8.2435523  -6.3253831  -4.6552624  -2.8087407
##           F           G           H           I           J
##  -0.9794636   0.9609360   2.7568233   4.4134081   6.2113577
##           K           L
##   8.1880249  10.0337231
```

Mais uma vez, os valores simulados de interceptos são muito próximos dos valores obtidos na estimação do modelo. Podemos compará-los diretamente em um dataframe:

```
par.df <- data_frame(real.alpha = my.alphas,
                     estimated.alpha = fixef(my.pdm),
                     perc.diff = (estimated.alpha-real.alpha)/
                                  real.alpha)

print(par.df)
```

```
## # A tibble: 12 x 3
##    real.alpha estimated.alpha perc.diff
##         <dbl>           <dbl>     <dbl>
##  1    -10             -10.1    0.00934
##  2     -8.18           -8.24   0.00755
##  3     -6.36           -6.33  -0.00601
##  4     -4.55           -4.66   0.0242
##  5     -2.73           -2.81   0.0299
##  6     -0.909          -0.979  0.0774
##  7      0.909           0.961  0.0570
##  8      2.73            2.76   0.0108
##  9      4.55            4.41  -0.0291
## 10      6.36            6.21  -0.0239
## 11      8.18            8.19   0.000759
## 12     10              10.0    0.00337
```

Observe como as diferenças percentuais entre os valores simulados e estimados são relativamente pequenas.

Como um exemplo no mundo real, vamos usar o conjunto de dados de Grunfeld (1958). Esse

******ebook converter DEMO Watermarks*******

trabalho de pesquisa estudou os componentes de investimentos corporativos usando dados para dez empresas por vinte anos. Os dados estão disponíveis com o pacote `plm`, e podemos carregá-lo com função `data`. Vamos importá-lo e olhar em seu conteúdo.

```
library(plm)
library(dplyr)

# data from Grunfeld
data("Grunfeld")

# print it
glimpse(Grunfeld)
```

```
## Observations: 200
## Variables: 5
## $ firm    <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ year    <int> 1935, 1936, 1937, 1938, 1939, 1940, 194...
## $ inv     <dbl> 317.6, 391.8, 410.6, 257.7, 330.8, 461....
## $ value   <dbl> 3078.5, 4661.7, 5387.1, 2792.2, 4313.2,...
## $ capital <dbl> 2.8, 52.6, 156.9, 209.2, 203.4, 207.2, ...
```

Uma nota aqui é importante, dado o elevado número de períodos de tempo em proporção ao número de empresas, os dados de Grunfeld são mais adequados para um modelo econométrico mais avançado do tipo SUR (*seemly unrelated regression*). Para fins educacionais de aprendizagem do R, exploraremos outros tipos de modelos de painel com esse conjunto de dados. Veja Greene (2003) para mais detalhes.

Primeiro, vamos explorar os dados brutos estimando um modelo OLS diferente para cada empresa. Isso também é chamado de modelo de dados empilhados (*pooled model*). Podemos usar função `by` com uma função personalizada de estimação para esse fim (veja o capítulo 8 para obter detalhes sobre `by`).

```
my.fct <- function(df) {
  # Estimates a linear model from Grunfeld data
  #
  # Args:
  #   df - dataframe from Grunfeld
  #
  # Returns:
  #   lm object

  my.model <- lm(data = df,
                 formula = inv ~  value + capital)

  return(my.model)
}

# estimate model for each firm
my.l <- by(Grunfeld,
           INDICES = Grunfeld$firm,
           FUN = my.fct)

# print result
my.coefs <- sapply(my.l, coef)
print(my.coefs)
```

******ebook converter DEMO Watermarks*******

```
##                          1           2           3
## (Intercept) -149.7824533 -49.1983219 -9.95630645
## value          0.1192808   0.1748560  0.02655119
## capital        0.3714448   0.3896419  0.15169387
##                          4           5           6           7
## (Intercept) -6.18996051 22.707116014 -8.68554338 -4.4995344
## value        0.07794782  0.162377704  0.13145484  0.0875272
## capital      0.31571819  0.003101737  0.08537427  0.1237814
##                          8           9          10
## (Intercept) -0.50939018 -7.72283708 0.161518567
## value        0.05289413  0.07538794 0.004573432
## capital      0.09240649  0.08210356 0.437369190
```

Os resultados mostram uma grande discrepância entre os coeficientes obtidos para cada empresa. Isso é especialmente verdadeiro para o valor de intercepto, o qual varia de -149.8 para 22.71. Esse resultado mostra evidências de que pode ser mais realista assumir diferentes coeficientes para as diferentes empresas. Podemos testar formalmente essa hipótese com função `plm::polltest`, a qual testa a hipótese nula de que todos os coeficientes são os mesmos em todos os casos, contra a hipótese alternativa de que não são. Veja a seguir:

```
# test if all coef are the same across firms
my.pooltest <- pooltest(inv~value+capital,
                        data = Grunfeld,
                        model = "pooling")

# print result
print(my.pooltest)
```

```
##
##  F statistic
##
## data:  inv ~ value + capital
## F = 27.749, df1 = 27, df2 = 170, p-value < 2.2e-16
## alternative hypothesis: unstability
```

O alto valor do teste F e o pequeno valor de *p* sugerem a rejeição da hipótese nula. A evidência de que os mesmos coeficientes podem ser aplicados a todas as empresas é mínima. A motivação para o uso de modelos de dados de painel para o conjunto de dados de Grunfeld é justificada pelo teste estatístico.

Antes de estimar o modelo, precisamos entender qual o tipo de modelo de dados de painel é mais adequado. Por simplicidade, vamos assumir apenas duas opções possíveis, efeitos fixos ou aleatórios. Em ambos os modelos, cada grupo possui efeitos individuais não observados, mas compartilham o mesmo impacto (*beta*) das variáveis explicativas. A diferença entre os modelos é como o efeito individual não observado é percebido. Os efeitos individuais estão correlacionados com as variáveis explicativas no modelo de efeitos fixos, enquanto que no modelo de efeitos aleatórios são variáveis aleatórias. A estimativa correta do modelo e a análise econométrica irá mudar de acordo com a estrutura de correlação subjacente. Veja Greene (2003) para obter detalhes mais técnicos sobre a diferença entre modelos de efeitos fixos e aleatórios.

Podemos testar a especificação do modelo usando o pacote `plm`. A função `phtest` executa o teste de Hausman (Hausman 1978), um procedimento estatístico que testa a hipótese nula de que o melhor modelo é o efeito aleatório e não o efeito fixo. Vamos utilizar o teste para os nossos

******ebook converter DEMO Watermarks*******

dados.

```
# set options for Hausman test
my.formula <- inv ~ value + capital
my.index <- c('firm','year')

# do Hausman test
my.hausman.test <- phtest(x = my.formula,
                          data = Grunfeld,
                          model = c('within', 'random'),
                          index = my.index)

# print result
print(my.hausman.test)

##
##  Hausman Test
##
## data:  my.formula
## chisq = 2.3304, df = 2, p-value = 0.3119
## alternative hypothesis: one model is inconsistent
```

O valor de *p* igual a 31.19% é superior a um limite aceitável de 10%. Portanto, falhamos na rejeição da hipótese nula de que o tipo de modelo mais eficiente é com efeitos aleatórios. Ou seja, temos fortes evidências estatísticas de que um modelo com efeitos aleatórios é mais adequado para o conjunto de dados de Grunfeld.

Depois de identificar o modelo, vamos estimá-lo usando função `plm::plm`.

```
# set panel data model with random effects
my.model <- 'random'
my.formula <- inv ~ value + capital
my.index <- c('firm','year')

# estimate it
my.pdm.random <- plm(data = Grunfeld,
                     formula = my.formula,
                     model = my.model,
                     index = my.index)

# print result
print(summary(my.pdm.random))

## Oneway (individual) effect Random Effect Model
##    (Swamy-Arora's transformation)
##
## Call:
## plm(formula = my.formula, data = Grunfeld, model = my.model,
##     index = my.index)
##
## Balanced Panel: n = 10, T = 20, N = 200
##
## Effects:
##                   var std.dev share
## idiosyncratic 2784.46   52.77 0.282
## individual    7089.80   84.20 0.718
## theta: 0.8612
##
```

******ebook converter DEMO Watermarks*******

```
## Residuals:
##      Min.   1st Qu.    Median   3rd Qu.      Max.
## -177.6063  -19.7350    4.6851   19.5105  252.8743
##
## Coefficients:
##               Estimate Std. Error t-value Pr(>|t|)
## (Intercept) -57.834415  28.898935 -2.0013  0.04674 *
## value         0.109781   0.010493 10.4627  < 2e-16 ***
## capital       0.308113   0.017180 17.9339  < 2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Total Sum of Squares:    2381400
## Residual Sum of Squares: 548900
## R-Squared:      0.7695
## Adj. R-Squared: 0.76716
## F-statistic: 328.837 on 2 and 197 DF, p-value: < 2.22e-16
```

Como esperado, os coeficientes são estatisticamente significativos a 1%. O ajuste do modelo também é alto, com um R2 ajustado igual a 0.77. Isso significa que uma grande proporção da variação nos dados foi explicada pelo modelo. Os resultados do modelo indicam que o valor das empresas e seus ativos atuais estão positivamente relacionados ao montante de investimentos realizados. As empresas com maior valor de mercado e capital tendem a investir mais.

Como último exemplo para os dados de `Grunfeld`, vamos estimar um modelo SUR, que é mais adequado para esses dados. A especificação SUR assume que os diferentes modelos para cada grupo podem ser estimados individualmente, com uma correlação entre os distúrbios em todos os modelos. Este é mais adequado quando temos muitos períodos de tempo e poucos grupos, como nos dados de `Grunfeld`.

O pacote `systemfit` oferece uma função com o mesmo nome para a estimativa do modelo SUR. O primeiro passo no uso de `systemfit` é alocar os dados do `Grunfeld` para um formato específico de dataframe com a função `plm::pdata.frame`:

```
library(systemfit)

# set pdataframe
p.Grunfeld <- pdata.frame(Grunfeld, c( "firm", "year" ))

# estimate sur
my.SUR <- systemfit(formula = inv ~value + capital,
                    method =  "SUR",
                    data = p.Grunfeld)
print(my.SUR)
```

```
##
## systemfit results
## method: SUR
##
## Coefficients:
##  1_(Intercept)       X1_value     X1_capital 10_(Intercept)
##   -135.6061364      0.1138135      0.3861235      1.9893500
##      X10_value    X10_capital  2_(Intercept)       X2_value
##     -0.0161291      0.3768475    -10.9059829      0.1627658
##     X2_capital  3_(Intercept)       X3_value     X3_capital
```

******ebook converter DEMO Watermarks*******

```
##      0.3406261    -15.8959008      0.0349626      0.1257302
##  4_(Intercept)       X4_value     X4_capital  5_(Intercept)
##      1.8043270      0.0678437      0.3075528     26.4673602
##       X5_value     X5_capital  6_(Intercept)       X6_value
##      0.1274473      0.0119871     -6.1934512      0.1333107
##     X6_capital  7_(Intercept)       X7_value     X7_capital
##      0.0540052     -9.7701305      0.1134649      0.1281802
##  8_(Intercept)       X8_value     X8_capital  9_(Intercept)
##      3.1490972      0.0537015      0.0433622     -3.1568643
##       X9_value     X9_capital
##      0.0765949      0.0654245
```

O objeto de saída contém a estimativa de todas as equações, empresa por empresa. O uso de `print` é limitado nesse caso, mostrando apenas os coeficientes estimados. Função `summary` fornece mais informações, incluindo a estrutura de correlação entre os distúrbios. Mas, a sua saída é extensa e preencheria várias páginas deste livro. Deixamos isso como um exercício para o leitor.

******ebook converter DEMO Watermarks*******

11.4 Modelos ARIMA

A especificação ARIMA (*AutoRegressive Integrated Moving Average*) é um tipo especial de modelo que usa o passado de uma série temporal para explicar o seu comportamento presente. A estimação de um modelo ARIMA para retornos de ações indica como os retornos hoje estão relacionados aos retornos de ontem. Em pesquisa envolvendo modelos de previsão, podemos comparar o desempenho preditivo dos candidatos contra um modelo ARIMA. Se os modelos propostos possuírem superior capacidade preditiva, então esses devem fornecer previsões melhor que um modelo ARIMA.

Um exemplo simples de um modelo ARIMA é definido pela seguinte equação:

$$y_t = 0.5y_{t-1} - 0.2\epsilon_{t-1} + \epsilon_t$$

Neste exemplo, temos um modelo ARIMA(AR=1, D=0, MA=1) sem o intercepto. Essa notação específica informa a configuração do modelo e o número de parâmetros usados. O primeiro valor, AR=1, indica o máximo *lag* usado em *yt* no lado direito da equação. O segundo valor, D, indica o grau de diferenciação da série temporal (J. D. Hamilton 1994). Se *D=1*, usamos a primeira diferença de *yt* como variável dependente. O terceiro componente, *MA*, mostra o máximo *lag* usado para o erro do modelo. Esse processo de identificação pode ser arbitrário ou não. Um procedimento comum é procurar a combinação de termos AR, D e MA que maximizem uma função de ajuste, conforme será mostrado na seção 11.4.2.

11.4.1 Simulando Modelos ARIMA

Em primeiro lugar, vamos simular um modelo ARIMA usando função `stat::arima.sim`. Esse pacote é carregado por padrão na inicialização do R e não precisamos carregá-lo explicitamente com `library`.

```
set.seed(1)

# set number of observations
my.n <- 5000

# set model's parameters
my.model <- list(ar = 0.5, ma = -0.1)
my.sd <- 1

# simulate model
my.ts <- arima.sim(n = my.n,
                   model = my.model ,
                   sd = my.sd)
```

Podemos observar o resultado da simulação criando uma gráfico com a série temporal artificial:

```
library(ggplot2)

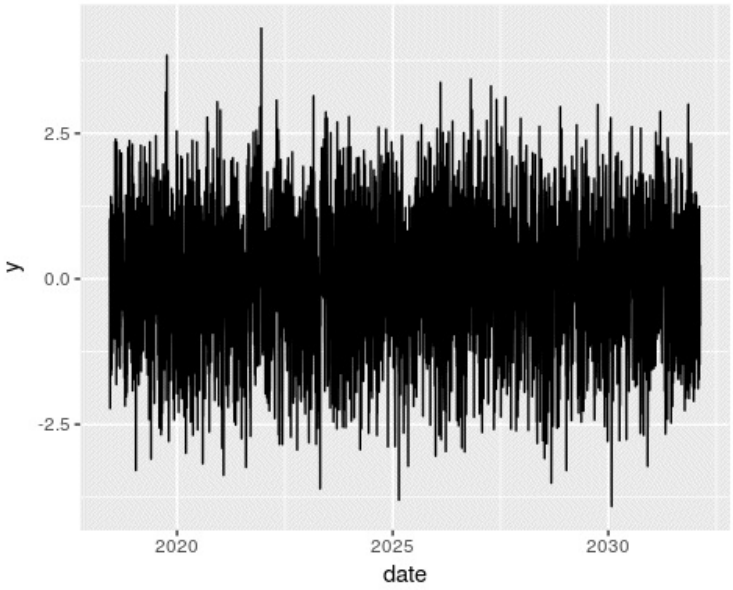
# set df
temp.df <- data.frame(y = unclass(my.ts),
```

******ebook converter DEMO Watermarks*******

```
                      date = Sys.Date() + 1:my.n)

p <- ggplot(temp.df, aes(x = date, y = y))
p <- p + geom_line(size=0.5)

print(p)
```



O gráfico mostra uma série de tempo com média próxima a zero e forte instabilidade. Essas são propriedades típicas de um modelo Arima.

11.4.2 Estimando Modelos ARIMA

Para estimar um modelo ARIMA, usamos função `arima` do mesmo pacote. Vamos estimar um modelo para nossos dados simulados.

```
# estimate arima model
my.arima <- arima(my.ts, order = c(1,0,1))

# print result
print(coef(my.arima))
```

```
##          ar1          ma1    intercept
##  0.482547196 -0.077376754 -0.007458499
```

Como esperado, os parâmetros estimados estão próximos dos valores simulados, com *ar1* igual a 0.4825 e *ma1* igual a -0.07738. Como fizemos para os modelos `lm` e `plm`, também podemos usar função `summary` para obter mais informações sobre o modelo estimado. Veja a seguir:

```
print(summary(my.arima))
```

```
##
## Call:
## arima(x = my.ts, order = c(1, 0, 1))
##
```

******ebook converter DEMO Watermarks*******

```
## Coefficients:
##          ar1      ma1  intercept
##       0.4825  -0.0774    -0.0075
## s.e.  0.0296   0.0337     0.0259
##
## sigma^2 estimated as 1.055:  log likelihood = -7228.15,  aic = 14464.29
##
## Training set error measures:
##                         ME    RMSE       MAE     MPE
## Training set -3.971605e-05 1.02703 0.8183845 223.964
##                   MAPE      MASE         ACF1
## Training set 402.7868 0.8370229 5.538459e-05
##                         ME    RMSE       MAE     MPE
## Training set -3.971605e-05 1.02703 0.8183845 223.964
##                   MAPE      MASE         ACF1
## Training set 402.7868 0.8370229 5.538459e-05
```

A identificação do modelo ARIMA, definindo valores AR, D, MA em Arima (AR, D, MA), também pode ser realizada automaticamente. O pacote `forecast` (Hyndman and Khandakar 2007) oferece função `auto.arima` que automatiza esse processo, escolhendo o melhor modelo de acordo com um critério de ajuste, como AIC (*Akaike information criteria*) e BIC (*bayesian information criteria*). Essa é uma função muito útil. Permitimos que os dados falem por si mesmos, evitando um possível viés de pesquisa na identificação do modelo.

No próximo exemplo usamos função `auto.arima` para encontrar o melhor modelo autoregressivo para os retornos diários do índice Ibovespa. Primeiro, carregamos os dados do arquivo `data/Ibov.csv` e adicionamos uma coluna para os retornos usando a função `calc.ret`.

```
library(readr)

# load Ibov data
df.ibov <- read_csv(file = 'data/Ibov.csv',
                    col_types = cols(ref.date = col_date(format = ""),
                                     price.close = col_integer()))

# calculate return for
df.ibov$ret <- calc.ret(df.ibov$price.close)
```

Antes de estimar o modelo, precisamos verificar a estacionaridade dos retornos. Se os dados não forem estacionários, pode ser necessário usar as primeiras diferenças da série original (Maddala 2001). Uma vez que estamos modificando os retornos, os dados brutos dos preços já foram diferenciados (veja equação de retorno no capítulo 8). Vale a pena testar essa propriedade dos dados antes de estimar o modelo ARIMA. Pacote `tseries` (Trapletti and Hornik 2017) fornece função `adf.test` que testa se um vetor temporal tem raiz unitária (não estacionária). A hipótese nula do teste é a não-estacionariedade dos dados, isto é, a existência de raízes unitárias.

```
library(tseries)
print(adf.test(na.omit(df.ibov$ret)))

##
##  Augmented Dickey-Fuller Test
##
## data:  na.omit(df.ibov$ret)
## Dickey-Fuller = -12.512, Lag order = 12, p-value =
## 0.01
```

******ebook converter DEMO Watermarks*******

```
## alternative hypothesis: stationary
```

O resultado do teste mostra um pequeno valor da estatística *p*, sugerindo a rejeição da hipótese nula. A evidência estatística indica que o vetor de retornos pode ser considerado estacionário. Por curiosidade, vamos executar o teste para a série de preços:

```
print(adf.test(na.omit(df.ibov$price.close)))
```

```
##
##  Augmented Dickey-Fuller Test
##
## data:  na.omit(df.ibov$price.close)
## Dickey-Fuller = -1.8494, Lag order = 12, p-value =
## 0.6421
## alternative hypothesis: stationary
```

Facilmente falhamos em rejeitar a hipótese nula com um grande valor da estatística *p*. O teste sugere que a série de preços não é estacionária. Do ponto de vista econométrico, estamos corretos em estimar um modelo ARIMA para retornos, e não preços.

Indo em frente, função `forecast::auto.arima` possibilita a identificação automática do melhor modelo para os dados. Vamos utilizá-la para os dados de retornos do Ibovespa:

```
library(forecast)

# estimate arima model with automatic identification
my.autoarima <- auto.arima(x = df.ibov$ret)

# print result
print(my.autoarima)
```

```
## Series:
## ARIMA(0,0,0) with zero mean
##
## sigma^2 estimated as 0.0002023:  log likelihood=5853.74
## AIC=-11705.48   AICc=-11705.48   BIC=-11699.83
```

O resultado nos diz que o melhor modelo para os retornos do índice Ibovespa é um ARIMA(AR=0, D=0, MA=0). Esse resultado implica que a série de retorno do índice financeiro não tem uma memória significativa, ou seja, o retorno passado não tem poder preditivo sobre o retorno futuro. Este resultado corrobora com a ideia de que preços do mercado financeiro são difíceis de prever.

11.4.3 Prevendo Modelos ARIMA

Podemos obter as previsões de um modelo ARIMA com função `forecast::forecast`. A previsão é do tipo estático: apenas informações até o momento *t* são usadas para fazer previsões em *t* + *k*. No exemplo a seguir, calculamos previsões para 5 períodos à frente, com seu intervalo de confiança correspondente.

```
# forecast model
print(forecast(my.autoarima, h = 5))
```

******ebook converter DEMO Watermarks*******

```
##      Point Forecast       Lo 80      Hi 80       Lo 95
## 2103              0 -0.01822751 0.01822751 -0.02787657
## 2104              0 -0.01822751 0.01822751 -0.02787657
## 2105              0 -0.01822751 0.01822751 -0.02787657
## 2106              0 -0.01822751 0.01822751 -0.02787657
## 2107              0 -0.01822751 0.01822751 -0.02787657
##           Hi 95
## 2103 0.02787657
## 2104 0.02787657
## 2105 0.02787657
## 2106 0.02787657
## 2107 0.02787657
```

Note que a previsão em $t+k$ converge para um valor. Esta é uma dinâmica típica e esperada de previsões estáticas.

******ebook converter DEMO Watermarks*******

11.5 Modelos GARCH

Os modelos do tipo GARCH (*generalized autoregressive conditional heteroskedasticity*) relacionam-se ao trabalho seminal de Engle (1982) e Bollerslev (1986). A principal inovação nessa classe de modelos é que a distribuição do vetor de resíduos é assumida como variante no tempo e modelada usando um processo autoregressivo específico. Lembre-se que nos modelos do tipo OLS assumimos que a distribuição dos erros é igual para qualquer período do tempo -- premissa de homoscesdasticidade.

Os modelos GARCH tornaram-se muito populares na literatura pois reproduzem mais fielmente as características encontradas na distribuição de retornos financeiros, tal como a existência de caudas gordas e agrupamento de períodos voláteis. Modelos GARCH são utilizados principalmente na avaliação e gerenciamento do risco financeiro.

Um modelo GARCH é modular. No formato mais simples, temos duas equações principais: um processo que define a média condicional e outro que define a variância do erro. Verifique o exemplo a seguir para um modelo ARIMA(1,0,0)-GARCH(1,1):

$$y_t = \mu + \theta y_{t-1} + \epsilon_t$$
$$\epsilon_t \sim N\left(0, h_t\right)$$
$$h_t = \omega + \alpha \epsilon_{t-1}^2 + \beta h_{t-1}$$

A equação para *yt* define o processo para a média condicional, o valor observado para a série modelada. Variável *ht* representa a variância do erro, isto é, a instabilidade do modelo. Diferentes modelos GARCH terão diferentes equações para *ht* e diferentes distribuições para o erro. No caso mais simples apresentado aqui, usamos a distribuição Normal. É importante entendermos essa notação básica de modelos Garch pois as funções de estimação e simulação seguem o mesmo formato.

11.5.1 Simulando Modelos GARCH

O R não tem função nativa para simular e estimar modelos GARCH. Aqui destacamos dois pacotes principais: `fGarch` (Wuertz et al. 2016) e `rugarch` (Ghalanos 2015). Ambos têm excelentes recursos e são otimizados para estimações ágeis. Você será bem servido na escolha de qualquer um deles. Por simplicidade, daremos preferência ao pacote `fGarch`, o qual tem uma interface semelhante à usada nos modelos ARIMA da seção anterior.

Simulamos um modelo usando a função `fGarch::garchSim`. O primeiro passo é carregar o pacote `fGarch` e criar a especificação do modelo:

```
library(fGarch)

# set list with model spec
my.model = list(omega=0.001,
```

******ebook converter DEMO Watermarks*******

```
                alpha=0.15,
                beta=0.8,
                mu=0.02,
                ar = 0.1)

# set garch spec
spec = garchSpec(model = my.model)

# print it
print(spec)
```

```
##
## Formula:
##  ~ ar(1) + garch(1, 1)
## Model:
##  ar:    0.1
##  mu:    0.02
##  omega: 0.001
##  alpha: 0.15
##  beta:  0.8
## Distribution:
##  norm
## Presample:
##   time         z    h          y
## 1    0 0.1145392 0.02 0.02222222
```

O código anterior define um modelo GARCH equivalente as seguintes equações:

$$
\begin{aligned}
y_t &= 0.02 + 0.1 y_{t-1} + \epsilon_t \\
\epsilon_t &\sim N\left(0, h_t\right) \\
h_t &= 0.001 + 0.15 \epsilon_{t-1}^2 + 0.8 h_{t-1}
\end{aligned}
$$

Para simular *1000* observações do modelo, usamos função `fGarch::garchSim`:

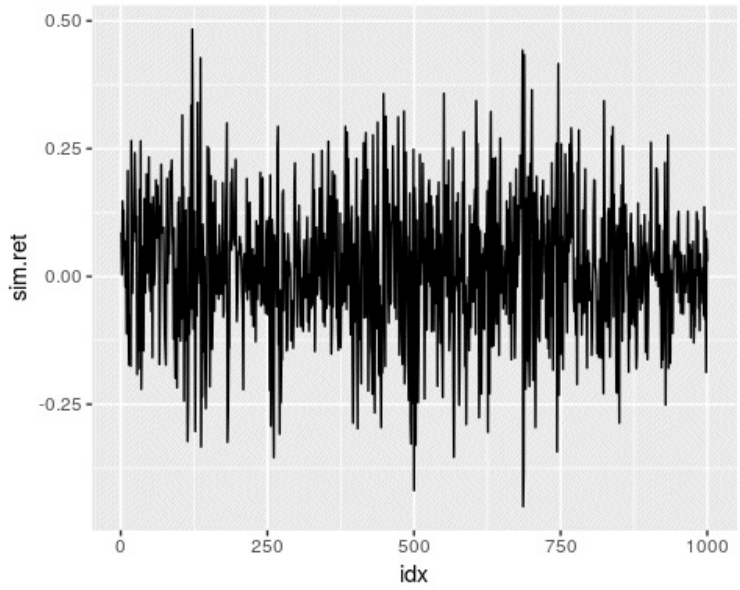
```
set.seed(20)
# simulate garch model
sim.garch = garchSim(spec, n = 1000)
```

Podemos visualizar a série temporal gerada com `ggplot2::ggplot`:

```
# set df for ggplot
temp.df <- data.frame(sim.ret = sim.garch$garch,
                      idx=seq_along(sim.garch$garch))

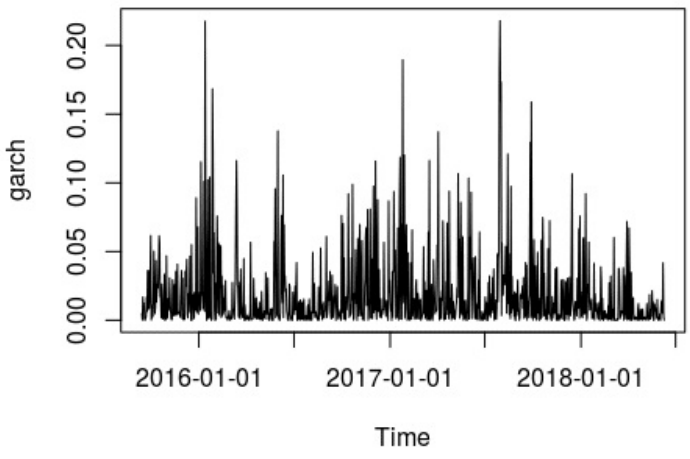
library(ggplot2)
p <- ggplot(temp.df, aes(x=idx, y=sim.ret))
p <- p + geom_line()
print(p)
```

******ebook converter DEMO Watermarks*******



O comportamento da série simulada é semelhante à série de retorno das ações apresentadas no capítulo 10. Ao utilizar um modelo GARCH na simulação de retornos, é difícil diferenciar visualmente uma série artificial de uma real. Ao contrário de outros modelos, onde a instabilidade é constante, um modelo GARCH pode retratar uma série de retorno de forma mais realista, com uma volatilidade que muda ao longo do tempo. No código a seguir plotamos a volatilidade da série simulada usando função `plot` e `fGarch::volatility`.

```
plot(volatility(sim.garch))
```



Claramente podemos ver a instabilidade da volatilidade da série simulada.

11.5.2 Estimando Modelos GARCH

******ebook converter DEMO Watermarks*******

Os parâmetros de um modelo GARCH geralmente são encontrados através de uma técnica de estimação chamada máxima verosimilhança (*maximum-likelihood*). Esse procedimento encontra os parâmetros que tornam a distribuição do modelo mais próxima possível da distribuição das séries temporais de interesse. O método envolve um processo de otimização numérica que exige uma quantidade razoável de tempo de processamento. Felizmente, pacote `fGarch` fornece a função `garchFit` que executa toda a operação.

No exemplo a seguir estimamos um modelo GARCH para os dados artificiais criados na seção anterior. Definimos a opção `trace = FALSE` para evitar a apresentação dos detalhes do processo de otimização, pois esses são extensos e ocupariam várias páginas deste livro.

```
# estimate garch model
my.garchfit <- garchFit(data = sim.garch,
                        formula = ~ arma(1,0) + garch(1,1),
                        trace = FALSE)
```

Para aprender mais sobre o modelo estimado, o apresentamos na tela com `print`:

```
print(my.garchfit)
```

```
##
## Title:
##  GARCH Modelling
##
## Call:
##  garchFit(formula = ~arma(1, 0) + garch(1, 1), data = sim.garch,
##     trace = FALSE)
##
## Mean and Variance Equation:
##  data ~ arma(1, 0) + garch(1, 1)
## <environment: 0x13fe6ca8>
##  [data = sim.garch]
##
## Conditional Distribution:
##  norm
##
## Coefficient(s):
##        mu        ar1      omega     alpha1      beta1
## 0.0164569  0.0695426  0.0010592  0.1292775  0.8175425
##
## Std. Errors:
##  based on Hessian
##
## Error Analysis:
##         Estimate  Std. Error  t value Pr(>|t|)
## mu     0.0164569   0.0039213    4.197 2.71e-05 ***
## ar1    0.0695426   0.0327651    2.122   0.0338 *
## omega  0.0010592   0.0004267    2.482   0.0131 *
## alpha1 0.1292775   0.0282136    4.582 4.60e-06 ***
## beta1  0.8175425   0.0405741   20.149  < 2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Log Likelihood:
##  604.1894    normalized:  0.6041894
```

******ebook converter DEMO Watermarks*******

```
##
## Description:
##  Sun Jun 10 14:57:07 2018 by user:
```

Os parâmetros resultantes da estimação do modelo estão próximos dos valores arbitrariamente definidos com `garchSpec`. Podemos alcançar maior precisão aumentando o número de observações no modelo simulado. Função `summary` também funciona para modelos GARCH. Devido à grande quantidade de informações no *prompt*, deixamos o seu uso como um exercício para o leitor.

Agora, vamos estimar um modelo GARCH para dados reais de retorno do índice Ibovespa. Primeiro, executamos o teste LM Arch (Engle 1982; Tsay 2005) para verificar se os retornos do índice de mercado têm o efeito Arch. Função `MTS::archTest` (Tsay 2015) executa essa tarefa.

```
library(MTS)

# test for Arch effects
archTest(rt = na.omit(df.ibov$ret))
```

```
## Q(m) of squared series(LM test):
## Test statistic:  148.6229  p-value:  0
## Rank-based Test:
## Test statistic:  80.20112  p-value:  4.585221e-13
```

A evidência é forte para efeitos Arch nos retornos do Ibovespa. A hipótese nula do teste é a inexistência de efeitos Arch, e podemos rejeitá-lo facilmente com um baixíssimo valor de *p*. Com esse resultado, partimos para a estimação de um Arma (1,0)-Garch(1,1) para os retornos.

```
# set object for estimation
df.est <- as.timeSeries(na.omit(df.ibov))

# estimate garch model
my.garchfit.ibov <- garchFit(data = df.est ,
                             formula = ret ~ arma(1,0) + garch(1,1),
                             trace = FALSE)

print(my.garchfit.ibov)
```

```
##
## Title:
##  GARCH Modelling
##
## Call:
##  garchFit(formula = ret ~ arma(1, 0) + garch(1, 1), data = df.est,
##     trace = FALSE)
##
## Mean and Variance Equation:
##  ret ~ arma(1, 0) + garch(1, 1)
##  [data = df.est]
##
## Conditional Distribution:
##  norm
##
## Coefficient(s):
##         mu         ar1       omega      alpha1       beta1
## 0.00031443  0.00604380  0.00000799  0.06628154  0.89539395
```

******ebook converter DEMO Watermarks*******

```
##
## Std. Errors:
##  based on Hessian
##
## Error Analysis:
##         Estimate  Std. Error  t value Pr(>|t|)
## mu     3.144e-04   2.919e-04    1.077  0.28148
## ar1    6.044e-03   2.320e-02    0.261  0.79448
## omega  7.990e-06   2.265e-06    3.527  0.00042 ***
## alpha1 6.628e-02   1.247e-02    5.317 1.05e-07 ***
## beta1  8.954e-01   1.976e-02   45.303  < 2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Log Likelihood:
##  5924.353    normalized:  2.860624
##
## Description:
##  Sun Jun 10 14:57:07 2018 by user:
```

Como esperado, todos os coeficientes GARCH (`omega`, `alpha1`, `beta1`) são significativos a 1%. Quanto à equação de média condicional, observamos novamente que o termo autoregressivo, `ar1`, não foi significativo.

11.5.3 Prevendo Modelos GARCH

A previsão de modelos GARCH envolve dois elementos: uma previsão para a média condicional (veja a primeira equação na fórmula GARCH) e uma previsão para valores futuros da volatilidade condicional (veja segunda equação). Enquanto a primeira define a previsão dos próximos valores da série analisada, a segunda quantifica a incerteza dessa previsão.

No pacote `fGarch`, ambas as previsões são calculadas usando função `predict` que, assim como `summary`, é uma função genérica que pode ser usada para diferentes modelos. Considere o seguinte exemplo, onde prevemos os próximos valores e as volatilidades futuras do modelo GARCH estimado para o índice Ibovespa.

```
# static forecast for garch
my.garch.forecast <- predict(my.garchfit.ibov, n.ahead = 3)

# print df
print(my.garch.forecast)
```

```
##   meanForecast  meanError standardDeviation
## 1 0.0002400358 0.01776460        0.01776460
## 2 0.0003158770 0.01764902        0.01764870
## 3 0.0003163353 0.01753684        0.01753651
```

A primeira coluna do resultado anterior é a previsão da média condicional; a segunda apresenta o erro esperado da previsão anterior e a terceira indica a volatilidade esperada no desvio padrão (raiz da variância). Todas as previsões são do tipo estático, isto é, informação até o tempo *t* são utilizadas para realizar previsões para *t* + *k*.

******ebook converter DEMO Watermarks*******

******ebook converter DEMO Watermarks*******

11.6 Modelos de Mudança de Regime

Modelos de mudança de regime são um tipo de especificação econométrica onde a inovação é a flexibilidade no manejo de diferentes estados nos dados (J. Hamilton 1994). Nos mercados financeiros, podemos ter dois regimes de incerteza, um regime em que a volatilidade é alta e outra onde é baixa. Podemos justificar esses regimes como períodos de tempo com maior ou menor quantidade de novas informações e incerteza. Cada regime pode ter suas próprias características. Como modelador, precisamos identificar esses regimes e estimar os parâmetros de nossos modelos de forma conjunta.

Para motivar o modelo de mudança de regime, considere o seguinte processo econométrico:

$$y_t = \mu_{S_t} + \epsilon_t$$

onde *St=1..k* e the residual seguem a distribuição Normal com média zero e variância igual a different regimes as well. Este é o caso mais simples de um modelo com uma dinâmica de mudança de estados. Se houver *k* estados no mundo, haverá *k* valores para a média e variância condicional. Se houver apenas um estado do mundo (*k = 1*), a fórmula anterior se torna um modelo de regressão linear simples sob condições gerais.

Agora, vamos assumir que o modelo anterior tem dois estados (*k = 2*). Uma representação alternativa é:

$$\begin{aligned} y_t &= \mu_1 + \epsilon_t \qquad \text{for State 1} \\ y_t &= \mu_2 + \epsilon_t \qquad \text{for State 2} \end{aligned}$$

Onde:

$$\begin{aligned} \epsilon_t &\sim (0, \sigma_1^2) \qquad \text{for State 1} \\ \epsilon_t &\sim (0, \sigma_2^2) \qquad \text{for State 2} \end{aligned}$$

Essa representação implica dois processos para a variável dependente. Quando o estado do mundo para o tempo *t* é 1, a expectativa da variável dependente é *u1* e a volatilidade das inovações é $^2_1 $. Da mesma forma, quando o estado é 2, a média e a volatilidade tomam outros valores.

Como exemplo em finanças, a variável dependente yty_t pode representar um vetor de retornos. O valor de *u1* é o retorno esperado em um estado de mercado com preços ascendentes (*bull market*), o que implica um retorno positivo para a ação ou índice. O valor inferior, e possivelmente negativo, de *2* mede o retorno esperado para o mercado nos estados de preços descendentes (*bear market*).

As diferentes volatilidades representam a maior incerteza sobre o poder preditivo do modelo em cada estado do mundo. Podemos esperar que o *bear market* seja mais volátil do que o *bull market*, ou seja, os preços caem mais rápido do que eles sobem. A usual explicação para esse

******ebook converter DEMO Watermarks*******

efeito é que operadores do mercado reagem mais rápido quando confrontado por más notícias em comparação a chegada de boas notícias. Isso também pode ser explicado por ordens do tipo limitada, as quais são enviadas ao mercado assim que um limite de preço é passado. Quando uma quantidade significativa de ordens limite de venda são encontradas em uma faixa de preço, observa-se um efeito cascata que acelera a queda dos preços.

As mudanças dos estados no modelo podem ser definidas de forma determinística. Poderíamos definir o estado 1 como verdadeiro para o tempo *t* quando outra série temporal é maior ou menor do que um limite conhecido. Isso simplifica muito o modelo, pois cada estado é observável; portanto, podemos tratar o modelo como uma regressão com variáveis dicotômicas (1 ou 0). A função `lm` pode ser usada para a estimativa desse modelo.

Uma classe especial de modelos de mudança de regime são aqueles relacionados a cadeias de *markov*. Sua principal inovação é que a identificação dos estados faz parte do processo de estimação. O modelo **descobre** os estados a partir dos dados. A transição dos estados em um modelo de mudança de *markov* não é determinística e sim estocástica. Isso significa que nunca se tem certeza se haverá uma mudança de estado, temos apenas uma indicação na forma de uma probabilidade. A dinâmica por trás do processo de mudança é conhecida e conduzida por uma matriz de transição. Essa matriz, também estimada a partir dos dados, controlará as probabilidades de se observar uma mudança entre um estado e outro. Podemos representar a matriz de transição como:

$$P = \begin{bmatrix} p_{11} & \cdots & p_{1k} \\ \vdots & \ddots & \vdots \\ p_{k1} & \cdots & p_{kk} \end{bmatrix}$$

Na matriz anterior, o elemento da linha *i* e coluna *j* controla a probabilidade de uma mudança do estado *j* para o estado *i*. Considere que, por algum tempo *t*, o estado do mundo é 2. Isso significa que a probabilidade de uma mudança do estado 2 para o estado 1 entre o tempo *t* e *t + 1* será dada por *p12*. Da mesma forma, a probabilidade de se permanecer no estado 2 é determinada por *p22*. Esse é um dos pontos centrais da estrutura de um modelo de mudança de regime do tipo *markov*: a mudança de estados é um processo estocástico por si mesmo.

11.6.1 Simulando Modelos de Mudança de Regime

No R, dois pacotes estão disponíveis para manipular modelos univariados de mudanças de regime: `MSwM` (Sanchez-Espigares and Lopez-Moreno 2014) e `fMarkovSwitching` (Perlin 2014). Ambos pacotes oferecem uma gama de funcionalidade e o leitor estará bem servido com qualquer um. Porém, `fMarkovSwitching` oferece também funções de simulação e, por isso, daremos preferência ao mesmo. Antes de usá-lo, vamos instalar `fMarkovSwitching` do repositório R-Forge. Até o momento, esse pacote não está disponível no CRAN.

```
install.packages("fMarkovSwitching",
                 repos="http://R-Forge.R-project.org")
```

Após o término da instalação, vamos verificar as suas funções:

******ebook converter DEMO Watermarks*******

```
library(fMarkovSwitching)
print(ls('package:fMarkovSwitching'))
```

```
## [1] "dim.MS_Model"     "MS_Regress_Fit"   "MS_Regress_For"
## [4] "MS_Regress_Lik"   "MS_Regress_Simul" "plot.MS_Model"
## [7] "plot.MS_Simul"    "print.MS_Model"   "print.MS_Simul"
```

O pacote inclui funções para simular, estimar e prever um modelo univaridado de mudanças de *markov*. Como exemplo, vamos simular o modelo de mudança de regime representados pelas seguintes equações:

$$
\begin{aligned}
y_t &= +0.5x_t + \epsilon_t \qquad \text{State 1} \\
y_t &= -0.5x_t + \epsilon_t \qquad \text{State 2} \\
\epsilon_t &\sim N(0, 0.25) \qquad \text{State 1} \\
\epsilon_t &\sim N(0, 1) \qquad \text{State 2}
\end{aligned}
$$

A matriz de transição será dada por:

$$
P = \begin{bmatrix} 0.90 & 0.2 \\ 0.10 & 0.8 \end{bmatrix}
$$

Esse modelo tem dois estados com diferentes volatilidades. Em cada estado, o impacto da variável explicativa será diferente. Utilizamos função `fMarkovSwitching::MS_Regress_Simul` para simular esse modelo:

```
set.seed(10)
library(fMarkovSwitching)

# number of obs
nr <- 500

# distribution of residuals
distrib <- "Normal"

# number of states
k <- 2

# set transition matrix
P <- matrix(c(.9 ,.2,
              .1 ,.8),
            nrow = 2,
            byrow = T)

# set switching flag
S <- c(0,1)

# set parameters of model (see manual for details)
nS_param <- matrix(0)
S_param <- matrix(0,sum(S),k)
S_param[,1] <-  .5
S_param[,2] <- -.5

# set variance of model
```

******ebook converter DEMO Watermarks*******

```
sigma <- matrix(0,1,k)
sigma[1,1] <- sqrt(0.25)  # state 1
sigma[1,2] <- 1           # state 2

# build list
Coeff <- list(P = P               ,
              S = S               ,
              nS_param = nS_param ,
              S_param = S_param   ,
              sigma = sigma       )

# simulate model
my.ms.simul <- MS_Regress_Simul(nr,Coeff,k,distrib)
```

Na função de simulação, argumento `nS_param` define os parâmetros que não mudam de regime. Esses são os coeficientes no lado direito da equação que não alternarão estados. Usamos um valor de zero, pois nosso modelo simulado não possui coeficientes que não mudam de estado. Mesmo que não seja usado, precisamos definir esse argumento de forma explícita em `MS_Regress_Simul`.

Os elementos em `S_param` definem os coeficientes em cada estado para os parâmetros de entrada. No nosso exemplo, temos um efeito positivo de *x* em *y* no estado um e um efeito negativo no estado dois. Finalmente, entrada `sigma` define a volatilidade (desvio padrão) do resíduo em cada regime.

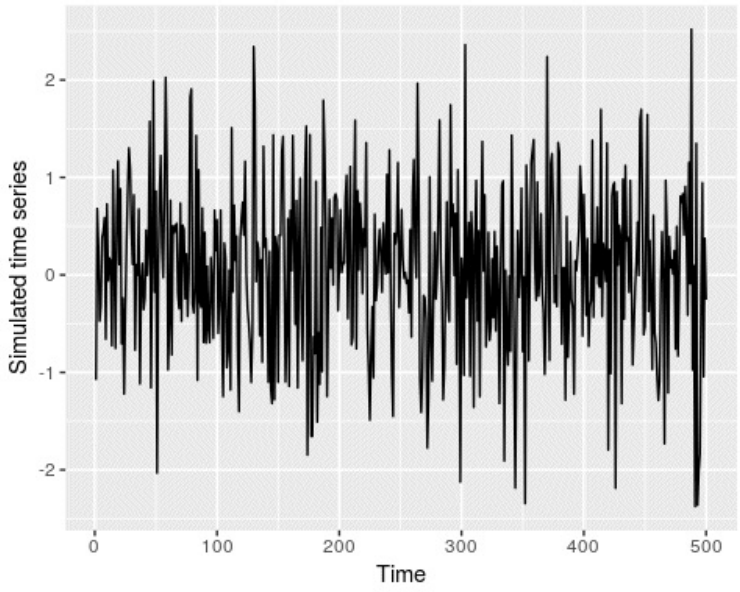
Uma vez que o modelo é simulado e disponível, vamos criar uma figura com a série temporal de valores artificiais. Uma nota aqui é importante; a saída de `my.ms.simul` é um objeto do tipo `S3` e foi projetado para interagir com as funções `print` e `plot`. Para acessar seus elementos, usamos `@` em vez de `$`.

```
library(ggplot2)
df.to.plot <- data.frame(y = my.ms.simul@dep,
                         x = Sys.Date()+1:my.ms.simul@nr,
                         states = my.ms.simul@trueStates[,1])

p <- ggplot(data = df.to.plot, aes(y=y, x=seq_along(y))) +
  geom_line() + labs(x='Time', y = 'Simulated time series')

print(p)
```

******ebook converter DEMO Watermarks*******



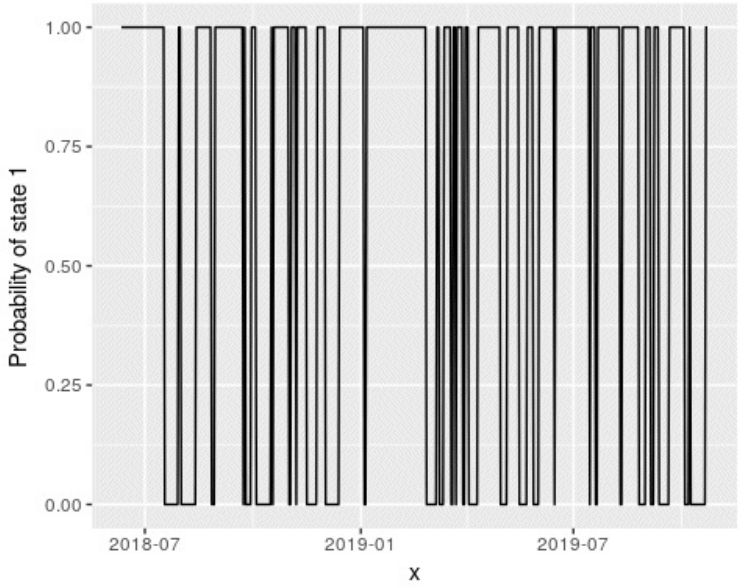
Também podemos verificar os estados simulados:

```
library(ggplot2)
library(dplyr)

df.to.plot <- data_frame(y = as.numeric(my.ms.simul@dep),
                         x = Sys.Date()+1:my.ms.simul@nr,
                         states = my.ms.simul@trueStates[,1])

p <- ggplot(data = df.to.plot, aes(y=states, x=x)) +
  geom_line() + labs(y='Probability of state 1')

print(p)
```



Como esperado, o modelo está mudando de um estado para o outro. Nenhum estado é fortemente

******ebook converter DEMO Watermarks*******

predominante ao longo do tempo, porém o estado 1 parece ter uma duração média maior do que o estado 2. Essa propriedade é controlada pelas probabilidades de transição definidas no elemento `P` de `Coeff`.

11.6.2 Estimando Modelos de Mudança de Regime

Estimamos um modelo univariado de mudança de regime com função `MS_Regress_Fit` do pacote `fMarkovSwitching`. Vamos utilizá-la para os dados simulados:

```
# set dep and indep
dep <- my.ms.simul@dep
indep <- my.ms.simul@indep

# set switching parameters and distribution
S <- c(0,1)
k <- 2
distIn <- "Normal"

# estimate the model
my.MS.model <- MS_Regress_Fit(dep,indep,S,k)    # fitting the model
```

Argumentos `dep` e `indep` definem as variáveis dependente e independente, lado esquerdo e direito da equação econométrica. Entrada `S` define onde o processo de troca de regimes ocorrerá. Uma vez que temos apenas duas variáveis independentes, onde a primeira não troca de estados, usamos `S <- c(0,1)`. O objeto `k` define o número de estados no modelo, nesse caso dois. Após a execução do código, verificamos o resultado.

```
# print estimation output
print(my.MS.model)
```

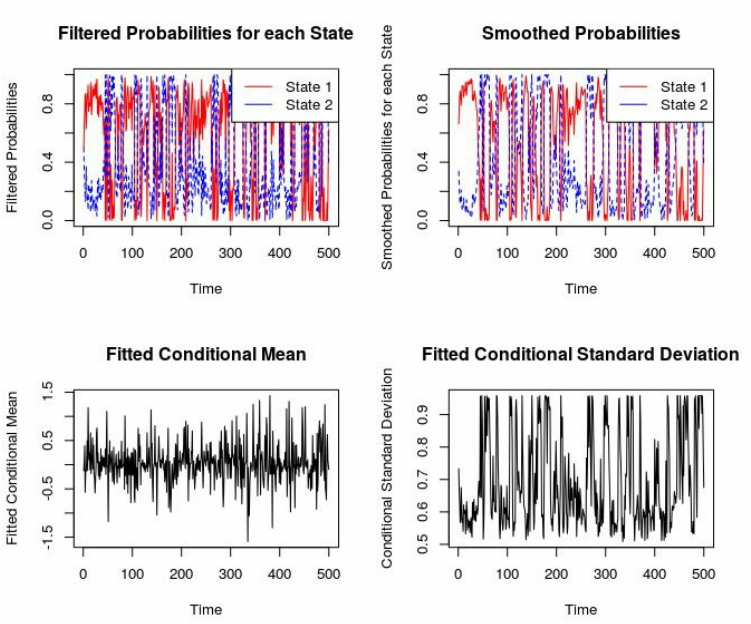
```
##
##
## ***** Numerical Optimization for MS Model Converged *****
##
## Final log Likelihood: -544.5191
## Number of parameters: 7
## Distribution Assumption -> Normal
##
## ***** Final Parameters *****
##
## ---> Non Switching Parameters <---
##
##  Non Switching Parameter at Indep  Column  1
##       Value:     -0.0333
##       Std error: 0.0266 (0.21)
##
## --->   Switching Parameters   <---
##
##   State 1
##       Model Standard Deviation: 0.5082
##       Std Error:                0.0235 (0.00)
##   State 2
##       Model Standard Deviation: 0.9584
##       Std Error:                0.0613 (0.00)
##
##   Switching Parameters for Indep  Column  2
```

******ebook converter DEMO Watermarks*******

```
##
##   State  1
##      Value:      0.5471
##      Std error:  0.0313 (0.00)
##   State  2
##      Value:      -0.4563
##      Std error:  0.0979 (0.00)
##
## ---> Transition Probabilities Matrix <---
##
##       0.90   0.21
##       0.10   0.79
##
## ---> Expected Duration of Regimes <---
##
##      Expected duration of Regime #1: 10.52 time periods
##      Expected duration of Regime #2: 4.81 time periods
```

Os coeficientes estimados são próximos dos utilizados na simulação. O objeto de saída de `MS_Regress_Fit` também pode ser usado com função `plot` para uma figura personalizada:

```
plot(my.MS.model)   # plotting output
```



O nosso exemplo com dados reais será com o índice Ibovespa. Vamos estimar um modelo de dois estados para os seus retornos.

```
df.ibov <- na.omit(df.ibov)

# set input objects to MS_Regress_Fit
ret <- na.omit(df.ibov$ret)
dep <- matrix(ret, nrow = length(ret))
indep <- matrix(rep(1, length(dep)),nrow = length(dep))

S <- c(1)   # where to switch (in this case in the only indep)
k <- 2      # number of states
```

******ebook converter DEMO Watermarks*******

```
distIn <- "Normal" #distribution assumption

my.ibov.MS.model <- MS_Regress_Fit(dep,indep,S,k)   # fitting the model
```

Por fim, checamos a saída do modelo com `print`:

```
# printing output
print(my.ibov.MS.model)
```

```
##
##
## ***** Numerical Optimization for MS Model Converged *****
##
## Final log Likelihood: 5850.509
## Number of parameters: 6
## Distribution Assumption -> Normal
##
## ***** Final Parameters *****
##
## ---> Non Switching Parameters <---
##
## There was no Non Switching Parameters. Skipping this result
##
## --->   Switching Parameters   <---
##
##   State 1
##       Model Standard Deviation: 0.0143
##       Std Error:                 NaN ( NaN)
##   State 2
##       Model Standard Deviation: 0.0143
##       Std Error:                 NaN ( NaN)
##
##   Switching Parameters for Indep  Column  1
##
##   State  1
##      Value:      0.0001
##      Std error:   NaN ( NaN)
##   State  2
##      Value:      0.0001
##      Std error:   NaN ( NaN)
##
## ---> Transition Probabilities Matrix <---
##
##       0.90   0.10
##       0.10   0.90
##
## ---> Expected Duration of Regimes <---
##
##      Expected duration of Regime #1: 10.00 time periods
##      Expected duration of Regime #2: 10.00 time periods
```

O modelo identificou e separou os dois regimes de volatilidade dos retornos do Ibovespa. No primeiro regime de alta volatilidade, o desvio padrão dos retornos é 1.43%. No segundo estado, o valor do desvio padrão é 1.43%. Conforme esperado, o estado de alta volatilidade tem uma média negativa de 0.0124% e o estado de baixa volatilidade tem uma média positiva de 0.0132%. Os resultados da estimação do modelo implicam que o índice Ibovespa cai mais rápido do que sobe. Informações mais interessantes estão relacionadas à duração esperada dos estados.

******ebook converter DEMO Watermarks*******

O estado com retornos médios positivos tende a durar cerca de 106 dias, enquanto o ciclo do estado com retornos negativos dura cerca de 44 dias. Ou seja, o mercado sobe devagar e por bastante tempo. Quando entra no ciclo de baixa, os preços caem rapidamente em um breve período.

Uma figura comum na análise de modelos de mudança de *markov* é a dinâmica do preço modeloado em diferentes estados. Vamos recriar este gráfico no R. Primeiro, separamos as variáveis de interesse e construimos um dataframe para o gráfico. Em seguida, usamos `ggplot` e `facet_wrap` para construir os painéis.

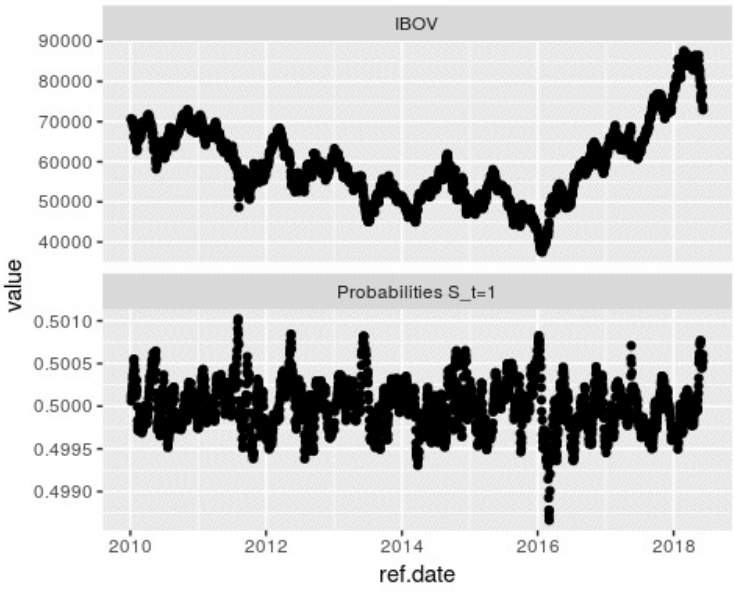
```
library(dplyr)

# get variables for plot
smooth.prob <- as.numeric(my.ibov.MS.model@smoothProb[-1,1])
price <- df.ibov$price.close[2:nrow(df.ibov)]
ref.dates <- df.ibov$ref.date[2:nrow(df.ibov)]

# build long df to plot
df.to.plot <- data.frame(type = c(rep('Probabilities S_t=1', length(smooth.prob)),
                                  rep('IBOV', length(smooth.prob))),
                         ref.date = rep(ref.dates ,2),
                         value = c(smooth.prob,
                                   price) )

# plot with ggplot
p <- ggplot(df.to.plot,
            aes(y=value, x =ref.date)) +
  geom_point() + facet_wrap(~type, nrow = 2, scales = 'free_y')

# plot it!
print(p)
```



A figura mostra como os preços diminuiram fortemente no estado 1 e aumentaram no estado 2.

******ebook converter DEMO Watermarks*******

11.6.3 Prevendo Modelos de Mudança de Regime

Pacote `MS_Regress` fornece função `MS_Regress_For` para a previsão estática de um modelo de mudança de regime markoviano. As entradas da função são: um modelo estimado com `MS_Regress_Fit`, argumento `myModel`, e o conjunto de novas variáveis explicativas na entrada `newIndep`. Vamos usá-lo para prever o próximo retorno do Ibovespa. No nosso caso, uma vez que o modelo de mudança de regime teve apenas um coeficiente para o intercepto, definimos `newIndep = 1`.

```
# make static forecast of regime switching model
newIndep <- 1

my.for <- MS_Regress_For(my.ibov.MS.model, newIndep)

# print output
print(my.for)
```

```
## $condMean
##              [,1]
## [1,] 0.0001276427
##
## $condStd
##            [,1]
## [1,] 0.01433179
```

O modelo prevê que, para o dia depois da última data disponível nos dados de `Ibov.csv`, 2018-06-08, o índice do mercado de ações Brasileiro irá aumentar seu valor em 0.013%, com uma volatilidade igual a 1.4%.

******ebook converter DEMO Watermarks*******

11.7 Trabalhando com Diversos Modelos

Na prática da pesquisa, é provável que o estudo necessite a estimação de mais de um modelo. Podemos ter diferentes grupos nos dados ou então necessitar a execução de testes de robustez, estimando o mesmo modelo em diferentes períodos de tempo. Aprender a estimar e gerenciar diferentes modelos de forma eficiente é importante. Essa questão torna-se extremamente relevante quando a escala da pesquisa aumenta. Mais dados e mais modelos exigem uma estrutura computacional eficiente.

No capítulo 8 aprendemos sobre o uso das funções da família `apply`, e pacotes `purrr` e `dplyr` para fazer tarefas iterativas. Da mesma forma, podemos usar essas funções para estimar vários modelos a partir dos dados. Vamos começar com um exemplo. Aqui, vamos estimar um modelo ARIMA para os retornos de quatro ações selecionadas aleatoriamente. Primeiro vamos carregar os dados e filtrar para as quatro ações:

```
library(readr)

set.seed(10)

# set number of stocks
n.stocks <- 4

# load data from .RData
my.df <- read_csv('data/IbovStocks_long.csv',
                  col_types = cols(ref.date = col_date(format = ""),
                                   ticker = col_character(),
                                   price.adjusted = col_double()))

# calc ret
my.df$ret <- calc.ret(my.df$price.adjusted, my.df$ticker)

# select tickers
my.tickers <- sample(unique(my.df$ticker), n.stocks)

# set my.df
my.df.stocks <- my.df[my.df$ticker %in% my.tickers, ]

# renew factors in ticker
my.df.stocks$ticker <- as.factor(as.character(my.df.stocks$ticker))
```

Agora, queremos ter um modelo ARIMA diferente para cada ação. Existem diferentes maneiras de fazer isso. Vamos apresentar a primeira solução, através do uso da função `tapply` e `arima`:

```
my.l <- tapply(X = my.df.stocks$ret,
               INDEX = my.df.stocks$ticker,
               FUN = arima,
               order = c(1,0,0))
```

Cada modelo está disponível na lista `my.l`. Para resgatar todos os coeficientes, usamos funções `sapply` e `coef`:

```
print(sapply(X = my.l, FUN = coef))
```

******ebook converter DEMO Watermarks*******

```
##                    CYRE3         ENBR3        GGBR4
## ar1       -6.215237e-03 -0.1561646111 2.098670e-02
## intercept  9.311904e-05  0.0005830582 8.714789e-05
##                    LREN3
## ar1       -0.029417004
## intercept  0.001459845
```

Uma limitação no uso de `tapply` é que estamos restritos ao acesso a uma única coluna de `my.df`. Observe como a entrada `X` de `tapply` aceita apenas um vetor. Para usar mais colunas do `dataframe` em uma operação de grupo, devemos usar função `by`. Esta função irá quebrar um `dataframe` em vários menores com base em um fator.

Para um exemplo de estimativa de vários modelos com função `by`, vamos calcular o coeficiente beta para todas as ações da nossa base de dados. Lembre-se que, para a estimação do modelo de mercado, precisamos de dois vetores, um com os retornos do ativo e outro com os retornos do índice. Primeiro, vamos carregar os dados do Ibovespa e adicionar uma nova coluna em `my.df` com os retornos do índice.

```
# load data
df.ibov <- read_csv(file = 'data/Ibov.csv',
                    col_types = cols(ref.date = col_date(format = ""),
                                     price.close = col_integer()))


# calculate return
df.ibov$ret <- calc.ret(df.ibov$price.close)

# find location of dates
idx <- match(my.df$ref.date, df.ibov$ref.date)

# create column in my.df
my.df$ret.ibov <- df.ibov$ret[idx]
```

O próximo passo é criar uma função que irá tomar um dataframe como entrada e usar as colunas de retornos do ativo e do índice para calcular o beta:

```
estimate.beta <- function(df) {
  # Function to estimate beta from dataframe of stocks returns
  #
  # Args:
  #   df - Dataframe with columns ret and ret.ibov
  #
  # Returns:
  #   The value of beta

  my.model <- lm(data = df, formula = ret ~ ret.ibov)

  return(coef(my.model)[2])
}
```

Agora vamos utilizar `estimate.beta` com `by`:

```
# calculate beta for each stock
my.betas <- by(data = my.df,
               INDICES = my.df$ticker,
```

******ebook converter DEMO Watermarks*******

```
                FUN = estimate.beta)

glimpse(as.numeric(my.betas))

##  num [1:53] 0.458 1.271 1.019 1.091 1.169 ...
```

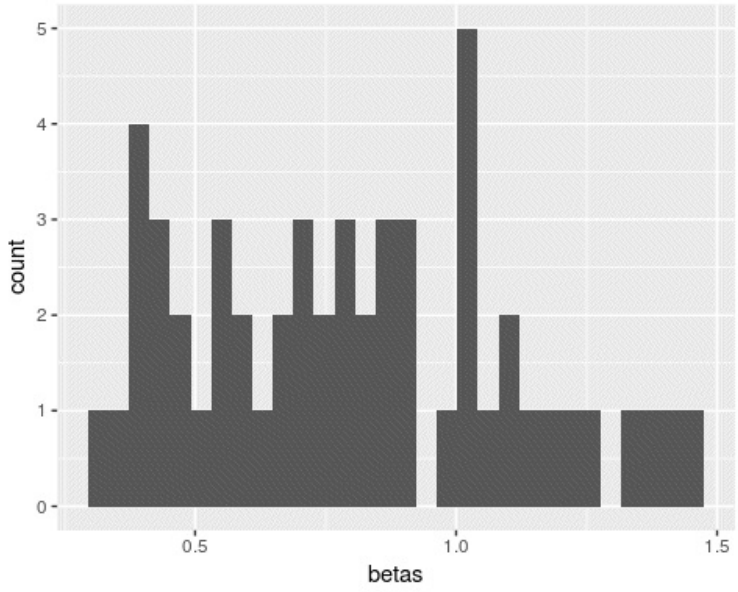
Os valores dos betas estão disponíveis no objeto `my.betas`. Vamos observar a distribuição de seus valores:

```
library(ggplot2)

df.to.plot <- data.frame(betas = as.numeric(my.betas))

p <- ggplot(df.to.plot, aes(x=betas)) +
  geom_histogram()

print(p)
```



Como podemos ver, os *betas* são positivos na maioria dos casos. Este resultado não surpreende. As ações fazem parte do índice e, portanto, uma correlação positiva entre seus retornos é esperada.

Outra maneira de armazenar e gerenciar vários modelos é usar os recursos de colunas do tipo lista do pacote `dplyr`. Veja o exemplo a seguir, onde replicamos o procedimento anterior com as funções do `dplyr`:

```
library(dplyr)

my.tab <- my.df %>%
  group_by(ticker) %>%
  do(my.model = arima(x = .$ret,
                      order = c(1,0,0)))

glimpse(my.tab)
```

******ebook converter DEMO Watermarks*******

```
## Observations: 53
## Variables: 2
## $ ticker   <chr> "ABEV3", "BBAS3", "BBDC3", "BBDC4", "B...
## $ my.model <list> [<-0.034508708, 0.001154925, 0.000221...
```

Temos uma coluna lista chamada `my.model`. Essa armazena cada saída da função de estimação. Indo além, podemos usar `mutate` para construir outras colunas com informações sobre os modelos estimados. Veja a seguir:

```
my.model.tab <- my.df %>%
  group_by(ticker) %>%
  do(my.model = arima(x = .$ret, order = c(1,0,0))) %>%
  mutate(alpha = coef(my.model)[2],
         ar1 = coef(my.model)[1])

print(head(my.model.tab))
```

```
## # A tibble: 6 x 4
##   ticker my.model       alpha     ar1
##   <chr>  <list>         <dbl>   <dbl>
## 1 ABEV3  <S3: Arima> 0.00115  -0.0345
## 2 BBAS3  <S3: Arima> 0.000638  0.0513
## 3 BBDC3  <S3: Arima> 0.000680 -0.0257
## 4 BBDC4  <S3: Arima> 0.000568 -0.0387
## 5 BRAP4  <S3: Arima> 0.000431  0.0440
## 6 BRFS3  <S3: Arima> 0.000459 -0.0117
```

Um truque muito interessante para manejar diversos modelos com `dplyr` é usar pacote `broom` (Robinson 2017) para acessar e organizar as informações de saída. No código anterior, adicionamos duas colunas em `my.tab`, com os valores de *alpha* and *ar1* para cada modelo. Uma versão mais simples e direta é reportar todos os coeficientes com função `broom::tidy`:

```
library(broom)

# get coefs with tidy
my.coef.tab <- my.model.tab %>%
  tidy(my.model)

# print result
print(head(my.coef.tab))
```

```
## # A tibble: 6 x 6
## # Groups:   ticker, alpha, ar1 [3]
##   ticker    alpha     ar1 term       estimate std.error
##   <chr>     <dbl>   <dbl> <fct>         <dbl>     <dbl>
## 1 ABEV3  0.00115  -0.0345 ar1        -0.0345    0.0224
## 2 ABEV3  0.00115  -0.0345 intercept   0.00115   0.000323
## 3 BBAS3  0.000638  0.0513 ar1         0.0513    0.0223
## 4 BBAS3  0.000638  0.0513 intercept   0.000638  0.000595
## 5 BBDC3  0.000680 -0.0257 ar1        -0.0257    0.0224
## 6 BBDC3  0.000680 -0.0257 intercept   0.000680  0.000421
```

Note que função `broom::tidy` incluiu os erros padrões dos coeficientes. Caso tivéssemos mais coeficientes, esses seriam reportados em `my.coef.tab`. Para informações mais gerais do modelo, podemos usar função `broom::glance`:

******ebook converter DEMO Watermarks*******

```r
# get info on models
my.info.models <- my.model.tab %>%
  glance(my.model)

print(head(my.info.models))
```

```
## # A tibble: 6 x 7
## # Groups:   ticker, alpha, ar1 [6]
##   ticker    alpha     ar1  sigma logLik     AIC     BIC
##   <chr>     <dbl>   <dbl>  <dbl>  <dbl>   <dbl>   <dbl>
## 1 ABEV3  0.00115  -0.0345 0.0149  5570. -11134. -11117.
## 2 BBAS3  0.000638  0.0513 0.0252  4518.  -9029.  -9012.
## 3 BBDC3  0.000680 -0.0257 0.0193  5056. -10106. -10089.
## 4 BBDC4  0.000568 -0.0387 0.0203  4955.  -9904.  -9887.
## 5 BRAP4  0.000431  0.0440 0.0265  4416.  -8827.  -8810.
## 6 BRFS3  0.000459 -0.0117 0.0174  5260. -10513. -10496.
```

Essa inclui informações sobre coeficiente e estatísticas tal como valor de *log-likelihood*, AIC e BIC.

******ebook converter DEMO Watermarks*******

11.8 Exercícios

1. Simule o seguinte processo linear no R:

```
set.seed(5)

# number of obs
nT <- 100

# set x as Normal (0, 1)
x <- rnorm(nT)

# set coefficients
my.alpha <- 1.5
my.beta <- 0.5

# build y
y <- my.alpha + my.beta*x + rnorm(nT)
```

A partir de `x` e `y`, estime um modelo linear onde `x` é a variável explicativa e `y` é a variável explicada. Use função `summary` no objeto de retorno da estimação para obter mais detalhes sobre o modelo. O coeficiente beta é significativo a 5%?

2. Utilizando pacote `car`, teste a hipótese conjunta de que o valor de alpha é igual a 1.5 e beta igual a 0.5. A hipótese nula do teste pode ser rejeitada a 5%?

3. Utilize pacote `gvlma` para testar as premissas do OLS para o modelo estimado anteriormente. O modelo passa em todos os testes? Em caso negativo, aumente o valor de `nT` para 1000 e tente novamente. O aumento do número de observações do modelo impactou no teste das premissas? De que forma?

4. **DESAFIO** - Usando suas habilidades de programação, realize um estudo para verificar como o valor de `nT` impacta no teste da hipótese conjunta de que os coeficientes são iguais aos valores definidos. Sugestão: para cada valor de `nT` testado, simule 1000 séries e estime o mesmo modelo para cada uma delas. Verifique, para cada `nT`, quantas vezes a hipótese nula é rejeitada a 5%. Um gráfico de dispersão mostrando a relação entre `nT` e o número de rejeições seria interessante para a análise.

5. Utilize função `BatchGetSymbols::GetSP500Stocks` para baixar dados de todas ações pertencentes ao atual índice SP500. Usando o SP500 como o índice de mercado, calcule o beta para cada uma das ações. Apresente o histograma dos *betas* estimados. Note que os retornos do SP500 (`'^GSPC'`) não estão disponíveis na base de dados original e devem ser baixados e agregados a base de dados original.

6. Para os dados importados anteriormente, estime uma versão em dados de painel para o modelo de mercado (beta). Nesta versão, cada ação possui um intercepto diferente, porém compartilham o mesmo beta. O beta estimado é significativo a 5%?

7. Utilizando as funções do tidyverse, `dplyr::group_by` e `dplyr::do`, estime um modelo

******ebook converter DEMO Watermarks*******

ARIMA para os retornos de cada ação dos dados importados no exercício 5. No mesmo dataframe de saída, crie uma nova coluna com a previsão em *t+1* de cada modelo. Qual ação possui maior expectativa de retorno?

8. **DESAFIO** - Separe os dados do SP500 em duas partes, etapa de estimação e etapa de previsão. Você pode, por exemplo, comprar a ação quando a previsão de retorno for positiva, vendendo-a no dia seguinte. As previsões dos modelos ARIMA permitem a construção de uma estratégia de negociação lucrativa?

9. No mesmo código utilizado na questão anterior, adicione uma nova coluna-lista com a estimação de um modelo ARMA(1,0)-GARCH(1,1) para os retornos de cada ação. Adicione outra coluna com a previsão de volatilidade (desvio padrão) em *t+1*. Ao dividir o retorno esperado calculado no item anterior pelo risco previsto, temos um índice de direção do mercado, onde aquelas ações com maior valor de índice apresentam maior retorno esperado por menor risco. Qual ação é mais atrativa e possui maior valor deste índice?

10. Para a mesma base de dados do SP500, selecione 4 ações aleatoriamente e estime um modelo de mudança de regime markoviano equivalente ao apresentado no item 11.6 para cada ação. Utilize função `plot` para apresentar o gráfico das probabilidades suavizadas e salve cada figura em uma pasta chamada `'fig'`.

11. **DESAFIO** - Utilize a mesma estrutura de pesquisa da questão 8 para verificar se o uso de modelos de mudança de regime melhoram ou pioram os resultados de uma estratégia de negociação.

******ebook converter DEMO Watermarks*******

12 Reportando Resultados e Criando Relatórios

Nos capítulos anteriores aprendemos a utilizar o R para criar figuras e estimar modelos. Agora é o momento de estudarmos a forma de reportar os resultados do nosso estudo, o processo de criação e exportação de tabelas e figuras. Essa é a etapa final do ciclo computacional da pesquisa. Visando eficiência, devemos levar em consideração dois pontos, qualidade dos itens exportados e facilidade para alterações.

A qualidade refere-se a atratividade visual no conjunto de informações exportadas. Por exemplo, uma figura pode ser exportada com alta resolução. Uma tabela com testes estatísticos pode ser criada com uma estrutura particular, destacando os resultados mais significativos. Na prática de pesquisa, é muito comum em tabelas de estimação de modelos que os coeficientes estatisticamente significativos sejam destacados com asteriscos ou então em negrito. Isso facilita a análise para o leitor, cuja atenção será focada para os resultados mais importantes.

A facilidade de alteração de tabelas relaciona-se a maneira como editamos e finalizamos os itens exportados. Uma tabela pode ser manualmente editada para atingir alta qualidade. Podemos copiar as informações para uma planilha eletrônica e modificar a tabela até atingir o formato desejado. Porém, toda vez que modificarmos a tabela original, seja por quaisquer motivos, teremos que re-copiar e re-formatar a mesma na planilha eletrônica. Isso adiciona um custo de trabalho muito alto pois é muito provável que a pesquisa evolva ao longo do tempo, exigindo diversas modificações.

Caso o trabalho de montagem de tabelas for demasiado, uma grande quantidade de horas será exigida e desperdiçada na formatação manual das informações. O ideal, portanto, é que o processo da pesquisa seja integrado a criação das tabelas, de forma que uma mudança nos dados não implique em trabalho extra na exportação das informações. O tempo economizado pode ser alocado em atividades mais produtivas. Felizmente, o R possui pacotes que facilitam e automatizam este processo. Na maioria dos casos e principalmente para quem escreve no Latex, o processo de exportação de figuras e tabelas é fluido e não necessita trabalho adicional.

Existem duas estratégias para reportar resultados de uma pesquisa. A primeira, e mais comum, é separar a etapa de produção de tabelas e figuras da etapa de escrita. Nesse caso, figuras e tabelas são exportadas do R como arquivos e podem ser copiadas diretamente pro texto em programas como *Word/Writer/LaTeX*. Vale destacar que o uso de LaTeX facilita muito esse processo pois cada tabela vira um arquivo que pode ser importado via código no próprio arquivo principal. A segunda maneira de escrever artigos no R é utilizar a capacidade de criação de relatórios dinâmicos com **Rmarkdown**. Essa é uma maneira mais dinâmica, onde texto e código mesclam-se no mesmo arquivo. Vamos iniciar explorando a primeira estratégia. Vale destacar que o material das próximas seções vale também para o *Rmarkdown*. A única diferença é que elimina-se o processo intermediário de utilizar arquivos para exportar as tabelas.

******ebook converter DEMO Watermarks*******

******ebook converter DEMO Watermarks*******

12.1 Reportando Tabelas

Tabelas simples, tal como estatísticas descritivas, devem ser criadas a partir de um dataframe. Vamos começar com um exemplo para a LaTeX. No próximo código, importaremos dados para 4 ações usando `BatchGetSymbols` e, a partir destes, uma tabela descritiva sobre os retornos dos diferentes ativos será exportada para um arquivo LaTeX usando o pacote `xtable`.

```
library(dplyr)
library(BatchGetSymbols)
library(DistributionUtils) # Kurtosis() and Assimetry()

# set number of rows in table
my.tickers <- c('PETR4.SA', 'GGBR4.SA',
                'USIM5.SA', 'MDIA3.SA')

first.date = '2015-01-01'
last.date = '2018-01-01'

df.stocks <- BatchGetSymbols(tickers = my.tickers,
                             first.date = first.date,
                             last.date = last.date,
                             bench.ticker = '^BVSP')[[2]]

##
## Running BatchGetSymbols for:
##    tickers = PETR4.SA, GGBR4.SA, USIM5.SA, MDIA3.SA
##    Downloading data for benchmark ticker | Found cache file
## PETR4.SA | yahoo (1|4) | Found cache file - Feels good!
## GGBR4.SA | yahoo (2|4) | Found cache file - Feels good!
## USIM5.SA | yahoo (3|4) | Found cache file - OK!
## MDIA3.SA | yahoo (4|4) | Found cache file - Got it!

# create descriptive table
my.desc.table <- df.stocks %>%
  group_by(Ticker = ticker ) %>%
  summarise('Mean Ret' = mean(ret.adjusted.prices, na.rm = TRUE),
            'StDev Ret' = sd(ret.adjusted.prices, na.rm = TRUE),
            'Max Ret' = max(ret.adjusted.prices, na.rm = TRUE),
            'Min Ret' = min(ret.adjusted.prices, na.rm = TRUE),
            'Assimetry' = skewness(ret.adjusted.prices, na.rm = TRUE),
            'Kurtosis' = kurtosis(ret.adjusted.prices, na.rm = TRUE))

print(my.desc.table)

## # A tibble: 4 x 7
##   Ticker   `Mean Ret` `StDev Ret` `Max Ret` `Min Ret`
##   <chr>         <dbl>       <dbl>     <dbl>     <dbl>
## 1 GGBR4.SA    0.00103      0.0341    0.161     -0.120
## 2 MDIA3.SA    0.00109      0.0199    0.0937    -0.113
## 3 PETR4.SA    0.00126      0.0343    0.163     -0.158
## 4 USIM5.SA    0.00189      0.0458    0.351     -0.161
## # ... with 2 more variables: Assimetry <dbl>,
## #   Kurtosis <dbl>
```

Na criação do dataframe, observe como definimos os nomes das colunas como texto (aspas). Isso

******ebook converter DEMO Watermarks*******

é necessário pois utilizamos espaços nos nomes. Agora que temos nossa tabela, vamos exportá-la para a LaTeX.

```
library(xtable)

# set xtable object
my.xtable <- xtable(x = my.desc.table,
                    label = 'tab:DescRetStats',
                    caption = 'Descriptive Statistics for Returns',
                    digits = 4)

# check if folder exists
if (!dir.exists('tabs')) {
    dir.create('tabs')
}

# print output to latex file
my.f.tex <- 'tabs/MyTable.tex'

# save it
print(my.xtable,
      include.rownames = FALSE,
      file = my.f.tex,
      type='latex')
```

Na função `xtable`, usamos apenas entradas `label` e `caption` para editar os detalhes da tabela. Usuários do LaTeX sabem que existem diversas outras opções. Essas estão disponíveis como outras entradas na função `xtable`. O arquivo LaTeX resultante é salvo na pasta `tabs` do diretório atual. O código verifica a existência desse diretório e, caso não existir, cria o mesmo. O resultado em um arquivo de LaTeX compilado será idêntico a 12.1, uma tabela pronta para impressão em um artigo ou relatório científico.

Ticker	Mean Ret	StDev Ret	Max Ret	Min Ret	Assimetry	Kurtosis
GGBR4.SA	0.0010	0.0341	0.1609	-0.1202	0.2832	1.3923
MDIA3.SA	0.0011	0.0199	0.0937	-0.1126	-0.1057	3.1040
PETR4.SA	0.0013	0.0343	0.1628	-0.1576	0.2165	2.0972
USIM5.SA	0.0019	0.0458	0.3511	-0.1614	0.9106	6.3831

Table 1: Descriptive Statistics for Returns

Figura 12.1 Exemplo de tabela com xtable

Quanto à exportação de tabelas para arquivos do *Word* (*Microsoft*) ou *Writer* (*Libreoffice*), não existe maneira direta de fazer isso usando `xtable`. Mas, uma solução simples é usar o pacote para exportar a tabela para um arquivo com extensão *html* ou *doc* temporário e, em seguida, copiar e colar o resultado no relatório final. Essa operação funciona pois arquivos com extensão *html* e *doc* compartilham um formato similar para tabelas. Como exemplo, vamos realizar esta operação:

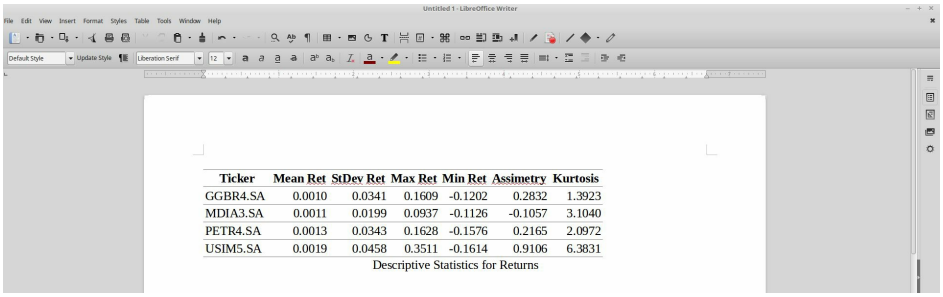
```
# set html file for output
my.f.html <- 'tabs/MyTable.html'

# write it!
```

******ebook converter DEMO Watermarks*******

```
print(x = my.xtable,
      file = my.f.html,
      type = 'html',
      include.rownames = FALSE )
```

Uma vez que o arquivo está disponível, podemos abrir tabs/MyTable.html com qualquer navegador de Internet, selecionar e copiar a tabela e, finalmente, colá-lo no nosso documento. O resultado deve ser semelhante ao apresentado na Figura 12.2.



Ticker	Mean Ret	StDev Ret	Max Ret	Min Ret	Assimetry	Kurtosis
GGBR4.SA	0.0010	0.0341	0.1609	-0.1202	0.2832	1.3923
MDIA3.SA	0.0011	0.0199	0.0937	-0.1126	-0.1057	3.1040
PETR4.SA	0.0013	0.0343	0.1628	-0.1576	0.2165	2.0972
USIM5.SA	0.0019	0.0458	0.3511	-0.1614	0.9106	6.3831

Descriptive Statistics for Returns

Figura 12.2 Exemplo de tabela no Writer (LibreOffice)

Indo além, se você lida com muitas figuras e tabelas em arquivos *.docx* (Word/Writer) ou *.ppt* (Powerpoint/Impress), o pacote `ReporteRs` (Gohel 2017) pode poupar muito tempo e esforço. Esse propões uma forma muito interessante de automatizar o processo de criação de relatórios no formato Microsoft Office. Ao invés de realizar uma exportação explícita via arquivos, podes definir *placeholders* no texto do arquivo *.docx* para inserir as tabelas e figuras direto do código. Por exemplo, podes escrever o texto `'FIGURA_1_AQUI'` no arquivo *Word* e, pelo R, substituir o texto por uma figura qualquer. Relatar mais sobre o uso desse pacote, porém, vai além dos nossos objetivos. Leitores interessados podem devem consultar o manual do autor.

******ebook converter DEMO Watermarks*******

12.2 Reportando Modelos

Um tipo especial de tabela refere-se a estimação de um ou mais modelos. Essas possuem uma estrutura própria com a apresentação de diferentes medidas de performance, comparação de coeficientes para modelos diferentes, além dos parâmetros individuais e suas estatísticas. Felizmente para usuários do R, existem pacotes que facilitam a construção e customização de tabelas de estimação de um ou mais modelos, tornando a vida do pesquisador mais fácil. Os mais populares são `xtable`, `texreg` e `stargazer`.

Como exemplo, vamos usar o pacote `texreg` para reportar os resultados do cálculo do *beta* de quatro ações. Carregamos os dados, estimamos os modelos e, depois disso, utilizamos função `screenreg` para exibir uma representação dos modelos em formato de texto.

```
library(texreg)
library(dplyr)

# get Ibov data
my.tickers <- c('^BVSP')

first.date = '2015-01-01'
last.date = '2018-01-01'

df.ibov <- BatchGetSymbols(tickers = my.tickers,
                           first.date = first.date,
                           last.date = last.date,
                           bench.ticker = '^BVSP')[[2]]

##
## Running BatchGetSymbols for:
##    tickers = ^BVSP
##    Downloading data for benchmark ticker | Found cache file
## ^BVSP | yahoo (1|1) | Found cache file - You got it!

# set ibov ret column
idx <- match(df.stocks$ref.date, df.ibov$ref.date)
df.stocks$ret.ibov <- df.ibov$ret.adjusted.prices[idx]

# estimate betas
beta.tab <- df.stocks %>%
  group_by(ticker) %>%
  do(beta.model = lm(data=., ret.adjusted.prices ~ ret.ibov))

# report result
est.table <- screenreg(l = beta.tab$beta.model,
                       custom.model.names = beta.tab$ticker,
                       custom.coef.names = c('Alpha', 'Beta'),
                       digits = 2)

# print it
print(est.table)

##
## ========================================================
##             GGBR4.SA    MDIA3.SA    PETR4.SA    USIM5.SA
## --------------------------------------------------------
```

******ebook converter DEMO Watermarks*******

```
## Alpha         0.00         0.00        -0.00         0.00
##              (0.00)       (0.00)      (0.00)       (0.00)
## Beta          1.41 ***     0.55 ***    1.87 ***     1.64 ***
##              (0.07)       (0.05)      (0.05)       (0.10)
## ------------------------------------------------------------
## R^2           0.36         0.16         0.64         0.27
## Adj. R^2      0.36         0.16         0.64         0.27
## Num. obs.   736          736          701          736
## RMSE          0.03         0.02         0.02         0.04
## ============================================================
## *** p < 0.001, ** p < 0.01, * p < 0.05
```

No código anterior, coluna `beta.model` de `beta.tab` contém todos os modelos estimados em um objeto do tipo lista. O argumento `custom.model.names` em `screenreg` define o nome dos modelos, enquanto `custom.coef.names` permite o uso de nomes customizados para os coeficientes. Para o nosso caso, podemos ver claramente que os *betas* das ações são diferentes e com significância estatística, isto é, a probabilidade deles serem iguais a zero é muito baixa.

O pacote `texreg` também oferece diversas outras opções para o usuário. Você pode personalizar sua tabela de estimação de várias maneiras. A exportação para outros formatos, como LaTeX e *html*, é possível e recomendada. Para usuários do Microsoft Office, a função `htmlreg` também permite exportar uma tabela para um arquivo do Word. Você pode fazer isso configurando entrada `file` como um arquivo *.doc*. Se você trabalha com tabelas de estimativas no dia-a-dia, o pacote `texreg` economizará muito tempo.

A seguir apresentamos um exemplo de uso de `texreg`:

```
# report result
est.table <- texreg(l = beta.tab$beta.model,
                    file = 'tabs/Example_texreg.tex',
                    custom.model.names = beta.tab$ticker,
                    custom.coef.names = c('Alpha', 'Beta'),
                    digits = 2)
```

O resultado em um arquivo LaTex compilado para pdf será equivalente a figura 12.3.

	GGBR4.SA	MDIA3.SA	PETR4.SA	USIM5.SA
Alpha	0.00	0.00	−0.00	0.00
	(0.00)	(0.00)	(0.00)	(0.00)
Beta	1.41***	0.55***	1.87***	1.64***
	(0.07)	(0.05)	(0.05)	(0.10)
R^2	0.36	0.16	0.64	0.27
Adj. R^2	0.36	0.16	0.64	0.27
Num. obs.	736	736	701	736
RMSE	0.03	0.02	0.02	0.04

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

Table 3: Statistical models

******ebook converter DEMO Watermarks*******

Figura 12.3 Exemplo de tabela com xtable

******ebook converter DEMO Watermarks*******

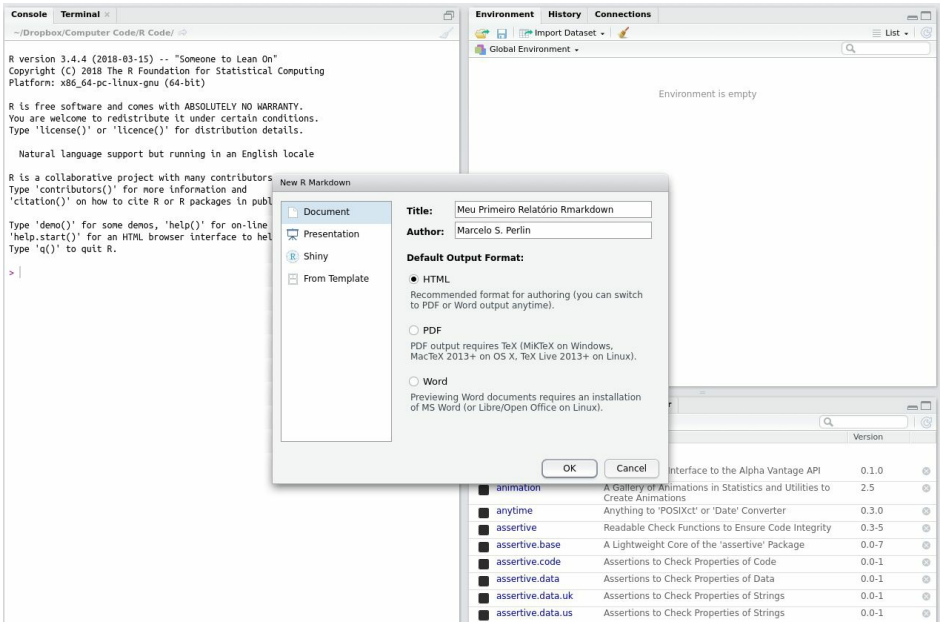
12.3 Criando Relatórios com o *RMarkdown*

O *RMarkdown* inova o processo de escrita de relatórios ao mesclar texto com código. O grande benefício é que todo o processo de produção do texto técnico fica contido em um único arquivo com extensão *.Rmd*. Já na estratégia mais comum, destacada anteriormente, separamos as etapas (e nossas pastas do computador) entre código e escrita.

A estrutura de texto do *Rmarkdown* baseia-se no *markdown*, uma linguagem voltada a escrita e formatação de texto na Internet. O grande diferencial em relação a programas como *Word* e *Writer* é que toda formatação do texto é realizada via código. Por exemplo, um texto em negrito em *markdown* é escrito como `**texto negrito**` e itálico como `_texto itálico_`. Um programa compilador lê o código original e o tranforma em texto formatado. Usuários de LaTeX irão se sentir muito confortável com o *markdown*, pois a forma de trabalho é bastante semelhante.

Vamos explicar o funcionamento do `Rmarkdown` com um exemplo. Abra o RStudio e clique em *New Files* e depois em *New R Markdown*. Note que uma caixa de diálogo aparece, com as opções para *Document*, *Presentation*, *Shiny* e *From Template*. O primeiro caso, *Document* refere-se a escrita de documentos de texto corrido, isto é, artigos e relatórios técnicos. A segunda opção, *Presentation*, é relativa a criação de *slides* de apresentação, comparável a arquivos do *Powerpoint/Impress* e LaTeX (*beamer*). A terceira opção, `Shiny`, cria aplicativos para a internet, usando a tecnologia de mesmo nome. Não cabe aqui explicar as funcionalidades de todos os casos. Iremos, portanto, dar preferência a opção *Document*, a qual temos maior interesse.

Indo em frente, vamos editar o título do relatório como `'Meu Primeiro Relatório RMarkdown'` e o nome do autor. O resultado de nossas escolhas deve ser semelhante a tela destacada na Figura 12.4.



******ebook converter DEMO Watermarks*******

Figura 12.4 Criando um relatório Rmarkdown

As próximas escolhas referem-se ao tipo de arquivo de saída na compilação do arquivo contendo o relatório: HTML, PDF ou Word. O conteúdo do documento será todo escrito em *RMarkdown* e depois compilado para um dos formatos destacados anteriormente. O resultado será um arquivo com uma das três extensões. Qual o melhor formato? Depende muito de quem irá ler o documento. Eu, particularmente, gosto muito do formato *html* devido a facilidade de distribuição. Hoje em dia, qualquer computador ou celular/tablet tem um navegador de internet pré-instalado. Os formatos PDF e Word, porém, são mais populares. De qualquer maneira, não se preocupe muito com isso. Podes modificar o formato de saída futuramente. Vale salientar que a produção em pdf exige a instalação do compilador do LaTex. No *Windows*, esse software é o miktex.

Após dar o *OK*, aparecerá um arquivo na tela de edição de *scripts*. Vamos dissecar o mesmo com figuras, explicando o funcionamento de cada elemento. A primeira parte é o cabeçalho do arquivo, Figura 12.5.

```
1 ---
2 title: "Meu Primeiro Relatório Rmarkdown"
3 author: "Marcelo S. Perlin"
4 date: "March 28, 2018"
5 output: html_document
6 ---
```

Figura 12.5 Cabeçalho de arquivo Rmarkdown

O cabeçalho do documento é identificado pelo uso do separador `---`. Cada item é pré-definido e reconhecido pelo `Rmarkdown`, incluindo o título (`title:`), autor (`author:`), data do documento (`date:`) e o tipo de saída (`output:`). Existem várias outras opções de customização. Podes configurar tamanho da página, estilo do texto de saída, tamanho das figuras, entre outras coisas. Interessados em customizar o arquivo de saída devem consultar o manual.

Avançando, o próximo pedaço é identificado na Figura 12.6. Esse é o código de configuração do relatório. O termo *setup* é o identificador do pedaço de código, o qual pode ser usado futuramente ou não. No *RMarkdown*, todo código começa e termina com triplo acento inverso. O item `r` indica a linguagem usada para rodar o código, nesse caso o próprio R. Caso fosse de interesse, poderíamos executar códigos de outra linguagem, tal como *Python*, *SQL* ou *Bash*.

O item `include=FALSE` é uma opção particular a aquele pedaço de código. Particularmente, o comando `include=FALSE` diz para o RMarkdown não incluir o pedaço de código na saída do documento, porém rodar o código contido nele. Existem diversas outras opções para cada pedaço de código. O manual do *knitr*, disponível na internet, apresenta todas as possibilidades. Por enquanto, sugiro conhecer apenas as opções mais simples. Nesse caso, além da entrada `include`, deves conhecer a opção `eval = TRUE|FALSE`, a qual define se o código em questão será avaliado (executado) ou não e `echo = TRUE|FALSE`, que permite esconder ou mostrar o código no texto final.

******ebook converter DEMO Watermarks*******

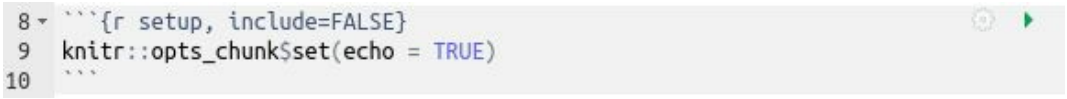


Figura 12.6 Código em Rmarkdown

Para o primeiro pedaço de código, a linha `knitr::opts_chunk$set(echo = TRUE)` define uma opção global para todos os pedaços de códigos futuros. Nesse caso, `echo=TRUE`, ou seja, todo o código será apresentado no documento final. Essa opção será *default* e respeitada caso o pedaço de código não tenha a opção `echo=FALSE` explicitamente em sua configuração.

Um ponto importante aqui é que, no momento de compilação do arquivo *Rmarkdown*, o diretório de trabalho sempre será o local do arquivo de origem. Portanto, ao contrário do desenvolvimento de *scripts*, não é necessário trocar de diretório no código com a função `setwd`.

Seguindo em frente, o texto `## R Markdown` na Figura 12.7 define um título, nesse caso no segundo nível de profundidade. No *Word/Writer*, esse formato seria denominado *Título 2*. A profundidade do título define, no documento final, o tamanho e fonte das letras do título, e o sumário da obra. Caso necessário, um título no primeiro nível poderia ser definido como `# Titulo 1` e um título no terceiro nível como `### Título 3`. Note que a quantidade de *hashtags* (#) define o nível de profundidade do título.



Figura 12.7 Texto em Rmarkdown

A segunda parte em 12.7 é o texto em si. Nesse, pode-se escrever livremente, atentando apenas a estrutura de formatação utilizada. Por exemplo, texto em negrito é escrito como `**texto negrito**`, itálico como `*texto itálico*` ou `_texto itálico_`, links como `(texto link)[url]`. Como para os casos anteriores, existem diversos outras opções. Refiro novamente ao manual para maiores detalhes.

Um ponto importante aqui é o uso de código em texto corrido, o qual não faz parte do documento *template* do *RMarkdown*. Para explicar, imagine uma situação em que queres reportar o valor de um teste estatístico no texto, tal como na frase *"O valor do teste T foi X"*, onde *X* é um resultado numérico obtido em código R. Assumindo que o resultado de interesse está disponível em um objeto chamado `result.ttest`, podemos escrever o texto em *Rmarkdown* como *"O valor do teste T foi @`r result.ttest`@"*, onde o símbolo @ deve ser substituído pelo acento inverso. Veja um exemplo na Figura 12.8

******ebook converter DEMO Watermarks*******

```
```{r}
x <- runif(100)
y <- runif(100)

my.lm <- lm(formula = y ~ x)
result.ttest <- summary(my.lm)$coefficients[1, 3]
```

O valor do teste T foi `r my.ttest` ...
```

Figura 12.8 Exemplo de código em texto corrido

O R irá avaliar o valor de `result.ttest` no momento da compilação do arquivo e a frase estará com o valor correto, obtido via código. Note aqui a grande vantagem do *RMarkdown*, texto e resultados são integrados dinamicamente. Caso a base de dados fosse trocada, não seria necessário reescrever a frase uma vez que ela não perderia o sentido, apenas trocaria o valor de `result.ttest`.

A última parte em 12.7 apresenta o código `summary(cars)`, isto é, apresenta o sumário de um objeto já existente na memória do R. Esse código será mostrado no texto (veja uso de `echo=TRUE` em `knitr::opts_chunk`) e apresentará o resultado de sua execução no texto compilado, também. A apresentação de resultados não é relativa somente para o comando `print`. Podes utilizar, também, comando `cat` para apresentar textos de saída no documento final.

Na Figura 12.9 apresenta-se a última seção do arquivo *Rmarkdown* de exemplo. Essa cria e apresenta uma figura no relatório final. Para isso, basta chamar função `plot` ou então funções do `ggplot2` normalmente, conforme aprendido no capítulo 10. No caso de criação de mais de um gráfico, esses serão apresentados um embaixo do outro no documento. Uma observação importante, o uso de `x11()` para criar janelas (veja capítulo 10) não deve ser feito em um arquivo *Rmarkdown*. Isso ocasionará um erro na compilação do relatório.

```
## Including Plots

You can also embed plots, for example:

```{r pressure, echo=FALSE}
plot(pressure)
```

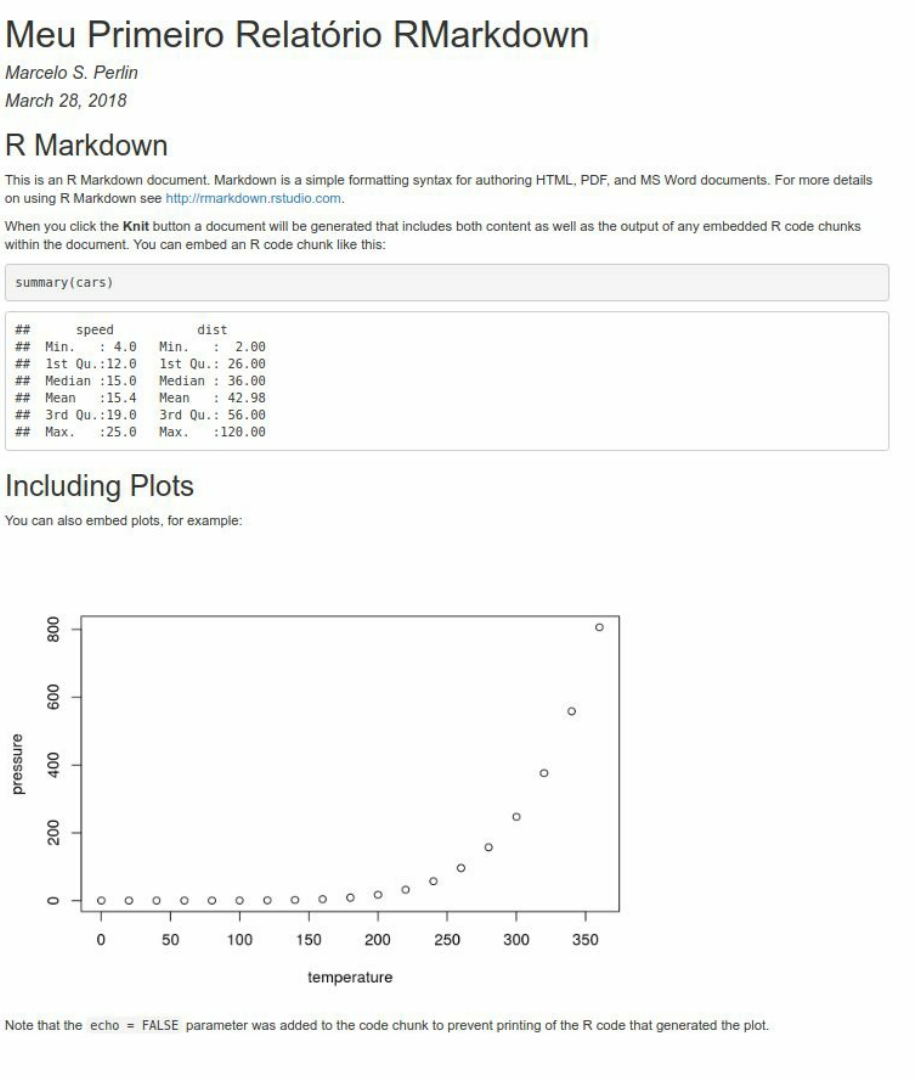
Note that the `echo = FALSE` parameter was added to the code chunk to prevent
printing of the R code that generated the plot.
```

Figura 12.9 Texto com figuras no Rmarkdown

Agora que já entendemos os componentes que formam o nosso relatório *Rmarkdown*, é hora de compilar e analisar o resultado. Primeiro salve o arquivo *Rmd* em uma pasta local com um nome qualquer. Após isso, clique no botão *Knit* (topo do editor) e selecione *Knit to html*. No canto inferior esquerdo do RStudio aparecerá as etapas do processo. Por fim, uma nova janela com o documento já compilado irá surgir. Podes clicar em *Open in Browser* para abrir o arquivo no seu navegador de internet padrão. Este arquivo compilado deve ter o mesmo nome e pasta do arquivo original *Rmd*, mas com extensão *html*. O resultado deve ser idêntico ao apresentado na Figura 12.10. Como exercício, sugiro compilar o arquivo para outros formatos, *pdf* e *docx*, e verificar o

******ebook converter DEMO Watermarks*******

resultado.



Meu Primeiro Relatório RMarkdown

Marcelo S. Perlin

March 28, 2018

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
summary(cars)
```

```
##      speed           dist
##  Min.   : 4.0   Min.   :  2.00
##  1st Qu.:12.0   1st Qu.: 26.00
##  Median :15.0   Median : 36.00
##  Mean   :15.4   Mean   : 42.98
##  3rd Qu.:19.0   3rd Qu.: 56.00
##  Max.   :25.0   Max.   :120.00
```

Including Plots

You can also embed plots, for example:

Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.

Figura 12.10 Exemplo de relatório compilado em html

Vale destacar que os procedimentos de exportação de tabelas e modelos destacados na seção anterior com o uso dos pacotes `xtable` e `texreg`, também são válidos. Portanto, tens a disposição todas as ferramentas necessárias para a criação de artigos científicos no *RMarkdown*, integrando código com texto.

Essa seção mostrou uma pequena porção do universo de possibilidades no uso do *Rmarkdown*. Leitores com interesse em aprender mais, sugiro a leitura completa do manual destacado na sua página. Essa oferece, também, uma galeria de casos de uso do Rmarkdown. Alternativamente, uma coleção de *templates* é oferecida no GitHub.

******ebook converter DEMO Watermarks*******

******ebook converter DEMO Watermarks*******

12.4 Exercícios

1. Observe os dados disponíveis no arquivo `grunfeld.csv`. Importe os dados no R e monte uma tabela descritiva das variáveis. Essa tabela deve oferecer informações suficientes para o leitor entender os dados. Utilize pacote `xtable` para reportar a mesma no formato LaTeX ou Word/Writer.

2. Utilizando função `BatchGetSymbols::GetIbovStocks`, selecione 4 ações aleatoriamente e baixe dados da mesma utilizando as funções do mesmo pacote. Estime um modelo ARIMA(1,0,1) para cada ação e reporte o resultado na tela do R com função `texreg::screenreg`.

3. Crie um novo relatório em *Rmarkdown* contemplando os dois exercícios anteriores. Compile o relatório em html e abra o mesmo no seu *browser*.

******ebook converter DEMO Watermarks*******

13 A Performance de Investimentos no Brasil

Neste capítulo iremos explorar um exemplo completo de pesquisa científica com o R, percorrendo as etapas de obtenção dos dados até a produção dos elementos do relatório como tabelas e figuras. A pesquisa inteira será reproduzível. Isto é, podes copiar e colar o código no R e o mesmo deve rodar sem modificação. Além disso, o arquivo *Rmarkdown* com este capítulo está disponível no site do autor.

******ebook converter DEMO Watermarks*******

13.1 Motivação

Um dos temas mais populares em Finanças é a análise da performance de diferentes investimentos. De fato, esse tema é quase sempre o primeiro tópico nos capítulos sobre mercado de capitais em livros didáticos. Isso não é acidental. A possibilidade de ganhar dinheiro no mercado de capitais é real, o que instiga uma curiosidade coletiva sobre o quanto poderia ser ganho historicamente. Além disso, a teoria financeira prevê que instrumentos financeiros com maior risco também seriam aqueles que, na média, ofereceriam maior retorno. Essa premissa faz parte de modelos teóricos de precificação de ativos. Portanto, a análise da performance histórica de diferentes investimentos é motivada tanto pelo interesse público quanto acadêmico.

O interesse principal na pesquisa proposta é analisar a performance de diferentes investimentos disponíveis para a pessoa física. Aqui, iremos analisar o desempenho de títulos do tesouro direto (renda fixa) e ações da B3 (renda variável). Para isso, iremos importar da internet dados de preços e retornos de vários instrumentos do mercado financeiro brasileiro no período entre 2008 e 2017.

******ebook converter DEMO Watermarks*******

13.2 Dados e Metodologia

Os seguintes instrumentos financeiros foram selecionados para essa análise:

- Ibovespa - Índice amplo do mercado brasileiro de ações, composto aproximadamente por 55 ações selecionadas por critérios de liquidez e negociabilidade. Esse índice representa a performance média no mercado de ações.

- PETR3, VALE3, ITUB3 - Ações ordinárias de diferentes empresas.

- NTN-B Principal 150824 - Título de dívida pública federal, indexado ao IPCA e com vencimento em 15/08/2024, disponível para investimento no Tesouro Direto.

- CDI - Taxa de juros no mercado bancário, geralmente utilizado como indicador do retorno médio no mercado de renda fixa.

A partir desses dados, a análise de performance envolverá a criação de figuras com o retorno anual e o retorno acumulado de cada instrumento. Além disso, iremos verificar o risco e o retorno esperado de cada alternativa de investimento. À primeira vista, essa pesquisa pode parecer fácil de implementar. Entretanto, não se engane. A combinação de dados de diferentes fontes irá demandar um considerável investimento de tempo no processo de limpeza e estruturação dos dados.

A nossa rotina de pesquisa terá etapas muito claras: 1) obtenção dos dados, 2) reestruturaçao e limpeza dos dados, 3) modelagem e teste de hipóteses e 4) reporte dos resultados. Vamos detalhar cada fase:

1. Obtenção dos Dados (2008-2018)

- Dados do Ibovespa e ações obtidos com `BatchGetSymbols`
- Dados do Tesouro direto com `GetTDData`
- Dados do CDI com o `Quandl`

2. Reestruturação e limpeza dos dados

- Dados de preços diários convertidos para frequência anual
- Criação da coluna de retorno anual
- Consolidação de todos dados em um único dataframe

3. Modelagem e teste de hipóteses

- Cálculos de média e desvio padrão dos retornos anuais
- Cálculo do índice *Sharpe* de cada instrumento

4. Reporte dos resultados

- Reportar os dados em uma tabela descritivas

******ebook converter DEMO Watermarks*******

- Criação de tabela com médias e desvios padrões
- Criação da figura de média variância com `ggplot2`

Esse *script* de pesquisa irá necessitar a instalação dos pacotes `BatchGetSymbols`, `GetTDData`, `dplyr`, `Quandl` e `ggplot2`. Caso você não o tenha instalado, pode fazê-lo com o seguinte comando:

```
install.packages(c('BatchGetSymbols',
                   'GetTDData',
                   'dplyr',
                   'Quandl',
                   'ggplot2'))
```

Ressalta-se, também , que o acesso aos dados do pacote `Quandl` exige um registro na plataforma, conforme explicado em capítulo anterior.

13.2.1 Importando Dados da B3

Os dados da B3 podem ser baixados diretamente do site do *Yahoo Finance* com o pacote `BatchGetSymbols`, conforme destacado em capítulo anterior e apresentado a seguir.

```
library(BatchGetSymbols)

first.date = as.Date('2008-01-01')
last.date = as.Date('2017-12-31')
my.assets <- c('^BVSP','PETR4.SA',
               'VALE3.SA', 'ITUB3.SA')

my.l <- BatchGetSymbols(tickers = my.assets,
                        first.date = first.date,
                        last.date = last.date)
```

```
##
## Running BatchGetSymbols for:
##    tickers = ^BVSP, PETR4.SA, VALE3.SA, ITUB3.SA
##    Downloading data for benchmark ticker | Found cache file
## ^BVSP | yahoo (1|4) | Found cache file - Well done!
## PETR4.SA | yahoo (2|4) | Found cache file - Good stuff!
## VALE3.SA | yahoo (3|4) | Found cache file - Good stuff!
## ITUB3.SA | yahoo (4|4) | Found cache file - Good stuff!
```

Após a importação, deve-se verificar se tudo ocorreu bem. Lembre que a função `BatchGetSymbols` retorna uma lista onde o elemento `df.control` mostra detalhes do processo de importação.

```
print(my.l$df.control)
```

```
##     ticker   src download.status total.obs
## 1    ^BVSP yahoo              OK      2483
## 2 PETR4.SA yahoo              OK      2484
## 3 VALE3.SA yahoo              OK      2483
## 4 ITUB3.SA yahoo              OK      2483
##   perc.benchmark.dates threshold.decision
## 1            0.9614774               KEEP
## 2            0.9614774               KEEP
```

******ebook converter DEMO Watermarks*******

```
## 3            0.9614774               KEEP
## 4            0.9614774               KEEP
```

Como pode ser visto, os dados foram baixados corretamente. Agora vamos observar o conteúdo do dataframe.

```
print(head(my.l$df.tickers))
```

```
##   price.open price.high price.low price.close  volume
## 1      63885      63906     62659       62815 2980200
## 2      62814      63129     62556       62892 1895800
## 3      62898      63436     60574       61037 2613800
## 4      61054      61505     60092       60772 2223400
## 5      60771      62650     60771       62081 3314000
## 6      62078      62674     61487       62674 3480400
##   price.adjusted   ref.date ticker ret.adjusted.prices
## 1          62815 2008-01-02  ^BVSP                  NA
## 2          62892 2008-01-03  ^BVSP         0.001225822
## 3          61037 2008-01-04  ^BVSP        -0.029495007
## 4          60772 2008-01-07  ^BVSP        -0.004341629
## 5          62081 2008-01-08  ^BVSP         0.021539525
## 6          62674 2008-01-09  ^BVSP         0.009552037
##   ret.closing.prices
## 1                 NA
## 2        0.001225822
## 3       -0.029495007
## 4       -0.004341629
## 5        0.021539525
## 6        0.009552037
```

Note que o dataframe possui diversas informações. Para a pesquisa em questão, estamos interessados apenas nas colunas de preços ajustados de fechamento, datas e tickers. Além disso, vamos organizar os dados com a modificação do nome das colunas para algo mais intuitivo. É importante definir os nomes pois os dados de ações serão agregados com os demais dados através da função `rbind`, a qual utiliza o nome das colunas para a agregação. Esse procedimento é apresentado a seguir:

```
df.stocks <- my.l$df.tickers[, c('price.adjusted','ref.date','ticker')]
colnames(df.stocks) <- c('price','ref.date','ticker')
```

Os nomes `price`, `ref.date` e `ticker` serão utilizados em todos os dados.

Após essa última etapa, temos em mãos um dataframe com dados limpos dos ativos da B3. Utilizando a função `glimpse`, verifica-se que as classes das colunas estão corretas:

```
dplyr::glimpse(df.stocks)
```

```
## Observations: 9,933
## Variables: 3
## $ price    <dbl> 62815, 62892, 61037, 60772, 62081, 626...
## $ ref.date <date> 2008-01-02, 2008-01-03, 2008-01-04, 2...
## $ ticker   <chr> "^BVSP", "^BVSP", "^BVSP", "^BVSP", "^...
```

13.2.2 Importando Dados de Títulos Públicos

******ebook converter DEMO Watermarks*******

Dados de títulos do Tesouro Direto, neste caso *NTN-B Principal 150824*, podem ser baixados do site do Tesouro Direto com pacote `GetTDData`. Veja a seguir:

```
library(GetTDData)

download.TD.data(asset.codes = 'NTN-B Principal')
df.TD <- read.TD.files(maturity = '150824')
```

Vamos verificar o formato do *dataframe* importado com as funções `head` e `print`:

```
print(head(df.TD))
```

```
##       ref.date yield.bid price.bid             asset.code
## 155 2005-08-11    0.0882    310.05 NTN-B Principal 150824
## 156 2005-08-12    0.0882    310.23 NTN-B Principal 150824
## 157 2005-08-15    0.0882    310.37 NTN-B Principal 150824
## 158 2005-08-16    0.0882    310.50 NTN-B Principal 150824
## 159 2005-08-17    0.0882    310.64 NTN-B Principal 150824
## 160 2005-08-18    0.0882    310.78 NTN-B Principal 150824
##     matur.date
## 155 2024-08-15
## 156 2024-08-15
## 157 2024-08-15
## 158 2024-08-15
## 159 2024-08-15
## 160 2024-08-15
```

Note que, novamente, o dataframe possui diversas informações. De todas colunas de `df.TD`, estamos interessados apenas em `ref.date`, `price.bid` e `asset.code`. No próximo código iremos primeiro garantir que os dados importados sejam após 2008-01-01 e, depois, iremos remover as colunas que não interessam para a pesquisa. Por fim, mudaremos o nome das colunas para manter o mesmo formato do dataframe de ações (`df.stocks`).

```
# remove rows before first.date and after last.date
idx <- (df.TD$ref.date>=first.date)&(df.TD$ref.date<=last.date)
df.TD <- df.TD[idx, ]

# remove  cols that don't matter to the study
df.TD$yield.bid <- NULL
df.TD$matur.date <- NULL

# modify names of cols
colnames(df.TD) <- c('ref.date','price','ticker')
```

E pronto! Verifica-se o resultado final com os comandos `print` e `head`:

```
print(head(df.TD))
```

```
##       ref.date  price                 ticker
## 255 2008-01-02 570.94 NTN-B Principal 150824
## 312 2008-01-03 573.00 NTN-B Principal 150824
## 412 2008-01-04 576.23 NTN-B Principal 150824
## 512 2008-01-07 573.85 NTN-B Principal 150824
## 612 2008-01-08 578.60 NTN-B Principal 150824
## 712 2008-01-09 581.59 NTN-B Principal 150824
```

******ebook converter DEMO Watermarks*******

Note que a estrutura dos dados está igual ao *dataframe* `df.stocks`. Essa equivalência é importante, pois iremos agregar ambos os *dataframes* em etapa futura.

13.2.3 Importando Dados do CDI

A última etapa na importação dos dados é buscar informações sobre o retorno do CDI. Para isso, utilizamos o pacote `Quandl`, onde o símbolo do CDI anual é `BCB/4389`. Para utilizar o sistema, o primeiro passo é registrar o `api_key`:

```
library (Quandl)

my.key <- 'YOUR_API_HERE'
```

Note que você deverá utilizar a sua própria chave, substituindo o texto no local onde se lê `YOUR_API_HERE`. Caso contrário, a importação dos dados não será efetivada. Avançando, os dados são acessados com o seguinte código:

```
Quandl.api_key(my.key)

quandl.codes <- c('BCB/4389')

df.CDI <- Quandl(quandl.codes,
                 type='raw',
                 start_date = first.date,
                 end_date = last.date)

dplyr::glimpse(df.CDI)
```

```
## Observations: 2,513
## Variables: 2
## $ Date  <date> 2017-12-29, 2017-12-28, 2017-12-27, 2017...
## $ Value <dbl> 6.89, 6.89, 6.89, 6.89, 6.89, 6.89, 6.89,...
```

Note as seguintes informações sobre os dados importados:

1. Os valores do CDI são informados como retorno anual. Essa é uma peculiaridade do mercado de CDI, onde se cotam retornos, e não preços. Por exemplo, na data `2016-12-23`, o CDI estava cotado como `13.6` de retorno anual. Esse valor irá mudar de acordo com os movimentos de demanda e oferta no mercado. Isso é diferente do que esperamos no formato da nossa pesquisa, uma vez que as demais bases de dados contêm **preços diários** e não retornos. Será necessário, portanto, converter esses retornos diários para o formato anual, com preços diários.

2. As datas não estão ordenadas. Essas deverão ser reordenadas no dataframe.

3. Falta uma coluna com o *ticker*, o símbolo do CDI.

Em primeiro lugar, modificamos os nomes das colunas de acordo com o formato anterior e calculamos o retorno diário equivalente para cada taxa de retorno anual. Essa transformação de retorno anual para retorno diário é bastante simples. Basta descapitalizar a taxa de retorno com base em um período de 252 dias.

******ebook converter DEMO Watermarks*******

```
colnames(df.CDI) <- c('ref.date','ret_year')
df.CDI$ret_day <- (1+df.CDI$ret_year/100)^(1/252) - 1
```

Após a criação dos retornos diários, ordenam-se as datas e se cria uma série de preços que irá simular o investimento de R$ 1 no início do período:

```
df.CDI <- df.CDI[order(df.CDI$ref.date), ]
df.CDI$price <- cumprod(1+df.CDI$ret_day)
```

Finalizando, adiciona-se a coluna `ticker` com o texto `CDI` e elimina-se as colunas que não interessam mais.

```
df.CDI$ticker <- 'CDI'

df.CDI$ret_year <- NULL
df.CDI$ret_day <- NULL
```

Novamente, verificamos se o dataframe final está correto.

```
dplyr::glimpse(df.CDI)
```

```
## Observations: 2,513
## Variables: 3
## $ ref.date <date> 2008-01-02, 2008-01-03, 2008-01-04, 2...
## $ price    <dbl> 1.000417, 1.000835, 1.001251, 1.001669...
## $ ticker   <chr> "CDI", "CDI", "CDI", "CDI", "CDI", "CD...
```

Como pode ser visto, temos as três colunas de interesse: `ref.date`, `price` e `ticker`. A partir dos dados da Bovespa, Tesouro Direto e CDI, partimos para a etapa de agregação de todas as informações anteriores para um único dataframe.

13.2.4 Agregando os Dados

Visto que, para cada tipo de dados, as colunas foram organizadas e nomeadas no mesmo formato, podemos agregar os diferentes dataframes com a função `dplyr::bind_rows`.

```
my.df <- bind_rows(df.stocks, df.TD, df.CDI)
```

Verifica-se o resultado com função `dplyr::glimpse`:

```
dplyr::glimpse(my.df)
```

```
## Observations: 14,941
## Variables: 3
## $ price    <dbl> 62815, 62892, 61037, 60772, 62081, 626...
## $ ref.date <date> 2008-01-02, 2008-01-03, 2008-01-04, 2...
## $ ticker   <chr> "^BVSP", "^BVSP", "^BVSP", "^BVSP", "^...
```

Essa última parte finaliza a etapa de limpeza e de estruturação dos dados. Iremos avançar com a fase de processamento e análise dos dados estruturados.

13.2.5 Calculando Retornos Anuais

******ebook converter DEMO Watermarks*******

O próximo passo da pesquisa é calcular os retornos anuais para cada ativo. O retorno anual nada mais é do que o valor percentual que um investidor ganharia caso comprasse o instrumento no início do ano e o vendesse no final do mesmo ano. Nesta etapa, para simplificar, ignoramos os efeitos do imposto de renda sobre o investimento em diferentes instrumentos. Notarás, porém, que a inclusão ou não do imposto não modifica as principais conclusões obtidas neste estudo.

Observa-se que os dados importados são para preços diários. Portanto, precisamos processar os preços ano por ano. Esse procedimento é fácil de realizar com o pacote `dplyr`. O primeiro passo é criar uma nova coluna em `my.df` com o ano de cada observação (linha). Essa nova coluna é então utilizada para calcular os retornos anuais de cada ativo. Calcula-se esse ativo pela divisão do preço final pelo preço inicial de cada ano. Veja, a seguir, o código que realiza essa operação:

```
library(dplyr)

my.ret <- my.df %>%
  mutate(year = format(my.df$ref.date,'%Y')) %>%
  group_by(ticker, year) %>%
  summarise(last.price = price[length(price)]) %>%
  mutate(ret = c(0, last.price[2:length(last.price)]/
                  last.price[1:(length(last.price)-1)]-1))

print(head(my.ret))
```

```
## # A tibble: 6 x 4
## # Groups:   ticker [1]
##   ticker year  last.price     ret
##   <chr>  <chr>      <dbl>   <dbl>
## 1 ^BVSP  2008       37550  0
## 2 ^BVSP  2009       68588  0.827
## 3 ^BVSP  2010       69305  0.0105
## 4 ^BVSP  2011       56754 -0.181
## 5 ^BVSP  2012       60952  0.0740
## 6 ^BVSP  2013       51507 -0.155
```

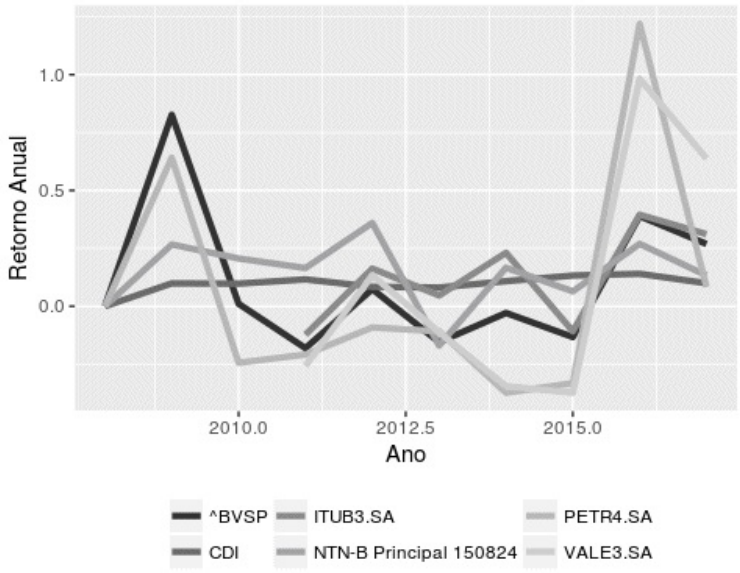
Note que o *dataframe* resultante é exatamente o esperado, onde temos um retorno para cada ticker e ano. Para visualizar os retornos calculados, vamos plotar os retornos com o pacote `ggplot2`.

```
library(ggplot2)

p <- ggplot(my.ret,
            aes(x = as.numeric(year),
                y = ret,
                color = ticker)) +
  geom_line(size=1.5) + labs(x='Ano',
                             y = 'Retorno Anual') +
  theme(legend.position="bottom") +
  theme(legend.title=element_blank()) +
  scale_colour_grey(start = 0.2,
                    end = 0.8)

print(p)
```

******ebook converter DEMO Watermarks*******



Conforme pode ser visto no gráfico, o investimento no mercado de ações é mais volátil em relação aos demais. O mercado de renda fixa, CDI e NTN-B Principal, foram os que apresentaram menor volatilidade (risco) durante o período. A ação PETR4 teve diversos anos com retornos negativos, porém, apresentou um alto retorno em 2016.

Para verificar o retorno total que um investidor teria ao investir R$ 1 em cada instrumento em 2008-01-01, é necessário acumular os retornos. Para isso, utilizamos o seguinte código baseado na função `tapply`.

```
my.l <- tapply(X = my.ret$ret,
               INDEX = my.ret$ticker,
               FUN = function(x) cumprod(c(x+1)))

my.l <- my.l[unique(my.ret$ticker)]
my.ret$acum.ret <- unlist(my.l)
```

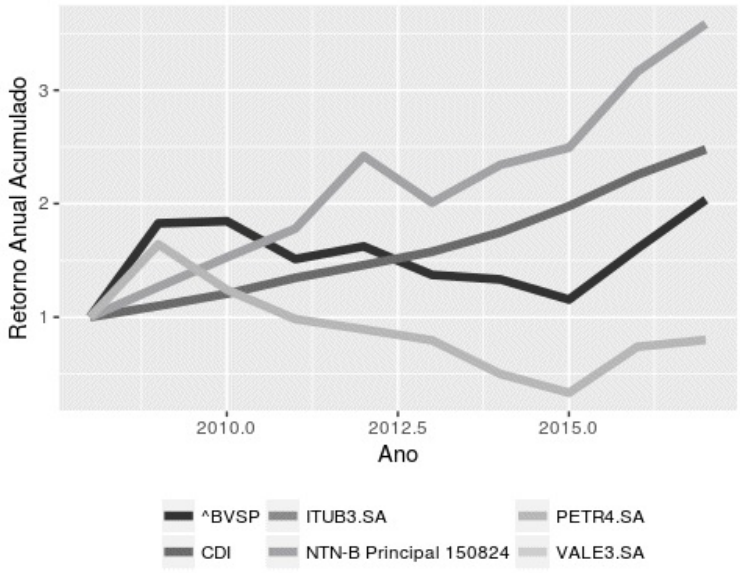
O código anterior calcula o retorno acumulado para cada ação. Esses são armazenados na lista `my.l`. Logo após, utiliza-se o comando `my.l <- my.l[unique(my.ret$ticker)]` para que a ordem da lista esteja de acordo com a ordem dos ativos em `my.ret` (lembre que `tapply` altera a ordem dos grupos). Por fim, acrescenta-se a coluna `acum.ret` com o comando `unlist(my.l)`.

O resultado desse cálculo é apresentado a seguir:

```
p <- ggplot(my.ret, aes(x=as.numeric(year),
                        y=acum.ret,
                        color=ticker)) +
  geom_line(size=2) +
  labs(x='Ano', y = 'Retorno Anual Acumulado') +
  theme(legend.position="bottom",legend.title=element_blank()) +
  scale_colour_grey( start = 0.2, end = 0.8)

print(p)
```

******ebook converter DEMO Watermarks*******



Como pode ser visto na figura anterior, o investimento em NTN-B Principal foi o que obteve maior retorno. Cada R$ 1 investido nesse ativo em 2008 resultou em 3,75 R$ no final de 2017! Nota-se também que o segundo ativo que representa o mercado de renda fixa, o CDI, ficou em segundo lugar em retorno acumulado. Esse é um resultado conhecido para o caso brasileiro onde, na média histórica, a renda fixa apresentou-se como um investimento bastante rentável.

Buscando uma análise mais completa sobre performance, deve-se também levar em conta o risco dos investimentos. Em uma análise simples, o risco pode ser representado pelo desvio padrão dos retornos anuais. Essa simples medida de volatilidade mensura a incerteza esperada a respeito do retorno futuro do investimento. Por exemplo, o CDI sempre rende algo próximo de sua média durante os anos e, devido a isso, é razoável esperar que essa dinâmica continue, de forma que a média dos seus retornos pode ser considerada como um bom indicador da sua rentabilidade futura. Essa qualidade já não pode ser atribuída ao índice IBovespa, uma vez que os retornos anuais tendem a ser dispersos em relação à sua média histórica.

O código a seguir irá realizar três procedimentos: calcular o retorno médio dos investimentos, determinar o risco (desvio padrão) e informar o índice *sharpe*. Esse último é o retorno médio dividido pelo risco. Ou seja, ele indica a relação quantitativa entre retorno e risco do investimento. Quanto maior o *sharpe*, melhor o investimento em termos de retorno ponderado pelo risco.

```
my.tab <- my.ret %>% group_by(ticker) %>%
  summarise(mean.ret = mean(ret, na.rm = TRUE),
            sd.ret = sd(ret, na.rm = TRUE),
            sharpe = mean.ret/sd.ret)

print(my.tab)
```

```
## # A tibble: 6 x 4
##   ticker                mean.ret sd.ret sharpe
##   <chr>                    <dbl>  <dbl>  <dbl>
## 1 ^BVSP                    0.107  0.311   0.344
```

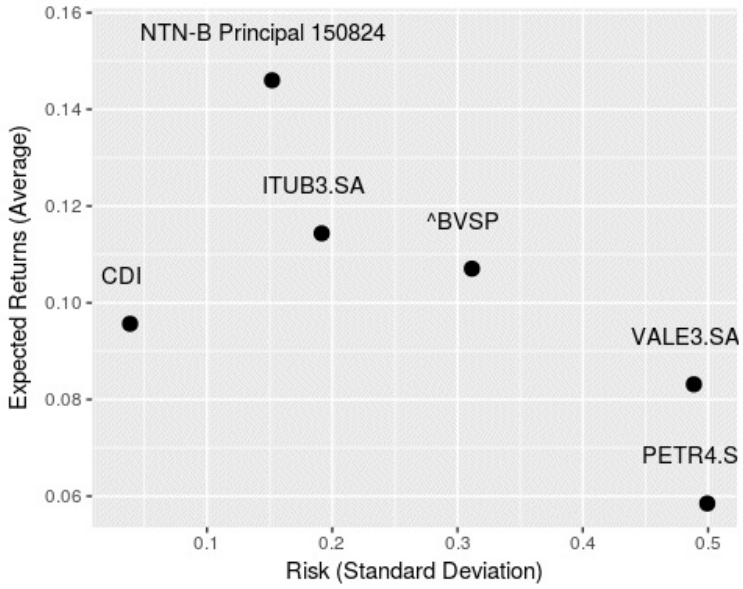
******ebook converter DEMO Watermarks*******

```
## 2 CDI                       0.0957 0.0386  2.48
## 3 ITUB3.SA                  0.114  0.192   0.597
## 4 NTN-B Principal 150824    0.146  0.152   0.961
## 5 PETR4.SA                  0.0585 0.499   0.117
## 6 VALE3.SA                  0.0831 0.489   0.170
```

Esse resultado fica mais fácil de analisar com o chamado gráfico de média/variância, que é muito familiar para quem estuda finanças. Ele não é nada mais do que um gráfico de dispersão, onde o eixo horizontal é o risco do investimento e o eixo vertical é o retorno esperado. O código a seguir constrói esse gráfico com base no pacote `ggplot2`.

```
p <- ggplot(my.tab, aes(x=sd.ret,
                        y=mean.ret)) +
  geom_point(size=3) +
  annotate('text', x = my.tab$sd.ret-0.007,
           y = my.tab$mean.ret+0.01,
           label = my.tab$ticker) +
  labs(x='Risk (Standard Deviation)',
       y='Expected Returns (Average)')

print(p)
```



Os gráficos e tabelas anteriores confirmam o resultado de que o mercado de renda fixa foi o melhor investimento nos últimos anos. Na média, o investidor que investiu nesse mercado conseguiu mais retorno por menos risco. Quando analisamos apenas o retorno total, o contrato de NTN-B Principal 150824 foi o que obteve maior retorno. Quando ponderamos o retorno pelo risco, com a análise do índice *sharpe*, o melhor investimento foi em títulos atrelados ao CDI.

Em relação ao *script* construído, destaca-se que o mesmo pode ser facilmente atualizado para dados recentes. Para isso, basta modificar a última data da pesquisa no objeto `last.date` para `last.date <- Sys.Date()`. Caso o pesquisador queira realizar a mesma pesquisa daqui a dois anos, basta executar o código com esta pequena modificação. Além disso, a adição de outras ações é tarefa trivial, basta incluir os *tickers* no início do *script*. De forma semelhante, a adição

******ebook converter DEMO Watermarks*******

de outros instrumentos do Tesouro Direto também é muito simples. Conforme foi destacado anteriormente, replicabilidade e reusabilidade do código são atributos essenciais para aumentar a produtividade do trabalho a médio e longo prazo.

******ebook converter DEMO Watermarks*******

Referências

Assaf Neto, Alexandre. 2010. *Estrutura E análise de Balanços: Um Enfoque Econômico-Financeiro*. Atlas.

Bache, Stefan Milton, and Hadley Wickham. 2014. *Magrittr: A Forward-Pipe Operator for R*. https://CRAN.R-project.org/package=magrittr.

Bollerslev, Tim. 1986. "Generalized Autoregressive Conditional Heteroskedasticity." *Journal of Econometrics* 31 (3). Elsevier: 307–27.

Brooks, Chris. 2014. *Introductory Econometrics for Finance*. Cambridge university press.

Croissant, Yves, and Giovanni Millo. 2008. "Panel Data Econometrics in R: The plm Package." *Journal of Statistical Software* 27 (2). http://www.jstatsoft.org/v27/i02/.

Dahl, David B. 2016. *Xtable: Export Tables to Latex or Html*. https://CRAN.R-project.org/package=xtable.

Dowle, M, A Srinivasan, T Short, S Lianoglou with contributions from R Saporta, and E Antonyan. 2015. *Data.table: Extension of Data.frame*. https://CRAN.R-project.org/package=data.table.

Dragulescu, Adrian A. 2014. *Xlsx: Read, Write, Format Excel 2007 and Excel 97/2000/Xp/2003 Files*. https://CRAN.R-project.org/package=xlsx.

Engle, Robert F. 1982. "Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation." *Econometrica: Journal of the Econometric Society*. JSTOR, 987–1007.

Farnsworth, Grant V. 2008. "Econometrics in R." Technical report, October 2008. Available at http://cran. rproject. org/doc/contrib/Farnsworth-EconometricsInR. pdf.

Fox, John, and Sanford Weisberg. 2011. *An R Companion to Applied Regression*. Second. Thousand Oaks CA: Sage. http://socserv.socsci.mcmaster.ca/jfox/Books/Companion.

Freitas, Wilson. 2016. *Bizdays: Business Days Calculations and Utilities*. https://CRAN.R-project.org/package=bizdays.

Gentzkow, Matthew, Bryan T Kelly, and Matt Taddy. 2017. "Text as Data." National Bureau of Economic Research.

Ghalanos, Alexios. 2015. *Rugarch: Univariate Garch Models.*

Gohel, David. 2017. *ReporteRs: Microsoft Word and Powerpoint Documents Generation*. https://CRAN.R-project.org/package=ReporteRs.

******ebook converter DEMO Watermarks*******

Greene, William H. 2003. *Econometric Analysis*. Pearson Education India.

Grolemund, Garrett, and Hadley Wickham. 2011. “Dates and Times Made Easy with lubridate.” *Journal of Statistical Software* 40 (3): 1–25. http://www.jstatsoft.org/v40/i03/.

Grunfeld, Y. 1958. “The Determinants of Corporate Investment, Unpublished Ph. d.” *D Thesis, the University of Chicago*.

Gujarati, Damodar N, and Dawn C Porter. 2011. *Econometria Básica-5*. AMGH Editora.

Hamilton, James. 1994. *Time Series Analysis*. Princeton University Press.

Hamilton, James Douglas. 1994. *Time Series Analysis*. Vol. 2. Princeton university press Princeton.

Harrison, John. 2016. *RSelenium: R Bindings for Selenium Webdriver*. https://CRAN.R-project.org/package=RSelenium.

Hausman, Jerry A. 1978. “Specification Tests in Econometrics.” *Econometrica: Journal of the Econometric Society*. JSTOR, 1251–71.

Henry, Lionel, and Hadley Wickham. 2017. *Purrr: Functional Programming Tools*. https://CRAN.R-project.org/package=purrr.

Hsiao, Cheng. 2014. *Analysis of Panel Data*. 54. Cambridge university press.

Hyndman, RJ, and Y Khandakar. 2007. “Automatic Time Series Forecasting: The Forecast Package for R 7. 2008.” *URL: Https://Www. Jstatsoft. Org/Article/View/V027i03 [Accessed 2016-02-24][WebCite Cache]*.

James, David, and Kurt Hornik. 2017. *Chron: Chronological Objects Which Can Handle Dates and Times*. https://CRAN.R-project.org/package=chron.

Kleiber, Christian, and Achim Zeileis. 2008. *Applied Econometrics with R*. Springer Science & Business Media.

Lang, Duncan Temple, and the CRAN Team. 2016. *XML: Tools for Parsing and Generating Xml Within R and S-Plus*. https://CRAN.R-project.org/package=XML.

Leifeld, Philip. 2013. “texreg: Conversion of Statistical Model Output in R to LaTeX and HTML Tables.” *Journal of Statistical Software* 55 (8): 1–24. http://www.jstatsoft.org/v55/i08/.

Maddala, GS. 2001. “Introduction to Econometrics.” John Wiley & Sons.

Matarazzo, Dante Carmine, and Armando Oliveira Pestana. 2010. *Analise Financeira de Balanços: Abordagem Básica E Gerencial: Livro de Exercícios*. Atlas.

Mirai Solutions GmbH. 2016. *XLConnect: Excel Connector for R*. https://CRAN.R-project.org/package=XLConnect.
******ebook converter DEMO Watermarks*******

Ooms, Jeroen. 2017. *Writexl: Export Data Frames to 'Xlsx' Format*. https://CRAN.R-project.org/package=writexl.

Pena, Edsel A., and Elizabeth H. Slate. 2014. *Gvlma: Global Validation of Linear Models Assumptions*. https://CRAN.R-project.org/package=gvlma.

Perlin, Marcelo. 2014. *FMarkovSwitching: R Package for Estimation, Simulation and Forecasting of a Univariate Markov Switching Model*. https://R-Forge.R-project.org/projects/rmetrics/.

———. 2016a. "A Microestrutura Do Tesouro Direto: Sazonalidade Do Fluxo de Ordens E O Processo de Formação de Spreads." *Economia Aplicada* 20 (3). Faculdade de Economia, Administração e Contabilidade de Ribeirão Preto-USP, Dept de Economia: 253.

———. 2016b. *BatchGetSymbols: Downloads and Organizes Financial Data for Multiple Tickers*.

———. 2017. *GetDFPData: Reading Annual Financial Reports from Bovespa's Dfp and Fre System*. https://github.com/msperlin/GetDFPData/.

Perlin, Marcelo, and Henrique Ramos. 2016. *GetHFData: A R Package for Downloading and Aggregating High Frequency Trading Data from Bovespa*. *Available at SSRN*. https://ssrn.com/abstract=2824058.

R Core Team. 2015. *Foreign: Read Data Stored by Minitab, S, Sas, Spss, Stata, Systat, Weka, dBase, ...* https://CRAN.R-project.org/package=foreign.

Robinson, David. 2017. *Broom: Convert Statistical Analysis Objects into Tidy Data Frames*. https://CRAN.R-project.org/package=broom.

Ryan, Jeffrey A. 2015. *Quantmod: Quantitative Financial Modelling Framework*. https://CRAN.R-project.org/package=quantmod.

Ryan, Jeffrey A., and Joshua M. Ulrich. 2014. *Xts: EXtensible Time Series*. https://CRAN.R-project.org/package=xts.

Sanchez-Espigares, Josep A., and Alberto Lopez-Moreno. 2014. *MSwM: Fitting Markov Switching Models*. https://CRAN.R-project.org/package=MSwM.

Team, Rmetrics Core, Diethelm Wuertz, Tobias Setz, Yohan Chalabi, Martin Maechler, and Joe W. Byers. 2015. *TimeDate: Rmetrics - Chronological and Calendar Objects*. https://CRAN.R-project.org/package=timeDate.

Teetor, Paul. 2011. *R Cookbook*. " O'Reilly Media, Inc."

Thompson, Ken. 1968. "Programming Techniques: Regular Expression Search Algorithm." *Communications of the ACM* 11 (6). ACM: 419–22.

******ebook converter DEMO Watermarks*******

Trapletti, Adrian, and Kurt Hornik. 2017. *Tseries: Time Series Analysis and Computational Finance*. https://CRAN.R-project.org/package=tseries.

Tsay, Ruey S. 2005. *Analysis of Financial Time Series*. Vol. 543. John Wiley & Sons.

Tsay, Ruey S. 2015. *MTS: All-Purpose Toolkit for Analyzing Multivariate Time Series (Mts) and Estimating Multivariate Volatility Models*. https://CRAN.R-project.org/package=MTS.

Vaughan, Davis, and Matt Dancho. 2018. *Tibbletime: Time Aware Tibbles*. https://CRAN.R-project.org/package=tibbletime.

Venables, William N, David M Smith, R Development Core Team, and others. 2004. "An Introduction to R." Network Theory Limited.

Wickham, Hadley. 2007. "Reshaping Data with the reshape Package." *Journal of Statistical Software* 21 (12): 1–20. http://www.jstatsoft.org/v21/i12/.

———. 2009. *Ggplot2: Elegant Graphics for Data Analysis*. Springer Science & Business Media.

———. 2014a. *Advanced R*. CRC Press.

———. 2014b. "Tidy Data." *The Journal of Statistical Software* 59 (10). http://www.jstatsoft.org/v59/i10/.

———. 2015. *Stringr: Simple, Consistent Wrappers for Common String Operations*. https://CRAN.R-project.org/package=stringr.

———. 2016a. *Readxl: Read Excel Files*. https://CRAN.R-project.org/package=readxl.

———. 2016b. *Rvest: Easily Harvest (Scrape) Web Pages*. https://CRAN.R-project.org/package=rvest.

———. 2016c. *Tidyr: Easily Tidy Data with Spread() and Gather() Functions*. https://CRAN.R-project.org/package=tidyr.

———. 2016d. *Tidyr: Easily Tidy Data with 'Spread()' and 'Gather()' Functions*. https://CRAN.R-project.org/package=tidyr.

Wickham, Hadley, and Romain Francois. 2016. *Dplyr: A Grammar of Data Manipulation*. https://CRAN.R-project.org/package=dplyr.

Wuertz, Diethelm, Yohan Chalabi with contribution from Michal Miklovic, Chris Boudt, Pierre Chausse, and others. 2016. *FGarch: Rmetrics - Autoregressive Conditional Heteroskedastic Modelling*. https://CRAN.R-project.org/package=fGarch.

Xie, Yihui. 2016. *Bookdown: Authoring Books and Technical Documents with R Markdown*. CRC Press.

******ebook converter DEMO Watermarks*******

Zeileis, Achim. 2004. “Econometric Computing with Hc and Hac Covariance Matrix Estimators.” Institut für Statistik und Mathematik, WU Vienna University of Economics; Business.

Zeileis, Achim, and Torsten Hothorn. 2002. “Diagnostic Checking in Regression Relationships.” *R News* 2 (3): 7–10. https://CRAN.R-project.org/doc/Rnews/.

******ebook converter DEMO Watermarks*******