

Scikit-learn Cheatsheet

Optimised for ML learners and Data Science Enthusiasts



Made with ♥
by @thegeekyb0y

Data Preprocessing

- **Standard Scaler**

Standardizes data by removing the mean and scaling to unit variance.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(data)
scaler.transform(data)
```

- **Normalizer**

Each sample with at least one non zero component is rescaled independently of other samples so that its norm (l1, l2 or inf) equals one.

```
from sklearn.preprocessing import Normalizer
transformer = Normalizer()
transformer.fit(data)
transformer.transform(data)
```

- **Binarizer**

Sets feature values to 0 or 1 according to a threshold.

```
from sklearn.preprocessing import Binarizer
transformer = Binarizer().fit(data)
transformer.transform(data)
```

- **Ordinal Encoder**

OrdinalEncoder will encode each category with a different number.

```
from sklearn.preprocessing import OrdinalEncoder
encoder = OrdinalEncoder()
data_encoded = encoder.fit_transform(data)
```

- **make_pipeline**

Allows to construct a pipeline – a set of commands/models/etc. which will be executed consequently.

```
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression

model = make_pipeline(
    OneHotEncoder(handle_unknown="ignore"),
    LogisticRegression(max_iter=500))
```

- **MinMax Scaler**

Transforms the data so that its values appear in the given range.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(data)
scaler.transform(data)
```

- **SimpleImputer**

Replaces missing values using a descriptive statistic (e.g. mean, median, or most frequent) along each column, or using a constant value.

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(missing_values=np.nan,
    strategy='mean')
imputer.fit(data)
imputer.transform(data)
```

- **Polynomial Features**

Generates polynomial and interaction features. Parameters: degree specifies the maximal degree

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree = 2)
poly.fit_transform(data)
```

- **OneHot Encoder**

OneHotEncoder will create as many new columns as there are possible categories

```
from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder()
data_encoded = encoder.fit_transform(data)
```

- **set_config**

Allows to visualize the pipelines in Jupyter, needs to be set once at the beginning of your notebook.

```
from sklearn import set_config
set_config(display="diagram")
```

Encoding

Pipelines

Column Transformer

Applies transformations to the subset of columns in data.

```
from sklearn.compose import ColumnTransformer

preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), ['num_col1', 'num_col2']),
        ('cat', OneHotEncoder(), ['cat_col'])
    ])
```

- **mean_squared_error**

Average of the squares of the differences between predicted and actual values, used to measure the accuracy of a model.

```
from sklearn.metrics import mean_squared_error
mean_squared_error(y_true, y_pred)
```

- **precision_score**

It measures the proportion of true positive predictions among all positive predictions made by the model.

```
from sklearn.metrics import precision_score
precision_score(y_true, y_pred, average='macro')
```

- **recall_score**

It measures the proportion of true positive predictions among all actual positive instances

```
from sklearn.metrics import recall_score
recall_score(y_true, y_pred, average='macro')
```

- **f1_score**

It measures the proportion of true positive predictions among all actual positive instances

```
from sklearn.metrics import f1_score
f1 = f1_score(y_true, y_pred)
```

- **train_test_split**

Split arrays or matrices into random train and test subsets.

random_state is Seed for the random number generator to ensure reproducibility.

```
from sklearn.model_selection import train_test_split
data_train, data_test, target_train, target_test = train_test_split(data, target, random_state=42)
```

Learning Curve

Allows to see how the model performance changes when choosing different train/test split size.

```
from sklearn.tree import DecisionTreeRegressor
regressor = DecisionTreeRegressor()

from sklearn.model_selection import ShuffleSplit
cv = ShuffleSplit(n_splits=30, test_size=0.2)

from sklearn.model_selection import learning_curve

train_sizes=[0.3, 0.6, 0.9]
results = learning_curve(
    regressor, data, target, train_sizes=train_sizes,
    cv=cv,
    scoring="neg_mean_absolute_error", n_jobs=2)
```

- **confusion_matrix**

It is a table that is used to define the performance of a classification algorithm.

```
from sklearn.metrics import confusion_matrix
labels=["a", "b", "c"]
cm = confusion_matrix(y_true, y_pred, labels=labels)
```

- **roc_curve**

visual representation of model performance.

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_true,
y_scores, pos_label=1)
```

- **auc** Area under roc

```
from sklearn.metrics import roc_curve, auc
fpr, tpr, thresholds = roc_curve(y_true,
y_scores, pos_label=1)
roc_auc = auc(fpr, tpr)
```



Zoom to read better

Use laptop/tab for
better experience.

Supervised Learning

Classification

Classification is an algorithm that finds functions that help divide the dataset into classes based on various parameters. For example, classification can predict whether an email is spam or not.

- **Logistic Regression**

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()
model.fit(X_train, y_train)
```

- **KNeighborsClassifier**

```
from sklearn.neighbors import KNeighborsClassifier

model = KNeighborsClassifier(n_neighbors=3)
model.fit(X_train, y_train)
```

- **Gaussian NB**

```
from sklearn.naive_bayes import GaussianNB

model = GaussianNB()
model.fit(X_train, y_train)
```

- **Linear SVC**

```
from sklearn.svm import LinearSVC

model = LinearSVC()
model.fit(X_train, y_train)
```

- **SVC**

```
from sklearn.svm import SVC

model = SVC(kernel='linear')
model.fit(X_train, y_train)
```

- **Decision Tree Classifier**

```
from sklearn.tree import DecisionTreeClassifier

model = DecisionTreeClassifier()
model.fit(X_train, y_train)
```

- **Random Forest Classifier**

```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier()
model.fit(X_train, y_train)
```

Regression

Regression finds correlations between dependent and independent variables. Therefore, regression algorithms help predict continuous variables such as house prices, market trends, weather patterns, oil and gas prices (a critical task these days!), etc.

- **Linear Regression**

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()
model.fit(X_train, y_train)
```

- **Polynomial Regression**

```
from sklearn.linear_model import PolynomialRegression

model = PolynomialRegression(degree=2)
model.fit(X, y)
```

- **SVR**

```
from sklearn.svm import SVR

model = SVR(kernel='linear')
model.fit(X_train, y_train)
```

- **Ridge**

```
from sklearn.linear_model import Ridge

model = Ridge(alpha=1.0)
model.fit(X_train, y_train)
```

- **Lasso**

```
from sklearn.linear_model import Lasso

model = Lasso(alpha=0.1)
model.fit(X_train, y_train)
```

- **Decision Tree Regressor**

```
from sklearn.tree import DecisionTreeRegressor

model = DecisionTreeRegressor()
model.fit(X_train, y_train)
```

- **Random Forest**

```
from sklearn.ensemble import
RandomForestRegressor

model = RandomForestRegressor()
model.fit(X_train, y_train)
```

- **Elastic NET**

```
from sklearn.linear_model import ElasticNet

dmodel = ElasticNet(alpha=0.1, l1_ratio=0.5)
model.fit(X_train, y_train)
```

- **KNearest Neighbours**

```
from sklearn.neighbors import KNeighborsRegressor

model = KNeighborsRegressor(n_neighbors=3)
model.fit(X_train, y_train)
```

Unsupervised Learning

Clustering

Clustering is an unsupervised machine learning technique designed to group unlabeled examples based on their similarity to each other.

- **Kmeans**

```
from sklearn.cluster import KMeans

data = np.array([[1, 2], [1, 4], [1, 0],
                 [4, 2], [4, 0], [4, 4]])

kmeans = KMeans(n_clusters=2)
kmeans.fit(data)
clusters = kmeans.labels_
```

- **AgglomerativeClustering**

```
from sklearn.cluster import AgglomerativeClustering

data = np.array([[1, 2], [1, 4], [1, 0],
                 [4, 2], [4, 0], [4, 4]])

model = AgglomerativeClustering(n_clusters=2)
clusters = model.fit_predict(data)
```

- **DBScan**

```
from sklearn.cluster import DBSCAN

data = np.array([[1, 2], [1, 4], [1, 0],
                 [4, 2], [4, 0], [4, 4]])

model = DBSCAN(eps=1, min_samples=2)
clusters = model.fit_predict(data)
```

- **GaussianMixture**

```
from sklearn.mixture import GaussianMixture

data = np.array([[1, 2], [1, 4], [1, 0],
                 [4, 2], [4, 0], [4, 4]])

gmm = GaussianMixture(n_components=2)
gmm.fit(data)
clusters = gmm.predict(data)
```

Dimensionality Reduction

- **Principal Component Analysis**

Technique used to reduce the dimensionality of a dataset while preserving as much variance as possible.

```
from sklearn.decomposition import PCA

data = np.array([[1, 2], [2, 3], [3, 4]])
pca = PCA(n_components=1)
reduced_data = pca.fit_transform(data)
```

- **t-SNE (t-distributed Stochastic Neighbor Embedding)**

Converts similarities between data points into joint probabilities and minimizes the Kullback-Leibler divergence between the original and reduced distributions.

```
from sklearn.manifold import TSNE

data = np.array([[1, 2], [2, 3], [3, 4]])
tsne = TSNE(n_components=2)
reduced_data = tsne.fit_transform(data)
```

- **UMAP (Uniform Manifold Approximation and Projection)**

A technique that preserves the global structure of data while allowing for local relationships. It is effective for visualizing complex datasets.

```
import umap

data = np.array([[1, 2], [2, 3], [3, 4]])
umap_model = umap.UMAP(n_components=2)
reduced_data = umap_model.fit_transform(data)
```

HyperParameter Tuning

- **GridSearchCV**

Used to find the optimal hyperparameters for a model by exhaustively searching through a specified parameter grid. It evaluates all combinations of parameters and selects the best-performing set based on cross-validation.

```
from sklearn.model_selection import GridSearchCV

data = load_iris()
model = RandomForestClassifier()
param_grid = {'n_estimators': [10, 50, 100]}
grid_search = GridSearchCV(model, param_grid, cv=5)
grid_search.fit(data.data, data.target)
```

- **RandomizedSearchCV**

This method samples a specified number of parameter combinations from a grid, making it more efficient than GridSearchCV, especially when dealing with a large parameter space.

```
from sklearn.model_selection import RandomizedSearchCV

data = load_iris()
model = RandomForestClassifier()
param_dist = {'n_estimators': randint(10, 100)}
random_search = RandomizedSearchCV(model,
param_dist, n_iter=10, cv=5)
random_search.fit(data.data, data.target)
```

Deep Learning

• Multi Layer Perceptron

A type of feedforward artificial neural network that consists of multiple layers of nodes, where each layer is fully connected to the next. MLPs are capable of learning complex patterns and are widely used for classification and regression tasks.

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(64, activation='relu', input_shape=(input_dim,)),
    layers.Dense(64, activation='relu'),
    layers.Dense(1)
])
model.compile(optimizer='adam', loss='mean_squared_error')
```

• Recurrent Neural Networks

Designed for sequential data, allowing information to persist across time steps. RNNs are particularly effective for tasks such as time series prediction and natural language processing.

```
model = keras.Sequential([
    layers.SimpleRNN(64, input_shape=(timesteps, features)),
    layers.Dense(1)
])
model.compile(optimizer='adam', loss='mean_squared_error')
```

• Long Short-Term Memory

A type of RNN capable of learning long-term dependencies. It uses special units called memory cells to store information over long periods, making it effective for tasks such as speech recognition and language modeling.

```
model = keras.Sequential([
    layers.LSTM(64, input_shape=(timesteps, features)),
    layers.Dense(1)
])
model.compile(optimizer='adam', loss='mean_squared_error')
```

• Gated Recurrent Unit

A variant of LSTM that uses gating mechanisms to control the flow of information. GRUs are simpler and often faster to train than LSTMs while achieving comparable performance on many tasks.

```
model = keras.Sequential([
    layers.GRU(64, input_shape=(timesteps, features)),
    layers.Dense(1)
])
model.compile(optimizer='adam', loss='mean_squared_error')
```

• Convolutional Neural Networks

Specialized neural networks designed for processing structured grid data, such as images.

```
model = keras.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu',
        input_shape=(height, width, channels)),

    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(num_classes, activation='softmax')
])
model.compile(optimizer='adam', loss='categorical_crossentropy',
    metrics=['accuracy'])
```



Zoom to read better

Use laptops/tabs for
better experience.