

Machine Learning using Scikit-Learn (Python)

In general, a learning problem considers a set of n samples of data and then tries to predict properties of unknown data. If each sample is more than a single number and, for instance, a multi-dimensional entry (aka multivariate data), is it said to have several attributes or **features**.

We can separate learning problems in a few large categories:

- **supervised learning**, in which the data comes with additional attributes that we want to predict. This problem can be either:
 - **classification**: samples belong to two or more classes and we want to learn from already labeled data how to predict the class of unlabeled data. An example of classification problem would be the handwritten digit recognition example, in which the aim is to assign each input vector to one of a finite number of discrete categories. Another way to think of classification is as a discrete (as opposed to continuous) form of supervised learning where one has a limited number of categories and for each of the n samples provided, one is to try to label them with the correct category or class.
 - **regression**: if the desired output consists of one or more continuous variables, then the task is called *regression*. An example of a regression problem would be the prediction of the length of a salmon as a function of its age and weight.
- **unsupervised learning**, in which the training data consists of a set of input vectors x without any corresponding target values. The goal in such problems may be to discover groups of similar examples within the data, where it is called **clustering**, or to determine the distribution of data within the input space, known as **density estimation**, or to project the data from a high-dimensional space down to two or three dimensions for the purpose of *visualization*.

Training set and testing set

Machine learning is about learning some properties of a data set and applying them to new data. This is why a common practice in machine learning to evaluate an algorithm is to split the data at hand into two sets, one that we call the **training set** on which we learn data properties and one that we call the **testing set** on which we test these properties.

Loading an example dataset

scikit-learn comes with a few standard datasets, for instance the iris and digits datasets for classification and the boston house prices dataset for regression.

Python

```
from sklearn import datasets
```

```
# load iris dataset
iris = datasets.load_iris()
```

```
# print the iris data
print iris.data
```

```
# print the names of the four features
print iris.feature_names
```

```
# print integers representing the species of each
observation
print iris.target
```

```
# store feature matrix in "X"
X = iris.data
```

```
# store response vector in "y"
y = iris.target
```

Training a machine learning model with Scikit-Learn

ML Classification Models:

K-nearest neighbors (KNN) classification

1. Pick a value for K .
2. Search for the K observations in the training data that are "nearest" to the measurements of the unknown iris.
3. Use the most popular response value from the K nearest neighbors as the predicted response value for the unknown iris.

Python

```
from sklearn.neighbors import KNeighborsClassifier
```

```
# instantiate the model (using K=1)
knn = KNeighborsClassifier(n_neighbors=1)
```

```
# check default settings
print(knn)

# fit the model with data (aka "model training")
knn.fit(X, y)

# predict the response for a new observation
knn.predict([3, 5, 4, 2])

# predict for multiple observations at once
X_new = [[3, 5, 4, 2], [5, 4, 3, 2]]
knn.predict(X_new)

# instantiate the model (using the value k=5)
knn = KNeighborsClassifier(n_neighbors=5)

# fit the model with data
knn.fit(X, y)

# predict the response for new observations
knn.predict(X_new)
```

Linear Regression

It is used to estimate real values (cost of houses, number of calls, total sales etc.) based on continuous variable(s). Here, we establish relationship between independent and dependent variables by fitting a best line. This best fit line is known as regression line and represented by a linear equation $Y = a * X + b$. These coefficients a and b are derived based on minimizing the sum of squared difference of distance between data points and regression line.

Python

```
from sklearn import linear_model

# instantiate the model (using the default parameters)
linear = linear_model.LinearRegression()

# fit the model with data
linear.fit(X, y)

# predict the response for new observations
linear.predict(X_new)

# equation coefficient and Intercept
print('Coefficient: \n', linear.coef_)
print('Intercept: \n', linear.intercept_)
```

Logistic Regression

Logistic regression is a linear model for classification rather than regression. Logistic regression is also known in the literature as logit regression, maximum-entropy classification (MaxEnt) or the log-linear classifier. In this model, the probabilities describing the

possible outcomes of a single trial are modeled using a logistic function.

Python

```
from sklearn.linear_model import LogisticRegression

# instantiate the model (using the default parameters)
logreg = LogisticRegression()

# fit the model with data
logreg.fit(X, y)

# predict the response for new observations
logreg.predict(X_new)
```

Decision Trees

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.

Python

```
from sklearn.tree import DecisionTreeClassifier

# instantiate the model (using the default parameters)
clf = DecisionTreeClassifier()

# fit the model with data
clf = clf.fit(X, y)

# predict the response for new observations
clf.predict(X_new)
```

Support vector machines

Support vector machines (**SVMs**) are a set of supervised learning methods used for classification, regression and outliers' detection. SVM uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.

Python

```
from sklearn import svm

# instantiate the model (using the default parameters)
svmcflf = svm.SVC()

# fit the model with data
svmcflf.fit(X, y)

# predict the response for new observations
svmcflf.predict(X_new)
```

Random Forest

Random Forest is a trademark term for an ensemble of decision trees. In Random Forest, we've collection of decision trees (so known as "Forest"). To classify a new object based on attributes, each tree gives a classification and we say the tree "votes" for that class. The forest chooses the classification having the most votes (over all the trees in the forest).

Python

```
from sklearn.ensemble import RandomForestClassifier
```

```
# instantiate the model (using the default parameters)
rclf = RandomForestClassifier()
```

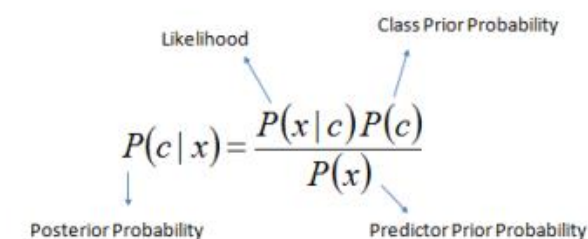
```
# fit the model with data
rclf.fit(X, y)
```

```
# predict the response for new observations
rclf.predict(X_new)
```

Naïve Bayes

It is a classification technique based on Bayes' theorem with an assumption of independence between predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. Naive Bayesian model is easy to build and particularly useful for very large data sets. Along with simplicity, Naive Bayes is known to outperform even highly sophisticated classification methods.

Bayes theorem provides a way of calculating posterior probability $P(c|x)$ from $P(c)$, $P(x)$ and $P(x|c)$. Look at the equation below:


$$P(c|x) = \frac{P(x|c)P(c)}{P(x)}$$

$$P(c|X) = P(x_1|c) \times P(x_2|c) \times \dots \times P(x_n|c) \times P(c)$$

- $P(c|x)$ is the posterior probability of *class (target)* given *predictor (attribute)*.
- $P(c)$ is the prior probability of *class*.
- $P(x/c)$ is the likelihood which is the probability of *predictor* given *class*.
- $P(x)$ is the prior probability of *predictor*.

Python

```
from sklearn.naive_bayes import GaussianNB
```

```
# instantiate the model (using the default parameters)
nbclf = GaussianNB()
```

```
# fit the model with data
nbclf.fit(X, y)
```

```
# predict the response for new observations
nbclf.predict(X_new)
```

Model evaluation procedures

Evaluation procedure #1: Train and test on the entire dataset

1. Train the model on the **entire dataset**.
2. Test the model on the **same dataset**, and evaluate how well we did by comparing the **predicted** response values with the **true** response values.

Python

```
from sklearn.linear_model import LogisticRegression
```

```
# instantiate the model (using the default parameters)
logreg = LogisticRegression()
```

```
# fit the model with data
logreg.fit(X, y)
```

```
# predict the response values for the observations in X
logreg.predict(X)
```

```
# store the predicted response values
y_pred = logreg.predict(X)
```

```
# compute classification accuracy for the logistic regression model
```

```
from sklearn import metrics
print(metrics.accuracy_score(y, y_pred))
```

Evaluation procedure #2: Train/test split

1. Split the dataset into two pieces: a **training set** and a **testing set**.
2. Train the model on the **training set**.
3. Test the model on the **testing set**, and evaluate how well we did.

```
# STEP 1: split X and y into training and testing sets
```

Python

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.4, random_state=4)

# STEP 2: train the model on the training set

logreg = LogisticRegression()
logreg.fit(X_train, y_train)

# STEP 3: make predictions on the testing set
y_pred = logreg.predict(X_test)

# compare actual response values (y_test) with
predicted response values (y_pred)

print(metrics.accuracy_score(y_test, y_pred))
```

Evaluation procedure #3: Cross-validation

K-fold cross-validation

1. Split the dataset into K **equal** partitions (or "folds").
2. Use fold 1 as the **testing set** and the union of the other folds as the **training set**.
3. Calculate **testing accuracy**.
4. Repeat steps 2 and 3 K times, using a **different fold** as the testing set each time.
5. Use the **average testing accuracy** as the estimate of out-of-sample accuracy.

Python

```
from sklearn.model_selection import cross_val_score

# 10-fold cross-validation
scores = cross_val_score(logreg, X, y, cv=10,
scoring='accuracy')
print(scores)

# use average accuracy as an estimate of out-of-sample
accuracy
print(scores.mean())
```

Feature Selection in Python with Scikit-Learn

Feature selection is a process where you automatically select those features in your data that contribute most to the prediction variable or output in which you are interested. Having too many irrelevant features in your data can decrease the accuracy of the models. Three benefits of performing feature selection before modeling your data are:

- **Reduces Overfitting:** Less redundant data means less opportunity to make decisions based on noise.
- **Improves Accuracy:** Less misleading data means modeling accuracy improves.
- **Reduces Training Time:** Less data means that algorithms train faster.

Recursive Feature Elimination

The Recursive Feature Elimination (RFE) method is a feature selection approach. It works by recursively removing attributes and building a model on those attributes that remain. It uses the model accuracy to identify which attributes (and combination of attributes) contribute the most to predicting the target attribute.

This recipe shows the use of RFE on the Iris flowers dataset to select 3 attributes.

```
# Recursive Feature Elimination
from sklearn import datasets
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression

# load the iris datasets
dataset = datasets.load_iris()
# create a base classifier used to evaluate a subset of
attributes
model = LogisticRegression()
# create the RFE model and select 3 attributes
rfe = RFE(model, 3)
rfe = rfe.fit(dataset.data, dataset.target)
# summarize the selection of the attributes
print(rfe.support_)
print(rfe.ranking_)
```

Feature Importance

Methods that use ensembles of decision trees (like Random Forest or Extra Trees) can also compute the relative importance of each attribute. These importance values can be used to inform a feature selection process.

This recipe shows the construction of an Extra Trees ensemble of the iris flowers dataset and the display of the relative feature importance.

```
# Feature Importance
from sklearn import datasets
from sklearn import metrics
from sklearn.ensemble import ExtraTreesClassifier

# load the iris datasets
dataset = datasets.load_iris()
# fit an Extra Trees model to the data
model = ExtraTreesClassifier()
model.fit(dataset.data, dataset.target)
# display the relative importance of each attribute
print(model.feature_importances_)
```