Python’s Scikit-learn

Nicolás García-Pedrajas

Computational Intelligence and Bioinformatics Research Group

October 23, 2019

Table of contents

Scikit-learn

What is Scikit-Learn?

Extensions to SciPy (Scientific Python) are called SciKits. SciKit-Learn provides machine learning algorithms:

- Algorithms for supervised & unsupervised learning
- Built on SciPy and Numpy
- Standard Python API interface
- Sits on top of c libraries, LAPACK, LibSVM, and Cython
- Open Source: BSD License (part of Linux)

Probably the best general ML framework out there.

Scikit-learn

Where did it come from?

Started as a Google summer of code project in 2007 by David Cournapeau, then used as a thesis project by Matthieu Brucher.

In 2010, INRIA pushed the first public release, and sponsors the project, as do Google, Tinyclues, and the Python Software Foundation.

Scikit-learn

Primary features

- Generalized Linear Models
- SVMs, kNN, Bayes, Decision Trees, Ensembles
- Clustering and Density algorithms
- Cross Validation
- Grid Search
- Pipelining
- Model Evaluations
- Dataset Transformations
- Dataset Loading

Scikit-learn

API

Object-oriented interface centered around the concept of an Estimator:

> *An estimator is any object that learns from data; it may be a classification, regression or clustering algorithm or a transformer that extracts/filters useful features from raw data.*
>
> Scikit-Learn Tutorial

Scikit-learn

Estimator class

Class definition

```
class Estimator(object):

        def fit(self, X, y=None):
                """Fits estimator to data. """
                # set state of ``self``
                return self

        def predict(self, X):
                """Predict response of ``X``. """
                # compute predictions ``pred``
                return pred
```

Scikit-learn

Estimator

Estimators

- fit(X,y) sets the state of the estimator.
- X is usually a 2D numpy array of shape (num_samples, num_features).
- y is a 1D array with shape (n_samples,)
- predict(X) returns the class or value
- predict_proba() returns a 2D array of shape (n_samples, n_classes)

Example:

Estimator (SVM)

```
rom sklearn import svm

stimator = svm.SVC(gamma=0.001)
stimator.fit(X, y)
stimator.predict(x)
```

Scikit-learn

Load and transform data

Load data using appropriate methods.

Transform data:

Transformers

```
class Transformer(Estimator):
        def transform(self, X):
                """Transforms the input data. """
                # transform ``X`` to ``X_prime``
                return X_prime

from sklearn import preprocessing

Xt = preprocessing.normalize(X) # Normalizer
Xt = preprocessing.scale(X)

# StandardScaler: Imputation of missing values
imputer =Imputer(missing_values='Nan', strategy='mean')

Xt = imputer.fit_transform(X)
```

Scikit-learn

Classification models

Scikit-learn includes the following models:

- Generalized linear models.
- Linear an quadratic discriminant analysis
- Support vector machines.
- Nearest neighbors.
- Decision trees.
- Naive Bayes.
- Ensemble methods.
- Neural networks (deep learning with Keras)
- Multi-class methods.
- Multi-label methods.

Decision trees

First step: import required libraries.

Libraries

```
# Load libraries
import pandas as pd
from sklearn.tree import DecisionTreeClassifier # Import Decision Tree Classifier
from sklearn.model_selection import train_test_split # Import train_test_split function
from sklearn import metrics #Import scikit-learn metrics module for accuracy calculation
```

Decision trees

Second step: load data.

Load data

```
# load dataset
col_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin', 'bmi', 'pedigree', 'age', 'label']
pima = pd.read_csv("pima-indians-diabetes.csv", header=None, names=col_names)

# split dataset in features and target variable
feature_cols = ['pregnant', 'glucose', 'bp', 'skin', 'insulin', 'bmi', 'pedigree', 'age']
X = pima[feature_cols] # Features
y = pima.label # Target variable
```

Decision trees

Data partitioning

We can split the data randomly (random partition, problem for reproduction):

Random partition

```
# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
↪  random_state=1) # 70% training and 30% test
```

Or we can have two separate files:

Separate files

```
# load dataset
col_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin', 'bmi',
↪  'pedigree', 'age', 'label']
pima_train = pd.read_csv("pima-indians-diabetes.train.csv", header=None,
↪  names=col_names)
pima_test = pd.read_csv("pima-indians-diabetes.test.csv", header=None,
↪  names=col_names)
```

Decision tree

Building model

The procedure is common for all classification models.

Building decision tree

```
# Create Decision Tree classifer object
clf = DecisionTreeClassifier()

# Train Decision Tree Classifer
clf = clf.fit(X_train,y_train)

#Predict the response for test dataset
y_pred = clf.predict(X_test)
```

Visualizing decision trees

Decision trees can be visualize using `graphviz` and `pydotplus`:

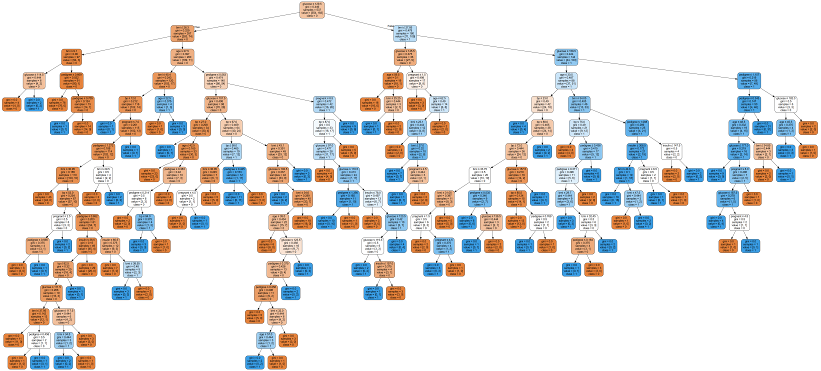
Building decision tree

```
from sklearn.tree import export_graphviz
from sklearn.externals.six import StringIO
from IPython.display import Image
import pydotplus

dot_data = StringIO()
export_graphviz(clf, out_file=dot_data,
                filled=True, rounded=True,
                special_characters=True,feature_names = feature_cols,class_names=['0','1'])
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_png('diabetes.png')
Image(graph.create_png())
```

Visualizing decision trees

Decision tree:



Optimizing the decision tree

There are a few hyper-parameters:

- criterion: optional (default="gini") or Choose attribute selection measure: This parameter allows us to use the different-different attribute selection measure. Supported criteria are"gini" for the Gini index and "entropy" for the information gain.
- splitter: string, optional (default="best") or Split Strategy: This parameter allows us to choose the split strategy. Supported strategies are "best" to choose the best split and "random" to choose the best random split.
- max_depth: int or None, optional (default=None) or Maximum Depth of a Tree: The maximum depth of the tree. If None, then nodes are expanded until all the leaves contain less than min_samples_split samples. The higher value of maximum depth causes overfitting, and a lower value causes underfitting.

Decision trees

Probabilities output

The model can output categories and/or probabilities.

Models output

```
# Get predictions
preds = clf.predict(X_test)
np.savetxt("predictions", preds, fmt='%d')

# Get probabilities
probs = clf.predict_proba(X_test))
np.savetxt("probabilities", probs)
```

Metric module

The evaluation is common for all models.
Metric module implements many classification performance metrics:

```
metrics.accuracy_score(y_true, y_pred[,
↪  …])
metrics.auc(x, y[, reorder])
metrics.average_precision_score(y_true,
↪  y_score)
metrics.balanced_accuracy_score(y_true,
↪  y_pred)
metrics.brier_score_loss(y_true, y_prob[,
↪  …])
metrics.classification_report(y_true,
↪  y_pred)
metrics.cohen_kappa_score(y1, y2[, labels,
↪  …])
metrics.confusion_matrix(y_true, y_pred[,
↪  …])
metrics.f1_score(y_true, y_pred[, labels,
↪  …])
metrics.fbeta_score(y_true, y_pred, beta[,
↪  …])
metrics.hamming_loss(y_true, y_pred[, …])
metrics.hinge_loss(y_true, pred_decision[,
↪  …])
metrics.jaccard_score(y_true, y_pred[, …])
metrics.log_loss(y_true, y_pred[, eps, …])
metrics.matthews_corrcoef(y_true, y_pred[,
↪  …])
metrics.multilabel_confusion_matrix(y_true,
↪  …)
metrics.precision_recall_curve(y_true, …)
metrics.precision_recall_fscore_support(…)
metrics.precision_score(y_true, y_pred[,
↪  …])
metrics.recall_score(y_true, y_pred[, …])
metrics.roc_auc_score(y_true, y_score[,
↪  …])
metrics.roc_curve(y_true, y_score[, …])
metrics.zero_one_loss(y_true, y_pred[, …])
```

Example of evaluation

Once trained and tested the model can be evaluated:

Evaluating the decision tree

```
# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))

Accuracy: 0.6753246753246753
```

The confusion matrix can also be obtained from the predictions:

Confusion matrix for the decision tree

```
cm = confusion_matrix(y_test, y_pred)
```

Evaluating the models

Cross-validation

Python implements cross-validation as an option for evaluating the models
The following example demonstrates how to estimate the accuracy of a linear kernel support vector machine on the iris dataset by splitting the data, fitting a model and computing the score 5 consecutive times (with different splits each time):

k-Fold cross-validation

```
>>> from sklearn.model_selection import cross_val_score
>>> clf = svm.SVC(kernel='linear', C=1)
>>> scores = cross_val_score(clf, iris.data, iris.target, cv=5, scoring='accuracy')
>>> scores
array([0.96..., 1.  ..., 0.96..., 0.96..., 1.        ])
```

Optimizing hyper-parameters

- Hyper-parameters are critic for many classification models (e.g: Support vector machines)
- A step of obtaining the best set of hyper-parameters is usually needed
- Scikit provides a grid search using cross-validation

Optimizing hyper-parameters

Grid search

Grid search

```
#Grid Search
from sklearn.model_selection import GridSearchCV
clf = LogisticRegression()
grid_values = {'penalty': ['l1',
↪  'l2'],'C':[0.001,.009,0.01,.09,1,5,10,25]}
grid_clf_acc = GridSearchCV(clf, param_grid = grid_values,scoring =
↪  'recall')
grid_clf_acc.fit(X_train, y_train)

#Predict values based on new parameters
y_pred_acc = grid_clf_acc.predict(X_test)

# New Model Evaluation metrics
print('Accuracy Score : ' + str(accuracy_score(y_test,y_pred_acc)))
print('Precision Score : ' + str(precision_score(y_test,y_pred_acc)))
print('Recall Score : ' + str(recall_score(y_test,y_pred_acc)))
print('F1 Score : ' + str(f1_score(y_test,y_pred_acc)))

#Logistic Regression (Grid Search) Confusion matrix
confusion_matrix(y_test,y_pred_acc)
```