FREE eBook

LEARNING scikit-learn

Free unaffiliated eBook created from
Stack Overflow contributors.

#scikit-learn

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: scikit-learn

It is an unofficial and free scikit-learn ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official scikit-learn.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with scikit-learn

Remarks

`scikit-learn` is a general-purpose open-source library for data analysis written in python. It is based on other python libraries: NumPy, SciPy, and matplotlib

`scikit-learn`contains a number of implementation for different popular algorithms of machine learning.

Examples

Installation of scikit-learn

The current stable version of scikit-learn requires:

- Python (>= 2.6 or >= 3.3),
- NumPy (>= 1.6.1),
- SciPy (>= 0.9).

For most installation `pip` python package manager can install python and all of its dependencies:

```
pip install scikit-learn
```

However for linux systems it is recommended to use `conda` package manager to avoid possible build processes

```
conda install scikit-learn
```

To check that you have `scikit-learn`, execute in shell:

```
python -c 'import sklearn; print(sklearn.__version__)'
```

Windows and Mac OSX Installation:

Canopy and Anaconda both ship a recent version of *scikit-learn*, in addition to a large set of scientific python library for Windows, Mac OSX (also relevant for Linux).

Train a classifier with cross-validation

Using iris dataset:

```
import sklearn.datasets
iris_dataset = sklearn.datasets.load_iris()
```

```
X, y = iris_dataset['data'], iris_dataset['target']
```

Data is split into train and test sets. To do this we use the `train_test_split` utility function to split both `X` and `y` (data and target vectors) randomly with the option `train_size=0.75` (training sets contain 75% of the data).

Training datasets are fed into a k-nearest neighbors classifier. The method `fit` of the classifier will fit the model to the data.

```
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.75)
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=3)
clf.fit(X_train, y_train)
```

Finally predicting quality on test sample:

```
clf.score(X_test, y_test) # Output: 0.94736842105263153
```

By using one pair of train and test sets we might get a biased estimation of the quality of the classifier due to the arbitrary choice the data split. By using *cross-validation* we can fit of the classifier on different train/test subsets of the data and make an average over all accuracy results. The function `cross_val_score` fits a classifier to the input data using cross-validation. It can take as input the number of different splits (folds) to be used (5 in the example below).

```
from sklearn.cross_validation import cross_val_score
scores = cross_val_score(clf, X, y, cv=5)
print(scores)
# Output: array([ 0.96666667,  0.96666667,  0.93333333,  0.96666667,  1.        ])
print "Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() / 2)
# Output: Accuracy: 0.97 (+/- 0.03)
```

Creating pipelines

Finding patterns in data often proceeds in a chain of data-processing steps, e.g., feature selection, normalization, and classification. In `sklearn`, a pipeline of stages is used for this.

For example, the following code shows a pipeline consisting of two stages. The first scales the features, and the second trains a classifier on the resulting augmented dataset:

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier

pipeline = make_pipeline(StandardScaler(), KNeighborsClassifier(n_neighbors=4))
```

Once the pipeline is created, you can use it like a regular stage (depending on its specific steps). Here, for example, the pipeline behaves like a classifier. Consequently, we can use it as follows:

```
# fitting a classifier
```

```
pipeline.fit(X_train, y_train)
# getting predictions for the new data sample
pipeline.predict_proba(X_test)
```

Interfaces and conventions:

Different operations with data are done using special classes.

Most of the classes belong to one of the following groups:

- classification algorithms (derived from `sklearn.base.ClassifierMixin`) to solve classification problems
- regression algorithms (derived from `sklearn.base.RegressorMixin`) to solve problem of reconstructing continuous variables (regression problem)
- data transformations (derived from `sklearn.base.TransformerMixin`) that preprocess the data

Data is stored in `numpy.array`s (but other array-like objects like `pandas.DataFrame`s are accepted if those are convertible to `numpy.array`s)

Each object in the data is described by set of features the general convention is that data sample is represented with array, where first dimension is data sample id, second dimension is feature id.

```
import numpy
data = numpy.arange(10).reshape(5, 2)
print(data)

Output:
[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
```

In `sklearn` conventions dataset above contains 5 objects each described by 2 features.

Sample datasets

For ease of testing, `sklearn` provides some built-in datasets in `sklearn.datasets` module. For example, let's load Fisher's iris dataset:

```
import sklearn.datasets
iris_dataset = sklearn.datasets.load_iris()
iris_dataset.keys()
['target_names', 'data', 'target', 'DESCR', 'feature_names']
```

You can read full description, names of features and names of classes (`target_names`). Those are stored as strings.

We are interested in the data and classes, which stored in `data` and `target` fields. By convention those are denoted as `X` and `y`

```
X, y = iris_dataset['data'], iris_dataset['target']
X.shape, y.shape
((150, 4), (150,))
```

```
numpy.unique(y)
array([0, 1, 2])
```

Shapes of `X` and `y` say that there are 150 samples with 4 features. Each sample belongs to one of following classes: 0, 1 or 2.

`X` and `y` can now be used in training a classifier, by calling the classifier's `fit()` method.

Here is the full list of datasets provided by the `sklearn.datasets` module with their size and intended use:

Load with	Description	Size	Usage
`load_boston()`	Boston house-prices dataset	506	regression
`load_breast_cancer()`	Breast cancer Wisconsin dataset	569	classification (binary)
`load_diabetes()`	Diabetes dataset	442	regression
`load_digits(n_class)`	Digits dataset	1797	classification
`load_iris()`	Iris dataset	150	classification (multi-class)
`load_linnerud()`	Linnerud dataset	20	multivariate regression

Note that (source: http://scikit-learn.org/stable/datasets/):

> These datasets are useful to quickly illustrate the behavior of the various algorithms implemented in the scikit. They are however often too small to be representative of real world machine learning tasks.

In addition to these built-in toy sample datasets, `sklearn.datasets` also provides utility functions for loading external datasets:

- `load_mlcomp` for loading sample datasets from the mlcomp.org repository (note that the datasets need to be downloaded before). Here is an example of usage.
- `fetch_lfw_pairs` and `fetch_lfw_people` for loading Labeled Faces in the Wild (LFW) pairs dataset from http://vis-www.cs.umass.edu/lfw/, used for face verification (resp. face recognition). This dataset is larger than 200 MB. Here is an example of usage.

Read Getting started with scikit-learn online: https://riptutorial.com/scikit-learn/topic/1035/getting-started-with-scikit-learn

Chapter 2: Classification

Examples

Using Support Vector Machines

Support vector machines is a family of algorithms attempting to pass a (possibly high-dimension) hyperplane between two labelled sets of points, such that the distance of the points from the plane is optimal in some sense. SVMs can be used for classification or regression (corresponding to `sklearn.svm.SVC` and `sklearn.svm.SVR`, respectively.

Example:

Suppose we work in a 2D space. First, we create some data:

```
import numpy as np
```

Now we create *x* and *y*:

```
x0, x1 = np.random.randn(10, 2), np.random.randn(10, 2) + (1, 1)
x = np.vstack((x0, x1))

y = [0] * 10 + [1] * 10
```

Note that *x* is composed of two Gaussians: one centered around *(0, 0)*, and one centered around *(1, 1)*.

To build a classifier, we can use:

```
from sklearn import svm

svm.SVC(kernel='linear').fit(x, y)
```

Let's check the prediction for *(0, 0)*:

```
>>> svm.SVC(kernel='linear').fit(x, y).predict([[0, 0]])
array([0])
```

The prediction is that the class is 0.

For regression, we can similarly do:

```
svm.SVR(kernel='linear').fit(x, y)
```

RandomForestClassifier

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-

samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting.

A simple usage example:

Import:

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

Define train data and target data:

```
train = [[1,2,3],[2,5,1],[2,1,7]]
target = [0,1,0]
```

The values in `target` represent the label you want to predict.

Initiate a RandomForest object and perform learn (fit):

```
rf = RandomForestClassifier(n_estimators=100)
rf.fit(train, target)
```

Predict:

```
test = [2,2,3]
predicted = rf.predict(test)
```

Analyzing Classification Reports

Build a text report showing the main classification metrics, including the precision and recall, f1-score (the harmonic mean of precision and recall) and support (the number of observations of that class in the training set).

Example from `sklearn` docs:

```
from sklearn.metrics import classification_report
y_true = [0, 1, 2, 2, 2]
y_pred = [0, 0, 2, 2, 1]
target_names = ['class 0', 'class 1', 'class 2']
print(classification_report(y_true, y_pred, target_names=target_names))
```

Output -

```
             precision    recall  f1-score   support

    class 0       0.50      1.00      0.67         1
    class 1       0.00      0.00      0.00         1
    class 2       1.00      0.67      0.80         3

avg / total       0.70      0.60      0.61         5
```

GradientBoostingClassifier

Gradient Boosting for classification. The Gradient Boosting Classifier is an additive ensemble of a base model whose error is corrected in successive iterations (or stages) by the addition of Regression Trees which correct the residuals (the error of the previous stage).

Import:

```
from sklearn.ensemble import GradientBoostingClassifier
```

Create some toy classification data

```
from sklearn.datasets import load_iris

iris_dataset = load_iris()

X, y = iris_dataset.data, iris_dataset.target
```

Let us split this data into training and testing set.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=0)
```

Instantiate a `GradientBoostingClassifier` model using the default params.

```
gbc = GradientBoostingClassifier()
gbc.fit(X_train, y_train)
```

Let us score it on the test set

```
# We are using the default classification accuracy score
>>> gbc.score(X_test, y_test)
1
```

By default there are 100 estimators built

```
>>> gbc.n_estimators
100
```

This can be controlled by setting `n_estimators` to a different value during the initialization time.

A Decision Tree

A decision tree is a classifier which uses a sequence of verbose rules (like a>7) which can be easily understood.

The example below trains a decision tree classifier using three feature vectors of length 3, and then predicts the result for a so far unknown fourth feature vector, the so called test vector.

```
from sklearn.tree import DecisionTreeClassifier

# Define training and target set for the classifier
train = [[1,2,3],[2,5,1],[2,1,7]]
target = [10,20,30]

# Initialize Classifier.
# Random values are initialized with always the same random seed of value 0
# (allows reproducible results)
dectree = DecisionTreeClassifier(random_state=0)
dectree.fit(train, target)

# Test classifier with other, unknown feature vector
test = [2,2,3]
predicted = dectree.predict(test)

print predicted
```

Output can be visualized using:

```
import pydot
import StringIO

dotfile = StringIO.StringIO()
tree.export_graphviz(dectree, out_file=dotfile)
(graph,)=pydot.graph_from_dot_data(dotfile.getvalue())
graph.write_png("dtree.png")
graph.write_pdf("dtree.pdf")
```

Classification using Logistic Regression

In LR Classifier, he probabilities describing the possible outcomes of a single trial are modeled using a logistic function. It is implemented in the `linear_model` library

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

The sklearn LR implementation can fit binary, One-vs- Rest, or multinomial logistic regression with optional L2 or L1 regularization. For example, let us consider a binary classification on a sample sklearn dataset

```
from sklearn.datasets import make_hastie_10_2

X,y = make_hastie_10_2(n_samples=1000)
```

Where X is a `n_samples X 10` array and y is the target labels -1 or +1.

Use train-test split to divide the input data into training and test sets (70%-30%)

```
from sklearn.model_selection import train_test_split
#sklearn.cross_validation in older scikit versions

data_train, data_test, labels_train, labels_test = train_test_split(X,y, test_size=0.3)
```

Using the LR Classifier is similar to other examples

```
# Initialize Classifier.
LRC = LogisticRegression()
LRC.fit(data_train, labels_train)

# Test classifier with the test data
predicted = LRC.predict(data_test)
```

Use Confusion matrix to visualise results

```
from sklearn.metrics import confusion_matrix

confusion_matrix(predicted, labels_test)
```

Read Classification online: https://riptutorial.com/scikit-learn/topic/2468/classification

Chapter 3: Dimensionality reduction (Feature selection)

Examples

Reducing The Dimension With Principal Component Analysis

Principal Component Analysis finds sequences of linear combinations of the features. The first linear combination maximizes the variance of the features (subject to a unit constraint). Each of the following linear combinations maximizes the variance of the features in the subspace orthogonal to that spanned by the previous linear combinations.

A common dimension reduction technique is to use only the k first such linear combinations. Suppose the features are a matrix X of n rows and m columns. The first k linear combinations form a matrix β_k of m rows and k columns. The product $X\,\beta$ has n rows and k columns. Thus, the resulting matrix $\beta\,k$ can be considered a reduction from m to k dimensions, retaining the high-variance parts of the original matrix X.

In `scikit-learn`, PCA is performed with `sklearn.decomposition.PCA`. For example, suppose we start with a 100 X 7 matrix, constructed so that the variance is contained only in the first two columns (by scaling down the last 5 columns):

```
import numpy as np
np.random.seed(123) # we'll set a random seed so that our results are reproducible
X = np.hstack((np.random.randn(100, 2) + (10, 10), 0.001 * np.random.randn(100, 5)))
```

Let's perform a reduction to 2 dimensions:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pca.fit(X)
```

Now let's check the results. First, here are the linear combinations:

```
pca.components_
# array([[ -2.84271217e-01,  -9.58743893e-01,  -8.25412629e-05,
#           1.96237855e-05,  -1.25862328e-05,   8.27127496e-05,
#          -9.46906600e-05],
#        [ -9.58743890e-01,   2.84271223e-01,  -7.33055823e-05,
#          -1.23188872e-04,  -1.82458739e-05,   5.50383246e-05,
#           1.96503690e-05]])
```

Note how the first two components in each vector are several orders of magnitude larger than the others, showing that the PCA recognized that the variance is contained mainly in the first two columns.

To check the ratio of the variance explained by this PCA, we can examine `pca.explained_variance_ratio_`:

```
pca.explained_variance_ratio_
# array([ 0.57039059,  0.42960728])
```

Read Dimensionality reduction (Feature selection) online: https://riptutorial.com/scikit-learn/topic/4829/dimensionality-reduction--feature-selection-

Chapter 4: Feature selection

Examples

Low-Variance Feature Removal

This is a very basic feature selection technique.

Its underlying idea is that if a feature is constant (i.e. it has 0 variance), then it cannot be used for finding any interesting patterns and can be removed from the dataset.

Consequently, a heuristic approach to feature elimination is to first remove all features whose variance is below some (low) threshold.

Building off the example in the documentation, suppose we start with

```
X = [[0, 0, 1], [0, 1, 0], [1, 0, 0], [0, 1, 1], [0, 1, 0], [0, 1, 1]]
```

There are *3* boolean features here, each with *6* instances. Suppose we wish to remove those that are constant in at least 80% of the instances. Some probability calculations show that these features will need to have variance lower than *0.8 * (1 - 0.8)*. Consequently, we can use

```
from sklearn.feature_selection import VarianceThreshold
sel = VarianceThreshold(threshold=(.8 * (1 - .8)))
sel.fit_transform(X)
# Output: array([[0, 1],
                 [1, 0],
                 [0, 0],
                 [1, 1],
                 [1, 0],
                 [1, 1]])
```

Note how the first feature was removed.

This method should be used with caution because a low variance doesn't necessarily mean that a feature is “uninteresting”. Consider the following example where we construct a dataset that contains *3* features, the first two consisting of randomly distributed variables and the third of uniformly distributed variables.

```
from sklearn.feature_selection import VarianceThreshold
import numpy as np

# generate dataset
np.random.seed(0)

feat1 = np.random.normal(loc=0, scale=.1, size=100) # normal dist. with mean=0 and std=.1
feat2 = np.random.normal(loc=0, scale=10, size=100) # normal dist. with mean=0 and std=10
feat3 = np.random.uniform(low=0, high=10, size=100) # uniform dist. in the interval [0,10)
data = np.column_stack((feat1,feat2,feat3))
```

```
data[:5]
# Output:
# array([[  0.17640523,  18.83150697,   9.61936379],
#        [  0.04001572, -13.47759061,   2.92147527],
#        [  0.0978738 , -12.70484998,   2.4082878 ],
#        [  0.22408932,   9.69396708,   1.00293942],
#        [  0.1867558 , -11.73123405,   0.1642963 ]])

np.var(data, axis=0)
# Output: array([  1.01582662e-02,   1.07053580e+02,   9.07187722e+00])

sel = VarianceThreshold(threshold=0.1)
sel.fit_transform(data)[:5]
# Output:
# array([[ 18.83150697,   9.61936379],
#        [-13.47759061,   2.92147527],
#        [-12.70484998,   2.4082878 ],
#        [  9.69396708,   1.00293942],
#        [-11.73123405,   0.1642963 ]])
```

Now the first feature has been removed because of its low variance, while the third feature (that's the most uninteresting) has been kept. In this case it would have been more appropriate to consider a *coefficient of variation* because that's independent of scaling.

Read Feature selection online: https://riptutorial.com/scikit-learn/topic/4909/feature-selection

Chapter 5: Model selection

Examples

Cross-validation

Learning the parameters of a prediction function and testing it on the same data is a methodological mistake: a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data. This situation is called **overfitting**. To avoid it, it is common practice when performing a (supervised) machine learning experiment to hold out part of the available data as a **test set** `X_test, y_test`. Note that the word “experiment” is not intended to denote academic use only, because even in commercial settings machine learning usually starts out experimentally.

In scikit-learn a random split into training and test sets can be quickly computed with the train_test_split helper function. Let’s load the iris data set to fit a linear support vector machine on it:

```
>>> import numpy as np
>>> from sklearn import cross_validation
>>> from sklearn import datasets
>>> from sklearn import svm

>>> iris = datasets.load_iris()
>>> iris.data.shape, iris.target.shape
((150, 4), (150,))
```

We can now quickly sample a training set while holding out 40% of the data for testing (evaluating) our classifier:

```
>>> X_train, X_test, y_train, y_test = cross_validation.train_test_split(
...     iris.data, iris.target, test_size=0.4, random_state=0)

>>> X_train.shape, y_train.shape
((90, 4), (90,))
>>> X_test.shape, y_test.shape
((60, 4), (60,))
```

Now, after we have train and test sets, lets use it:

```
>>> clf = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
```

K-Fold Cross Validation

K-fold cross-validation is a systematic process for repeating the train/test split procedure multiple times, in order to reduce the variance associated with a single trial of train/test split. You essentially split the entire dataset into K equal size "folds", and each fold is used once for testing

the model and K-1 times for training the model.

Multiple folding techniques are available with the scikit library. Their usage is dependent on the input data characteristics. Some examples are

K-Fold

You essentially split the entire dataset into K equal size "folds", and each fold is used once for testing the model and K-1 times for training the model.

```
from sklearn.model_selection import KFold
X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
y = np.array([1, 2, 1, 2])
cv = KFold(n_splits=3, random_state=0)

for train_index, test_index in cv.split(X):
...     print("TRAIN:", train_index, "TEST:", test_index)

TRAIN: [2 3] TEST: [0 1]
TRAIN: [0 1 3] TEST: [2]
TRAIN: [0 1 2] TEST: [3]
```

`StratifiedKFold` is a variation of k-fold which returns stratified folds: each set contains approximately the same percentage of samples of each target class as the complete set

ShuffleSplit

Used to generate a user defined number of independent train / test dataset splits. Samples are first shuffled and then split into a pair of train and test sets.

```
from sklearn.model_selection import ShuffleSplit
X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
y = np.array([1, 2, 1, 2])
cv = ShuffleSplit(n_splits=3, test_size=.25, random_state=0)

for train_index, test_index in cv.split(X):
...     print("TRAIN:", train_index, "TEST:", test_index)

TRAIN: [3 1 0] TEST: [2]
TRAIN: [2 1 3] TEST: [0]
TRAIN: [0 2 1] TEST: [3]
```

`StratifiedShuffleSplit` is a variation of ShuffleSplit, which returns stratified splits, i.e which creates splits by preserving the same percentage for each target class as in the complete set.

Other folding techniques such as Leave One/p Out, and TimeSeriesSplit (a variation of K-fold) are available in the scikit model_selection library.

Read Model selection online: https://riptutorial.com/scikit-learn/topic/4901/model-selection

Chapter 6: Receiver Operating Characteristic (ROC)

Examples

Introduction to ROC and AUC

Example of Receiver Operating Characteristic (ROC) metric to evaluate classifier output quality.

ROC curves typically feature true positive rate on the Y axis, and false positive rate on the X axis. This means that the top left corner of the plot is the “ideal” point - a false positive rate of zero, and a true positive rate of one. This is not very realistic, but it does mean that a larger area under the curve (AUC) is usually better.

The “steepness” of ROC curves is also important, since it is ideal to maximize the true positive rate while minimizing the false positive rate.

A simple example:

```
import numpy as np
from sklearn import metrics
import matplotlib.pyplot as plt
```

Arbitrary `y` values - in real case this is the predicted target values (`model.predict(x_test)`):

```
y = np.array([1,1,2,2,3,3,4,4,2,3])
```

Scores is the mean accuracy on the given test data and labels (`model.score(X,Y)`):

```
scores = np.array([0.3, 0.4, 0.95,0.78,0.8,0.64,0.86,0.81,0.9, 0.8])
```

Calculate the ROC curve and the AUC:

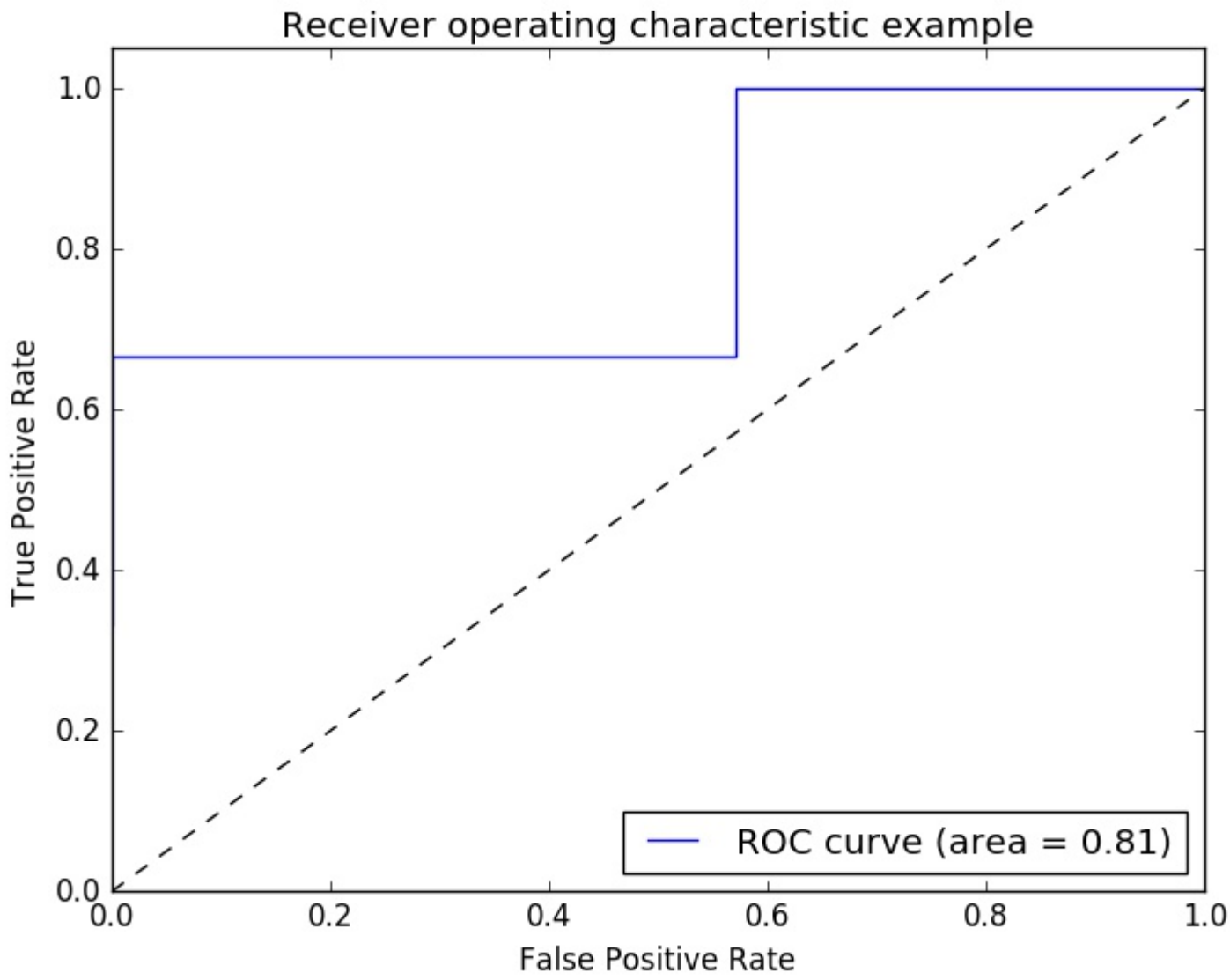
```
fpr, tpr, thresholds = metrics.roc_curve(y, scores, pos_label=2)
roc_auc = metrics.auc(fpr, tpr)
```

Plotting:

```
plt.figure()
plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic example')
plt.legend(loc="lower right")
```

```
plt.show()
```

Output:



Note: the sources were taken from these link1 and link2

ROC-AUC score with overriding and cross validation

One needs the predicted probabilities in order to calculate the ROC-AUC (area under the curve) score. The `cross_val_predict` uses the `predict` methods of classifiers. In order to be able to get the ROC-AUC score, one can simply subclass the classifier, overriding the `predict` method, so that it would act like `predict_proba`.

```
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.cross_validation import cross_val_predict
from sklearn.metrics import roc_auc_score

class LogisticRegressionWrapper(LogisticRegression):
```

```
    def predict(self, X):
        return super(LogisticRegressionWrapper, self).predict_proba(X)

X, y = make_classification(n_samples = 1000, n_features=10, n_classes = 2, flip_y = 0.5)

log_reg_clf = LogisticRegressionWrapper(C=0.1, class_weight=None, dual=False,
             fit_intercept=True)

y_hat = cross_val_predict(log_reg_clf, X, y)[:,1]

print("ROC-AUC score: {}".format(roc_auc_score(y, y_hat)))
```

output:

```
ROC-AUC score: 0.724972396025
```

Read Receiver Operating Characteristic (ROC) online: https://riptutorial.com/scikit-learn/topic/5945/receiver-operating-characteristic--roc-

Chapter 7: Regression

Examples

Ordinary Least Squares

Ordinary Least Squares is a method for finding the linear combination of features that best fits the observed outcome in the following sense.

If the vector of outcomes to be predicted is y, and the explanatory variables form the matrix X, then OLS will find the vector β solving

$min_\beta |y\hat{} - y|_2^2$,

where $y\hat{} = X\beta$ is the linear prediction.

In sklearn, this is done using `sklearn.linear_model.LinearRegression`.

Application Context

OLS should only be applied to regression problems, it is generally unsuitable for classification problems: Contrast

- Is an email spam? (Classfication)
- What is the linear relationship between upvotes depend on the length of answer? (Regression)

Example

Let's generate a linear model with some noise, then see if `LinearRegression` Manages to reconstruct the linear model.

First we generate the `X` matrix:

```
import numpy as np

X = np.random.randn(100, 3)
```

Now we'll generate the `y` as a linear combination of `X` with some noise:

```
beta = np.array([[1, 1, 0]])
y = (np.dot(x, beta.T) + 0.01 * np.random.randn(100, 1))[:, 0]
```

Note that the true linear combination generating `y` is given by `beta.

To try to reconstruct this from `X` and `y` alone, let's do:

```
>>> linear_model.LinearRegression().fit(x, y).coef_
```

```
array([  9.97768469e-01,   9.98237634e-01,   7.55016533e-04])
```

Note that this vector is very similar to `beta`.

Read Regression online: https://riptutorial.com/scikit-learn/topic/5190/regression

Credits

S. No	Chapters	Contributors
1	Getting started with scikit-learn	Alleo, Ami Tavory, Community, Gabe, Gal Dreiman, panty, Sean Easter, user2314737
2	Classification	Ami Tavory, Drew, Gal Dreiman, hashcode55, Mechanic, Raghav RV, Sean Easter, tfv, user6903745, Wayne Werner
3	Dimensionality reduction (Feature selection)	Ami Tavory, DataSwede, Gal Dreiman, Sean Easter, user2314737
4	Feature selection	Ami Tavory, user2314737
5	Model selection	Gal Dreiman, Mechanic
6	Receiver Operating Characteristic (ROC)	Gal Dreiman, Gorkem Ozkaya
7	Regression	Ami Tavory, draco_alpine