Community Experience Distilled

Learning scikit-learn: Machine Learning in Python

Experience the benefits of machine learning techniques by applying them to real-world problems using Python and the open source scikit-learn library

Raúl Garreta
Guillermo Moncecchi

[PACKT] open source*
PUBLISHING
community experience distilled

Learning scikit-learn: Machine Learning in Python

Experience the benefits of machine learning techniques by applying them to real-world problems using Python and the open source scikit-learn library

Raúl Garreta

Guillermo Moncecchi

[PACKT] open source
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

Learning scikit-learn: Machine Learning in Python

First published: November 2013

Production Reference: 1181113

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-193-0

`www.packtpub.com`

Cover Image by Faiz Fattohi (`faizfattohi@gmail.com`)

This shows that the dictionary is composed of 61236 tokens. Let's print the feature names.

```
>>> clf_7.named_steps['vect'].get_feature_names()
```

The following table presents an extract of the results:

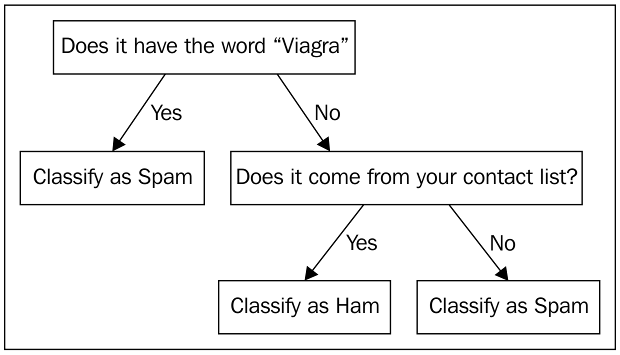
Extract of features obtained by vectorizer	
`u''sanctuaries'',`	`u''sanderson'',`
`u''sanctuary'',`	`u''sandia'',`
`u''sanctum'',`	`u''sandiego.ncr.com'',`
`u''sand'',`	`u''sanding'',`
`u''sandals'',`	`u''sandlak'',`
`u''sandbags'',`	`u''sandman.caltech.edu'',`
`u''sandberg'',`	`u''sandman.ece.clarkson.edu'',`
`u''sandblasting'',`	`u''sandra'',`
`u''sanders'',`	`u''sandro'',`
	`u''sands''`

You can see that some words are semantically very similar, for example, sand and sands, sanctuaries and sanctuary. Perhaps if the plurals and the singulars are counted to the same bucket, we would better represent the documents. This is a very common task, which could be solved using stemming, a technique that relates two words having the same lexical root.

Explaining Titanic hypothesis with decision trees

A common argument against linear classifiers and against statistical learning methods is that it is difficult to explain how the built model decides its predictions for the target classes. If you have a highly dimensional SVM, it is impossible for a human being to even imagine how the hyperplane built looks like. A Naïve Bayes classifier will tell you something like: "this class is the most probable, assuming it comes from a similar distribution as the training data, and making a few more assumptions" something not very useful, for example, we want to know why this or that mail should be considered as spam.

decision trees are very simple yet powerful supervised learning methods, which constructs a decision tree model, which will be used to make predictions. The following figure shows a very simple decision tree to decide if an e-mail should be considered spam:


Does it have the word "Viagra"
Yes
No
Classify as Spam
Does it come from your contact list?
Yes
No
Classify as Ham
Classify as Spam

It first asks if the e-mail contains the word **Viagra**; if the answer is yes, it classifies it as spam; if the answer is no, it further asks if it comes from somebody in your contacts list; this time, if the answer is yes, it classifies the e-mail as Ham; if the answer is no, it classify it as spam. The main advantage of this model is that a human being can easily understand and reproduce the sequence of decisions (especially if the number of attributes is small) taken to predict the target class of a new instance. This is very important for tasks such as medical diagnosis or credit approval, where we want to show a reason for the decision, rather than just saying this is what the training data suggests (which is, by definition, what every supervised learning method does). In this section, we will show you through a working example what decision trees look like, how they are built, and how they are used for prediction.

The problem we would like to solve is to determine if a Titanic's passenger would have survived, given her age, passenger class, and sex. We will work with the Titanic dataset that can be downloaded from `http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/titanic.txt`. Like every other example in this chapter, we start with a dataset that includes the list of Titanic's passengers and a feature indicating whether they survived or not. Each instance in the dataset has the following form:

```
"1","1st",1,"Allen, Miss Elisabeth Walton",29.0000,"Southampton","St
Louis, MO","B-5","24160 L221","2","female"
```

The list of attributes is: `Ordinal`, `Class`, `Survived` (`0`=no, `1`=yes), `Name`, `Age`, `Port of Embarkation`, `Home/Destination`, `Room`, `Ticket`, `Boat`, and `Sex`. We will start by loading the dataset into a `numpy` array.

```
>>> import csv
>>> import numpy as np
>>> with open('data/titanic.csv', 'rb') as csvfile:
>>>     titanic_reader = csv.reader(csvfile, delimiter=',',
>>>     quotechar='"')
>>>
>>>     # Header contains feature names
>>>     row = titanic_reader.next()
>>>     feature_names = np.array(row)
>>>
>>>     # Load dataset, and target classes
>>>     titanic_X, titanic_y = [], []
>>>     for row in titanic_reader:
>>>         titanic_X.append(row)
>>>         titanic_y.append(row[2]) # The target value is
            "survived"
>>>
>>>     titanic_X = np.array(titanic_X)
>>>     titanic_y = np.array(titanic_y)
```

The code shown uses the Python `csv` module to load the data.

```
>>> print feature_names
['row.names' 'pclass' 'survived' 'name' 'age' 'embarked' 'home.dest'
'room' 'ticket' 'boat' 'sex']

>>> print titanic_X[0], titanic_y[0]
['1' '1st' '1' 'Allen, Miss Elisabeth Walton' '29.0000' 'Southampton'
'St Louis, MO' 'B-5' '24160 L221' '2' 'female'] 1
```

Preprocessing the data

The first step we must take is to select the attributes we will use for learning:

```
>>> # we keep class, age and sex
>>> titanic_X = titanic_X[:, [1, 4, 10]]
>>> feature_names = feature_names[[1, 4, 10]]
```

We have selected feature numbers 1, 4, and 10 that is class, age, and sex, based on the assumption that the remaining attributes have no effect on the passenger's survival. Feature selection is an extremely important step while creating a machine learning solution. If the algorithm does not have good features as input, it will not have good enough material to learn from, results won't be good, no matter even if we have the best machine learning algorithm ever designed.

Sometimes the feature selection will be made manually, based on our knowledge of the problem's domain and the machine learning method we are planning to use. Sometimes feature selection may be done by using automatic tools to evaluate and select the most promising ones. In *Chapter 4*, *Advanced Features*, we will talk a bit about these techniques, but for now, we will manually select our attributes. Very specific attributes (such as Name in our case) could result in overfitting (consider a tree that just asks if the name is X, she survived); attributes where there is a small number of instances with each value, present a similar problem (they might not be useful for generalization). We will use class, age, and sex because a priori, we expect them to have influenced the passenger's survival.

Now, our learning data looks like:

```
>>> print feature_names
['pclass' 'age' 'sex']

>>> print titanic_X[12],titanic_y[12]
['1st' 'NA' 'female'] 1
```

We have shown instance number 12 because it poses a problem to solve; one of its features (the age) is not available. We have **missing values**, a usual problem with datasets. In this case, we decided to substitute missing values with the mean age in the training data. We could have taken a different approach, for example, using the most common value in the training data, or the median value. When we substitute missing values, we have to understand that we are modifying the original problem, so we have to be very careful with what we are doing. This is a general rule in machine learning; when we change data, we should have a clear idea of what we are changing, to avoid skewing the final results.

```
>>> # We have missing values for age
>>> # Assign the mean value
>>> ages = titanic_X[:, 1]
>>> mean_age = np.mean(titanic_X[ages != 'NA',
    1].astype(np.float))
>>> titanic_X[titanic_X[:, 1] == 'NA', 1] = mean_age
```

The implementation of decision trees in scikit-learn expects as input a list of real-valued features, and the decision rules of the model would be of the form:

```
Feature <= value
```

For example, `age <= 20.0`. Our attributes (except for age) are categorical; that is, they correspond to a value taken from a discrete set such as male and female. So, we have to convert categorical data into real values. Let's start with the sex feature. The preprocessing module of scikit-learn includes a `LabelEncoder` class, whose `fit` method allows conversion of a categorical set into a `0..K-1` integer, where `K` is the number of different classes in the set (in the case of sex, just 0 or 1):

```
>>> # Encode sex
>>> from sklearn.preprocessing import LabelEncoder
>>> enc = LabelEncoder()
>>> label_encoder = enc.fit(titanic_X[:, 2])
>>> print "Categorical classes:", label_encoder.classes_
Categorical classes: ['female' 'male']

>>> integer_classes =
    label_encoder.transform(label_encoder.classes_)
>>> print "Integer classes:", integer_classes
Integer classes: [0 1]

>>> t = label_encoder.transform(titanic_X[:, 2])
>>> titanic_X[:, 2] = t
```

The last two sentences transform the values of the sex attribute into `0-1` values, and modify the training set.

```
print feature_names
['pclass' 'age' 'sex']

print titanic_X[12], titanic_y[12]
['1st' '31.1941810427' '0'] 1
```

We still have a categorical attribute: `class`. We could use the same approach and convert its three classes into 0, 1, and 2. This transformation implicitly introduces an ordering between classes, something that is not an issue in our problem. However, we will try a more general approach that does not assume an ordering, and it is widely used to convert categorical classes into real-valued attributes. We will introduce an additional encoder and convert the class attributes into three new binary features, each of them indicating if the instance belongs to a feature value `(1)` or `(0)`. This is called **one hot encoding**, and it is a very common way of managing categorical attributes for real-based methods:

```
>>> from sklearn.preprocessing import OneHotEncoder
>>>
>>> enc = LabelEncoder()
>>> label_encoder = enc.fit(titanic_X[:, 0])
>>> print "Categorical classes:", label_encoder.classes_
Categorical classes: ['1st' '2nd' '3rd']

>>> integer_classes =
    label_encoder.transform(label_encoder.classes_).reshape(3, 1)
>>> print "Integer classes:", integer_classes
Integer classes: [[0] [1] [2]]

>>> enc = OneHotEncoder()
>>> one_hot_encoder = enc.fit(integer_classes)
>>> # First, convert classes to 0-(N-1) integers using
    label_encoder
>>> num_of_rows = titanic_X.shape[0]
>>> t = label_encoder.transform(titanic_X[:,
    0]).reshape(num_of_rows, 1)
>>> # Second, create a sparse matrix with three columns, each one
    indicating if the instance belongs to the class
>>> new_features = one_hot_encoder.transform(t)
>>> # Add the new features to titanix_X
>>> titanic_X = np.concatenate([titanic_X,
    new_features.toarray()], axis = 1)
>>> #Eliminate converted columns
>>> titanic_X = np.delete(titanic_X, [0], 1)
>>> # Update feature names
>>> feature_names = ['age', 'sex', 'first_class', 'second_class',
    'third_class']
>>> # Convert to numerical values
>>> titanic_X = titanic_X.astype(float)
>>> titanic_y = titanic_y.astype(float)
```

The preceding code first converts the classes into integers and then uses the `OneHotEncoder` class to create the three new attributes that are added to the array of features. It finally eliminates from training data the original `class` feature.

```
>>> print feature_names
['age', 'sex', 'first_class', 'second_class', 'third_class']

>>> print titanic_X[0], titanic_y[0]
[29.   0.   1.   0.   0.] 1.0
```

We have now a suitable learning set for scikit-learn to learn a decision tree. Also, standardization is not an issue for decision trees because the relative magnitude of features does not affect the classifier performance.

The preprocessing step is usually underestimated in machine learning methods, but as we can see even in this very simple example, it can take some time to make data look as our methods expect. It is also very important in the overall machine learning process; if we fail in this step (for example, incorrectly encoding attributes, or selecting the wrong features), the following steps will fail, no matter how good the method we use for learning.

Training a decision tree classifier

Now to the interesting part; let's build a decision tree from our training data. As usual, we will first separate training and testing data.

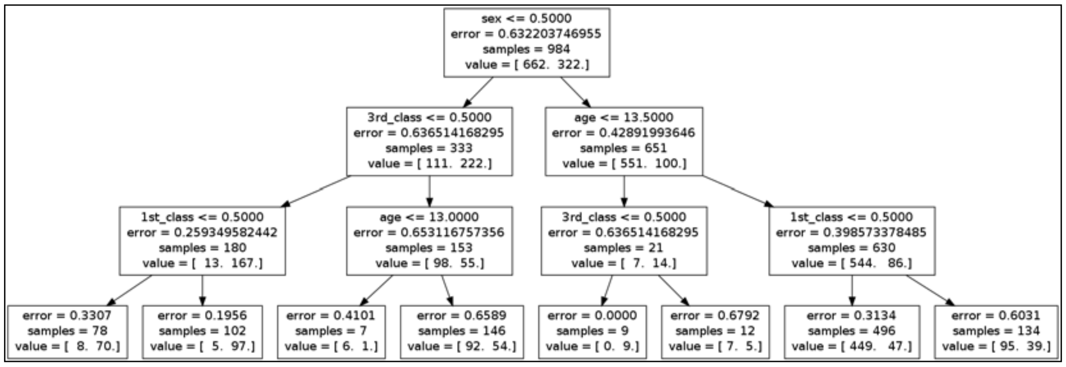
```
>>> from sklearn.cross_validation import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(titanic_X, >>>
titanic_y, test_size=0.25, random_state=33)
```

Now, we can create a new `DecisionTreeClassifier` and use the `fit` method of the classifier to do the learning job.

```
>>> from sklearn import tree
>>> clf = tree.DecisionTreeClassifier(criterion='entropy',
    max_depth=3,min_samples_leaf=5)
>>> clf = clf.fit(X_train,y_train)
```

`DecisionTreeClassifier` accepts (as most learning methods) several hyperparameters that control its behavior. In this case, we used the **Information Gain** (**IG**) criterion for splitting learning data, told the method to build a tree of at most three levels, and to accept a node as a leaf if it includes at least five training instances. To explain this and show how decision trees work, let's visualize the model built. The following code assumes you are using IPython and that your Python distribution includes the `pydot` module. Also, it allows generation of **Graphviz** code from the tree and assumes that Graphviz itself is installed. For more information about Graphviz, please refer to `http://www.graphviz.org/`.

```
>>> import pydot,StringIO
>>> dot_data = StringIO.StringIO()
>>> tree.export_graphviz(clf, out_file=dot_data,
    feature_names=['age','sex','1st_class','2nd_class'
    '3rd_class'])
>>> graph = pydot.graph_from_dot_data(dot_data.getvalue())
>>> graph.write_png('titanic.png')
>>> from IPython.core.display import Image
>>> Image(filename='titanic.png')
```



The decision tree we have built represents a series of decisions based on the training data. To classify an instance, we should answer the question at each node. For example, at our root node, the question is: Is sex<=0.5? (are we talking about a woman?). If the answer is yes, you go to the left child node in the tree; otherwise you go to the right child node. You keep answering questions (was she in the third class?, was she in the first class?, and was she below 13 years old?), until you reach a leaf. When you are there, the prediction corresponds to the target class that has most instances (that is if the answers are given to the previous questions). In our case, if she was a woman from second class, the answer would be 1 (that is she survived), and so on.

You might be asking how our method decides which questions should be asked in each step. The answer is **Information Gain** (**IG**) (or the Gini index, which is a similar measure of disorder used by scikit-learn). IG measures how much entropy we lose if we answer the question, or alternatively, how much surer we are after answering it. **Entropy** is a measure of disorder in a set, if we have zero entropy, it means all values are the same (in our case, all instances of the target classes are the same), while it reaches its maximum when there is an equal number of instances of each class (in our case, when half of the instances correspond to survivors and the other half to non survivors). At each node, we have a certain number of instances (starting from the whole dataset), and we measure its entropy. Our method will select the questions that yield more homogeneous partitions (with the lowest entropy), when we consider only those instances for which the answer for the question is yes or no, that is, when the entropy after answering the question decreases.

Interpreting the decision tree

As you can see in the tree, at the beginning of the decision tree growing process, you have the 984 instances in the training set, 662 of them corresponding to class `0` (fatalities), and 322 of them to class `1` (survivors). The measured entropy for this initial group is about 0.632. From the possible list of questions we can ask, the one that produces the greatest information gain is: Was she a woman? (remember that the female category was encoded as `0`). If the answer is yes, entropy is almost the same, but if the answer is no, it is greatly reduced (the proportion of men who died was much greater than the general proportion of casualties). In this sense, the woman question seems to be the best to ask. After that, the process continues, working in each node only with the instances that have feature values that correspond to the questions in the path to the node.

If you look at the tree, in each node we have: the question, the initial Shannon entropy, the number of instances we are considering, and their distribution with respect to the target class. In each step, the number of instances gets reduced to those that answer yes (the left branch) and no (the right branch) to the question posed by that node. The process continues until a certain stopping criterion is met (in our case, until we have a fourth-level node, or the number of considered samples is lower than five).

At prediction time, we take an instance and start traversing the tree, answering the questions based on the instance features, until we reach a leaf. At this point, we look at to how many instances of each class we had in the training set, and select the class to which most instances belonged.

For example, consider the question of determining if a 10-year-old girl, from first class would have survived. The answer to the first question (was she female?) is yes, so we take the left branch of the tree. In the two following questions the answers are no (was she from third class?) and yes (was she from first class?), so we take the left and right branch respectively. At this time, we have reached a leaf. In the training set, we had 102 people with these attributes, 97 of them survivors. So, our answer would be survived.

In general, we found reasonable results: the group with more casualties (449 from 496) corresponded to adult men from second or third class, as you can check in the tree. Most girls from first class, on the other side, survived. Let's measure the accuracy of our method in the training set (we will first define a helper function to measure the performance of a classifier):

```
>>> from sklearn import metrics
>>> def measure_performance(X,y,clf, show_accuracy=True,
    show_classification_report=True, show_confussion_matrix=True):
>>>     y_pred=clf.predict(X)
>>>     if show_accuracy:
>>>         print "Accuracy:{0:.3f}".format(
>>>             metrics.accuracy_score(y, y_pred)
>>>         ),"\n"
>>>
>>>     if show_classification_report:
>>>         print "Classification report"
>>>         print metrics.classification_report(y,y_pred),"\n"
>>>
>>>     if show_confussion_matrix:
>>>       print "Confussion matrix"
>>>       print metrics.confusion_matrix(y,y_pred),"\n"

>>> measure_performance(X_train,y_train,clf,
    show_classification=False, show_confusion_matrix=False))
Accuracy:0.838
```

Our tree has an accuracy of 0.838 on the training set. But remember that this is not a good indicator. This is especially true for decision trees as this method is highly susceptible to overfitting. Since we did not separate an evaluation set, we should apply cross-validation. For this example, we will use an extreme case of cross-validation, named **leave-one-out cross-validation**. For each instance in the training sample, we train on the rest of the sample, and evaluate the model built on the only instance left out. After performing as many classifications as training instances, we calculate the accuracy simply as the proportion of times our method correctly predicted the class of the left-out instance, and found it is a little lower (as we expected) than the resubstitution accuracy on the training set.

```
>>> from sklearn.cross_validation import cross_val_score, LeaveOneOut
>>> from scipy.stats import sem
>>>
>>> def loo_cv(X_train, y_train,clf):
>>>     # Perform Leave-One-Out cross validation
>>>     # We are preforming 1313 classifications!
>>>     loo = LeaveOneOut(X_train[:].shape[0])
>>>     scores = np.zeros(X_train[:].shape[0])
>>>     for train_index, test_index in loo:
>>>         X_train_cv, X_test_cv = X_train[train_index],
            X_train[test_index]
>>>         y_train_cv, y_test_cv = y_train[train_index],
            y_train[test_index]
>>>         clf = clf.fit(X_train_cv,y_train_cv)
>>>         y_pred = clf.predict(X_test_cv)
>>>         scores[test_index] = metrics.accuracy_score(
   y_test_cv.astype(int), y_pred.astype(int))
>>>     print ("Mean score: {0:.3f} (+/-{1:.3f})").format(np.
mean(scores), sem(scores))

   >>> loo_cv(X_train, y_train,clf)
Mean score: 0.837 (+/-0.012)
```

The main advantage of leave-one-out cross-validation is that it allows almost as much data for training as we have available, so it is particularly well suited for those cases where data is scarce. Its main problem is that training a different classifier for each instance could be very costly in terms of the computation time.

A big question remains here: how we selected the hyperparameters for our method instantiation? This problem is a general one, it is called model selection, and we will address it in more detail in *Chapter 4, Advanced Features*.

Random Forests – randomizing decisions

A common criticism to decision trees is that once the training set is divided after answering a question, it is not possible to reconsider this decision. For example, if we divide men and women, every subsequent question would be only about men or women, and the method could not consider another type of question (say, age less than a year, irrespective of the gender). **Random Forests** try to introduce some level of randomization in each step, proposing alternative trees and combining them to get the final prediction. These types of algorithms that consider several classifiers answering the same question are called **ensemble methods**. In the Titanic task, it is probably hard to see this problem because we have very few features, but consider the case when the number of features is in the order of thousands.

Random Forests propose to build a decision tree based on a subset of the training instances (selected randomly, with replacement), but using a small random number of features at each set from the feature set. This tree growing process is repeated several times, producing a set of classifiers. At prediction time, each grown tree, given an instance, predicts its target class exactly as decision trees do. The class that most of the trees vote (that is the class most predicted by the trees) is the one suggested by the ensemble classifier.

In scikit-learn, using Random Forests is as simple as importing `RandomForestClassifier` from the `sklearn.ensemble` module, and fitting the training data as follows:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> clf = RandomForestClassifier(n_estimators=10, random_state=33)
>>> clf = clf.fit(X_train, y_train)
>>> loo_cv(X_train, y_train, clf)
Mean score: 0.817 (+/-0.012)
```

We find that results are actually worse for Random Forests. It seems that introducing randomization was, after all, not a good idea because the number of features was too small. However, for bigger datasets, with a bigger number of features, Random Forests is a very fast, simple, and popular method to improve accuracy, retaining the virtues of decision trees. Actually, in the next section, we will use them for regression.

Evaluating the performance

The final step in every supervised learning task should be to evaluate our best classifier on the previously unseen data, to get an idea of its prediction performance. Remember, this step should not be used to select among competing methods or parameters. That would be cheating (because again, we risk overfitting the new data). So, in our case, let's measure the performance of decision trees on the testing data.

```
>>> clf_dt = tree.DecisionTreeClassifier(criterion='entropy', max_
depth=3, min_samples_leaf=5)
>>> clf_dt.fit(X_train, y_train)
>>> measure_performance(X_test, y_test, clf_dt)
Accuracy:0.793
Classification report
             precision    recall  f1-score   support

          0       0.77      0.96      0.85       202
          1       0.88      0.54      0.67       127

avg / total       0.81      0.79      0.78       329
Confusion matrix
[[193   9]
 [ 59  68]]
```