Machine Learning

Intelligent Systems (IS)

September, 15th 2015

Machine Learning

Any machine learning problem can be represented with the following three concepts:

- We will have to learn to solve a task T. For example, build a spam filter that learns to classify e-mails as spam or ham.
- We will need some experience E to learn to perform the task. Usually, experience is represented through a dataset. For the spam filter, experience comes as a set of e-mails, manually classified by a human as spam or ham.
- We will need a measure of performance P to know how well we are solving the task and also to know whether after doing some modifications, our results are improving or getting worse. The percentage of e-mails that our spam filtering is correctly classifying as spam or ham could be P for our spam-filtering task.

Scikit-Learn

Scikit-learn is an open source Python library of popular machine learning.

- The project was started in 2007 as a Google Summer of Code project by David Cournapeau.
- Later that year, Matthieu Brucher started working on this project as part of his thesis.
- In 2010, Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, and Vincent Michel of INRIA took the project leadership and produced the first public release.
- Nowadays, the project is being developed very actively by an enthusiastic community of contributors.
- It is built upon NumPy (http://www.numpy.org/) and SciPy (http://scipy.org/).

Installing Scikit-Learn

Installation instructions for scikit-learn are available at http://scikit-learn.org/stable/install.html.

Several examples include visualizations, so you should also install the matplotlib package from http://matplotlib.org/.

An easy way to install all packages is to download and install the Anaconda distribution for scientific computing from https://store.continuum.io/

Installing Scikit-Learn on Linux

In the case of Debian-based operating systems, such as Ubuntu, you can install the packages by running the following commands:

- Firstly, to install the package we enter the following command:

  ```
  # sudo apt-get install build-essential python-dev
  python-numpy python-setuptools python-scipy
  libatlas-dev
  ```

- Then, to install matplotlib, run the following command:

  ```
  # sudo apt-get install python-matplotlib
  ```

- After that, we should be ready to install scikit-learn by issuing this command:

  ```
  # sudo pip install scikit-learn
  ```

Checking your installation

To check that everything is ready to run, just open your Python console and type the following:

```
>>> import sklearn as sk
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

Datasets

Scikit-learn includes a few well-known datasets.

A dataset is a set of items or instances, where each instance is a set of features or attributes. There can be an special attribute called goal or class.

We will use one of them, the Iris flower dataset, introduced in 1936 by Sir Ronald Fisher. Includes information about 150 instances from three different Iris flower species, including sepal and petal length and width.

The natural task to solve using this dataset is to learn to guess the Iris species knowing the sepal and petal measures.

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> X_iris, y_iris = iris.data, iris.target
>>> print X_iris.shape, y_iris.shape
(150, 4) (150,)
>>> print X_iris[0], y_iris[0]
[ 5.1 3.5 1.4 0.2] 0
>>> print iris.target_names
['setosa' 'versicolor' 'virginica']
```

Datasets

Every method will require this data array, where each instance is represented as a list of features or attributes, and another target array representing a certain value we want our learning method to learn to predict.

In our example, the petal and sepal measures are our real-valued attributes, while the flower species is the one-of-a-list class we want to predict.

First machine learning method: linear classification

Try to predict the Iris flower species using only two attributes: sepal width and sepal length.

Let's first build our training dataset.

- Import the dataset
- Randomly select about 75 percent of the instances (training set)
- Reserve the remaining ones (test set) for evaluation purposes
- Standarize the features

Get the train and test set

```
>>> from sklearn.cross_validation import train_test_split
>>> from sklearn import preprocessing
>>> # Get dataset with only the first two attributes
>>> X, y = X_iris[:, :2], y_iris
>>> # Split the dataset into a training and a testing set
>>> # Test set will be the 25% taken randomly
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=33)
>>> print X_train.shape, y_train.shape
(112, 2) (112,)
```

Standarize the features

For each feature, calculate the average, subtract the mean value from the feature value, and divide the result by their standard deviation.

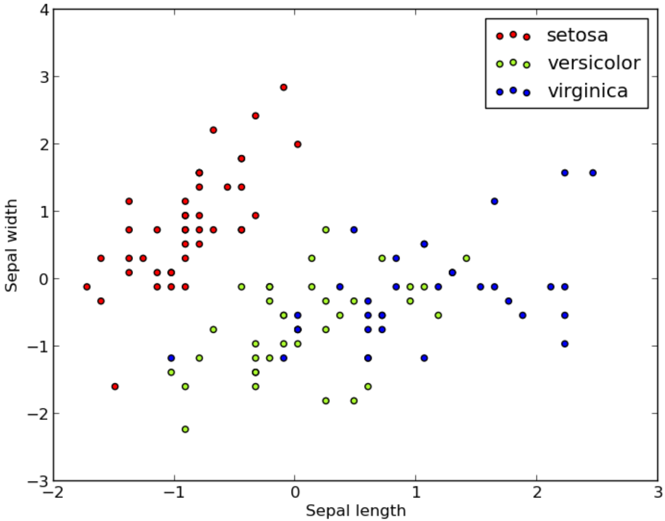
After scaling, each feature will have a zero average, with a standard deviation of one. This standardization of values (which does not change their distribution) is a common requirement of machine learning methods, to avoid that features with large values may weight too much on the final results.

```
>>> # Standardize the features
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> X_train = scaler.transform(X_train)
>>> X_test = scaler.transform(X_test)
```

Look at how our training instances are distributed

```
>>> import matplotlib.pyplot as plt
>>> colors = ['red', 'greenyellow', 'blue']
>>> for i in xrange(len(colors)):
>>>   xs = X_train[:, 0][y_train == i]
>>>   ys = X_train[:, 1][y_train == i]
>>>   plt.scatter(xs, ys, c=colors[i])
>>> plt.legend(iris.target_names)
>>> plt.xlabel('Sepal length')
>>> plt.ylabel('Sepal width')
>>> plt.show()
```

Look at how our training instances are distributed



Look at how our training instances are distributed

Given the available data, let’s, for a moment, redefine our learning task: suppose we aim, given an Iris flower instance, to predict if it is a setosa or not. We have converted our problem into a binary classification task (that is, we only have two possible target classes).

Our first classification method, linear classification models, try to do: build a line (or, more generally, a hyperplane in the feature space) that best separates both the target classes, and use it as a decision boundary (that is, the class membership depends on what side of the hyperplane the instance is).

Linear classification models

Given the available data, let’s, for a moment, redefine our learning task: suppose we aim, given an Iris flower instance, to predict if it is a setosa or not. We have converted our problem into a binary classification task (that is, we only have two possible target classes).

Our first classification method, linear classification models, try to do: build a line (or, more generally, a hyperplane in the feature space) that best separates both the target classes, and use it as a decision boundary (that is, the class membership depends on what side of the hyperplane the instance is).

Linear classification models

We will use the SGDClassifier from scikit-learn. SGD stands for Stochastic Gradient Descent, a very popular numerical procedure to find the local minimum of a function (in this case, the loss function, which measures how far every instance is from our boundary). The algorithm will learn the coefficients of the hyperplane by minimizing the loss function.

To use any method in scikit-learn, we must:

- First create the corresponding classifier
- Initialize its parameters
- Train the model that better fits the training data

```
>>> from sklearn.linear_model import SGDClassifier
>>> clf = SGDClassifier()
>>> clf.fit(X_train, y_train)
```

Linear classification models

What does the classifier look like in our linear model method?

```
>>> print clf.coef_
[[-28.53692691 15.05517618]
[ -8.93789454 -8.13185613]
[ 14.02830747 -12.80739966]]
>>> print clf.intercept_
[-17.62477802 -2.35658325 -9.7570213 ]
```

The coef_ attribute of the clf object (consider, for the moment, only the first row of the matrices), now has the coefficients of the linear boundary and the intercept_ attribute, the point of intersection of the line with the y axis.

Linear classification models

Indeed in the real plane, with these three values, we can draw a line, represented by the following equation:

$$-17.62477802 - 28.53692691 * x_1 + 15.05517618 * x_2 = 0$$

Now, given x_1 and x_2 (our real-valued features), we just have to compute the value of the left-side of the equation: if its value is greater than zero, then the point is above the decision boundary (the red side), otherwise it will be beneath the line (the green or blue side). Our prediction algorithm will simply check this and predict the corresponding class for any new iris flower.

But, why does our coefficient matrix have three rows?

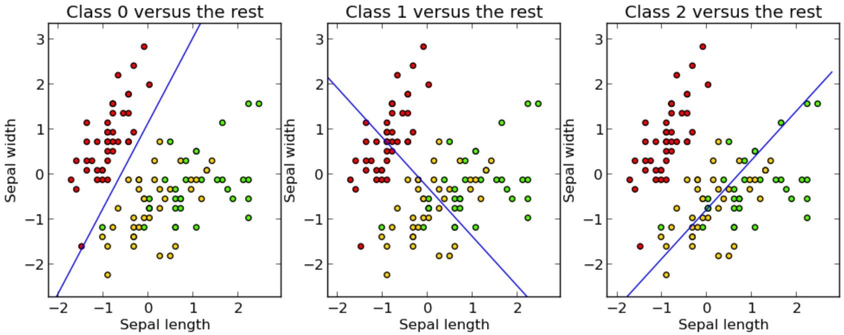
Linear classification models

The method is facing a three-class problem, not a binary decision problem. What, in this case, the classifier does is the same we did it converts the problem into three binary classification problems in a one-versus-all setting (it proposes three lines that separate a class from the rest).

Linear classification models

```
>>> x_min, x_max = X_train[:, 0].min() - .5, X_train[:, 0].max() + .5
>>> y_min, y_max = X_train[:, 1].min() - .5, X_train[:, 1].max() + .5
>>> xs = np.arange(x_min, x_max, 0.5)
>>> fig, axes = plt.subplots(1, 3)
>>> fig.set_size_inches(10, 6)
>>> for i in [0, 1, 2]:
>>>   axes[i].set_aspect('equal')
>>>   axes[i].set_title('Class '+ str(i) + ' versus the rest')
>>>   axes[i].set_xlabel('Sepal length')
>>>   axes[i].set_ylabel('Sepal width')
>>>   axes[i].set_xlim(x_min, x_max)
>>>   axes[i].set_ylim(y_min, y_max)
>>>   plt.sca(axes[i])
>>>   plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=plt.cm.prism)
>>>   ys = (-clf.intercept_[i] - xs*clf.coef_[i, 0]) / clf.coef_[i, 1]
>>>   plt.plot(xs, ys, hold=True)
>>> plt.show()
```

Linear classification models



Using the classifier

Suppose that we have a new flower with a sepal width of 4.7 and a sepal length of 3.1, and we want to predict its class.

We just have to apply our brand new classifier to it (after normalizing!). The predict method takes an array of instances (in this case, with just one element) and returns a list of predicted classes:

```
>>>print clf.predict(scaler.transform([[4.7, 3.1]]))
[0]
```

Our prediction procedure combines the result of the three binary classifiers and selects the class in which it is more confident. In this case, we will select the boundary line whose distance to the instance is longer. We can check that using the classifier decision_function method:

```
>>>print clf.decision_function(scaler.transform([[4.7, 3.1]]))
[[ 19.73905808 8.13288449 -28.63499119]]
```

Evaluating our results

The performance of a classifier is a measure of its effectiveness.

The simplest performance measure is accuracy: given a classifier and an evaluation dataset, it measures the proportion of instances correctly classified by the classifier.

```
>>> from sklearn import metrics
>>> y_train_pred = clf.predict(X_train)
>>> print metrics.accuracy_score(y_train, y_train_pred)
0.821428571429
```

Probably, the most important thing you should learn so far is that measuring accuracy on the training set is really a bad idea.

Overfitting

You have built your model using this data, and it is possible that your model adjusts well to them but performs poorly in future (previously unseen data), which is its purpose.

This phenomenon is called overfitting. If you measure based on your training data, you will never detect overfitting. So, never measure based on your training data.

Let's check the accuracy again, now on the test set.

```
>>> y_pred = clf.predict(X_test)
>>> print metrics.accuracy_score(y_test, y_pred)
0.684210526316
```

Our goal will always be to produce models that avoid overfitting when trained over a training set, so they have enough generalization power to also correctly model the unseen data.

Accuracy on the test set is a good performance measure when the number of instances of each class is similar, that is, we have a uniform distribution of classes.

Overfitting

Our goal will always be to produce models that avoid overfitting when trained over a training set, so they have enough generalization power to also correctly model the unseen data.

Evaluation functions within scikit-learn

We show three popular ones: precision, recall, and F1-score (or f-measure). They assume a binary classification problem and two classes, a positive one and a negative one.

- Precision: this computes the proportion of instances predicted as positives that were correctly evaluated (it measures how right our classifier is when it says that an instance is positive).
- Recall: this counts the proportion of positive instances that were correctly evaluated (measuring how right our classifier is when faced with a positive instance).
- F1-score: this is the harmonic mean of precision and recall, and tries to combine both in a single number.

The harmonic mean is used instead of the arithmetic mean because the latter compensates low values for precision and with high values for recall (and vice versa). On the other hand, with harmonic mean we will always have low values if either precision or recall is low.

Evaluation functions within scikit-learn

We can define these measures in terms of True and False, and Positives and Negatives:

	Prediction: Positive	**Prediction: Negative**
Target class: Positive	True Positive (TP)	False Negative (FN)
Target class: Negative	False Positive (FP)	True Negative (TN)

With m being the sample size (that is, $TP + TN + FP + FN$), we have the following formulae:

- $Accuracy = (TP + TN)/m$
- $Precision = TP/(TP + FP)$
- $Recall = TP/(TP + FN)$
- $F1\text{-}score = 2 * Precision * Recall/(Precision + Recall)$

Evaluation functions within scikit-learn

```
>>> print metrics.classification_report(y_test, y_pred,
target_names=iris.target_names)

            precision recall f1-score support
setosa      1.00      1.00   1.00     8
versicolor  0.43      0.27   0.33     11
virginica   0.65      0.79   0.71     19
avg / total 0.66      0.68   0.66     38
```

Now, we can see that our method (as we expected) is very good at predicting setosa, while it suffers when it has to separate the versicolor or virginica classes. The support value shows how many instances of each class we had in the testing set.

Evaluation functions within scikit-learn

Another useful metric (especially for multi-class problems) is the **confusion matrix**: in its (i, j) cell, it shows the number of class instances i that were predicted to be in class j.

```
>>> print metrics.confusion_matrix(y_test, y_pred)
[[ 8 0 0]
[ 0 3 8]
[ 0 4 15]]
```

A good classifier will accumulate the values on the confusion matrix diagonal, where correctly classified instances belong.

Evaluation functions within scikit-learn

Another useful method is the so called cross-validation. As we explained before, we have to partition our dataset into a training set and a testing set. However, partitioning the data, results such that there are fewer instances to train on, and also, depending on the particular partition we make (usualy made randomly), we can get either better or worse results.

Cross-validation allows us to avoid this particular case, reducing result variance and producing a more realistic score for our models. The usual steps for k-fold cross-validation are the following:

- Partition the dataset into k different subsets.
- Create k different models by training on $k-1$ subsets and testing on the remaining subset.
- Measure the performance on each of the k models and take the average measure.

Evaluation functions within scikit-learn

We will chose to have k = 5 folds, so each time we will train on 80 percent of the data and test on the remaining 20 percent. Cross-validation, by default, uses accuracy as its performance measure, but we could select the measurement by passing any scorer function as an argument.

```
>>> from sklearn.cross_validation import cross_val_score, KFold
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.preprocessing import StandardScaler
>>> # create a composite estimator made by a pipeline of the
standarization and the linear model
>>> clf = Pipeline([
('scaler', StandardScaler()),
('linear_model', SGDClassifier())
])
>>> # create a k-fold cross validation iterator of k=5 folds
>>> cv = KFold(X.shape[0], 5, shuffle=True, random_state=33)
>>> # by default the score used is the one returned by score
method of the estimator (accuracy)
>>> scores = cross_val_score(clf, X, y, cv=cv)
>>> print scores
[ 0.73333333  0.63333333  0.73333333  0.66666667  0.6        ]
```

Evaluation functions within scikit-learn

We obtained an array with the k scores. We can calculate the mean and the standard error to obtain a final figure:

```
>>> from scipy.stats import sem
>>> def mean_score(scores):
return ("Mean score: {0:.3f} (+/-
{1:.3f})").format(np.mean(scores), sem(scores))
>>> print mean_score(scores)
Mean score: 0.673 (+/-0.027)
```

Our model has an average accuracy of 0.71.

Machine learning categories

We can organize machine learning problems in the following categories:

- **Supervised learning**: we have a set of instances represented by certain features and with a particular target attribute. Algorithms try to build a model from this data, which lets us predict the target attribute for new instances, knowing only these instance features. When the target class belongs to a discrete set (such as a list of flower species), we are facing a classification problem.
- **Regression**: in this case, the class we want to predict, instead of belonging to a discrete set, ranges on a continuous set, such as the real number line.

  For example, we could try to predict the petal width based on the other three features. We will see that the methods used for regression are quite different from those used for classification.

Machine learning categories

We can organize machine learning problems in the following categories:

- **Unsupervised learning**: In this case, we do not have a target class to predict but instead want to group instances according to some similarity measure based on the available set of features. For example, suppose you have a dataset composed of e-mails and want to group them by their main topic (the task of grouping instances is called clustering). We can use it as features, for example, the different words used in each of them.

Important concepts related to Machine Learning

- **The curse of dimensionality**: when the number of parameters of a model grows, the data needed to learn them grows exponentially.
- **Overfitting**: the general rule is that, in order to avoid overfitting, we should prefer simple (that is, with less parameters) methods, something that could be seen as an instantiation of the philosophical principle of Occam's razor, which states that among competing hypotheses, the hypothesis with the fewest assumptions should be selected.

  However, we should also take into account Einstein's words:

  "Everything should be made as simple as possible, but not simpler."
- **Underfitting**: underfitting problems arise when our model has such a low representation power that it cannot model the data even if we had all the training data we want. We clearly have underfitting when our algorithm cannot achieve good performance measures even when measuring on the training set.

Important concepts related to Machine Learning

- **Bias-variance tradeoff**: It can be shown that, no matter which method we are using, if we reduce bias, variance will increase, and vice versa.

  Error due to Bias: The error due to bias is taken as the difference between the expected (or average) prediction of our model and the correct value which we are trying to predict. Of course you only have one model so talking about expected or average prediction values might seem a little strange. However, imagine you could repeat the whole model building process more than once: each time you gather new data and run a new analysis creating a new model. Due to randomness in the underlying data sets, the resulting models will have a range of predictions. Bias measures how far off in general these models' predictions are from the correct value.

Important concepts related to Machine Learning

Error due to Variance: The error due to variance is taken as the variability of a model prediction for a given data point. Again, imagine you can repeat the entire model building process multiple times. The variance is how much the predictions for a given point vary between different realizations of the model.

Example: Linear classifiers have generally low-variance: no matter what subset we select for training, results will be similar. However, if the data distribution (as in the case of the versicolor and virginica species) makes target classes not separable by a hyperplane, these results will be consistently wrong, that is, the method is highly biased.

Important concepts related to Machine Learning

- **Data preprocessing:** In real-world applications our data may not come naturally as a list of real-valued features. We need to have methods to transform non real-valued features to real-valued ones. Besides, there are other steps related to feature standardization and normalization, which as we saw in our Iris example, are needed to avoid undesired effects regarding the different value ranges.
- **Feature selection:** Not all of the features that come in our original dataset could be useful for resolving our task

Supervised Learning

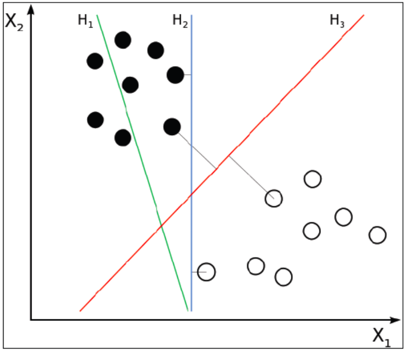
Supervised learning methods are nowadays a standard tool in a wide range of disciplines, from medical diagnosis to natural language processing, image recognition, and searching for new particles at the Large Hadron Collider (LHC).

Support Vector Machines

Imagine that the instances in your dataset are points in a multidimensional space; we can assume that the model built by our classifier can be a surface or using linear algebra terminology, a hyperplane that separates instances (points) of one class from the rest. Support Vector Machines (SVM) are supervised learning methods that try to obtain these hyperplanes in an optimal way, by selecting the ones that pass through the widest possible gaps between instances of different classes. New instances will be classified as belonging to a certain category based on which side of the surfaces they fall on.

Support Vector Machines

The following figure shows an example for a two-dimensional space with two features (X1 and X2) and two classes (black and white):



The blue and red hyperplanes separate both classes without errors.

Support Vector Machines

However, the Red separates both classes with maximum margin; it is the most distant hyperplane from the closest instances from the two categories.

The main advantage of this approach is that it will probably lower the generalization error, making this model resistant to overfitting, something that actually has been verified in several, different, classification tasks.
This can be extended to:

- construct hyperplanes in high or infinite dimensional spaces
- use nonlinear surfaces, such as polynomial or radial basis functions

Support Vector Machines

Advantages of SVMs:

- very effective on high-dimensional spaces
- very effective when the data is sparse
- very efficient in terms of memory storage (decision surfaces only require a subset of the points in the learning space)

Disadvantages of SVMs:

- very calculation intensive while training the model
- do not return a numerical indicator of how confident they are about a prediction (K-fold cross-validation can be used to avoid this, but more computational cost)

Image recognition with Support Vector Machines

We will apply SVM to image recognition, a classic problem with a very large dimensional space

The value of each pixel of the image is considered as a feature.

Given an image of a person's face, predict to which of the possible people from a list does it belongs.

Our learning set will be a group of labeled images of peoples' faces.

We will try to learn a model that can predict the label of unseen instances.

Image recognition with Support Vector Machines

Our dataset is provided within scikit-learn:

```
>>> import sklearn as sk
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from sklearn.datasets import fetch_olivetti_faces
>>> faces = fetch_olivetti_faces()
>>> print faces.DESCR
```

The dataset contains 400 images of 40 different persons. The photos were taken with different light conditions and facial expressions.

Image recognition with Support Vector Machines

Images contain the 400 images represented as 64 x 64 pixel matrices. data contains the same 400 images but as array of 4096 pixels. target is, as expected, an array with the target classes, ranging from 0 to 39.

```
>>> print faces.keys()
['images', 'data', 'target', 'DESCR']
>>> print faces.images.shape
(400, 64, 64)
>>> print faces.data.shape
(400, 4096)
>>> print faces.target.shape
(400,)
```

Image recognition with Support Vector Machines

Normalizing the data is important. In this case, we can verify by running the following snippet that our images already come as values in a very uniform range between 0 and 1 (pixel value):

```
>>> print np.max(faces.data)
1.0
>>> print np.min(faces.data)
0.0
>>> print np.mean(faces.data)
0.547046432495
```

Therefore, we do not have to normalize the data.

Image recognition with Support Vector Machines

Lets print some faces.

```
>>> import matplotlib.pyplot as plt
>>> def print_faces(images, target, top_n):
>>>   # set up the figure size in inches
>>>   fig = plt.figure(figsize=(12, 12))
>>>   fig.subplots_adjust(left=0, right=1, bottom=0, top=1,
hspace=0.05, wspace=0.05)
>>>   for i in range(top_n):
>>>     # plot the images in a matrix of 20x20
>>>     p = fig.add_subplot(20, 20, i + 1, xticks=[],
yticks=[])
>>>     p.imshow(images[i], cmap=plt.cm.bone)
>>>
>>>     # label the image with the target value
>>>     p.text(0, 14, str(target[i]))
>>>     p.text(0, 60, str(i))
>>> print_faces(faces.images, faces.target, 20)
>>> plt.show()
```

Training a Support Vector Machine

Lets print some faces.

```
>>> import matplotlib.pyplot as plt
>>> def print_faces(images, target, top_n):
>>>   # set up the figure size in inches
>>>   fig = plt.figure(figsize=(12, 12))
>>>   fig.subplots_adjust(left=0, right=1, bottom=0, top=1,
hspace=0.05, wspace=0.05)
>>>   for i in range(top_n):
>>>     # plot the images in a matrix of 20x20
>>>     p = fig.add_subplot(20, 20, i + 1, xticks=[],
yticks=[])
>>>     p.imshow(images[i], cmap=plt.cm.bone)
>>>
>>>     # label the image with the target value
>>>     p.text(0, 14, str(target[i]))
>>>     p.text(0, 60, str(i))
>>> print_faces(faces.images, faces.target, 20)
>>> plt.show()
```

Training a Support Vector Machine

The Support Vector Classifier (SVC) will be used for classification (SVM for regression tasks).
The most relevant parameter is the kernel function (think of the kernel functions as different similarity measures between instances). By default, the SVC class uses the rbf kernel, which allows us to model nonlinear problems. To start, we will use the simplest kernel, the linear one.

```
>>> from sklearn.svm import SVC
>>> svc_1 = SVC(kernel='linear')
```

Split our dataset into training and testing datasets

```
>>> from sklearn.cross_validation import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
faces.data, faces.target, test_size=0.25, random_state=0)
```

Training a Support Vector Machine

Define a function to evaluate K-fold cross-validation:

```
>>> from sklearn.cross_validation import cross_val_score, KFold
>>> from scipy.stats import sem
>>>
>>> def evaluate_cross_validation(clf, X, y, K):
>>>   # create a k-fold cross validation iterator
>>>   cv = KFold(len(y), K, shuffle=True, random_state=0)
>>>   # by default the score used is the one returned by score
method of the estimator (accuracy)
>>>   scores = cross_val_score(clf, X, y, cv=cv)
>>>   print scores
>>>   print ("Mean score: {0:.3f} (+/-{1:.3f})").format(
np.mean(scores), sem(scores))
>>> evaluate_cross_validation(svc_1, X_train, y_train, 5)
[ 0.93333333  0.86666667  0.91666667  0.93333333  0.91666667]
Mean score: 0.913 (+/-0.012)
```

Cross-validation with five folds, obtains pretty good results (accuracy of 0.933). In a few steps we obtained a face classifier.

Training a Support Vector Machine

Define a function to perform training on the training set and evaluate the performance on the testing set:

```
>>> from sklearn import metrics
>>>
>>> def train_and_evaluate(clf, X_train, X_test, y_train, y_test)
>>>
>>>   clf.fit(X_train, y_train)
>>>
>>>   print "Accuracy on training set:"
>>>   print clf.score(X_train, y_train)
>>>   print "Accuracy on testing set:"
>>>   print clf.score(X_test, y_test)
>>>
>>>   y_pred = clf.predict(X_test)
>>>
>>>   print "Classification Report:"
>>>   print metrics.classification_report(y_test, y_pred)
>>>   print "Confusion Matrix:"
>>>   print metrics.confusion_matrix(y_test, y_pred)
```

Training a Support Vector Machine

Define a function to perform training on the training set and evaluate the performance on the testing set:

```
>>> from sklearn import metrics
>>>
>>> def train_and_evaluate(clf, X_train, X_test, y_train, y_test)
>>>
>>>   clf.fit(X_train, y_train)
>>>
>>>   print "Accuracy on training set:"
>>>   print clf.score(X_train, y_train)
>>>   print "Accuracy on testing set:"
>>>   print clf.score(X_test, y_test)
>>>
>>>   y_pred = clf.predict(X_test)
>>>
>>>   print "Classification Report:"
>>>   print metrics.classification_report(y_test, y_pred)
>>>   print "Confusion Matrix:"
>>>   print metrics.confusion_matrix(y_test, y_pred)
```

Training a Support Vector Machine

Try to classify the faces as people with and without glasses? Let's do that. Define the range of the images that show faces wearing glasses

```
>>> # the index ranges of images of people with glasses
>>> glasses = [
(10, 19), (30, 32), (37, 38), (50, 59), (63, 64),
(69, 69), (120, 121), (124, 129), (130, 139), (160, 161),
(164, 169), (180, 182), (185, 185), (189, 189), (190, 192),
(194, 194), (196, 199), (260, 269), (270, 279), (300, 309),
(330, 339), (358, 359), (360, 369)
]
```

Training a Support Vector Machine

Define a function that from those segments returns a new target array that marks with 1 for the faces with glasses and 0 for the faces without glasses (our new target classes)

```
>>> def create_target(segments):
>>>   # create a new y array of target size initialized with zeros
>>>   y = np.zeros(faces.target.shape[0])
>>>   # put 1 in the specified segments
>>>   for (start, end) in segments:
>>>     y[start:end + 1] = 1
>>>   return y
>>> target_glasses = create_target(glasses)
```

Training a Support Vector Machine

Perform the training/testing split again.

```
>>> X_train, X_test, y_train, y_test = train_test_split(
faces.data, target_glasses, test_size=0.25, random_state=0)
```

Create a new SVC classifier:

```
>>> svc_2 = SVC(kernel='linear')
```

Check the performance with cross-validation:

```
>>> evaluate_cross_validation(svc_2, X_train, y_train, 5)
[ 0.98333333 0.98333333 0.93333333 0.96666667 0.96666667]
Mean score: 0.967 (+/-0.009)
```

Training a Support Vector Machine

Train and evaluate with test set:

```
>>> train_and_evaluate(svc_2, X_train, X_test, y_train, y_test)
Accuracy on training set:
1.0
Accuracy on testing set:
0.99
Classification Report:
             precision    recall  f1-score   support

        0.0       1.00      0.99      0.99        67
        1.0       0.97      1.00      0.99        33

avg / total       0.99      0.99      0.99       100

Confusion Matrix:
[[66  1]
 [ 0 33]]
```

Training a Support Vector Machine

Could it be possible that our classifier has learned to identify peoples' faces associated with glasses and without glasses precisely? How can we be sure that this is not happening and that if we get new unseen faces, it will work as expected?

```
>>> train_and_evaluate(svc_2, X_train, X_test, y_train, y_test)
Accuracy on training set:
1.0
Accuracy on testing set:
0.99
Classification Report:
             precision    recall  f1-score   support

        0.0       1.00      0.99      0.99        67
        1.0       0.97      1.00      0.99        33

avg / total       0.99      0.99      0.99       100

Confusion Matrix:
[[66  1]
 [ 0 33]]
```

Training a Support Vector Machine

Could it be possible that our classifier has learned to identify peoples' faces associated with glasses and without glasses precisely? How can we be sure that this is not happening and that if we get new unseen faces, it will work as expected? Let's separate all the images of the same person, sometimes wearing glasses and sometimes not. We will also separate all the images of the same person, the ones with indexes from 30 to 39, train by using the remaining instances, and evaluate on our new 10 instances set. With this experiment we will try to discard the fact that it is remembering faces, not glassed-related features.

Training a Support Vector Machine

```
>>> X_test = faces.data[30:40]
>>> y_test = target_glasses[30:40]
>>> print y_test.shape[0]
10
>>> select = np.ones(target_glasses.shape[0])
>>> select[30:40] = 0
>>> X_train = faces.data[select == 1]
>>> y_train = target_glasses[select == 1]
>>> print y_train.shape[0]
390
>>> svc_3 = SVC(kernel='linear')
>>> train_and_evaluate(svc_3, X_train, X_test, y_train, y_test)
```

Training a Support Vector Machine

From the 10 images, only one error, let's check out which one was incorrectly classified. First, we have to reshape the data from arrays to 64 x 64 matrices:

```
>>> y_pred = svc_3.predict(X_test)
>>> eval_faces = [np.reshape(a, (64, 64)) for a in X_eval]
```

Then plot with our print_faces function:

```
>>> print_faces(eval_faces, y_pred, 10)
```

Training a Support Vector Machine

Usually we would not get such good results in the first trial. In these cases:

- Look at different features
- Start tweaking the hyperparameters of our algorithm: (try different kernels, polynomial or RBF)

Text classification with Naïve Bayes

Naïve Bayes Classifier:

- Based on a probabilistic model derived from the Bayes' theorem
- Determines the probability that an instance belongs to a class based on each of the feature value probabilities
- Naïve term comes from the fact that it assumes that each feature is independent of the rest
- The independence assumption, although a naïve and strong simplification, is one of the features that make the model useful in practical applications
- One of the most successful applications of Naïve Bayes has been within the field of Natural Language Processing (NLP)

Text classification with Naïve Bayes

We will have a set of text documents with their corresponding categories, and we will train a Naïve Bayes algorithm to learn to predict the categories of new unseen instances.

This simple task has many practical applications; probably the most known and widely used one is spam filtering.

The dataset consists of around 19,000 newsgroup messages from 20 different topics ranging from politics and religion to sports and science.

Text classification with Naïve Bayes

```
>>> import numpy
>>> np = numpy
>>> from sklearn.datasets import fetch_20newsgroups
>>> news = fetch_20newsgroups(subset='all')
>>> print type(news.data), type(news.target), type(news.target_names)
<type 'list'> <type 'numpy.ndarray'> <type 'list'>
>>> print news.target_names
['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc',
'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware',
'comp.windows.x', 'misc.forsale', 'rec.autos', 'rec.motorcycles',
'rec.sport.baseball', 'rec.sport.hockey', 'sci.crypt',
'sci.electronics', 'sci.med', 'sci.space','soc.religion.christian',
'talk.politics.guns', 'talk.politics.mideast', 'talk.politics.misc',
'talk.religion.misc']
>>> print len(news.data)
18846
>>> print len(news.target)
18846
```

To print the first instance:

```
>>> print news.data[0]
>>> print news.target[0], news.target_names[news.target[0]]
```

Text classification with Naïve Bayes

Partition our data into training and testing set. The loaded data is already in a random order.

```
>>> SPLIT_PERC = 0.75
>>> split_size = int(len(news.data)*SPLIT_PERC)
>>> X_train = news.data[:split_size]
>>> X_test = news.data[split_size:]
>>> y_train = news.target[:split_size]
>>> y_test = news.target[split_size:]
```

Preprocessing the Data

Machine learning algorithms can work only on numeric data, so our next step will be to convert our text-based dataset to a numeric dataset.

We only have one feature, the text content of the message.

Intuitively one could try to look at which are the words (or more precisely, tokens, including numbers or punctuation signs) that are used in each of the text categories, and try to characterize each category with the frequency distribution of each of those words.

The sklearn. feature_extraction.text module has some useful utilities to build numeric feature vectors from text documents.

Preprocessing the Data

There are three classes inside sklearn.feature_extraction.text:

- CountVectorizer: creates a dictionary of words from the text corpus. Then, each instance is converted to a vector of numeric features where each element will be the count of the number of times a particular word appears in the document.
- HashingVectorizer:instead of constricting and maintaining the dictionary in memory, implements a hashing function that maps tokens into feature indexes, and then computes the count as in CountVectorizer.
- TfidfVectorizer: as CountVectorizer, but with a more advanced calculation called Term Frequency Inverse Document Frequency (TF-IDF). Looks for words that are more frequent in the current document, compared with their frequency in the whole corpus of documents. Avoids words that are too frequent, and thus not useful to characterize the instances.

Training a Naïve Bayes classifier

We will use the MultinomialNB class from the sklearn.naive_bayes module.

We will create three different classifiers by combining MultinomialNB with the three different text vectorizers.

```
>>> from sklearn.naive_bayes import MultinomialNB
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.feature_extraction.text import TfidfVectorizer,
HashingVectorizer, CountVectorizer
>>>
>>> clf_1 = Pipeline([
>>> ('vect', CountVectorizer()),
>>> ('clf', MultinomialNB()),
>>> ])
```

Training a Naïve Bayes classifier

We will use the MultinomialNB class from the sklearn.naive_bayes module.

We will create three different classifiers by combining MultinomialNB with the three different text vectorizers.

```
>>> clf_2 = Pipeline([
>>> ('vect', HashingVectorizer(non_negative=True)),
>>> ('clf', MultinomialNB()),
>>> ])
>>> clf_3 = Pipeline([
>>> ('vect', TfidfVectorizer()),
>>> ('clf', MultinomialNB()),
>>> ])
```

Training a Naïve Bayes classifier

Define a function that takes a classifier and performs the K-fold crossvalidation:

```
>>> from sklearn.cross_validation import cross_val_score, KFold
>>> from scipy.stats import sem
>>>
>>> def evaluate_cross_validation(clf, X, y, K):
>>>   # create a k-fold croos validation iterator of k=5 folds
>>>   cv = KFold(len(y), K, shuffle=True, random_state=0)
>>>   # by default the score used is the one returned by score >>
method of the estimator (accuracy)
>>>   scores = cross_val_score(clf, X, y, cv=cv)
>>>   print scores
>>>   print ("Mean score: {0:.3f} (+/-{1:.3f})").format(
>>> np.mean(scores), sem(scores))
```

Training a Naïve Bayes classifier

Perform a five-fold cross-validation by using each one of the classifiers.

```
>>> clfs = [clf_1, clf_2, clf_3]
>>> for clf in clfs:
>>> evaluate_cross_validation(clf, news.data, news.target, 5)
[ 0.85782493  0.85725657  0.84664367  0.85911382  0.8458477 ]
Mean score: 0.853 (+/-0.003)
[ 0.75543767  0.77659857  0.77049615  0.78508888  0.76200584]
Mean score: 0.770 (+/-0.005)
[ 0.84482759  0.85990979  0.84558238  0.85990979  0.84213319]
Mean score: 0.850 (+/-0.004)
```

Improving the naïve classifier

Parse the text documents into tokens with a different regular expression.

The default regular expression considers alphanumeric characters and the underscore. Perhaps also considering the slash and the dot could improve the tokenization, and begin considering tokens as Wi-Fi and site.com.

```
>>> clf_4 = Pipeline([
>>>   ('vect', TfidfVectorizer(
>>>     token_pattern=ur"\b[a-z0-9_\-\.]+[a-z][a-z0-9_\-\.]+\b",
>>>   )),
>>>   ('clf', MultinomialNB()),
>>> ])
```

```
>>> evaluate_cross_validation(clf_4, news.data, news.target, 5)
[ 0.86100796  0.8718493   0.86203237  0.87291059  0.8588485 ]
Mean score: 0.865 (+/-0.003)
```

Slight improvement ...

Improving the naïve classifier

Pass a list of words we do not want to take into account, such as too frequent words, or words we do not a priori expect to provide information about the particular topic.

```
>>> def get_stop_words():
>>>   result = set()
>>>   for line in open('stopwords_en.txt', 'r').readlines():
>>>     result.add(line.strip())
>>>   return result
```

Improving the naïve classifier

Create a new classifier with this new parameter as follows:

```
>>> clf_5 = Pipeline([
>>>   ('vect', TfidfVectorizer(
>>>     stop_words = get_stop_words(),
>>>     token_pattern=ur"\b[a-z0-9_\-\.]+[a-z][a-z0->>>
9_\-\.]+\b",
>>>   )),
>>>   ('clf', MultinomialNB()),
>>> ])
>>> evaluate_cross_validation(clf_5, news.data, news.target, 5)
[ 0.88116711  0.89519767  0.88325816  0.89227912  0.88113558]
Mean score: 0.887 (+/-0.003)
```

Keep this vectorizer and start looking at the MultinomialNB parameters.

Improving the naïve classifier

The most important is the alpha parameter, which is a smoothing parameter. Default value is 1.0, lets set it to 0.01.

```
>>> clf_7 = Pipeline([
>>> ('vect', TfidfVectorizer(
>>>   stop_words = get_stop_words(),
>>>   token_pattern=ur"\b[a-z0-9_\-\.]+[a-z][a-z0->>>
9_\-\.]+\b",
>>> )),
>>> ('clf', MultinomialNB(alpha=0.01)),
>>> ])
>>> evaluate_cross_validation(clf_7, news.data, news.target, 5)
[ 0.9204244   0.91960732  0.91828071  0.92677103  0.91854603]
Mean score: 0.921 (+/-0.002)
```

We got a better improvement.

Evaluating the performance

```
>>> from sklearn import metrics
>>>
>>> def train_and_evaluate(clf, X_train, X_test, y_train, y_test):
>>>
>>>   clf.fit(X_train, y_train)
>>>
>>>   print "Accuracy on training set:"
>>>   print clf.score(X_train, y_train)
>>>   print "Accuracy on testing set:"
>>>   print clf.score(X_test, y_test)
>>>   y_pred = clf.predict(X_test)
>>>
>>>   print "Classification Report:"
>>>   print metrics.classification_report(y_test, y_pred)
>>>   print "Confusion Matrix:"
>>>   print metrics.confusion_matrix(y_test, y_pred)
>>>
>>> train_and_evaluate(clf_7, X_train, X_test, y_train, y_test)
>>>
>>> print len(clf_7.named_steps['vect'].get_feature_names())
>>> clf_7.named_steps['vect'].get_feature_names()
```

Explaining Titanic hypothesis with decision trees

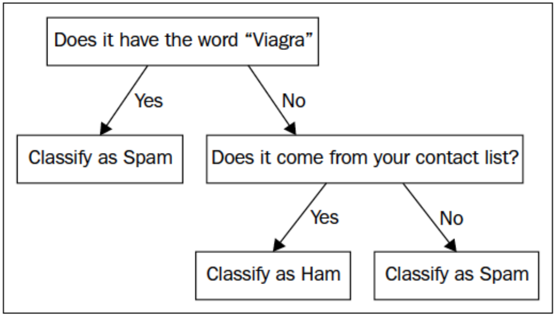
A common argument against linear classifiers and against statistical learning methods is that it is difficult to explain how the built model decides its predictions for the target classes.

Decision trees are very simple yet powerful supervised learning methods, which constructs a decision tree model.

The main advantage 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.

Explaining Titanic hypothesis with decision trees

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

Explaining Titanic hypothesis with decision trees

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.

The list of attributes is: Ordinal, Class, Survived (0=no, 1=yes), Name, Age, Port of Embarkation, Home/Destination, Room, Ticket, Boat, and Sex.

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

Explaining Titanic hypothesis with decision trees

Load 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)
```

Explaining Titanic hypothesis with decision trees

```
>>> 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

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]]
```

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).

```
>>> print feature_names
['pclass' 'age' 'sex']
>>> print titanic_X[12],titanic_y[12]
['1st' 'NA' 'female'] 1
```

Preprocessing the data

We have missing values!, a usual problem with datasets.

We will 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.

A 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
```

Preprocessing the data

The implementation of decision trees in scikit-learn expects as input a list of realvalued features.

The decision rules of the model would be of the form:

```
Feature <= value
```

We have to convert categorical data into real values. We use the 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.

```
>>> # 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
```

Preprocessing the data

```
print feature_names
['pclass' 'age' 'sex']
print titanic_X[12], titanic_y[12]
['1st' '31.1941810427' '0'] 1
```

This transformation implicitly introduces an ordering between classes, something that is not an issue in our problem.

For the categorical attribute *class* we will use the one hot encoding. It converts the class attributes into three new binary features, each of them indicating if the instance belongs to a feature value (1) or (0).

Preprocessing the data

```
>>> 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)
```

Preprocessing the data

```
>>> # 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)
```

```
>>> 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
```

Training a decision tree classifier

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)
```

Create a new DecisionTreeClassifier and use the fit method:

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

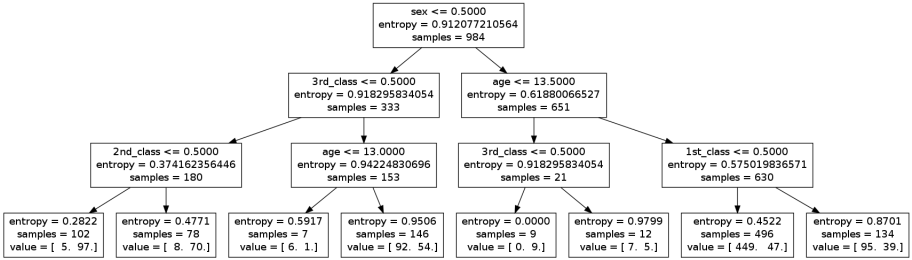
Training a decision tree classifier

Let’s visualize the model built. If required, install pyhton-pydot.

```
$ sudo apt-get install python-pydot
```

```
>>> 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')
```

Training a decision tree classifier



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.

Training a decision tree 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_report=False, show_confussion_matrix=False)
Accuracy:0.838
```

Good accuracy, but remember this is not a good indicator. This is especially true for decision trees as this method is highly susceptible to overfitting.

Training a decision tree classifier

Leave-one-out cross-validation:

```
>>> 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)
```

This method is well suited when data is scarce, but can be very costly in terms of the computation time.

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.

Random Forests try to introduce some level of randomization in each step, proposing alternative trees and combining them to get the final prediction.

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.

Random Forests: randomizing decisions

```
>>> 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)
```

Evaluating the performance

```
>>> 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]]
```

It seems that our method tends to predict too much that the person did not survive.

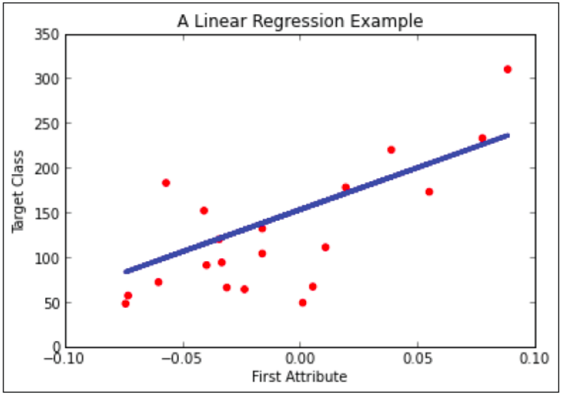
Evaluating the performance

```
>>> 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]]
```

It seems that our method tends to predict too much that the person did not survive.

Predicting house prices with regression

Instead of selecting a class from a list, now we want our classifier to act as a real-valued function, which for each of the (possibly infinite) combination of learning features returns a real number.



Predicting house prices with regression

We will compare several regression methods.

We will try to predict the price of a house as a function of its attributes.

We will use the Boston house-prices dataset, which includes 506 instances, representing houses in the suburbs of Boston by 14 features, one of them (the median value of owner-occupied homes) being the target class (for a detailed reference, see http://archive.ics.uci.edu/ml/datasets/Housing). Each attribute in this dataset is real-valued.

Predicting house prices with regression

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from sklearn.datasets import load_boston
>>> boston = load_boston()
>>> print boston.data.shape
(506, 13)
>>> print boston.feature_names
['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX'
'PTRATIO' 'B' 'LSTAT' 'MEDV']
>>> print np.max(boston.target), np.min(boston.target),
np.mean(boston.target)
50.0 5.0 22.5328063241
```

Predicting house prices with regression

Slice our learning set into training and testing datasets, and normalize the data:

```
>>> from sklearn.cross_validation import train_test_split
>>> X_train, X_test, y_train, y_test =
    train_test_split(boston.data, boston.target, test_size=0.25,
    random_state=33)
>>> from sklearn.preprocessing import StandardScaler
>>> scalerX = StandardScaler().fit(X_train)
>>> scalery = StandardScaler().fit(y_train)
>>> X_train = scalerX.transform(X_train)
>>> y_train = scalery.transform(y_train)
>>> X_test = scalerX.transform(X_test)
>>> y_test = scalery.transform(y_test)
```

Predicting house prices with regression

Regression poses an additional problem: how should we evaluate our results?

Accuracy is not a good idea, since we are predicting real values, it is almost impossible for us to predict exactly the final value.

There are several measures that can be used (you can look at the list of functions under sklearn.metrics module).

We will use the R^2 score, or coefficient of determination. It measures the proportion of the outcomes variation explained by the model.

This score reaches its maximum value of 1 when the model perfectly predicts all the test target values.

Predicting house prices with regression

```
>>> from sklearn.cross_validation import *
>>> def train_and_evaluate(clf, X_train, y_train):
>>>   clf.fit(X_train, y_train)
>>>   print "Coefficient of determination on training
      set:",clf.score(X_train, y_train)
>>>   # create a k-fold cross validation iterator of k=5 folds
>>>   cv = KFold(X_train.shape[0], 5, shuffle=True,
      random_state=33)
>>>   scores = cross_val_score(clf, X_train, y_train, cv=cv)
>>>   print "Average coefficient of determination using 5-fold
      crossvalidation:",np.mean(scores)
```

First try: Linear Model

The question that linear models try to answer is which hyperplane in the 14-dimensional space created by our learning features (including the target value) is located closer to them.

After this hyperplane is found, prediction reduces to calculate the projection on the hyperplane of the new point, and returning the target value coordinate.

But, what do we mean by closer? The usual measure is least squares: calculate the distance of each instance to the hyperplane, square it (to avoid sign problems), and sum them.

```
>>> from sklearn.cross_validation import *
>>> def train_and_evaluate(clf, X_train, y_train):
>>>   clf.fit(X_train, y_train)
>>>   print "Coefficient of determination on training
      set:",clf.score(X_train, y_train)
>>>   # create a k-fold cross validation iterator of k=5 folds
>>>   cv = KFold(X_train.shape[0], 5, shuffle=True,
      random_state=33)
>>>   scores = cross_val_score(clf, X_train, y_train, cv=cv)
>>>   print "Average coefficient of determination using 5-fold
      crossvalidation:",np.mean(scores)
```

First try: Linear Model

Start with a linear model called SGDRegressor, which tries to minimize squared loss.

```
>>> from sklearn import linear_model
>>> clf_sgd = linear_model.SGDRegressor(loss='squared_loss',
    penalty=None, random_state=42)
>>> train_and_evaluate(clf_sgd,X_train,y_train)
Coefficient of determination on training set: 0.743303511411
Average coefficient of determination using 5-fold crossvalidation:
0.715166411086
```

Print the hyperplane coefficients our method has calculated, which is as follows:

```
>>> print clf_sgd.coef_
[-0.07641527 0.06963738 -0.05935062 0.10878438 -0.06356188
0.37260998 -0.02912886 -0.20180631 0.08463607 -0.05534634
-0.19521922 0.0653966 -0.36990842]
```

First try: Linear Model

The penalization parameter for linear regression methods is introduced to avoid overfitting. It does this by penalizing those hyperplanes having some of their coefficients too large, seeking hyperplanes where each feature contributes more or less the same to the predicted value.

This parameter is generally the L2 norm (the squared sums of the coefficients) or the L1 norm (that is the sum of the absolute value of the coefficients).

```
>>> clf_sgd1 = linear_model.SGDRegressor(loss='squared_loss',
    penalty='l2', random_state=42)
>>> train_and_evaluate(clf_sgd1, X_train, y_train)
Coefficient of determination on training set: 0.743300616394
Average coefficient of determination using 5-fold crossvalidation:
0.715166962417
```

Second try: Support Vector Machines for regression

The regression version of SVM can be used instead to find the hyperplane.

```
>>> from sklearn import svm
>>> clf_svr = svm.SVR(kernel='linear')
>>> train_and_evaluate(clf_svr, X_train, y_train)
Coefficient of determination on training set: 0.71886923342
Average coefficient of determination using 5-fold crossvalidation:
0.707838419194
```

We can use a nonlinear function, for example, a polynomial function to approximate our data.

```
>>> clf_svr_poly = svm.SVR(kernel='poly')
>>> train_and_evaluate(clf_svr_poly, X_train, y_train)
Coefficient of determination on training set: 0.904109273301
Average coefficient of determination using 5-fold crossvalidation:
0.779288545488
```

Second try: Support Vector Machines for regression

Use a Radial Basis Function (RBF) kernel (actually it is the default kernel for SVMs in sci-kit)

```
>>> clf_svr_rbf = svm.SVR(kernel='rbf')
>>> train_and_evaluate(clf_svr_rbf, X_train, y_train)
Coefficient of determination on training set: 0.900131126377
Average coefficient of determination using 5-fold crossvalidation:
0.833665309761
```

Third try: Random Forests

The tree growing procedure is exactly the same, but at prediction time, when we arrive at a leaf, instead of reporting the majority class, we return a representative real value, for example, the average of the target values.

We will use Extra Trees, implemented in the ExtraTreesRegressor class within the sklearn.ensemble module. This method adds an extra level of randomization. It not only selects for each tree a different, random subset of features, but also randomly selects the threshold for each decision.

```
>>> from sklearn import ensemble
>>> clf_et=ensemble.ExtraTreesRegressor(n_estimators=10,
compute_importances=True, random_state=42)
>>> train_and_evaluate(clf_et, X_train, y_train)
Coefficient of determination on training set: 1.0
Average coefficient of determination using 5-fold crossvalidation: 0.8382
```

We completely eliminated underfitting.

Third try: Random Forests

An interesting feature of Extra Trees is that they allow computing the importance of each feature for the regression task.

```
>>> from sklearn import ensemble
>>> clf_et=ensemble.ExtraTreesRegressor(n_estimators=10,
compute_importances=True, random_state=42)
>>> train_and_evaluate(clf_et, X_train, y_train)
Coefficient of determination on training set: 1.0
Average coefficient of determination using 5-fold crossvalidation: 0.83823
```

We completely eliminated underfitting.

Third try: Random Forests

An interesting feature of Extra Trees is that they allow computing the importance of each feature for the regression task.

```
>>> print np.sort(zip(clf_et.feature_importances_,boston.feature_names),
    axis=0)
[['0.00166' 'AGE']
 ['0.01529' 'B']
 ['0.01838' 'CHAS']
 ['0.01892' 'CRIM']
 ['0.02264' 'DIS']
 ['0.02718' 'INDUS']
 ['0.02727' 'LSTAT']
 ['0.02735' 'NOX']
 ['0.03565' 'PTRATIO']
 ['0.05316' 'RAD']
 ['0.11986' 'RM']
 ['0.28110' 'TAX']
 ['0.35148' 'ZN']]
```

Third try: Random Forests

An interesting feature of Extra Trees is that they allow computing the importance of each feature for the regression task.

```
>>> print np.sort(zip(clf_et.feature_importances_,boston.feature_names),
    axis=0)
[['0.00166' 'AGE']
 ['0.01529' 'B']
 ['0.01838' 'CHAS']
 ['0.01892' 'CRIM']
 ['0.02264' 'DIS']
 ['0.02718' 'INDUS']
 ['0.02727' 'LSTAT']
 ['0.02735' 'NOX']
 ['0.03565' 'PTRATIO']
 ['0.05316' 'RAD']
 ['0.11986' 'RM']
 ['0.28110' 'TAX']
 ['0.35148' 'ZN']]
```

Evaluating the performance

```
>>> from sklearn import metrics
>>> def measure_performance(X, y, clf, show_accuracy=True,
show_classification_report=True, show_confusion_matrix=True,
show_r2_score=False):
>>>   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_confusion_matrix:
>>>     print "Confusion matrix"
>>>     print metrics.confusion_matrix(y, y_pred),"\n"
>>>   if show_r2_score:
>>>     print "Coefficient of determination:{0:.3f}".format(
>>>     metrics.r2_score(y, y_pred)),"\n"
>>> measure_performance(X_test, y_test, clf_et,
show_accuracy=False, show_classification_report=False,
show_confusion_matrix=False, show_r2_score=True)
Coefficient of determination:0.784
```

Unsupervised Learning

Supervised learning methods have an obvious drawback: data must be curated; a human being should have annotated the target class for a certain number of instances.

However, there are some things we can do without annotated data. We basically want to identify clusters of instances based on a similarity meassure.

Clustering involves finding groups where all elements in the group are similar, but objects in different groups are not.

We will present several approximations for clustering: k-means , affinity propagation, mean shift, and a model-based method called Gaussian Mixture Models.

Unsupervised Learning

Supervised learning methods have an obvious drawback: data must be curated; a human being should have annotated the target class for a certain number of instances.

However, there are some things we can do without annotated data. We basically want to identify clusters of instances based on a similarity meassure.

Clustering involves finding groups where all elements in the group are similar, but objects in different groups are not.

We will present several approximations for clustering: k-means , affinity propagation, mean shift, and a model-based method called Gaussian Mixture Models.

Domains of application: medicine, psychology, botany, sociology, biology, archeology, marketing, insurance, libraries, etc.

Unsupervised Learning: Dimensionality Reduction

Another useful application of unsupervised learning:

If your number of features is high, it may be useful to reduce it with an unsupervised step prior to supervised steps. Many of the unsupervised learning methods implement a transform method that can be used to reduce the dimensionality.

Dimensionality Reduction methods present a way to represent data points of a high dimensional dataset in a lower dimensional space, keeping (at least partly) their pattern structure.

Principal Component Analysis

Principal Component Analysis (PCA) is an orthogonal linear transformation that turns a set of possibly correlated variables into a new set of variables that are as uncorrelated as possible.

The new variables lie in a new coordinate system such that the greatest variance is obtained by projecting the data in the first coordinate, the second greatest variance by projecting in the second coordinate, and so on.

These new coordinates are called principal components; we have as many principal components as the number of original dimensions, but we keep only those with high variance.

Each new principal component that is added to the principal components set must comply with the restriction that it should be orthogonal (that is, uncorrelated) to the remaining principal components.

https://georgemdallas.wordpress.com/2013/10/30/principal-component-analysis-4-dummies-eigenvectors-eigenvalues-and-dimension-reduction/
http://setosa.io/ev/principal-component-analysis/

Principal Component Analysis

PCA can be seen as a method that reveals the internal structure of data; it supplies the user with a lower dimensional shadow of the original objects.

For our learning methods, PCA will allow us to reduce a high-dimensional space into a low-dimensional one while preserving as much variance as possible.

This is very useful for two major purposes: Visualization and feature selection.

Working example: dataset of handwritten digits digitalized in matrices of 8x8 pixels (64 attributes). How can we visualize the distribution of instances?

We will use PCA to visualize the distribution of instances in a two-dimensional scatter graph.

Principal Component Analysis

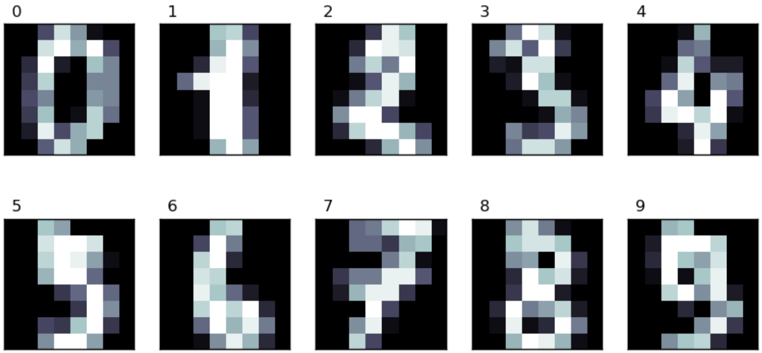
Load the dataset:

```
>>> from sklearn.datasets import load_digits
>>> digits = load_digits()
>>> X_digits, y_digits = digits.data, digits.target
>>> print digits.keys()
['images', 'data', 'target_names', 'DESCR', 'target']
```

Print the digits:

```
>>> import matplotlib.pyplot as plt
>>> n_row, n_col = 2, 5
>>> def print_digits(images, y, max_n=10):
>>>   # set up the figure size in inches
>>>   fig = plt.figure(figsize=(2. * n_col, 2.26 * n_row))
>>>   i=0
>>>   while i < max_n and i < images.shape[0]:
>>>     p = fig.add_subplot(n_row, n_col, i + 1, xticks=[], yticks=[])
>>>     p.imshow(images[i], cmap=plt.cm.bone, interpolation='nearest')
>>>    # label the image with the target value
>>>    p.text(0, -1, str(y[i]))
>>>    i = i + 1
>>>
>>> print_digits(digits.images, digits.target, max_n=10)
>>> plt.show()
```

Principal Component Analysis



Principal Component Analysis

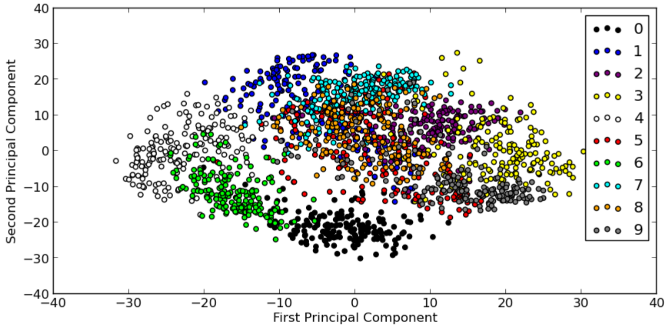
Define a function that will plot a scatter with the first two-dimensional points that will be obtained by a PCA transformation.

```
>>> def plot_pca_scatter():
>>> colors = ['black', 'blue', 'purple', 'yellow', 'white',
      'red', 'lime', 'cyan', 'orange', 'gray']
>>> for i in xrange(len(colors)):
>>>   px = X_pca[:, 0][y_digits == i]
>>>   py = X_pca[:, 1][y_digits == i]
>>>   plt.scatter(px, py, c=colors[i])
>>> plt.legend(digits.target_names)
>>> plt.xlabel('First Principal Component')
>>> plt.ylabel('Second Principal Component')
```

Principal Component Analysis

We have various classes: PCA, ProbabilisticPCA, RandomizedPCA, and KernelPCA. We transform instances of 64 features to instances of just 10.

```
from sklearn.decomposition import PCA
estimator = PCA(n_components=10)
X_pca = estimator.fit_transform(X_digits)
plot_pca_scatter()
plt.show()
```



Principal Component Analysis

Observations:

- There are 10 different classes corresponding to the 10 digits at first sight.
- Instances are clearly grouped in clusters according to their target class, except digit 5.
- Class corresponding to the digit 0 is the most separated cluster, i.e., easiest to separate from the rest.
- For topological distribution, we may predict that contiguous classes correspond to similar digits, i.e., more difficult to separate. Example: he clusters corresponding to digits 9 and 3 appear contiguous.

Clustering handwritten digits with k-means

K-means is the most popular clustering algorithm.

It belongs to the class of partition algorithms that simultaneously partition data points into distinct groups called clusters.

The main idea behind k-means is to find a partition of data points such that the squared distance between the cluster mean and each point in the cluster is minimized.

This method assumes that you know a priori the number of clusters your data should be divided into.

Clustering handwritten digits with k-means

We will work on the problem of clustering handwritten digits

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from sklearn.datasets import load_digits
>>> from sklearn.preprocessing import scale
>>> digits = load_digits()
>>> data = scale(digits.data)
>>> images_index = np.arange(digits.images.shape[0])
>>>
>>> def print_digits(images, images_index,y,max_n=10):
>>> # set up the figure size in inches
>>> fig = plt.figure(figsize=(12, 12))
>>> fig.subplots_adjust(left=0, right=1, bottom=0, top=1,
hspace=0.05, wspace=0.05)
>>> i = 0
>>> while i <max_n and i <images_index.shape[0]:
>>>   # plot the images in a matrix of 20x20
>>>   p = fig.add_subplot(20, 20, i + 1, xticks=[],yticks=[])
>>>   p.imshow(images[images_index[i]], cmap=plt.cm.bone)
>>>   # label the image with the target value
>>>   p.text(0, 14, str(y[i]))
>>>   i = i + 1
>>>
>>> print_digits(digits.images, images_index, max_n=10)
```

Clustering handwritten digits with k-means

0
1
2
3
4
5
6
7
8
9

As usual, we must separate train and testing sets as follows:

```
>>> from sklearn.cross_validation import train_test_split
>>> X_train, X_test, y_train, y_test, images_index_train,
images_index_test = train_test_split(data, digits.target, digits.images, t
random_state=42)
>>>
>>> n_samples, n_features = X_train.shape
>>> n_digits = len(np.unique(y_train))
>>> labels = y_train
```

Clustering handwritten digits with k-means

What the k-means algorithm does is:

1. Select an initial set of cluster centers at random.
2. Find the nearest cluster center for each data point, and assign the data point closest to that cluster.
3. Compute the new cluster centers, averaging the values of the cluster data points, and repeat until cluster membership stabilizes; that is, until a few data points change their clusters after each iteration.

The initial set of cluster centers could greatly affect the clusters found.

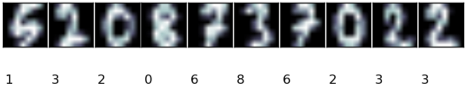
The usual approach to mitigate this is to try several initial sets and select the set with minimal value for the sum of squared distances between cluster centers (or inertia).

Clustering handwritten digits with k-means

```
>>> from sklearn import cluster
>>> clf = cluster.KMeans(init='k-means++', n_clusters=10, random_state=42)
>>> clf.fit(X_train)
```

Print the value of the labels_ attribute of the classifier, we get a list of the cluster numbers associated to each training instance.

```
>>> print_digits(digits.images, images_index_train, clf.labels_, max_n=10)
```



Note that the cluster number has nothing to do with the real number value!

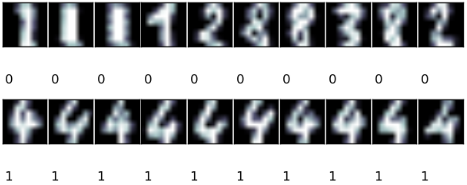
we have not used the class to classify; we only grouped images by similarity.

To predict the clusters for training data, we use the usual predict method of the classifier:

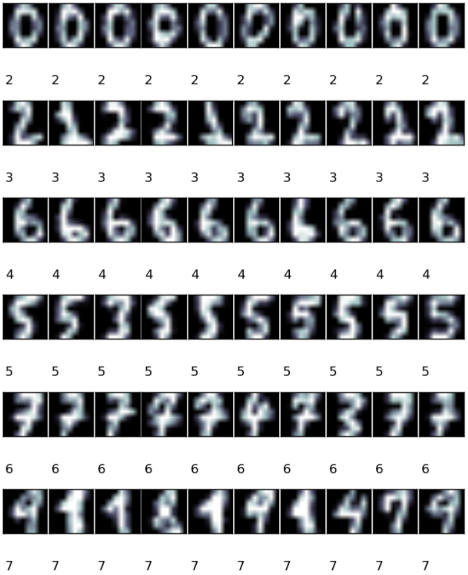
```
>>> y_pred=clf.predict(X_test)
>>> def print_cluster(images, images_index, y_pred, cluster_number):
>>>   images_index =  images_index[y_pred == cluster_number]
>>>   y_pred = y_pred[y_pred == cluster_number]
>>>   print_digits(images, images_index, y_pred, max_n=10)
```

Clustering handwritten digits with k-means

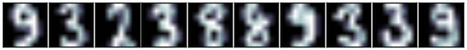
```
>>> y_pred=clf.predict(X_test)
>>> def print_cluster(images, images_index, y_pred, cluster_number):
>>>   images_index =  images_index[y_pred == cluster_number]
>>>   y_pred = y_pred[y_pred == cluster_number]
>>>   print_digits(images, images_index, y_pred, max_n=10)
>>> for i in range(10):
>>    print_cluster(digits.images, images_index_test, y_pred, i)
```

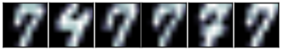


Clustering handwritten digits with k-means



Clustering handwritten digits with k-means

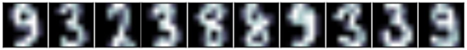


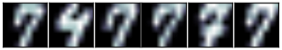


Recursos

http://hub.darcs.net/gh/ia2014/

Clustering handwritten digits with k-means





Clustering handwritten digits with k-means

How can we evaluate our performance?

Compute the adjusted Rand index between our cluster assignment and the expected one.

It is a similar measure for accuracy, but it takes into account the fact that classes can have different names in both assignments.

The adjusted index tries to deduct from the result coincidences that have occurred by chance.

When you have the exact same clusters in both sets, the Rand index equals one, while it equals zero when there are no clusters sharing a data point.

```
>>> from sklearn import metrics
>>> print "Adjusted rand score:{:.2}".format(metrics.
           adjusted_rand_score(y_test, y_pred))
Adjusted rand score:0.57
```

Clustering handwritten digits with k-means

The confusion matrix:

```
>>> print metrics.confusion_matrix(y_test, y_pred)
[[ 0  0 43  0  0  0  0  0  0  0]
 [20  0  0  7  0  0  0 10  0  0]
 [ 5  0  0 31  0  0  0  1  1  0]
 [ 1  0  0  1  0  1  4  0 39  0]
 [ 1 50  0  0  0  0  1  2  0  1]
 [ 1  0  0  0  1 41  0  0 16  0]
 [ 0  0  1  0 44  0  0  0  0  0]
 [ 0  0  0  0  0  1 34  1  0  5]
 [21  0  0  0  0  3  1  2 11  0]
 [ 0  0  0  0  0  2  3  3 40  0]]
```

Clustering handwritten digits with k-means

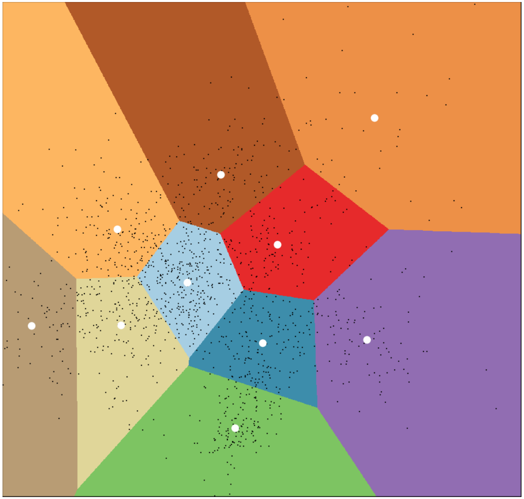
Graphically show how k-means clusters look like:

```
>>> from sklearn import decomposition
>>> pca = decomposition.PCA(n_components=2).fit(X_train)
>>> reduced_X_train = pca.transform(X_train)
>>> # Step size of the mesh.
>>> h = .01
>>> # point in the mesh [x_min, m_max]x[y_min, y_max].
>>> x_min, x_max = reduced_X_train[:, 0].min() + 1,
reduced_X_train[:, 0].max() - 1
>>> y_min, y_max = reduced_X_train[:, 1].min() + 1,
reduced_X_train[:, 1].max() - 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
np.arange(y_min, y_max, h))
>>> kmeans = cluster.KMeans(init='k-means++', n_clusters=n_digits,
n_init=10)
```

Clustering handwritten digits with k-means

```
>>> kmeans.fit(reduced_X_train)
>>> Z = kmeans.predict(np.c_[xx.ravel(), yy.ravel()])
>>> # Put the result into a color plot
>>> Z = Z.reshape(xx.shape)
>>> plt.figure(1)
>>> plt.clf()
>>> plt.imshow(Z, interpolation='nearest', extent=(xx.min(), xx.max(), yy.
>>> plt.plot(reduced_X_train[:, 0], reduced_X_train[:, 1], 'k.', markersiz
>>> # Plot the centroids as a white X
>>> centroids = kmeans.cluster_centers_
>>> plt.scatter(centroids[:, 0], centroids[:, 1],marker='.', s=169, linewi
>>> plt.title('K-means clustering on the digits dataset (PCA reduced data)
>>> plt.xlim(x_min, x_max)
>>> plt.ylim(y_min, y_max)
>>> plt.xticks(())
>>> plt.yticks(())
>>> plt.show()
```

Clustering handwritten digits with k-means



Alternative clustering methods

A typical problem for clustering is that most methods require the number of clusters we want to identify.

There are also some methods that try to automatically calculate the number of clusters.

Affinity Propagation: a method that looks for instances that are the most representative of others, and uses them to describe the clusters.

```
>>> aff = cluster.AffinityPropagation()
>>> aff.fit(X_train)
>>> print aff.cluster_centers_indices_.shape
(112,)
```

Affinity propagation detected 112 clusters in our training set!

The cluster_centers_indices attribute represents what Affinity Propagation found as the canonical elements of each cluster.

Alternative clustering methods

MeanShift():

```
>>> ms = cluster.MeanShift()
>>> ms.fit(X_train)
>>> print ms.cluster_centers_.shape
(18, 64)
```

The cluster_centers attribute shows the hyperplane cluster centroids.

For the last two methods, we cannot use the Rand score to evaluate performance because we do not have a canonical set of clusters to compare with.

But, we can measure the inertia of the clustering, since inertia is the sum of distances from each data point to the centroid.

Alternative clustering methods

A probabilistic approach to clustering: **Gaussian Mixture Models (GMM)**

GMM assumes that data comes from a mixture of finite Gaussian distributions with unknown parameters.

A Gaussian mixture model is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters.

One can think of mixture models as generalizing k-means clustering to incorporate information about the covariance structure of the data as well as the centers of the latent Gaussians.

```
>>> from sklearn import mixture
>>> gm = mixture.GMM(n_components=n_digits,
covariance_type='tied', random_state=42)
>>> gm.fit(X_train)
GMM(covariance_type='tied', init_params='wmc', min_covar=0.001,n_
components=10, n_init=1, n_iter=100, params='wmc',random_
state=42,thresh=0.01)
```

Alternative clustering methods

Let's see how it performs on our testing data:

```
>>> # Print train clustering and confusion matrix
>>> y_pred = gm.predict(X_test)
>>> print "Adjusted randscore:{:.2}".format(metrics.
           adjusted_rand_score(y_test,y_pred))
Adjusted rand score:0.65
>>> print "Homogeneity score:{:.2}".format(metrics.
           homogeneity_score(y_test, y_pred))
Homogeneity score:0.74
>>> print "Completeness score: {:.2}".format(metrics.
           completeness_score(y_test, y_pred))
Completeness score: 0.79
```

Homogeneity is a number between 0.0 and 1.0 (greater is better). A value of 1.0 indicates that clusters only contain data points from a single class; that is, clusters effectively group similar instances.

Completeness, on the other hand, is satisfied when every data point of a given class is within the same cluster (meaning that we have grouped all possible instances of the class, instead of building several uniform but smaller clusters).

Advanced Features

We will see more advanced topics on:

- **Feature extraction:** In real-world problems, usually data is not already expressed by attribute/ float value pairs, but through more complex structures or is not structured at all. We will see feature extraction techniques.
- **Feature selection:** Not all initial features will be useful for our algorithms to learn from; in fact, some of them may degrade our performance. We will see how to select the most adequate feature set.
- **Model selection:** Many of the machine learning algorithms have parameters that must be set in order to use them. We will see how to select the most promising hyperparameters to our algorithms.

Feature Extraction

We can identify two different steps in this task:

- **Obtain features:** Process the source data and extrac the learning instances, usually in the form of feature/value pairs where the value can be an integer or float value, a string, a categorical value, and so on.
- **Convert features:** Most scikit-learn algorithms assume as an input a set of instances represented as a list of float-valued features. We will focus on this point.

Feature Extraction: convert features

We can build ad hoc procedures to convert the source data.

However, there are tools that can help us to obtain a suitable representation. The Python package pandas (http://pandas.pydata. org/), for example, provides data structures and tools for data analysis.

```
$ sudo apt-get install python-pandas
```

Let’s start by importing the original titanic.csv data into a pandas DataFrame data structure.

```
>>> import pandas as pd
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

Feature Extraction: convert features

```
>>> titanic = pd.read_csv('titanic.csv')
>>> print titanic
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1313 entries, 0 to 1312
Data columns:
row.names     1313  non-null values
pclass        1313  non-null values
survived      1313  non-null values
name          1313  non-null values
age           633  non-null values
embarked      821  non-null values
home.dest     754  non-null values
room          77  non-null values
ticket        69  non-null values
boat          347  non-null values
sex           1313  non-null values
dtypes: float64(1), int64(2), object(8)
```

Feature Extraction: convert features

Lets inspect some features to see what they look like.

```
>>> print titanic.head()[['pclass', 'survived', 'age', 'embarked','boat',
  pclass  survived      age     embarked   boat     sex
0    1st         1  29.0000  Southampton      2  female
1    1st         0   2.0000  Southampton    NaN  female
2    1st         0  30.0000  Southampton  (135)    male
3    1st         0  25.0000  Southampton    NaN  female
4    1st         1   0.9167  Southampton     11    male
```

Remenber that scikit-learn methods expect real numbers as feature values.

We will use the scikit-learn method, DictVectorizer, which automatically builds these features from the different original feature values.

Feature Extraction: convert features

The one_hot_dataframe method (based on the script at https://gist.github.com/kljensen/5452382) takes a pandas DataFrame data structure and a list of columns and encodes each column into the necessary 1-of-K features. If the replace parameter is True, it will also substitute the original column with the new set.

```
>>> from sklearn import feature_extraction
>>> def one_hot_dataframe(data, cols, replace=False):
>>>   vec = feature_extraction.DictVectorizer()
>>>   mkdict = lambda row: dict((col, row[col]) for col in cols)
>>>   vecData = pd.DataFrame(vec.fit_transform(
>>>     data[cols].apply(mkdict, axis=1)).toarray())
>>>   vecData.columns = vec.get_feature_names()
>>>   vecData.index = data.index
>>>   if replace:
>>>     data = data.drop(cols, axis=1)
>>>     data = data.join(vecData)
>>> return (data, vecData)
```

Feature Extraction: convert features

The one_hot_dataframe method (based on the script at https://gist.github.com/kljensen/5452382) takes a pandas DataFrame data structure and a list of columns and encodes each column into the necessary 1-of-K features. If the replace parameter is True, it will also substitute the original column with the new set.

```
>>> from sklearn import feature_extraction
>>> def one_hot_dataframe(data, cols, replace=False):
>>>   vec = feature_extraction.DictVectorizer()
>>>   mkdict = lambda row: dict((col, row[col]) for col in cols)
>>>   #vecData = pd.DataFrame(vec.fit_transform(
>>>   #  data[cols].apply(mkdict, axis=1)).toarray())
>>> REVISAR!!! vecData = pd.DataFrame(vec.fit_transform(data[cols].
>>>             to_dict(outtype='records')).toarray())

>>>   vecData.columns = vec.get_feature_names()
>>>   vecData.index = data.index
>>>   if replace:
>>>     data = data.drop(cols, axis=1)
>>>     data = data.join(vecData)
>>> return (data, vecData)
```

Feature Extraction: convert features

```
>>> print titanic
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1313 entries, 0 to 1312
Data columns:
row.names                1313  non-null values
survived                 1313  non-null values
name                     1313  non-null values
age                      633  non-null values
home.dest                754  non-null values
room                     77  non-null values
ticket                   69  non-null values
boat                     347  non-null values
embarked                 821  non-null values
embarked=Cherbourg       1313  non-null values
embarked=Queenstown      1313  non-null values
embarked=Southampton     1313  non-null values
pclass=1st               1313  non-null values
pclass=2nd               1313  non-null values
pclass=3rd               1313  non-null values
sex=female               1313  non-null values
sex=male                 1313  non-null values
dtypes: float64(10), int64(2), object(5)
```

Feature Extraction: convert features

Note that the embarked feature has not disappeared, because the original embarked attribute included NaN values, indicating a missing value; in those cases, every feature based on embarked will be valued 0, but the original feature whose value is NaN remains, indicating the feature is missing for certain instances.

We encode the remaining categorical attributes:

```
>>> titanic, titanic_n = one_hot_dataframe(titanic, ['home.dest',
'room', 'ticket', 'boat'], replace=True)
```

Pandas allow us to replace misssing values with a fixed value using the fillna method. We will use the mean age for the age feature, and 0 for the remaining missing attributes.

```
>>> mean = titanic['age'].mean()
>>> titanic['age'].fillna(mean, inplace=True)
>>> titanic.fillna(0, inplace=True)
```

Feature Extraction: convert features

Lets test it:

```
>>> from sklearn.cross_validation import train_test_split
>>> titanic_target = titanic['survived']
>>> titanic_data = titanic.drop(['name', 'row.names', 'survived'],axis=1)
>>> X_train, X_test, y_train, y_test = train_test_split(titanic_data,
titanic_target, test_size=0.25, random_state=33)

>>> from sklearn import tree
>>> dt = tree.DecisionTreeClassifier(criterion='entropy')
>>> dt = dt.fit(X_train, y_train)
>>> from sklearn import metrics
>>> y_pred = dt.predict(X_test)
>>> print "Accuracy:{0:.3f}".
    format(metrics.accuracy_score(y_test, y_pred)), "\n"
#Accuracy:0.839
```

Feature Selection

There are two main reasons why we would want to restrict the number of features used:

- For those methods that reduce the number of instances used to refine the model at each step, it is possible that irrelevant features could suggest correlations between features and target classes that arise just by chance and do not correctly model the problem.
- Having certain over-specific features may lead to poor generalization. Besides, some features may be highly correlated, and will simply add redundant information.
- A large number of features could greatly increase the computation time without a corresponding classifier improvement.

Feature Selection

We will try to limit the features to the most relevant ones.

What do we mean by relevant?

A general approach is to find the smallest set of features that correctly characterize the training data.

If a feature always coincides with the target class (that is, it is a perfect predictor), it is enough to characterize the data. On the other hand, if a feature always has the same value, its prediction power will be very low.

We need kind kind of evaluation function that, when given a potential feature, returns a score of how useful the feature is.

We can, for instance, use a statistical test that measures how probable it is that two random variables (say, a given feature and the target class) are independent; that is, there is no correlation between them.

Feature Selection

Scikit-learn provides several methods in the feature_selection module. We will use the SelectPercentile method that, when given a statistical test, selects a userspecified percentile of features with the highest scoring.

The most popular statistical test is the chi-squared statistic.

```
>>> from sklearn import feature_selection
>>> fs = feature_selection.SelectPercentile(
feature_selection.chi2, percentile=20)
>>> X_train_fs = fs.fit_transform(X_train, y_train)
```

The X_train_fs array now has the statistically more important features.

```
>>> dt.fit(X_train_fs, y_train)
>>> X_test_fs = fs.transform(X_test)
>>> y_pred_fs = dt.predict(X_test_fs)
>>> print "Accuracy:{0:.3f}".format(metrics.
      accuracy_score(y_test,y_pred_fs)),"\n"
```

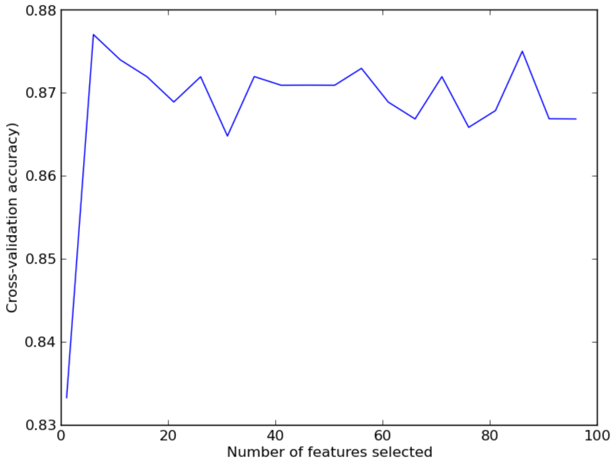
Feature Selection

Is it possible to find the optimal number of features?. Brute-force on different numbers of features while measuring performance with cross-validation.

```
>>> from sklearn import cross_validation
>>> percentiles = range(1, 100, 5)
>>> results = []
>>> for i in range(1,100,5):
>>>  fs = feature_selection.SelectPercentile(
>>>       feature_selection.chi2, percentile=i)
>>>  X_train_fs = fs.fit_transform(X_train, y_train)
>>>  scores = cross_validation.cross_val_score(dt, X_train_fs,
              y_train, cv=5)
>>>  results = np.append(results, scores.mean())
>>> optimal_percentil = np.where(results == results.max())[0]
>>> print "Optimal number of features:{0}".
               format(percentiles[optimal_percentil]), "\n"
>>> # Plot number of features VS. cross-validation scores
>>> import pylab as pl
>>> pl.figure()
>>> pl.xlabel("Number of features selected")
>>> pl.ylabel("Cross-validation accuracy)")
>>> pl.plot(percentiles, results)
```

Feature Selection

REVISAR ONE HOT!!! La figura parece rara ...



Feature Selection

Let’s see if this actually improved performance on the testing set:

```
>>> fs = feature_selection.SelectPercentile( feature_selection.chi2,
         percentile=percentiles[optimal_percentil])
>>> X_train_fs = fs.fit_transform(X_train, y_train)
>>> dt.fit(X_train_fs, y_train)
>>> X_test_fs = fs.transform(X_test)
>>> y_pred_fs = dt.predict(X_test_fs)
>>> print "Accuracy:{0:.3f}".
         format(metrics.accuracy_score(y_test, y_pred_fs)), "\n"
Accuracy:0.857
```

Feature Selection

Can we improve our model using different parameters?

```
>>> dt = tree.DecisionTreeClassifier(criterion='entropy')
>>> scores = cross_validation.cross_val_score(dt, X_train_fs, y_train, cv=
>>> print "Entropy criterion accuracy on cv: {0:.3f}".format(scores.mean()
Entropy criterion accuracy on cv: 0.878
>>> dt = tree.DecisionTreeClassifier(criterion='gini')
>>> scores = cross_validation.cross_val_score(dt, X_train_fs, y_train, cv=
>>> print "Gini criterion accuracy on cv: {0:.3f}".format(scores.mean())
Gini criterion accuracy on cv: 0.882
```

How about its performance on the test set?

```
>>> dt.fit(X_train_fs, y_train)
>>> X_test_fs = fs.transform(X_test)
>>> y_pred_fs = dt.predict(X_test_fs)
>>> print "Accuracy:{0:.3f}".
      format(metrics.accuracy_score(y_test,y_pred_fs)),"\n"
Accuracy:0.863
```

Model Selection

Selecting the best parameter (hyperparameters) configuration is also an important step.

This is known as the configuration algorithm problem.

We will work on the previous text-classification problem.

We used a TF-IDF vectorizer alongside a multinomial NaÃ¯ve Bayes (NB) algorithm, and saw that setting alpha parameter to 0.01 gave us good results.

How can we be sure 0.01 is the best value?.

Model Selection

We will work only with 3,000 instances.

```
>>> from sklearn.datasets import fetch_20newsgroups
>>> news = fetch_20newsgroups(subset='all')
>>> n_samples = 3000
>>> X_train = news.data[:n_samples]
>>> y_train = news.target[:n_samples]

>>> from sklearn.naive_bayes import MultinomialNB
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.feature_extraction.text import TfidfVectorizer

>>> def get_stop_words():
>>>   result = set()
>>>   for line in open('stopwords_en.txt', 'r').readlines():
>>>     result.add(line.strip())
>>>   return result
>>> stop_words = get_stop_words()
>>> clf = Pipeline([('vect', TfidfVectorizer(stop_words=stop_words,
>>>       token_pattern=ur"\b[a-z0-9_\-\.]+[a-z][a-z0-9_\-\.]+\b",
>>>       )), ('nb', MultinomialNB(alpha=0.01)),])
```

Model Selection

We evaluate our algorithm with a three-fold cross-validation:

```
>>> def evaluate_cross_validation(clf, X, y, K):
>>>   # create a k-fold croos validation iterator of k=5 folds
>>>   cv = KFold(len(y), K, shuffle=True, random_state=0)
>>>   # by default the score used is the one returned by score method of t
>>>   scores = cross_val_score(clf, X, y, cv=cv)
>>>   print scores
>>>   print ("Mean score: {0:.3f} (+/-{1:.3f})").format(np.mean(scores), s
>>> evaluate_cross_validation(clf, X_train, y_train, 3)
[ 0.812  0.808  0.822]
Mean score: 0.814 (+/-0.004)
```

The next function will train the algorithm with a list of values, each time obtaining an accuracy score calculated by performing k-fold cross-validation on the training instances.

Model Selection

```
import matplotlib.pyplot as plt
def calc_params(X, y, clf, param_values, param_name, K):
  # initialize training and testing scores with zeros
  train_scores = np.zeros(len(param_values))
  test_scores = np.zeros(len(param_values))

  # iterate over the different parameter values
  for i, param_value in enumerate(param_values):
    print param_name, ' = ', param_value
    # set classifier parameters
    clf.set_params(**{param_name:param_value})
    # initialize the K scores obtained for each fold
    k_train_scores = np.zeros(K)
    k_test_scores = np.zeros(K)
    # create KFold cross validation
    cv = KFold(n_samples, K, shuffle=True, random_state=0)
    # iterate over the K folds
    for j, (train, test) in enumerate(cv):
      clf.fit([X[k] for k in train], y[train])
      k_train_scores[j] = clf.score([X[k] for k in train], y[train])
      k_test_scores[j] = clf.score([X[k] for k in test], y[test])
    train_scores[i] = np.mean(k_train_scores)
    test_scores[i] = np.mean(k_test_scores)
```

Model Selection

```
# plot the training and testing scores in a log scale
plt.semilogx(param_values, train_scores, alpha=0.4, lw=2, c='b')
plt.semilogx(param_values, test_scores, alpha=0.4, lw=2, c='g')
plt.xlabel("Alpha values")
plt.ylabel("Mean cross-validation accuracy")
# return the training and testing scores on each parameter value
return train_scores, test_scores
```

The function accepts six arguments: the feature array, the target array, the classifier object to be used, the list of parameter values, the name of the parameter to adjust, and the number of K folds to be used in the crossvalidation evaluation.

Model Selection

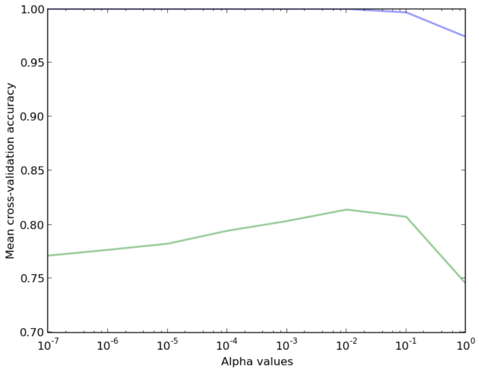
Ggenerate a list of alpha values spaced evenly on a log scale.

```
>>> alphas = np.logspace(-7, 0, 8)
>>> print alphas
[  1.00000000e-07   1.00000000e-06   1.00000000e-05   1.00000000e-04
   1.00000000e-03   1.00000000e-02   1.00000000e-01   1.00000000e+00]

>>> train_scores, test_scores = calc_params(X_train, y_train, clf,
alphas, 'nb__alpha', 3)
```

Model Selection

The line at the top corresponds to the training accuracy and the one at the bottom to the testing accuracy.



The best testing accuracy is obtained with an alpha value in the range of 10-2 and 10-1. Below this range, the classifier shows signs of overfitting (the training accuracy is high but the testing accuracy is lower than it could be).

Model Selection

Print the scores vector to look at the actual values.

```
print 'training scores: ', train_scores
print 'testing scores: ', test_scores
training scores:  [ 1.          1.          1.          1.          1.
  0.99683333  0.97416667]
testing scores:  [ 0.77133333  0.77666667  0.78233333  0.79433333  0.80333
  0.80733333  0.74533333]
```

The best value is obtained for alpha 0.01.

Let's use the previous function to adjust another classifier that uses a Support Vector Machines (SVM) instead of MultinomialNB:

Model Selection

Print the scores vector to look at the actual values.

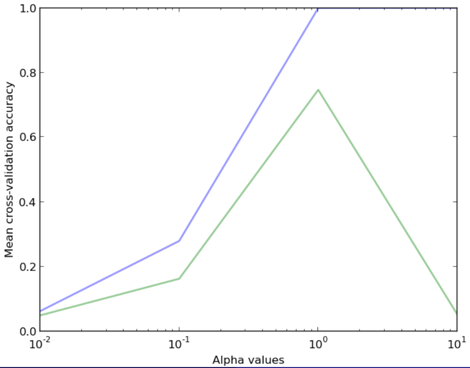
```
print 'training scores: ', train_scores
print 'testing scores: ', test_scores
training scores:  [ 1.          1.          1.          1.          1.
  0.99683333  0.97416667]
testing scores:  [ 0.77133333  0.77666667  0.78233333  0.79433333  0.8033
  0.80733333  0.74533333]
```

The best value is obtained for alpha 0.01.

Let's use the previous function to adjust another classifier that uses a Support Vector Machines (SVM) instead of MultinomialNB:

Model Selection

```
from sklearn.svm import SVC
clf = Pipeline([('vect', TfidfVectorizer(
  stop_words=stop_words,
  token_pattern=ur"\b[a-z0-9_\-\.]+[a-z][a-z0-9_\-\.]+\b",
  )), ('svc', SVC()),])
gammas = np.logspace(-2, 1, 4)
train_scores, test_scores = calc_params(X_train, y_train, clf,gammas,
'svc__gamma', 3)
```



Model Selection: Grid Search

The SVC class constructor has other parameters ... how to look for good combinations?

We have a very useful class named GridSearchCV within the sklearn.grid_search module.

Let's use it to adjust the C and the gamma parameters at the same time.

```
>>> from sklearn.grid_search import GridSearchCV
>>> parameters = {
>>>   'svc__gamma': np.logspace(-2, 1, 4),
>>>   'svc__C': np.logspace(-1, 1, 3),
>>> }
>>> clf = Pipeline([
>>>   ('vect', TfidfVectorizer(
>>>     stop_words=stop_words,
>>>     token_pattern=ur"\b[a-z0-9_\-\.]+[a-z][a-z0-9_\-\.]+\b",
>>>   )),
>>>   ('svc', SVC()),
>>> ])

>>> gs = GridSearchCV(clf, parameters, verbose=2, refit=False, cv=3)
>>> gs.fit(X_train, y_train)
>>> print gs.best_params_, gs.best_score
```

Model Selection: Parallel Grid Search

Grid search calculation grows exponentially with each parameter and its possible values we want to tune. We could reduce our response time if we calculate each of the combinations in parallel instead of sequentially.

TODO