Chapter 4. Advanced Features

In the previous chapters we have studied several algorithms for very different tasks, from classification and regression to clustering and dimensionality reduction. We showed how we can apply these algorithms to predict results when faced with new data. That is what machine learning is all about. In this last chapter, we want to show some important concepts and methods you should take into account if you want to do real-world machine learning.

- 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 learn **feature extraction** techniques that will allow us to extract scikit-learn features from data.
- From the initial set of available features, not all of them will be useful for our algorithms to learn from; in fact, some of them may degrade our performance. We will address the problem of selecting the most adequate feature set, a process known as **feature selection**.
- Finally, as we have seen in the examples in this book, many of the machine learning algorithms have parameters that must be set in order to use them. To do that, we will review **model selection** techniques; that is, methods to select the most promising hyperparameters to our algorithms.

All these steps are crucial in order to obtain decent results when working with machine learning applications.

Feature extraction

The usual scenario for learning tasks such as those presented in this book include a list of instances (represented as feature/value pairs) and a special feature (the target class) that we want to predict for future instances based on the values of the remaining features. However, the source data does not usually come in this format. We have to extract what we think are potentially useful features and convert them to our learning format. This process is called feature extraction or feature engineering, and it is an often underestimated but very important and time-consuming phase in most real-world machine learning tasks. We can identify two different steps in this task:

- **Obtain features**: This step involves processing the source data and extracting 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. The method used for extraction depends heavily on how the data is presented. For

example, we can have a set of pictures and generate an integer-valued feature for each pixel, indicating its color level, as we did in the face recognition example in Chapter 2, *Supervised Learning*. Since this is a very task-dependent job, we will not delve into details and assume we already have this setting for our examples.

- **Convert features**: Most scikit-learn algorithms assume as an input a set of instances represented as a list of float-valued features. How to get these features will be the main subject of this section.

We can, as we did in Chapter 2, *Supervised Learning*, build ad hoc procedures to convert the source data. There are, however, 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. It aims to provide similar features to those of R, the popular language and environment for statistical computing. We will use pandas to import the Titanic data we presented in Chapter 2, *Supervised Learning*, and convert them to the scikit-learn format.

Let's start by importing the original `titanic.csv` data into a pandas `DataFrame` data structure (`DataFrame` is essentially a two-dimensional labeled data structure where columns can potentially include different data types and each row represents an instance). As usual, we previously import the `numpy` and `pyplot` packages.

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

Then we import the Titanic data with pandas.

```
>>> titanic = pd.read_csv('data/titanic.csv')
>>> print titanic
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1313 entries, 0 to 1312
Data columns (total 11 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)
```

You can see that each `csv` column has a corresponding feature into the `DataFrame`, and that the feature type is induced from the available data. We can inspect some features to see what they look like.

```
>>> print titanic.head()[['pclass', 'survived', 'age',
'embarked',
    'boat', 'sex']]
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
```

The main difficulty we have now is that scikit-learn methods expect real numbers as feature values. In Chapter 2, *Supervised Learning*, we used the `LabelEncoder` and `OneHotEncoder` preprocessing methods to manually convert certain categorical features into 1-of-K values (generating a new feature for each possible value; valued `1` if the original feature had the corresponding value and `0` otherwise). This time, we will use a similar scikit-learn method, `DictVectorizer`, which automatically builds these features from the different original feature values. Moreover, we will program a method to encode a set of columns in a unique step.

```
>>> 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)
```

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. Let's see it applied to the categorical `pclass`, `embarked`, and `sex` features (`titanic_n` only contains the previously created columns):

```
>>> titanic,titanic_n = one_hot_dataframe(titanic, ['pclass',
    'embarked', 'sex'], replace=True)
>>> titanic.describe()
<class 'pandas.core.frame.DataFrame'>
```

```
Index: 8 entries, count to max
Data columns (total 12 columns):
row.names                8  non-null values
survived                 8  non-null values
age                      8  non-null values
embarked                 8  non-null values
embarked=Cherbourg       8  non-null values
embarked=Queenstown      8  non-null values
embarked=Southampton     8  non-null values
pclass=1st               8  non-null values
pclass=2nd               8  non-null values
pclass=3rd               8  non-null values
sex=female               8  non-null values
sex=male                 8  non-null values
dtypes: float64(12)
```

The `pclass` attribute has been converted to three `pclass=1st`, `pclass=2nd`, `pclass=3rd` features, and similarly for the other two features. Note that the `embarked` feature has not disappeared, This is due to the fact that 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. Next, we encode the remaining categorical attributes:

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

We also have to deal with missing values, since `DecisionTreeClassifier` we plan to use does not admit them on input. Pandas allow us to replace them 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)
```

Now, all of our features (except for `Name`) are in a suitable format. We are ready to build the test and training sets, as usual.

```
>>> 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)
```

We decided to simply drop the `name` attribute, since we do not expect it to be informative about the survival status (we have one different value for each instance, so we can generalize over it). We also specified the `survived` feature as the target class, and consequently eliminated it from the training vector.

Let's see how a decision tree works with the current feature set.

```
>>> 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

Until now, when training our decision tree, we used every available feature in our learning dataset. This seems perfectly reasonable, since we want to use as much information as there is available to build our model. There are, however, two main reasons why we would want to restrict the number of features used:

- Firstly, for some methods, especially those (such as decision trees) 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. This aspect is also related to overfitting; having certain over-specific features may lead to poor generalization. Besides, some features may be highly correlated, and will simply add redundant information.
- The second reason is a real-world one. A large number of features could greatly increase the computation time without a corresponding classifier improvement. This is of particular importance when working with Big Data, where the number of instances and features could easily grow to several thousand or more. Also, in relation to the curse of dimensionality, learning a generalizable model from a dataset with too many features relative to the number of instances can be difficult.

As a result, working with a smaller feature set may lead to better results. So we want to find some way to algorithmically find the best features. This task is called feature selection and is a crucial step when we aim to get decent results with machine learning algorithms. If we have poor features, our algorithm will return poor results no matter how sophisticated our machine learning algorithm is.

Consider, for example, our very simple Titanic example. We started with just 11 features, but after 1-of-K encoding they grew to `581`.

```
>>> print titanic
<class 'pandas.core.frame.DataFrame'> Int64Index: 1313
entries, 0 to 1312 Columns: 581 entries, row.names to
ticket=L15 1s dtypes: float64(578), int64(2), object(1)
```

This does not pose an important computational problem, but consider what could happen if, as previously demonstrated, we represent each document in a dataset as the number of occurrences of each possible word. Another problem is that decision trees suffer from overfitting. If branching is based on a very small number of instances, the prediction power of the built model will decrease on future data. One solution to this is to adjust model parameters (such as the maximum tree depth or the minimum required number of instances at a leaf node). In this example, however, we

will take a different approach: we will try to limit the features to the most relevant ones.

What do we mean by relevant? This is an important question. 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.

The general approach in feature selection is to get some kind of evaluation function that, when given a potential feature, returns a score of how useful the feature is, and then keeps the features with the highest scores. These methods may have the disadvantage of not detecting correlations between features. Other methods may be more brute force: try all possible subsets of the original feature list, train the algorithm on each combination, and keep the combination that gets the best results.

As an evaluation method, 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.

Scikit-learn provides several methods in the `feature_selection` module. We will use the `SelectPercentile` method that, when given a statistical test, selects a user-specified percentile of features with the highest scoring. The most popular statistical test is the χ^2 (chi-squared) statistic. Let's see how it works for our Titanic example; we will use it to select 20 percent of the most important features:

```
>>> 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. We can now train our decision tree on this data.

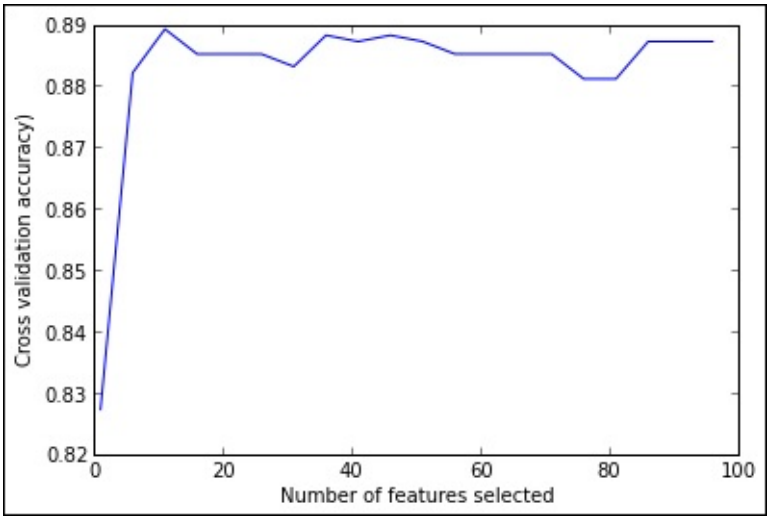
```
>>> dt.fit(X_train_fs, y_train)
>>> X_test_fs = fs.transform(X_test)
>>> y_pred_fs = dt.predict(X_test_fs)
>>> print "Accuracy:
{0:.3f}".format(metrics.accuracy_score(y_test,
    y_pred_fs)),"\n"
Accuracy:0.845
```

We can see that the accuracy on the training set improved half a point after feature selection on the training set.

Is it possible to find the optimal number of features? If by optimal we mean with the best performance on the training set, it is actually possible; we can simply use a brute-force approach and try with different numbers of features while measuring their performance on the training set using 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"
Optimal number of features:11
>>>
>>> # 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)
```

The following figure shows how cross-validation accuracy changes with the number of features:



We can see that accuracy quickly improves when we start adding features, remaining stable after the percentile of features turns about 10. In fact, the best accuracy is

achieved when using 64 of the original 581 features (at the 11 percent percentile). 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:0.848
```

The performance improved slightly, again. Compared with our initial performance, we have finally improved by almost one accuracy point using only 11 percent of the features.

The reader may have noted that while creating our classifier, we used the default parameters, except for the splitting criterion, where we have used `entropy`. Can we improve our model using different parameters? This task is called model selection, and we will address it in detail in the next section using a different learning example. For now, let's just test if the alternative method (`gini`) would result in better performance for our example. To do this, we will again use cross-validation.

```
>>> dt = tree.DecisionTreeClassifier(criterion='entropy')
>>> scores = cross_validation.cross_val_score(dt, X_train_fs,
    y_train, cv=5)
>>> print "Entropy criterion accuracy on
    cv: {0:.3f}".format(scores.mean())
Entropy criterion accuracy on cv: 0.889
>>> dt = tree.DecisionTreeClassifier(criterion='gini')
>>> scores = cross_validation.cross_val_score(dt, X_train_fs,
    y_train, cv=5)
>>> print "Gini criterion accuracy on
    cv: {0:.3f}".format(scores.mean())
Gini criterion accuracy on cv: 0.897
```

The Gini criterion performs better on our training set. 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: 0.848
```

It seems that performance improvement on the training set did not hold for the evaluation set. This is always possible. In fact, performance could have decreased (recall overfitting). Our model is still the best. If we changed our model to use the one with the best performance in the testing set, we can never measure its performance, since the testing dataset could not be considered "unseen data" anymore.

Model selection

In the previous section we worked on ways to preprocess the data and select the most promising features. As we stated, selecting a good set of features is a crucial step to obtain good results. Now we will focus on another important step: selecting the algorithm parameters, known as **hyperparameters** to distinguish them from the parameters that are adjusted within the machine learning algorithm. Many machine learning algorithms include hyperparameters (from now on we will simply call them parameters) that guide certain aspects of the underlying method and have great impact on the results. In this section we will review some methods to help us obtain the best parameter configuration, a process known as model selection.

We will look back at the text-classification problem we addressed in Chapter 2, *Supervised Learning*. In that example, we compounded a TF-IDF vectorizer alongside a multinomial **Naïve Bayes** (**NB**) algorithm to classify a set of newsgroup messages into a discrete number of categories. The `MultinomialNB` algorithm has one important parameter, named `alpha`, that adjusts the smoothing. We initially used the class with its default parameter values (`alpha` = `1.0`) and obtained an accuracy of `0.89`. But when we set `alpha` to `0.01`, we obtained a noticeable accuracy improvement to `0.92`. Clearly, the configuration of the `alpha` parameter has great impact on the performance of the algorithm. How can we be sure `0.01` is the best value? Perhaps if we try other possible values, we could still obtain better results.

Let's start again with our text-classification problem, but for now we will only use a reduced number of instances. We will work only with 3,000 instances. We start by importing our `pylab` environment and loading the data.

```
>>> %pylab inline
>>> 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]
```

After that, we need to import the classes to construct our classifier.

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

Then import the set of stop words and create a pipeline that compounds the TF-IDF vectorizer and the Naïve Bayes algorithms (recall that we had a `stopwords_en.txt`

file with a list of stop words).

```
>>> 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)),
>>>])
```

If we evaluate our algorithm with a three-fold cross-validation, we obtain a mean score of around 0.811.

```
>>> 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))
>>> evaluate_cross_validation(clf, X_train, y_train, 3)
[ 0.814  0.815  0.804]
Mean score: 0.811 (+/-0.004)
```

It looks like we should train the algorithm with a list of different parameter values and keep the parameter value that achieves the best results. Let's implement a helper function to do that. This 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. After that, it will plot the training and testing scores as a function of the parameter values.

```
>>> 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)
>>>
>>>     # 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.

Let's call this function; we will use numpy's `logspace` function to generate a list of alpha values spaced evenly on a log scale.

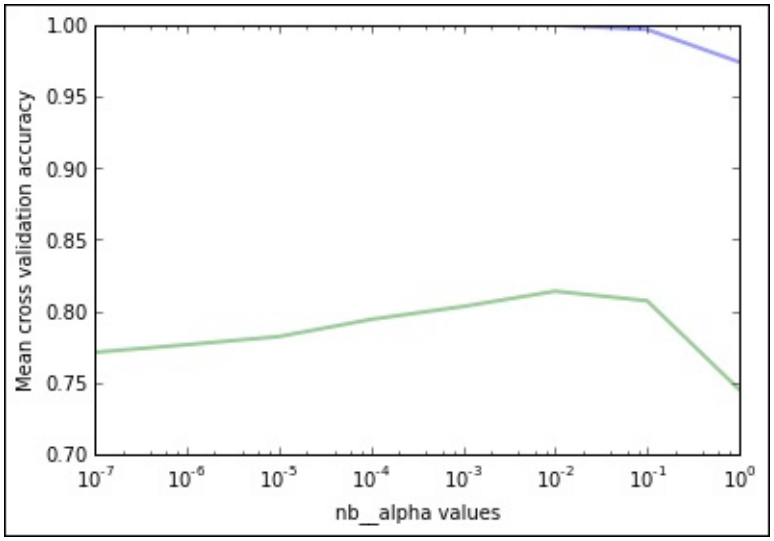
```
>>> alphas = np.logspace(-7, 0, 8)
>>> print alphas
[  1.00000000e-07   1.00000000e-06   1.00000000e-05
1.00000000e-04
1.00000000e-03   1.00000000e-02   1.00000000e-01
1.00000000e+00]
```

We will set the values of the `alpha` parameter of the NB classifier within the pipeline, which corresponds to the parameter name `nb__alpha`. We will use three folds for the

cross-validation.

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

In the following figure, the line at the top corresponds to the training accuracy and the one at the bottom to the testing accuracy:



As expected, the training accuracy is always greater than the testing accuracy. We can see in the graph that 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). Above this range, the classifier shows signs of underfitting (accuracy on the training set is lower than it could be).

It is worth mentioning that at this point a second pass could be performed in the range of 10-2 and 10-1with a finer grid to find an ever better alpha value.

Let's 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. 0.99933333 0.99633333
0.96933333]
testing scores:  [ 0.75 0.75666667 0.76433333 0.77533333
0.78866667 0.811 0.81233333 0.753]
```

The best results are obtained with an `alpha` value of `0.1` (accuracy of 0.812).

We created a very useful function to graph and obtain the best parameter value for a classifier. Let's use it to adjust another classifier that uses a **Support Vector**

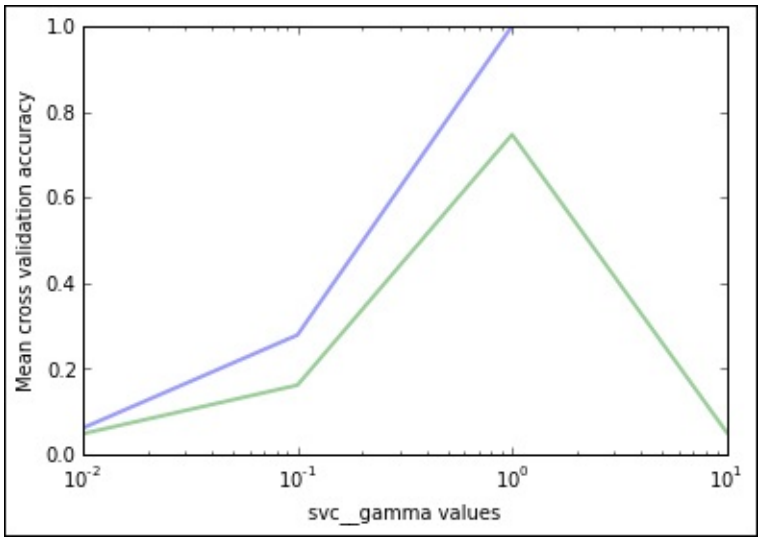
Machines (**SVM**) instead of `MultinomialNB`:

```
>>> 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()),
>>> ])
```

We created a pipeline as before, but now we use the SVC classifier with its default values. Now we will use our `calc_params` function to adjust the `gamma` parameter.

```
>>> gammas = np.logspace(-2, 1, 4)
>>> train_scores, test_scores = calc_params(X_train, y_train,
clf, gammas,'svc__gamma', 3)
```

For gamma values lesser than one we have underfitting and for gamma values greater than one we have overfitting.



So the best result is for a `gamma` value of `1`, where we obtain a training accuracy of 0.999 and a testing accuracy of 0.760.

If you take a closer look at the SVC class constructor parameters, we have other parameters, apart from gamma, that may also affect classifier performance. If we only adjust the gamma value, we implicitly state that the optimal `C` value is `1.0` (the default value that we did not explicitly set). Perhaps we could obtain better results with a new combination of `C` and `gamma` values. This opens a new degree of complexity; we should try all the parameter combinations and keep the better one.

Grid search

To mitigate this problem, we have a very useful class named `GridSearchCV` within the `sklearn.grid_search` module. What we have been doing with our `calc_params` function is a kind of grid search in one dimension. With `GridSearchCV`, we can specify a grid of any number of parameters and parameter values to traverse. It will train the classifier for each combination and obtain a cross-validation accuracy to evaluate each one.

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)
```

Let's execute our grid search and print the best parameter values and scores.

```
>>> %time _ = gs.fit(X_train, y_train)
>>> gs.best_params_, gs.best_score_
CPU times: user 304.39 s, sys: 2.55 s, total: 306.94 s
Wall time: 306.56 s
 ({'svc__C': 10.0, 'svc__gamma': 0.10000000000000001},
0.81166666666666665)
```

With the grid search, we obtained a better combination of `C` and `gamma` parameters, for values `10.0` and `0.10` respectively, with a three-fold cross-validation accuracy of `0.811`, which is much better than the best value we obtained (`0.76`) in the previous experiment by only adjusting `gamma` and keeping the `C` value at `1.0`.

At this point, we could continue performing experiments by trying not only to adjust other parameters of the SVC but also adjusting the parameters on `TfidfVectorizer`, which is also part of the estimator. Note that this additionally increases the

complexity. As you might have noticed, the previous grid search experiment took about five minutes to finish. If we add new parameters to adjust, the time will increase exponentially. As a result, these kinds of methods are very resource/time intensive; this is also the reason why we used only a subset of the total instances.

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, as we have done. In our previous example, we had four different values for `gamma` and three different values for `C`, summing up 12 parameter combinations. Additionally, we also needed to train each combination three times (in a three-fold cross-validation), so we summed up 36 trainings and evaluations. We could try to run these 36 tasks in parallel, since the tasks are independent.

Most modern computers have multiple cores that can be used to run tasks in parallel. We also have a very useful tool within IPython, called **IPython parallel**, that allows us to run independent tasks in parallel, each task in a different core of our machine. Let's do that with our text classifier example.

We will first declare a function that will persist all K folds for the cross-validation in different files. These files will be loaded by a process that will execute the corresponding fold. To do that, we will use the `joblib` library.

```
>>> from sklearn.externals import joblib
>>> from sklearn.cross_validation import ShuffleSplit
>>> import os
>>> def persist_cv_splits(X, y, K=3, name='data',
                          suffix="_cv_%03d.pkl"):
>>>     """Dump K folds to filesystem."""
>>>
>>>     cv_split_filenames = []
>>>
>>>     # create KFold cross validation
>>>     cv = KFold(n_samples, K, shuffle=True, random_state=0)
>>>
>>>     # iterate over the K folds
>>>     for i, (train, test) in enumerate(cv):
>>>         cv_fold = ([X[k] for k in train], y[train], [X[k]
for
                           k in test], y[test])
>>>         cv_split_filename = name + suffix % i
>>>         cv_split_filename =
os.path.abspath(cv_split_filename)
>>>         joblib.dump(cv_fold, cv_split_filename)
>>>         cv_split_filenames.append(cv_split_filename)
>>>
>>>     return cv_split_filenames
>>> cv_filenames = persist_cv_splits(X, y, name='news')
```

The following function loads a particular fold and fits the classifier with the specified

parameter set, returning the testing score. This function will be called by each of the parallel tasks.

```
>>> def compute_evaluation(cv_split_filename, clf, params):
>>>
>>>     # All module imports should be executed in the worker
       namespace
>>>     from sklearn.externals import joblib
>>>
>>>     # load the fold training and testing partitions from
the
       filesystem
>>>     X_train, y_train, X_test, y_test = joblib.load(
>>>         cv_split_filename, mmap_mode='c')
>>>
>>>     clf.set_params(**params)
>>>     clf.fit(X_train, y_train)
>>>     test_score = clf.score(X_test, y_test)
>>>     return test_score
```

Finally, the following function executes the grid search in parallel tasks. For each parameter combination (returned by the `IterGrid` iterator), it iterates over K folds and creates a task to compute the evaluation. It returns the parameter combinations alongside the tasks list.

```
>>> from sklearn.grid_search import IterGrid
>>>
>>> def parallel_grid_search(lb_view, clf, cv_split_filenames,
param_grid):
>>>     all_tasks = []
>>>     all_parameters = list(IterGrid(param_grid))
>>>
>>>     # iterate over parameter combinations
>>>     for i, params in enumerate(all_parameters):
>>>         task_for_params = []
>>>         # iterate over the K folds
>>>         for j, cv_split_filename in
                enumerate(cv_split_filenames):
>>>             t = lb_view.apply(
>>>                 compute_evaluation, cv_split_filename,
clf,
                    params)
>>>             task_for_params.append(t)
>>>
>>>         all_tasks.append(task_for_params)
>>>
>>>     return all_parameters, all_tasks
```

Now we use IPython parallel to get the client and a load balanced view. We must first create a local cluster of N engines (one for each core of your machine) using the

`Cluster` tab in the IPython Notebook. Then we create the client and the view and execute our `parallel_grid_search` function.

```
>>> from sklearn.svm import SVC
>>> from IPython.parallel import Client
>>>
>>> client = Client()
>>> lb_view = client.load_balanced_view()
>>>
>>> all_parameters, all_tasks = parallel_grid_search(
    lb_view, clf, cv_filenames, parameters)
```

IPython parallel will start to run the tasks in parallel. We can use this to monitor the progress of the whole task group.

```
>>> def print_progress(tasks):
>>>     progress = np.mean([task.ready() for task_group in
tasks
                                for task in task_group])
>>>     print "Tasks completed: {0}%".format(100 * progress)
```

After all the tasks are completed, use the following function:

```
>>> print_progress(all_tasks)
Tasks completed: 100.0%
```

We can define a function that computes the mean score of the completed tasks.

```
>>> def find_bests(all_parameters, all_tasks, n_top=5):
>>>     """Compute the mean score of the completed tasks"""
>>>     mean_scores = []
>>>
>>>     for param, task_group in zip(all_parameters,
all_tasks):
>>>         scores = [t.get() for t in task_group if
t.ready()]
>>>         if len(scores) == 0:
>>>             continue
>>>         mean_scores.append((np.mean(scores), param))
>>>
>>>     return sorted(mean_scores, reverse=True)[:n_top]
>>> print find_bests(all_parameters, all_tasks)

[(0.81733333333333336, {'svc__gamma': 0.10000000000000001,
'svc__C': 10.0}), (0.78733333333333333, {'svc__gamma': 1.0,
'svc__C': 10.0}), (0.76000000000000012, {'svc__gamma': 1.0,
'svc__C': 1.0}), (0.30099999999999999, {'svc__gamma': 0.01,
'svc__C': 10.0}), (0.19933333333333333, {'svc__gamma':
0.10000000000000001, 'svc__C': 1.0})]
```

You can observe that we computed the same results as in the previous section, but in half the time (if you used two cores) or in a quarter of the time (if you used four cores).

Summary

In this chapter we reviewed two important methods to improve our results when applying machine learning algorithms: feature selection and model selection. First, we used different techniques to preprocess data, extract features, and select the most promising features. Then we used techniques to automatically calculate the most promising hyperparameters of machine learning algorithms and used methods to parallelize these calculations.

The reader must be aware that this book covered only the main machine learning lines and some of their methods. Keep in mind that there is much more than supervised and unsupervised learning. For example:

- Semi-supervised learning methods are the middle ground between supervised and unsupervised learning. They combine small amounts of annotated data with huge amounts of unlabeled data. Usually, unlabeled data can reveal the underlying distribution of elements and obtain better results in combination with a small, labeled dataset.
- Active learning is a particular case within semi-supervised methods. Again, it is useful when labeled data is scarce or hard to obtain. In active learning, the algorithm actively queries a human expert to answer the label of certain unlabeled instances, and thus learn the concept over a reduced set of labeled instances.
- Reinforcement learning proposes methods where an agent learns from feedback (rewards or reinforcements) after performing actions within an environment. The agent learns to perform a task by trying to maximize the cumulative reward. These methods have been very successful in robotics and video games.
- Sequential classification (very commonly used in **Natural Language Processing** (**NLP**)) assigns a sequence of labels to a sequence of items; for example, the parts of speech of the words in a sentence.

Besides these, there are lots of supervised learning methods with radically different approaches to those we presented; for example, neural networks, maximum entropy models, memory-based models, and rule-based models. Machine learning is a very active research area with a growing literature; there are many books and courses that the reader can use to go deeper into the theory and details.

Scikit-learn has many of these algorithms implemented, and lacks others, but expect its active and enthusiastic contributors to build them soon. We encourage the reader to be part of the community!