Supervised Learning

5

5.1 Introduction

Machine learning involves coding programs that automatically adjust their performance in accordance with their exposure to information in data. This learning is achieved via a parameterized model with tunable parameters that are automatically adjusted according to different performance criteria. Machine learning can be considered a subfield of artificial intelligence (AI) and we can roughly divide the field into the following three major classes.

1. Supervised learning: Algorithms which learn from a training set of labeled examples (exemplars) to generalize to the set of all possible inputs. Examples of techniques in supervised learning: logistic regression, support vector machines, decision trees, random forest, etc.
2. Unsupervised learning: Algorithms that learn from a training set of unlabeled examples. Used to explore data according to some statistical, geometric or similarity criterion. Examples of unsupervised learning include k-means clustering and kernel density estimation. We will see more on this kind of techniques in Chap. 7.
3. Reinforcement learning: Algorithms that learn via reinforcement from criticism that provides information on the quality of a solution, but not on how to improve it. Improved solutions are achieved by iteratively exploring the solution space.

This chapter focuses on a particular class of supervised machine learning: *classification*. As a data scientist, the first step you apply given a certain problem is to identify the question to be answered. According to the type of answer we are seeking, we are directly aiming for a certain set of techniques.

L. Igual and S. Seguí, *Introduction to Data Science*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-50017-1_5

- If our question is answered by YES/NO, we are facing a classification problem. Classifiers are also the tools to use if our question admits only a discrete set of answers, i.e., we want to select from a finite number of choices.

  – Given the results of a clinical test, e.g., does this patient suffer from diabetes?
  – Given a magnetic resonance image, is it a tumor shown in the image?
  – Given the past activity associated with a credit card, is the current operation fraudulent?

- If our question is a prediction of a real-valued quantity, we are faced with a *regression* problem. We will go into details of regression in Chap. 6.

  – Given the description of an apartment, what is the expected market value of the flat? What will the value be if the apartment has an elevator?
  – Given the past records of user activity on Apps, how long will a certain client be connected to our App?
  – Given my skills and marks in computer science and maths, what mark will I achieve in a data science course?

Observe that some problems can be solved using both regression and classification. As we will see later, many classification algorithms are thresholded regressors. There is a certain skill involved in designing the correct question and this dramatically affects the solution we obtain.

5.2 The Problem

In this chapter we use data from the Lending Club[1] to develop our understanding of machine learning concepts. The Lending Club is a peer-to-peer lending company. It offers loans which are funded by other people. In this sense, the Lending Club acts as a hub connecting borrowers with investors. The client applies for a loan of a certain amount, and the company assesses the risk of the operation. If the application is accepted, it may or may not be fully covered. We will focus on the prediction of whether the loan will be fully funded, based on the scoring of and information related to the application.

We will use the partial dataset of period 2007–2011. Framing the problem a little bit more, based on the information supplied by the customer asking for a loan, we want to predict whether it will be granted up to a certain threshold thr. The attributes we use in this problem are related to some of the details of the loan application, such as amount of the loan applied for the borrower, monthly payment to be made by the borrower if the loan is accepted, the borrower's annual income, the number of

[1] https://www.lendingclub.com/info/download-data.action.

incidences of delinquency in the borrower's credit file, and interest rate of the loan, among others.

In this case we would like to predict unsuccessful accepted loans. A loan application is unsuccessful if the funded amount (`funded_amnt`) or the amount funded by investors (`funded_amnt_inv`) falls far short of the requested loan amount (`loan_amnt`). That is,

$$\frac{loan - funded}{loan} \geq 0.95.$$

5.3 First Steps

Note that in this problem we are predicting a binary value: either the loan is fully funded or not. Classification is the natural choice of machine learning tools for prediction with discrete known outcomes. According to the cardinality of the target set, one usually distinguishes between *binary* classifiers when the target output only takes two values, i.e., the classifier answers questions with a yes or a no; or *multiclass* classifiers, for a larger number of classes. This issue is important in that not all methods can naturally handle the multiclass setting.[2]

In a formal way, classification is regarded as the problem of finding a function $h(\mathbf{x}) : \mathbb{R}^d \to \mathbb{K}$ that maps an input space in $\mathbb{R}^d$ onto a discrete set of k target outputs or classes $\mathbb{K} = \{1, \ldots, k\}$. In this setting, the features are arranged as a vector $\mathbf{x}$ of d real-valued numbers.[3]

We can encode both target states in a numerical variable, e.g., a successful loan target can take value $+1$; and it is -1, otherwise.

Let us check the dataset,[4]

In [1]:
```
import pickle
ofname = open('./files/ch05/dataset_small.pkl','rb')
# x stores input data and y target values
(x,y) = pickle.load(ofname)
```

[2]Several well-known techniques such as support vector machines or adaptive boosting (adaboost) are originally defined in the binary case. Any binary classifier can be extended to the multiclass case in two different ways. We may either change the formulation of the learning/optimization process. This requires the derivation of a new learning algorithm capable of handling the new modeling. Alternatively, we may adopt ensemble techniques. The idea behind this latter approach is that we may divide the multiclass problem into several binary problems; solve them; and then aggregate the results. If the reader is interested in these techniques, it is a good idea to look for: one-versus-all, one-versus-one, or error correcting output codes methods.

[3]Many problems are described using categorical data. In these cases either we need classifiers that are capable of coping with this kind of data or we need to change the representation of those variables into numerical values.

[4]The notebook companion shows the preprocessing steps, from reading the dataset, cleaning and imputing data, up to saving a subsampled clean version of the original dataset.

A problem in Scikit-learn is modeled as follows:

- Input data is structured in Numpy arrays. The size of the array is expected to be [n_samples, n_features]:
  - n_samples: The number of samples (n). Each sample is an item to process (e.g., classify). A sample can be a document, a picture, an audio file, a video, an astronomical object, a row in a database or CSV file, or whatever you can describe with a fixed set of quantitative traits.
  - n_features: The number of features (d) or distinct traits that can be used to describe each item in a quantitative manner. Features are generally real-valued, but may be Boolean, discrete-valued or even categorical.

$$\text{feature matrix}: \quad \mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1d} \\ x_{21} & x_{22} & \cdots & x_{2d} \\ x_{31} & x_{32} & \cdots & x_{3d} \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nd} \end{bmatrix}$$

$$\text{label vector}: \quad \mathbf{y}^T = [y_1, y_2, y_3, \cdots y_n]$$

The number of features must be fixed in advance. However, it can be very great (e.g., millions of features).

In [2]:
```
dims = x.shape[1]
N = x.shape[0]
print 'dims: ' + str(dims) + ', samples: ' + str(N)
```

Out[2]:
```
dims: 15, samples: 4140
```

Considering data arranged as in the previous matrices we refer to:

- the columns as features, attributes, dimensions, regressors, covariates, predictors, or independent variables;
- the rows as instances, examples, or samples;
- the target as the label, outcome, response, or dependent variable.

All objects in Scikit-learn share a uniform and limited API consisting of three complementary interfaces:

- an estimator interface for building and fitting models (`fit()`);
- a predictor interface for making predictions (`predict()`);
- a transformer interface for converting data (`transform()`).

Let us apply a classifier using Python's Scikit-learn libraries,

In [3]:
```
from sklearn import neighbors
from sklearn import datasets
# Create an instance of K-nearest neighbor classifier
knn = neighbors.KNeighborsClassifier(n_neighbors = 11)
# Train the classifier
knn.fit(x, y)
# Compute the prediction according to the model
yhat = knn.predict(x)
# Check the result on the last example
print 'Predicted value: ' + str(yhat[-1]),
      ', real target: ' + str(y[-1])
```

Out[3]:
```
Predicted value: -1.0 , real target: -1.0
```

The basic measure of performance of a classifier is its *accuracy*. This is defined as the number of correctly predicted examples divided by the total amount of examples. Accuracy is related to the error as follows: $acc = 1 - err$.

$$acc = \frac{\text{Number of correct predictions}}{n}$$

Each estimator has a `score()` method that invokes the default scoring metric. In the case of k-nearest neighbors, this is the classification accuracy.

In [4]:
```
knn.score(x,y)
```

Out[4]:
```
0.83164251207729467
```

It looks like a really good result. But how good is it? Let us first understand a little bit more about the problem by checking the distribution of the labels.

Let us load the dataset and check the distribution of labels:

In [5]:
```
plt.pie(np.c_[np.sum(np.where(y == 1, 1, 0)),
        np.sum(np.where(y == -1, 1, 0))][0],
        labels = ['Not fully funded','Full amount'],
        colors = ['r', 'g'],shadow = False,
        autopct = '%.2f' )
plt.gcf().set_size_inches((7, 7))
```

with the result observed in Fig. 5.1.

Note that there are far more positive labels than negative ones. In this case, the dataset is referred to as *unbalanced*.[5] This has important consequences for a classifier as we will see later on. In particular, a very simple rule such as always predict the

[5]The term unbalanced describes the condition of data where the ratio between positives and negatives is a small value. In these scenarios, always predicting the majority class usually yields accurate performance, though it is not very informative. This kind of problems is very common when we want to model unusual events such as rare diseases, the occurrence of a failure in machinery, fraudulent credit card operations, etc. In these scenarios, gathering data from usual events is very easy but collecting data from unusual events is difficult and results in a comparatively small dataset.

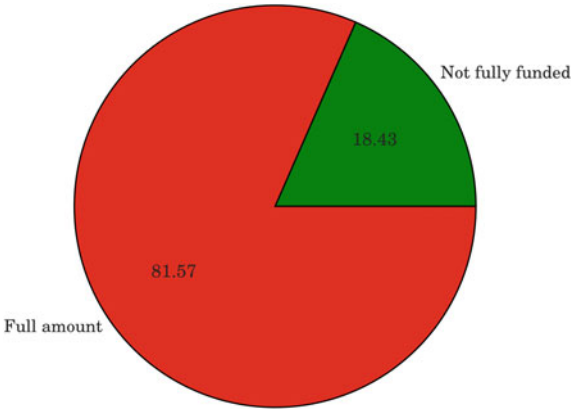

Fig. 5.1 Pie chart showing the distribution of labels in the dataset

majority class, will give us good performance. In our problem, always predicting that the loan will be fully funded correctly predicts 81.57% of the samples. Observe that this value is very close to that obtained using the classifier.

Although accuracy is the most normal metric for evaluating classifiers, there are cases when the business value of correctly predicting elements from one class is different from the value for the prediction of elements of another class. In those cases, accuracy is not a good performance metric and more detailed analysis is needed. The *confusion matrix* enables us to define different metrics considering such scenarios. The confusion matrix considers the concepts of the classifier outcome and the actual ground truth or gold standard. In a binary problem, there are four possible cases:

- *True positives (TP)*: When the classifier predicts a sample as positive and it really is positive.
- *False positives (FP)*: When the classifier predicts a sample as positive but in fact it is negative.
- *True negatives (TN)*: When the classifier predicts a sample as negative and it really is negative.
- *False negatives (FN)*: When the classifier predicts a sample as negative but in fact it is positive.

We can summarize this information in a matrix, namely the confusion matrix, as follows:

		Gold Standard: Positive	Gold Standard: Negative	
Prediction	Positive	TP	FP	→ Precision
	Negative	FN	TN	→ Negative Predictive Value
		↓	↓	
		Sensitivity (Recall)	Specificity	

The combination of these elements allows us to define several performance metrics:

- *Accuracy:*

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

- Column-wise we find these two partial performance metrics:
  - *Sensitivity or Recall:*

$$\text{sensitivity} = \frac{\text{TP}}{\text{Real Positives}} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

  - *Specificity:*

$$\text{specificity} = \frac{\text{TN}}{\text{Real Negatives}} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

- Row-wise we find these two partial performance metrics:
  - *Precision or Positive Predictive Value:*

$$\text{precision} = \frac{\text{TP}}{\text{Predicted Positives}} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

  - *Negative predictive value:*

$$\text{NPV} = \frac{\text{TN}}{\text{Predicted Negative}} = \frac{\text{TN}}{\text{TN} + \text{FN}}$$

These partial performance metrics allow us to answer questions concerning how often a classifier predicts a particular class, e.g., what is the rate of predictions for not fully funded loans that have actually not been fully funded? This question is answered by recall. In contrast, we could ask: Of all the fully funded loans predicted by the classifier, how many have been fully funded? This is answered by the precision metric.

Let us compute these metrics for our problem.

In [6]:
```
yhat = knn.predict(x)
TP = np.sum(np.logical_and(yhat == -1, y == -1))
TN = np.sum(np.logical_and(yhat == 1, y == 1))
FP = np.sum(np.logical_and(yhat == -1, y == 1))
FN = np.sum(np.logical_and(yhat == 1, y == -1))
print 'TP: '+ str(TP), ', FP: '+ str(FP)
print 'FN: '+ str(FN), ', TN: '+ str(TN)
```

Out[6]:
```
TP: 3370 , FP: 690
 FN: 7 ,  TN: 73
```

Scikit-learn provides us with the confusion matrix,

In [7]:
```
from sklearn import metrics
metrics.confusion_matrix(yhat, y)
# sklearn uses a transposed convention for the confusion
# matrix thus I change targets and predictions
```

Out[7]:
```
3370, 690
 7,   73
```

Let us check the following example. Let us select a nearest neighbor classifier with the number of neighbors equal to one instead of eleven, as we did before, and check the training error.

In [8]:
```
# Train a classifier using .fit()
knn = neighbors.KNeighborsClassifier(n_neighbors = 1)
knn.fit(x, y)
yhat = knn.predict(x)

print "classification accuracy:" +
      str(metrics.accuracy_score(yhat, y))
print "confusion matrix: \n" +
      str(metrics.confusion_matrix(yhat, y))
```

Out[8]:
```
classification accuracy: 1.0 confusion matrix:
3377  0
 0   763
```

The performance measure is perfect! 100% accuracy and a diagonal confusion matrix! This looks good. However, up to this point we have checked the classifier performance on the same data it has been trained with. During exploitation, in real applications, we will use the classifier on data not previously seen. Let us simulate this effect by splitting the data into two sets: one will be used for learning (*training set*) and the other for testing the accuracy (*test set*).

In [9]:
```
# Simulate a real case: Randomize and split data into
# two subsets PRC*100\% for training and the rest
# (1-PRC)*100\% for testing
perm = np.random.permutation(y.size)
PRC = 0.7
split_point = int(np.ceil(y.shape[0]*PRC))

X_train = x[perm[:split_point].ravel(),:]
y_train = y[perm[:split_point].ravel()]

X_test = x[perm[split_point:].ravel(),:]
y_test = y[perm[split_point:].ravel()]
```

If we check the shapes of the training and test sets we obtain,

Out[9]:
```
Training shape: (2898, 15), training targets shape: (2898,)
Testing shape: (1242, 15), testing targets shape: (1242,)
```

With this new partition, let us train the model

In [10]:
```
#Train a classifier on training data
knn = neighbors.KNeighborsClassifier(n_neighbors = 1)
knn.fit(X_train, y_train)
yhat = knn.predict(X_train)

print "\n TRAINING STATS:"
print "classification accuracy:" +
      str(metrics.accuracy_score(yhat, y_train))
print "confusion matrix: \n" +
      str(metrics.confusion_matrix(y_train, yhat))
```

Out[10]:
```
TRAINING STATS:
classification accuracy: 1.0
confusion matrix:
2355  0
  0   543
```

As expected from the former experiment, we achieve a perfect score. Now let us see what happens in the simulation with previously unseen data.

In [11]:
```
#Check on the test set
yhat = knn.predict(X_test)
print "TESTING STATS:"
print "classification accuracy:",
      metrics.accuracy_score(yhat, y_test)
print "confusion matrix: \n" +
      str(metrics.confusion_matrix(yhat, y_test))
```

Out[11]:
```
TESTING STATS:
classification accuracy: 0.754428341385
confusion matrix:
865 148
157  72
```

Observe that each time we run the process of randomly splitting the dataset and train a classifier we obtain a different performance. A good simulation for approximating the test error is to run this process many times and average the performances. Let us do this![6]

In [12]:
```
# Spitting done by using the tools provided by sklearn:
from sklearn.cross_validation import train_test_split

PRC = 0.3
acc = np.zeros((10,))
for i in xrange(10):
    X_train, X_test, y_train, y_test =
        train_test_split(x, y, test_size = PRC)
    knn = neighbors.KNeighborsClassifier(n_neighbors = 1)
    knn.fit(X_train, y_train)
    yhat = knn.predict(X_test)
    acc[i] = metrics.accuracy_score(yhat, y_test)
acc.shape = (1, 10)
print "Mean expected error:" + str(np.mean(acc[0]))
```

Out[12]:
```
Mean expected error: 0.754669887279
```

As we can see, the resulting error is below 81%, which was the result of the most naive decision process. What is wrong with this result?

Let us introduce the nomenclature for the quantities we have just computed and define the following terms.

- *In-sample error* E_{in}: The in-sample error or training error is the error measured over all the observed data samples in the training set, i.e.,

$$E_{\text{in}} = \frac{1}{N} \sum_{i=1}^{N} e(x_i, y_i)$$

- *Out-of-sample error* E_{out}: The out-of-sample error or generalization error measures the expected error on unseen data. We can approximate/simulate this quantity by holding back some training data for testing purposes.

$$E_{\text{out}} = \mathbb{E}_{x,y}(e(x, y))$$

Note that the definition of the instantaneous error $e(x_i, y_i)$ is still missing. For example, in classification we could use the indicator function to account for a correctly classified sample as follows:

$$e(x_i, y_i) = I[h(x_i) = y_i] = \begin{cases} 1, \text{ if } \ h(x_i) = y_i \\ 0 \text{ otherwise.} \end{cases}$$

[6] *sklearn* allows us to easily automate the train/test splitting using the function `train_test_split(...)`.

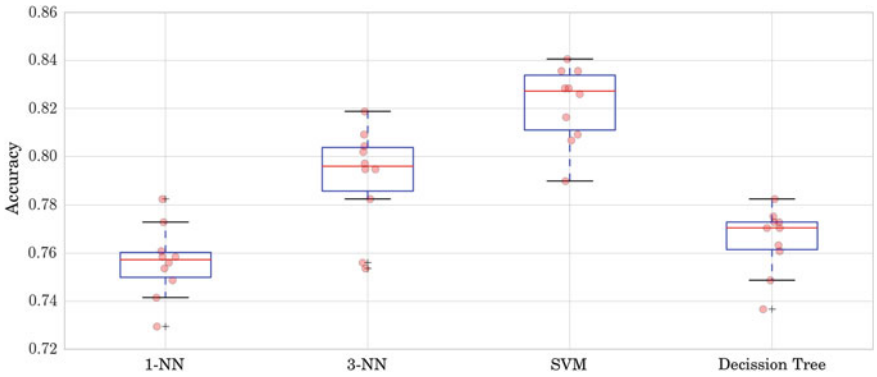

Fig. 5.2 Comparison of the methods using the accuracy metric

Observe that:

$$E_{\text{out}} \geq E_{\text{in}}$$

Using the expected error on the test set, we can select the best classifier for our application. This is called model selection. In this example we cover the most simplistic setting. Suppose we have a set of different classifiers and want to select the "best" one. We may use the one that yields the lowest error rate.

In [13]:
```
from sklearn import tree
from sklearn import svm
PRC = 0.1
acc_r = np.zeros((10, 4))
for i in xrange(10):
    X_train, X_test, y_train, y_test =
        train_test_split(x, y, test_size = PRC)
    nn1 = neighbors.KNeighborsClassifier(n_neighbors = 1)
    nn3 = neighbors.KNeighborsClassifier(n_neighbors = 3)
    svc = svm.SVC()
    dt = tree.DecisionTreeClassifier()

    nn1.fit(X_train, y_train)
    nn3.fit(X_train, y_train)
    svc.fit(X_train, y_train)
    dt.fit(X_train, y_train)

    yhat_nn1 = nn1.predict(X_test)
    yhat_nn3 = nn3.predict(X_test)
    yhat_svc = svc.predict(X_test)
    yhat_dt = dt.predict(X_test)

    acc_r[i][0] = metrics.accuracy_score(yhat_nn1, y_test)
    acc_r[i][1] = metrics.accuracy_score(yhat_nn3, y_test)
    acc_r[i][2] = metrics.accuracy_score(yhat_svc, y_test)
    acc_r[i][3] = metrics.accuracy_score(yhat_dt, y_test)
```

Figure 5.2 shows the results of applying the code.

This process is one particular form of a general model selection technique called *cross-validation*. There are other kinds of cross-validation, such as *leave-one-out* or *K-fold cross-validation*.

- In leave-one-out, given N samples, the model is trained with $N - 1$ samples and tested with the remaining one. This is repeated N times, once per training sample and the result is averaged.
- In K-fold cross-validation, the training set is divided into K nonoverlapping splits. K-1 splits are used for training and the remaining one used to assess the mean. This process is repeated K times leaving one split out each time. The results are then averaged.

5.4 What Is Learning?

Let us recall the two basic values defined in the last section. We talk of *training error* or in-sample error, E_{in}, which refers to the error measured over all the observed data samples in the training set. We also talk of *test error* or *generalization error*, E_{out}, as the error expected on unseen data.

We can empirically estimate the generalization error by means of cross-validation techniques and observe that:

$$E_{\text{out}} \geq E_{\text{in}}.$$

The goal of learning is to minimize the generalization error; but how can we guarantee this minimization using only training data?

From the above inequality it is easy to derive a couple of very intuitive ideas.

- Because E_{out} is greater than or equal to E_{in}, it is desirable to have
$$E_{\text{in}} \to 0.$$
- Additionally, we also want the training error behavior to track the generalization error so that if one minimizes the in-sample error the out-of-sample error follows, i.e.,
$$E_{\text{out}} \approx E_{\text{in}}.$$

We can rewrite the second condition as

$$E_{\text{in}} \leq E_{\text{out}} \leq E_{\text{in}} + \Omega,$$

with $\Omega \to 0$.

We would like to characterize Ω in terms of our problem parameters, i.e., the number of samples (N), dimensionality of the problem (d), etc.

Statistical analysis offers an interesting characterization of this quantity[7]

[7]The reader should note that there are several bounds in machine learning to characterize the generalization error. Most of them come from variations of Hoeffding's inequality.

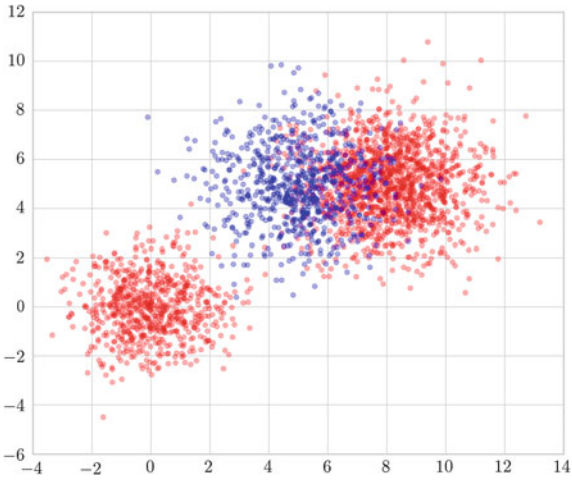

Fig. 5.3 Toy problem data

$$E_{\text{out}} \leq E_{\text{in}}(C) + \mathcal{O}\left(\sqrt{\frac{\log C}{N}}\right),$$

where C is a measure of the complexity of the model class we are using. Technically, we may also refer to this model class as the hypothesis space.

5.5 Learning Curves

Let us simulate the effect of the number of examples on the training and test errors for a given complexity. This curve is called the *learning curve*. We will focus for a moment in a more simple case. Consider the toy problem in Fig. 5.3.

Let us take a classifier and vary the number of examples we feed it for training purposes, then check the behavior of the training and test accuracies as the number of examples grows. In this particular case, we will be using a decision tree with fixed maximum depth.

Observing the plot in Fig. 5.4, we can see that:

- As the number of training samples increases, both errors tend to the same value.
- When we have few training data, the training error is very small but the test error is very large.

Now check the learning curve when the degree of complexity is greater in Fig. 5.5. We simulate this effect by increasing the maximum depth of the tree.

And if we put both curves together, we have the results shown in Fig. 5.6.

Although both show similar behavior, we can note several differences:

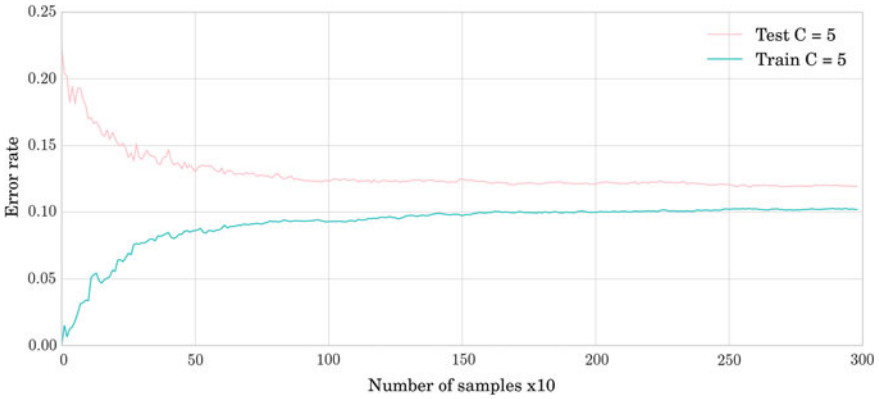

Fig. 5.4 Learning curves (training and test errors) for a model with a high degree of complexity

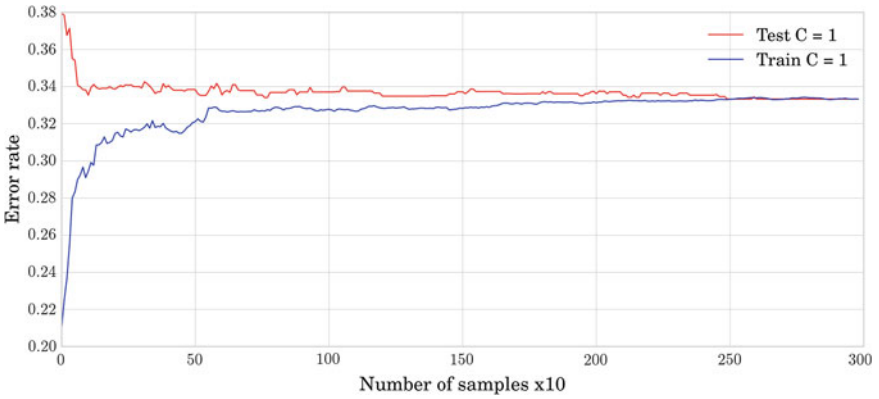

Fig. 5.5 Learning curves (training and test errors) for a model with a low degree of complexity

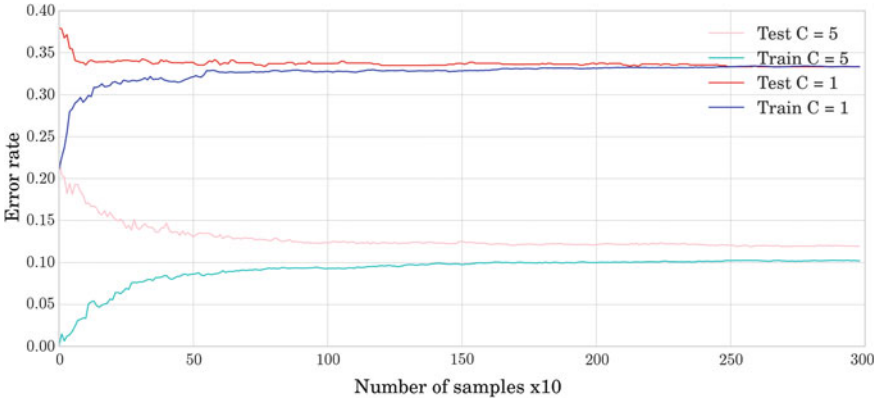

Fig. 5.6 Learning curves (training and test errors) for models with a low and a high degree of complexity

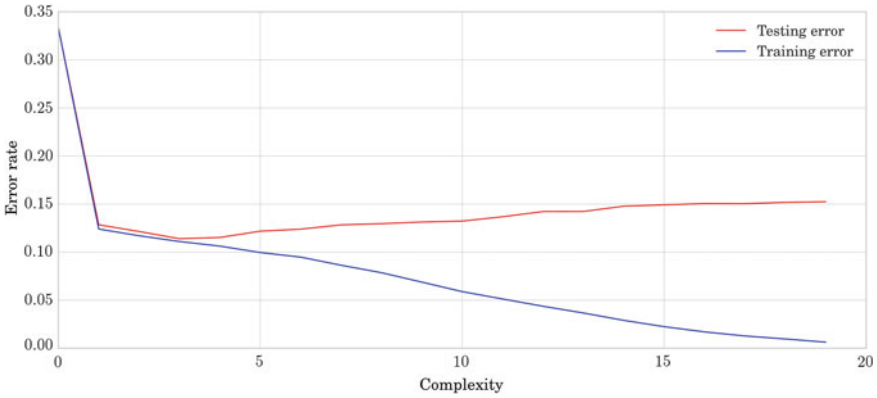

Fig. 5.7 Learning curves (training and test errors) for a fixed number of data samples, as the complexity of the decision tree increases

- With a low degree of complexity, the training and test errors converge to the bias sooner/with fewer data.
- Moreover, with a low degree of complexity, the error of convergence is larger than with increased complexity.

The value both errors converge towards is also called the *bias*; and the difference between this value and the test error is called the *variance*. The *bias/variance* decomposition of the learning curve is an alternative approach to the training and generalization view.

Let us now plot the learning behavior for a fixed number of examples with respect to the complexity of the model. We may use the same data but now we will change the maximum depth of the decision tree, which governs the complexity of the model.

Observe in Fig. 5.7 that as the complexity increases the training error is reduced; but above a certain level of complexity, the test error also increases. This effect is called *overfitting*. We may enact several cures for overfitting:

- Observe that models are usually parameterized by some hyperparameters. Selecting the complexity is usually governed by some such parameters. Thus, we are faced with a model selection problem. A good heuristic for selecting the model is to choose the value of the hyperparameters that yields the smallest estimated test error. Remember that this can be done using cross-validation.
- We may also change the formulation of the objective function to penalize complex models. This is called *regularization*. Regularization accounts for estimating the value of Ω in our out-of-sample error inequality. In other words, it models the complexity of the technique. This usually becomes implicit in the algorithm but has huge consequences in real applications. The most common regularization strategies are as follows:

  - L2 weight regularization: Adding an L2 penalization term to the weights of a weight-controlled model implies looking for solutions with small weight values. Intuitively, adding an L2 penalization term can be seen as a surrogate for the notion of smoothness. In this sense, a low complexity model means a very smooth model.
  - L1 weight regularization: Adding an L1 regularization term forces sparsity in the weights of the model. In this sense, a low complexity model means a model with few components or few active terms.

  These terms are added to the objective function. They trade off with the error function in the objective and are governed by a hyperparameter. Thus, we still have to select this parameter by means of model selection.
- We can use "ensemble techniques". A third cure for overfitting is to use ensemble techniques. The best known are *bagging* and *boosting*.

5.6 Training, Validation and Test

Going back to our problem, we have to select a model and control its complexity according to the number of training data. In order to do this, we can start by using a model selection technique. We have seen model selection before when we wanted to compare the performance of different classifiers. In that case, our best bet was to select the classifier with the smallest E_{out}. Analogous to model selection, we may think of selecting the best hyperparameters as choosing the classifier with parameters that performs the best. Thus, we may select a set of hyperparameter values and use cross-validation to select the best configuration.

The process of selecting the best hyperparameters is called *validation*. This introduces a new set into our simulation scheme; we now need to divide the data we have into three sets: training, validation, and test sets. As we have seen, the process of assessing the performance of the classifier by estimating the generalization error is called testing. And the process of selecting a model using the estimation of the generalization error is called validation. There is a subtle but critical difference between the two and we have to be aware of it when dealing with our problem.

- Test data is used exclusively for assessing performance at the end of the process and will never be used in the learning process.[8]
- Validation data is used explicitly to select the parameters/models with the best performance according to an estimation of the generalization error. This is a form of learning.
- Training data are used to learn the instance of the model from a model class.

[8]This set cannot be used to select a classifier, model or hyperparameter; nor can it be used in any decision process.

In practice, we are just given training data, and in the most general case we explicitly have to tune some hyperparameter. Thus, how do we select the different splits?

How we do this will depend on the questions regarding the method that we want to answer:

- Let us say that our customer asks us to deliver a classifier for a given problem. If we just want to provide the best model, then we may use cross-validation on our training dataset and select the model with the best performance. In this scenario, when we return the trained classifier to our customer, we know that it is the one that achieves the best performance. But if the customer asks about the expected performance, we cannot say anything.
  A practical issue: once we have selected the model, we use the complete training set to train the final model.
- If we want to know about the performance of our model, we have to use unseen data. Thus, we may proceed in the following way:

1. Split the original dataset into training and test data. For example, use 30% of the original dataset for testing purposes. This data is held back and will only be used to assess the performance of the method.
2. Use the remaining training data to select the hyperparameters by means of cross-validation.
3. Train the model with the selected parameter and assess the performance using the test dataset.

A practical issue: Observe that by splitting the data into three sets, the classifier is trained with a smaller fraction of the data.

- If we want to make a good comparison of classifiers but we do not care about the best parameters, we may use *nested cross-validation*. Nested cross-validation runs two cross-validation processes. An external cross-validation is used to assess the performance of the classifier and in each loop of the external cross-validation another cross-validation is run with the remaining training set to select the best parameters.

If we want to select the best complexity of a decision tree, we can use tenfold cross-validation checking for different complexity parameters. If we change the maximum depth of the method, we obtain the results in Fig. 5.8.

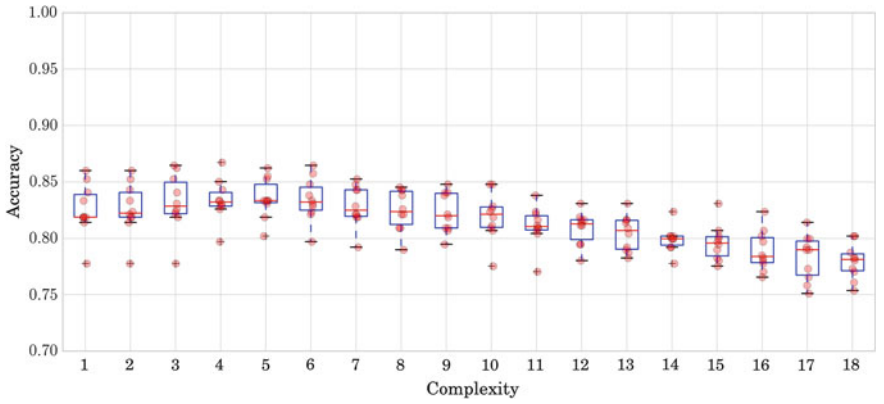

Fig. 5.8 Box plot showing accuracy for different complexities of the decision tree

In [14]:

```
# Create a 10-fold  cross-validation set
kf = cross_validation.KFold(n = y.shape[0],
                            n_folds = 10,
                            shuffle = True,
                            random_state = 0)

# Search for the parameter among the following:
C = np.arange(2, 20,)

acc = np.zeros((10, 18))
i = 0
for train_index, val_index in kf:
    X_train, X_val = X[train_index], X[val_index]
    y_train, y_val = y[train_index], y[val_index]
    j = 0
    for c in C:
        dt = tree.DecisionTreeClassifier(
            min_samples_leaf = 1,
            max_depth = c)
        dt.fit(X_train, y_train)
        yhat = dt.predict(X_val)
        acc[i][j] = metrics.accuracy_score(yhat, y_val)
        j = j + 1
    i = i + 1
```

Checking Fig. 5.8, we can see that the best average accuracy is obtained by the fifth model, a maximum depth of 6. Although we can report that the best accuracy is estimated to be found with a complexity value of 6, we cannot say anything about the value it will achieve. In order to have an estimation of that value, we need to run the model on a new set of data that are completely unseen, both in training and in model selection (the model selection value is positively biased). Let us put everything together. We will be considering a simple train_test split for testing purposes and then run cross-validation for model selection.

In [15]:
```
# Train_test split
X_train, X_test, y_train, y_test = cross_validation
    .train_test_split(X, y, test_size = 0.20)

# Create a 10-fold  cross-validation set
kf = cross_validation.KFold(n = y_train.shape[0],
                            n_folds = 10,
                            shuffle = True,
                            random_state = 0)

# Search the parameter among the following
C = np.arange(2, 20,)
acc = np.zeros((10, 18))
i = 0
for train_index, val_index in kf:
    X_t, X_val = X_train[train_index], X_train[val_index]
    y_t, y_val = y_train[train_index], y_train[val_index]
    j = 0
    for c in C:
        dt = tree.DecisionTreeClassifier(
            min_samples_leaf = 1,
            max_depth = c)
        dt.fit(X_t, y_t)
        yhat = dt.predict(X_val)
        acc[i][j] = metrics.accuracy_score(yhat, y_val)
        j = j + 1
    i = i + 1
print 'Mean accuracy: ' + str(np.mean(acc, axis = 0))
print 'Selected model index: ' +
      str(np.argmax(np.mean(acc, axis = 0)))
```

Out[15]:
```
Mean accuracy: [0.8254832 0.83031158 0.83091854 0.83423816
0.83363939 0.83303516 0.82759983 0.82337022 0.82034725
0.81642795 0.80947567 0.79951316 0.80162614 0.79226695
0.79589324 0.785928 0.78049267 0.78320988]
Selected model index: 3
```

If we run the output of this code, we observe that the best accuracy is provided by the fourth model. In this example it is a model with complexity 5.[9] The selected model achieves a success rate of 0.83423816 in validation. We then train the model with the complete training set and verify its test accuracy.

[9] This reduction in the complexity of the best model should not surprise us. Remember that complexity and the number of examples are intimately related for the learning to succeed. By using a test set we perform model selection with a smaller dataset than in the former case.

In [16]:
```
# Train the model with the complete training set with the
    selected complexity
dt = tree.DecisionTreeClassifier(
    min_samples_leaf = 1,
    max_depth = C[np.argmax(np.mean(acc, axis = 0))])
dt.fit(X_train,y_train)

# Test the model with the test set
yhat = dt.predict(X_test)
print 'Test accuracy: ' +
      str(metrics.accuracy_score(yhat, y_test))
```

Out[16]:
```
Test accuracy: 0.826086956522
```

As expected, the value is slightly reduced; it achieves 0.82608. Finally, the model is trained with the complete dataset. This will be the model used in exploitation and we expect to at least achieve an accuracy rate of 0.82608.

In [17]:
```
# Train the final model
dt = tree.DecisionTreeClassifier(min_samples_leaf = 1,
    max_depth = C[np.argmax(np.mean(acc, axis = 0))])
dt.fit(X, y)
```

5.7 Two Learning Models

Let us return to our problem and check the performance of different models. There are many learning models in the machine learning literature. However, in this short introduction we focus on two of the most important and pragmatically effective approaches[10]: support vector machines (SVM) and random forests (RF).

5.7.1 Generalities Concerning Learning Models

Before going into some of the details of the models selected, let us check the components of any learning algorithm. In order to be able to learn, an algorithm has to define at least three components:

- *The model class/hypothesis space* defines the family of mathematical models that will be used. The target decision boundary will be approximated from one element of this space. For example, we can consider the class of linear models. In this case our decision boundary will be a line if the problem is defined in $\mathbf{R}^2$ and the model class is the space of all possible lines in $\mathbf{R}^2$.

[10] These techniques have been shown to be two of the most powerful families for classification [1].

Model classes define the geometric properties of the decision function. There are different taxonomies but the best known are the families of *linear* and *nonlinear* models. These families usually depend on some parameters; and the solution to a learning problem is the selection of a particular set of parameters, i.e., the selection of an instance of a model from the model class space. The model class space is also called the *hypothesis space*.
The selection of the best model will depend on our problem and what we want to obtain from the problem. The primary goal in learning is usually to achieve the minimum error/maximum performance; but according to what else we want from the algorithm, we can come up with different algorithms. Other common desirable properties are interpretability, behavior when faced with missing data, fast training, etc.

- *The problem model* formalizes and encodes the desired properties of the solution. In many cases, this formalization takes the form of an optimization problem. In its most basic instantiation, the problem model can be the *minimization of an error function*. The error function measures the difference between our model and the target. Informally speaking, in a classification problem it measures how "irritated" we are when our model misses the right label for a training sample. For example, in classification, the ideal error function is the *0–1 loss*. This function takes value 1 when we incorrectly classify a training sample and zero otherwise. In this case, we can interpret it by saying that we are only irritated by "one unit of irritation" when one sample is misclassified.
  The problem model can also be used to impose other constraints on our solution,[11] such as finding a smooth approximation, a model with a low degree of small complexity, a sparse solution, etc.
- *The learning algorithm* is an optimization/search method or algorithm that, given a model class, fits it to the training data according to the error function. According to the nature of our problem there are many different algorithms. In general, we are talking about finding the minimum error approximation or maximum probable model. In those cases, if the problem is convex/quasi-convex we will typically use first- or second-order methods (i.e., gradient descent, coordinate descent, Newton's method, interior point methods, etc.). Other searching techniques such as genetic algorithms or Monte Carlo techniques can be used if we do not have access to the derivatives of the objective function.

5.7.2 Support Vector Machines

SVM is a learning technique initially designed to fit a linear boundary between the samples of a binary problem, ensuring the maximum robustness in terms of tolerance to isotropic uncertainty. This effect is observed in Fig. 5.9. Note that the boundary displayed has the largest distance to the closest point of both classes. Any other

[11] Remember the regularization cure for overfitting.

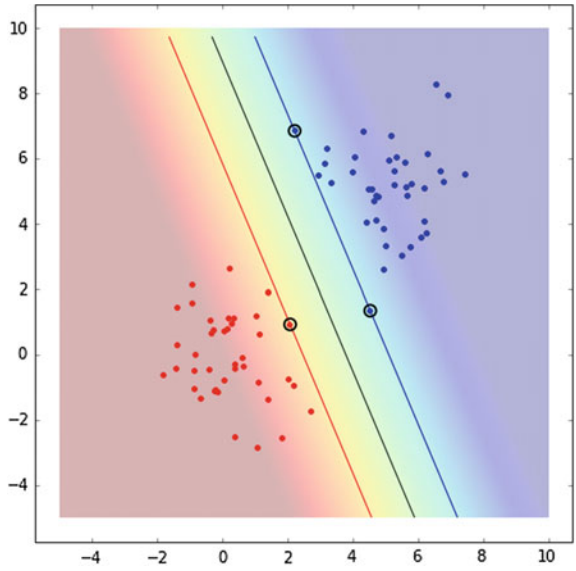

Fig. 5.9 Support vector machine decision boundary and the support vectors

separating boundary will have a point of a class closer to it than this one. The figure also shows the closest points of the classes to the boundary. These points are called *support vectors*. In fact, the boundary only depends on those points. If we remove any other point from the dataset, the boundary remains intact. However, in general, if any of these special points is removed the boundary will change.

5.7.2.1 A Brief Note on Deriving Hard Margin Support Vector Machines

In order to understand the model, we have to be able to approximately derive its formulation. For this purpose it is important to understand a couple of things about basic geometry of a hyperplane. A hyperplane in $\mathbf{R}^d$ is defined as an affine combination of the variables: $\pi \equiv a^T x + b = 0$. A hyperplane splits the space into two half-spaces. The evaluation of the equation of the hyperplane on any element belonging to one of the half-spaces is a positive value. It is a negative value for all the elements in the other half-space. The distance of a point $x \in \mathbf{R}^d$ to the hyperplane π is

$$d(x, \pi) = \frac{|a^T x + b|}{\|a\|_2}$$

Given a binary classification problem with training data $\mathcal{D} = \{(x_i, y_i)\}$, $i = 1 \ldots N$, $y_i \in \{+1, -1\}$, consider $\mathcal{S} \subseteq \mathcal{D}$ the subset of all data points belonging to class $+1$, $\mathcal{S} = \{x_i | y_i = +1\}$, and $\mathcal{R} = \{x_i | y_i = -1\}$ its complement.

Then the problem of finding a separating hyperplane consists of fulfilling the following constraints[12]

$$a^T s_i + b > 0 \text{ and } a^T r_i + b < 0 \quad \forall s_i \in \mathcal{S}, r_i \in \mathcal{R}.$$

This is a *feasibility problem* and it is usually written in the following way in optimization standard notation:

$$\begin{aligned} &\text{minimize} \quad 1 \\ &\text{subject to} \quad y_i(a^T x_i + b) \geq 1, \ \forall x_i \in \mathcal{D} \end{aligned}$$

The solution of this problem is not unique. Selecting the maximum margin hyperplane requires us to add a new constraint to our problem. Remember from the geometry of the hyperplane that the distance of any point to a hyperplane is given by: $d(x, \pi) = \frac{a^T x + b}{\|a\|_2}$.

Recall also that we want positive data to be beyond value 1 and negative data below -1. Thus, what is the distance value we want to maximize?

The positive point closest to the boundary is at $1/\|a\|_2$ and the negative point closest to the boundary data point is also at $1/\|a\|_2$. Thus, data points from different classes are at least $2/\|a\|_2$ apart.

Recall that our goal is to find the separating hyperplane with maximum margin, i.e., with maximum distance between elements in the different classes. Thus, we can complete the former formulation with our last requirement as follows:

$$\begin{aligned} &\text{minimize} \quad \|a\|_2/2 \\ &\text{subject to} \quad y_i(a^T x_i + b) \geq 1, \ \forall x_i \in \mathcal{D} \end{aligned}$$

This formulation has a solution as long as the problem is linearly separable.

In order to deal with misclassifications, we are going to introduce a new set of variables ξ_i, that represents the amount of violation in the i-th constraint. If the constraint is already satisfied, then $\xi_i = 0$; while $\xi_i > 0$ otherwise. Because ξ_i is related to the errors, we would like to keep this amount as close to zero as possible. This makes us introduce an element in the objective trade-off with the maximum margin.

[12]Note the strict inequalities in the formulation. Informally, we can consider the smallest satisfied constraint, and observe that the rest must be satisfied with a larger value. Thus, we can arbitrarily set that value to 1 and rewrite the problem as

$$a^T s_i + b \geq 1 \text{ and } a^T r_i + b \leq -1.$$

The new model becomes:

$$\begin{aligned} \text{minimize} \quad & \|a\|_2/2 + C\sum_{i=1}^{N}\xi_i \\ \text{subject to} \quad & y_i(a^T x_i + b) \geq 1 - \xi_i,\ i = 1 \ldots N \\ & \xi_i \geq 0 \end{aligned}$$

where C is the trade-off parameter that roughly balances the rates of margin and misclassification. This formulation is also called *soft-margin SVM*.

The larger the C value is, the more importance one gives to the error, i.e., the method will be more accurate according to the data at hand, at the cost of being more sensitive to variations of the data.

The decision boundary of most problems cannot be well approximated by a linear model. In SVM, the extension to the nonlinear case is handled by means of kernel theory. In a pragmatic way, a kernel can be referred to as any function that captures the similarity between any two samples in the training set. The kernel has to be a positive semi-definite function as follows:

- *Linear kernel*:
$$k(x_i, x_j) = x_i^T x_j$$
- *Polynomial kernel*:
$$k(x_i, x_j) = (1 + x_i^T x_j)^p$$
- *Radial Basis Function kernel*:
$$k(x_i, x_j) = e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}}$$

Note that selecting a polynomial or a Radial Basis Function kernel means that we have to adjust a second parameter p or σ, respectively. As a practical summary, the SVM method will depend on two parameters (C, γ) that have to be chosen carefully using cross-validation to obtain the best performance.

5.7.3 Random Forest

Random Forest (RF) is the other technique that is considered in this work. RF is an ensemble technique. Ensemble techniques rely on combining different classifiers using some aggregation technique, such as majority voting. As pointed out earlier, ensemble techniques usually have good properties for combating overfitting. In this case, the aggregation of classifiers using a voting technique reduces the variance of the final classifier. This increases the robustness of the classifier and usually achieves a very good classification performance. A critical issue in the ensemble of classifiers is that for the combination to be successful, the errors made by the members of the ensemble should be as uncorrelated as possible. This is sometimes referred to in the

literature as the diversity of the classifiers. As the name suggests, the base classifiers in RF are decision trees.

5.7.3.1 A Brief Note on Decision Trees

A decision tree is one of the most simple and intuitive techniques in machine learning, based on the divide and conquer paradigm. The basic idea behind decision trees is to partition the space into patches and to fit a model to a patch. There are two questions to answer in order to implement this solution:

- How do we partition the space?
- What model shall we use for each patch?

Tackling the first question leads to different strategies for creating decision tree. However, most techniques share the axis-orthogonal hyperplane partition policy, i.e., a threshold in a single feature. For example, in our problem "Does the applicant have a home mortgage?". This is the key that allows the results of this method to be interpreted. In decision trees, the second question is straightforward, each patch is given the value of a label, e.g., the majority label, and all data falling in that part of the space will be predicted as such.

The RF technique creates different trees over the same training dataset. The word "random" in RF refers to the fact that only a subset of features is available to each of the trees in its building process. The two most important parameters in RF are the number of trees in the ensemble and the number of features each tree is allowed to check.

5.8 Ending the Learning Process

With both techniques in mind, we are going to optimize and check the results using nested cross-validation. Scikit-learn allows us to do this easily using several model selection techniques. We will use a grid search, `GridSearchCV` (a cross-validation using an exhaustive search over all combinations of parameters provided).

In [16]:
```
parameters = {'C': [1e4, 1e5, 1e6],
              'gamma': [1e-5, 1e-4, 1e-3]}
N_folds = 5

kf=cross_validation.KFold(n = y.shape[0],
                          n_folds = N_folds,
                          shuffle = True,
                          random_state = 0)

acc = np.zeros((N_folds,))
i = 0
# We will build the predicted y from the partial predictions
     on the test of each of the folds
yhat = y.copy()
for train_index, test_index in kf:
    X_train, X_test = X[train_index,:], X[test_index,:]
    y_train, y_test = y[train_index], y[test_index]
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    clf = svm.SVC(kernel = 'rbf')
    clf = grid_search.GridSearchCV(clf, parameters, cv = 3)
    clf.fit(X_train, y_train.ravel())
    X_test = scaler.transform(X_test)
    yhat[test_index] = clf.predict(X_test)

print metrics.accuracy_score(yhat, y)
print metrics.confusion_matrix(yhat, y)
```

Out[16]:
```
classification accuracy: 0.856038647343
confusion matrix:
3371 590
   6   173
```

The result obtained has a large error in the non-fully funded class (negative). This is because the default scoring for cross-validation grid-search is mean accuracy. Depending on our business, this large error in recall for this class may be unacceptable. There are different strategies for diminishing the impact of this effect. On the one hand, we may change the default scoring and find the parameter setting that corresponds to the maximum average recall. On the other hand, we could mitigate this effect by imposing a different weight on an error on the critical class. For example, we could look for the best parameterization such than one error on the critical class is equivalent to one thousand errors on the noncritical class. This is important in business scenarios where monetization of errors can be derived.

5.9 A Toy Business Case

Consider that clients using our service yield a profit of 100 units per client (we will use abstract units but keep in mind that this will usually be accounted in euros/dollars). We design a campaign with the goal of attracting investors in order to cover all non-fully funded loans. Let us assume that the cost of the campaign is α units per client. With this policy we expect to keep our customers satisfied and engaged with our service, so they keep using it. Analyzing the confusion matrix we can

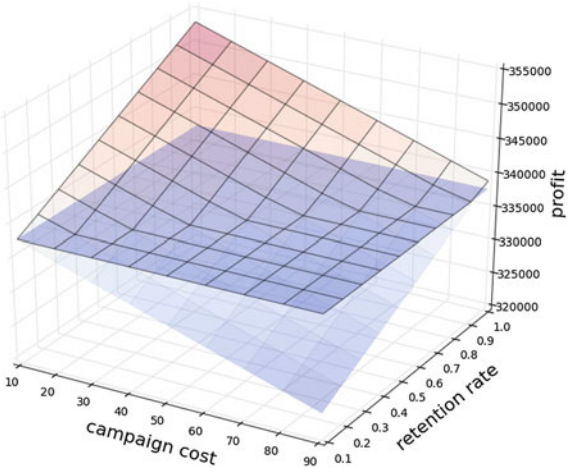

Fig. 5.10 Surfaces for two different campaign and attraction factors. The horizontal plane corresponds to the profit if no campaign is launched. The slanted plane is the profit for a certain confusion matrix

give precise meaning to different concepts in this campaign. The real positive set $(TP + FN)$ consists of the number of clients that are fully funded. According to our assumption, each of these clients generates a profit of 100 units. The total profit is $100 \cdot (TP + FN)$. The campaign to attract investors will be cast considering all the clients we predict are not fully funded. These are those that the classifier predict as negative, i.e., $(FN + TN)$. However, the campaign will only have an effect on the investors/clients that are actually not funded, i.e., TN; and we expect to attract a certain fraction β of them. After deploying our campaign, a simplified model of the expected profit is as follows:

$$100 \cdot (TP + FN) - \alpha(TN + FN) + 100\beta TN$$

When optimizing the classifier for accuracy, we do not consider the business needs. In this case, optimizing an SVM using cross-validation for different parameters of the C and γ, we have an accuracy of 85.60% and a confusion matrix with the following values:

$$\begin{pmatrix} 3371. & 590. \\ 6. & 173. \end{pmatrix}$$

If we check how the profit changes for different values of α and β, we obtain the plot in Fig. 5.10. The figure shows two hyperplanes. The horizontal plane is the expected profit if the campaign is not launched, i.e., $100 \cdot (TP + FN)$. The other hyperplane represents the profit of the campaign for different values of α and β using a particular classifier. Remember that the cost of the campaign is given by α, and the success rate of the campaign is represented by β. For the campaign to be successful we would like to select values for both parameters so that the profit of the campaign is larger than the cost of launching it. Observe in the figure that certain costs and attraction rates result in losses.

We may launch different classifiers with different configurations and toy with different weights (2, 4, 8, 16) for elements of different classes in order to bias the classi-

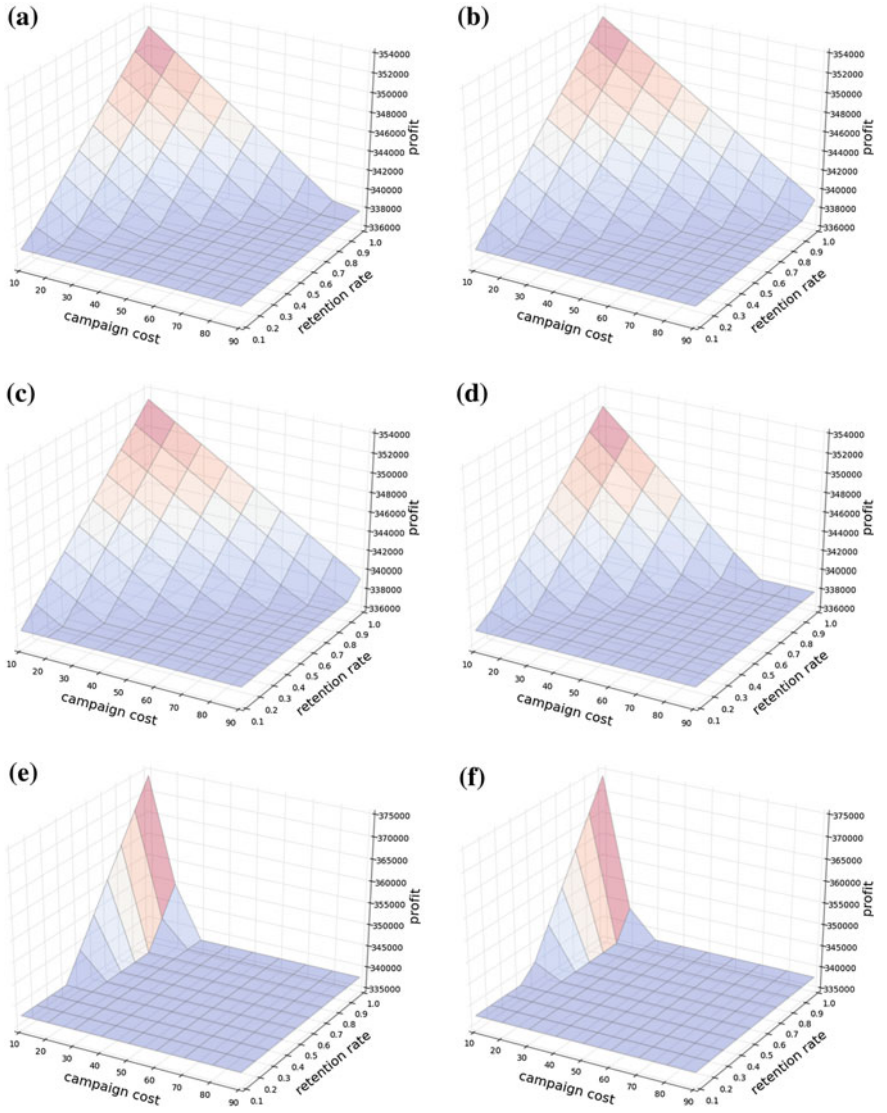

Fig. 5.11 3D surfaces of the profit obtained for different classifiers and configurations of retention campaign cost and retention rate. **a** RF, **b** SVM with the same cost per class, **c** SVM with double cost for the target class, **d** SVM with a cost for the target class equal to 4, **e** SVM with a cost for the target class equal to 8, **f** SVM with a cost for the target class equal to 16

fier towards obtaining different values for the confusion matrix.[13] The weights define

[13]It is worth mentioning that another useful tool for visualizing the trade-off between true positives and false positives in order to choose the operating point of the classifier is the receiver-operating

Table 5.1 Different configurations of classifiers and their respective profit rates and accuracies

	Max profit rate (%)	Profit rate at 60% (%)	Accuracy (%)
Random forest	4.41	2.41	87.87
SVM {1 : 1}	4.59	2.54	85.60
SVM {1 : 2}	4.52	2.50	85.60
SVM {1 : 4}	4.30	2.28	83.81
SVM {1 : 8}	10.69	3.57	52.51
SVM {1 : 16}	10.68	2.88	41.40

how much a misclassification in one class counts with respect to a misclassification in another. Figure 5.11 shows the different landscapes for different configurations of the SVM classifier and RF.

In order to frame the problem, we consider a very successful campaign with a 60% investor attraction rate. We can ask several questions in this scenario:

- What is the maximum amount to be spent on the campaign?
- How much will I gain?
- From all possible configurations of the classifier, which is the most profitable?
- Is it the one with the best accuracy?

Checking the values in Fig. 5.11, we find the results collected in Table 5.1. Observe that the most profitable campaign with 60% corresponds to a classifier that considers the cost of mistaking a sample from the non-fully funded class eight times larger than the one from the other class. Observe also that the accuracy in that case is much worse than in other configurations.

The take-home idea of this section is that business needs are often not aligned with the notion of accuracy. In such scenarios, the confusion matrix values have specific meanings. This must be taken into account when tuning the classifier.

5.10 Conclusion

In this chapter we have seen the basics of machine learning and how to apply learning theory in a practical case using Python. The example in this chapter is a basic one in which we can safely assume the data are independent and identically distributed, and that they can be readily represented in vector form. However, machine learning

(Footnote 13 continued)
characteristic (ROC) curve. This curve plots the true positive rate/sensitivity/recall (TP/(TP+FN)) with respect to the false positive rate (FP/(FP+TN)).

may tackle many more different settings. For example, we may have different target labels for a single example; this is called multilabel learning. Or, data can come from streams or be time dependent; in these settings, sequential learning or sequence learning can be the methods of choice. Moreover, each data example can be a non-vector or have a variable size, such as a graph, a tree, or a string. In such scenarios kernel learning or structural learning may be used. During these last years we are also seeing the revival of neural networks under the name of deep learning and achieving impressive results in different domains such as computer vision or natural language processing. Nonetheless, all of these methods will behave as explained in this chapter and most of the lessons learned here can be readily applied to these techniques.

Acknowledgements This chapter was co-written by Oriol Pujol and Petia Radeva.

Reference

1. M. Fernández-Delgado, E. Cernadas, S. Barro, D. Amorim, Do we Need Hundreds of Classifiers to Solve Real World Classification Problems? Journal of Machine Learning Research **15**, 3133 (2014). http://jmlr.org/papers/v15/delgado14a.html