IN THIS CHAPTER

- Revisiting the nonlinear separability problem
- Getting into the basic formulation of a linear SVM
- Grasping what kernels are and what they can achieve
- Discovering how R and Python implementations of SVMs work

Chapter 17

Going a Step beyond Using Support Vector Machines

Sometimes ideas sprout from serendipity, and sometimes from the urgent necessity to solve relevant problems. Knowing the next big thing in machine learning isn't possible yet because the technology continues to evolve. However, you can discover the tools available that will help machine learning capabilities grow, allowing you to solve more problems and feed new intelligent applications.

Support vector machines (SVMs) were the next big thing two decades ago, and when they appeared, they initially left many scholars wondering whether they'd really work. Many questioned whether the kind of representation SVMs were capable of could perform useful tasks. The representation is the capability of machine learning to approximate certain kinds of target functions expected at the foundations of a data problem. Having a good representation of a problem implies being able to produce reliable predictions on any new data.

SVMs not only demonstrate an incredible capability of representation but also a useful one, allowing the algorithm to find its niche applications in the large (and growing) world of machine learning applications. You use SVMs to drive the algorithm, not the other way around. This chapter helps you discover this fact mathematically by using a mind-blowing algorithm that, through complex calculations, can solve important problems in image recognition, medical diagnosis, and textual classification.

Revisiting the Separation Problem: A New Approach

As discussed in Chapter 12, when talking about the perceptron, nonseparability of classes can be a problem when you try to classify the examples of two classes in an exact manner: There is no straight line that traces a precise border between different examples. The set of machine learning algorithms that you choose offers some options in such occurrences:

- **K-Nearest Neighbors:** Adapts to nonlinear boundaries between classes when using a small k.
- **Logistic regression:** Solves the problem by estimating a probability of being in a particular class, thus allowing an estimate even if distinguishing between classes isn't possible due to partial overlap.
- **Transforming the features:** Solves the problem by employing both feature creation (adding human creativity and knowledge to the learning process) and automatic polynomial expansion (creating power transformations and interactions) to find a new set of features used to distinguish classes by means of a straight line.

Decision trees (discussed in Chapter 18) naturally don't fear any nonlinearity because the algorithm builds their classification boundaries using multiple rules that can easily approximate complex curves. The same is true of neural networks, which naturally create feature transformations and nonlinear approximations by making different strata of neurons connect (at the price of high estimate variability when the training process isn't handled carefully). Given an already large range of possible options, you may wonder why it was necessary to create another type of machine learning algorithm, such as SVM, to solve the nonseparability problem.

The lack of success that people have had in searching for a master algorithm that learns most problems means that you do have to believe in the no-free-lunch theorem and recognize that you can't deem one algorithm better than the other.

The kind of features you're dealing with and the kind of problem you need to solve are what determine which algorithms work better. Having access to one more effective learning technique is like getting an extra weapon to fight difficult data problems.

Moreover, SVMs feature a range of characteristics that can make SVMs quite appealing for many data problems:

- A comprehensive family of techniques for binary and multiclass classification, regression, and detection of anomalous or novelty data
- Robust handling of overfitting, noisy data, and outliers
- A capability to handle situations with many variables (and SVMs are still effective when you have more variables than examples)
- Easy and timely handling of up to about 10,000 training examples
- Automatic detection of nonlinearity in data, so that applying transformations directly to variables and feature creation isn't necessary

In particular, the last characteristic is effective thanks to the availability of special functions, the kernel functions. The special capability of kernel functions is to map the original feature space into a new feature space reconstructed to achieve better classification of regression results. It's similar to the polynomial expansion principle (one available kernel function provides polynomial expansion support), but the mathematics behind it require fewer computations, allowing the algorithm to map complex response functions in less time and with more precision.

Explaining the Algorithm

SVMs, a kind of algorithm used with Random Forests and Gradient Boosting Machines, are the creation of the mathematician Vladimir Vapnik and of some of his colleagues working at AT&T laboratories in the 1990s (such as Boser, Guyon, and Cortes). Even though many machine learning experts were initially skeptical of the algorithm because it didn't resemble any other then-existing approach, SVMs quickly gained momentum and success thanks to their performance in many image-recognition problems, such as handwritten input, that were challenging the machine learning community of the time.

Today, SVMs see widespread use among data scientists, who apply it to an incredible array of problems, from medical diagnosis to image recognition and textual classification. The technique is somewhat limited in its applicability to big data because of a lack of scalability when examples and features are too numerous.

The mathematics are a bit complex, but the idea that started everything is quite simple. With this in mind, the chapter uses easy examples and demonstrations to demonstrate the basic intuitions and mathematical knowledge.

We start by looking at the problem of separating two groups with a line. You rarely see a similar situation in real-life data, but it's the basic classification problem in machine learning, and many algorithms, such as the perceptron, are built from it. Without transforming the feature space of a hypothetical data matrix (made of two features, x1 and x2), Figure 17-1 shows how to solve the problem using (clockwise from upper left) a perceptron, a logistic regression, and an SVM.

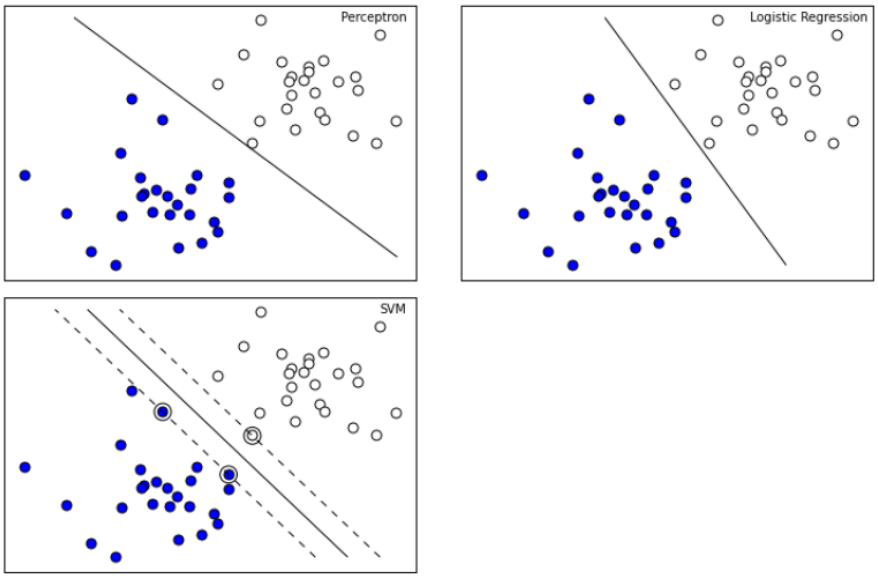

FIGURE 17-1: Comparing the different approaches.

Interestingly, although the perceptron aims only at separating the classes, and the mass of the points clearly influences the logistic regression, the separating line drawn by an SVM has clear and defined characteristics. By observing the line differences, you can see that the best line separating the classes is the one with the largest space between the line itself and the examples on the fringes of the classes, which is the solution from SVM. Using SVM terminology, the separating line is the one with the largest *margin* (the empty space between the boundaries of the classes). SVM places the separating line in the middle of the margin (described in textbooks as *maximum margin* or *optimal margin hyperplane*) and the examples near the line (thus being part of the boundaries) are *support vectors.*

TIP

Support vectors are actually examples represented as points in the feature space. In fact, in order to define the position of a point in a space, you have to state its coordinates in each dimension as a series of numbers, which is a vector. Moreover, you can view them as support vectors because the largest margin hyperplane depends on them — and changing any of them mutates the margin and the hyperplane defining it.

REMEMBER

Margin, separating hyperplane, and support vectors are key terms used to define how an SVM works. Support vectors decide the best margin possible, both in the case of linearly separable and nonseparable classes.

The strategy of an SVM is quite simple: By looking for the largest separating margin, it takes into account that the algorithm derives a classification function from a sample of observations, but that it can't rely on those observations. Sampled values do change, sometimes even greatly from sample to sample. Consequently, you can't rely on a *just fit* approach based on a single sample to understand all possible data variations (the way a perceptron does, for example). Because you can't know whether a successive sample will be similar to the one used for learning, keeping the fattest margin allows SVM flexibility in the feature space when working with successive samples. In addition, by looking at the class boundaries, SVM isn't influenced by distant points (contrary to a linear regression). The algorithm determines the margin using only the examples placed on the boundaries.

Getting into the math of an SVM

In terms of a mathematical formulation, you can define an SVM by starting with a perceptron formulation and enforcing certain constraints, that is, trying to obtain a separating line according to certain rules we are going to describe. Because the algorithm divides classes in a feature space using a separating line (or hyperplane, when you have more than two dimension features), ideally, extending the perceptron formula, the following expression would be true for every example i:

$$y(x^{T}w + b) \geq M$$

In this expression, the vector-vector multiplication of the transpose of the feature vector x by a coefficient vector w is summed with a constant bias (b). It provides a single value whose sign is indicative of the class. Because y can be only –1 or +1, you can have a value equal to or more than zero when the operations between parentheses have guessed the same sign as y. In the preceding expression, M is a constraint representing the margin: It is positive and the largest possible value to assure the best separation of predicted classes.

GETTING DEEPER INTO THE MATH

Understanding the underlying SVM math is a critical point in the formulation. It's a bit beyond the scope of this book to detail the mathematical passages for making the cost function of a perceptron into a distance. The paper "Support Vector Machines" (by Hearst and others), is available at `http://www.svms.org/tutorials/` and provides much more detail. At the same web address, you can find other interesting intermediate and advanced tutorials, too.

Linear algebra properties assure you that the preceding expression represents for each example i the distance from the separating hyperplane and that such distance is equal or more than M, the margin. In the end, SVM is based on an optimal distance calculation between examples and the hyperplane.

When you understand the preceding formula as a distance formulation, M naturally becomes the margin. The target of the optimization then becomes one of finding the parameters (the weights w) that correctly predict every example using the largest value of M possible.

Solving the set of formulations required by an SVM, given such constraints (that is, having the largest M possible and correctly predicting the training examples), requires that you use a quadratic optimization. You solve the quadratic programming problem differently from the gradient descent approach discussed so far in the book. Apart from the math difference, the solution is also more complex, requiring the algorithm to solve more computations. Complexity in an SVM depends on the number of features and the number of examples. In particular, the algorithm scales badly to the number of cases, because the number of computations is proportional to the number of examples, raised to the second or third power (depending on the problem type). For this reason, SVMs can scale well up to 10,000 examples, but beyond this limit, the computer time required for a solution may become too demanding.

REMEMBER

Even though they are created for classification, SVMs can also perform regression problems well. You use the prediction error as distance and the separating hyperplane surface values for determining the right prediction value at each feature value combination.

Avoiding the pitfalls of nonseparability

You need to consider a final issue concerning the SVM optimization process previously described. Because you know that classes are seldom linearly separable, it may not always be possible for all the training examples to force the distance

to equal or exceed M. In this case, it's necessary to consider the following expression:

$$y(x^{T}w+b) \geq M(1+\epsilon_{i})$$

Each example has an epsilon value that's always major or equal to zero, in such a way that epsilon turns into a value correcting the wrong classification of specific vectors. If an example is correctly classified, then epsilon is 0. Values of epsilon between 0 and 1 indicate that the example is on the right side of the separating line, yet it entered the margin. Finally, when epsilon is above 1, the case is misclassified and is located on the other side of the optimal separating hyperplane.

Epsilon allows the algorithm to correct mismatched cases so that the optimization process will accept them. You can define the extent of this correction by specifying that the sum of all epsilons should be less than a value, C, which acts as a way to create a solution that is more affected by bias (if the value of C is small) or variance (if the value of C is large). The value of C approximately indicates the number of mismatches that the SVM can ignore when learning.


REMEMBER

The C parameter is very important, if not the most important parameter of all in an SVM. Epsilon can create exceptions in classification and allow a more realistic fit of the SVM formulation when data is imprecise and noisy. However, if you introduce too much correction (thus, C is too large), the search for the optimal margin will accept too many exceptions, creating an overfitting to data. On the other hand, if C is too little, the SVM will look for a complete separation of the points, which results in an unsuccessful suboptimal hyperplane whose margin is too small.


TIP

Be sure not to confuse the C value in the preceding formulation with the C parameter required by SVM software implementations. Both R and Python implementations of SVMs accept a C parameter as a cost for a mismatch. As a software parameter, a high C value leads to a smaller margin, because costs are higher when an SVM classifies incorrectly. On the other hand, a low C implies a larger margin, open acceptance of examples wrongly predicted, and thus an increased variance of estimates.

Applying Nonlinearity

SVMs are quite mathematically demanding. Up to now, you've seen some formulations that help you understand that SVMs are an optimization problem that strives to classify all the examples of two classes. When solving the optimization in SVMs, you use a partitioning hyperplane having the largest distance from the class boundaries. If classes aren't linearly separable, the search for the optimal

separating hyperplane allows for errors (quantified by the value of C) in order to deal with noise.

In spite of allowing a small cost for errors, the SVM's linear hyperplane can't recover nonlinear relationships between classes unless you transform the features appropriately. For instance, you can correctly classify only a part of the examples like the ones depicted in Figure 17-2 if you don't transform the existing two dimensions into other dimensions using multiplication or powers.

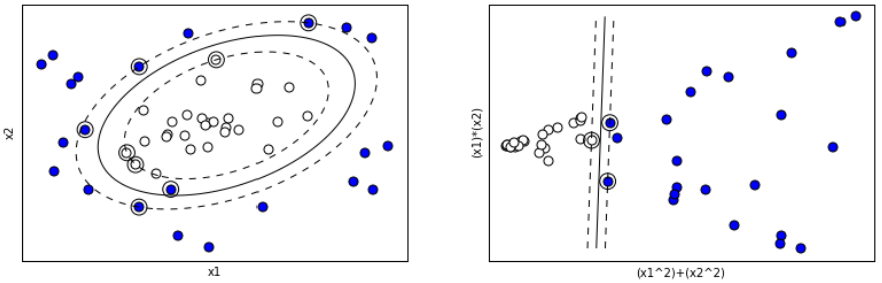

FIGURE 17-2: A case of nonlinearly separable points requiring feature transformation (left) to be fit by a line (right).

In other words, you map the existing features onto a feature space of higher dimensionality in hopes of finding a way to separate the classes linearly. This is the same process shown in Chapter 15 when trying polynomial expansion (where you discovered how a linear model captures nonlinear relationships between variables automatically). Automatic feature creation using polynomial expansion, which is also possible for SVMs, has some important limits:

- The number of features increases exponentially, making computations cumbersome and saturating the in-memory dataset. (Some datasets cannot be expanded beyond a power of 2 or 3.)
- The expansion creates many redundant features, causing overfitting.
- It's difficult to determine the level of expansion where the classes will become linearly separable, thus requiring many iterations of expansion and test.

As a consequence of these limitations, SVM has adopted a different way, called *kernel functions*, to redefine the feature space without occupying more memory or increasing the number of computations too much. Kernel functions aren't magic; they rely on algebra calculations, but they are even more mathematically demanding than SVM. As a means of understanding how they work, you could say that kernel functions project the original features into a higher dimensional space by combining them in a nonlinear way. They do so in an implicit way because they don't provide SVM with a new feature space to use for learning (as a polynomial expansion would do). Instead, they rely on a vector of values that the SVM can use directly to fit a nonlinear separating hyperplane.

Kernel functions, therefore, provide the result of a combination of features (precisely a dot-product, a multiplication between vectors), without calculating all the combinations involved in such result. This is called the *kernel trick* and is possible for SVMs because their optimization processes can be reformulated into another form, called the *dual formulation* (in contrast with the previous formula, called the *primal formulation*). The dual formulation operates directly on the results of the kernel functions and not on the original features.


REMEMBER

Kernels aren't just for SVMs. You can apply them to any machine learning algorithm using a formulation operating on the dot-product between examples, as the SVM dual formulation does.


TIP

If you would like to know more about the dual formulation and how it differs from the primal one, you can find more details in this Quora question at `https://www.quora.com/Support-Vector-Machines/Why-is-solving-in-the-dual-easier-than-solving-in-the-primal`, or more formally in this tutorial from Microsoft Research at `http://research.microsoft.com/en-us/um/people/manik/projects/trade-off/svm.html`.

Demonstrating the kernel trick by example

In this section, as an example of how a kernel function operates, you see how to make an explicit and implicit transformation by using Python. The cost function is a dual one and operates on dot-products of examples. The dataset is minimal: two examples with three features. The goal is to map our existing features to a higher dimensionality, using all the possible combinations between the features. So if an example is made of the feature vector of values (1,2,3), projecting it this way results in a new vector of values (1, 2, 3, 2, 4, 6, 3, 6, 9). The projection grows the dimensionality from three to nine features, occupying three times more computer memory.

```
import numpy as np
X = np.array([[1,2,3],[3,2,1]])
def poly_expansion(A):
    return np.array([[x*y for x in
        row for y in row] for row in A])

poly_X = poly_expansion(X)
print ('Dimensions after expanding: %s'
   % str(poly_X.shape))
print (poly_X)

Dimensions after expanding: (2, 9)
[[1 2 3 2 4 6 3 6 9]
 [9 6 3 6 4 2 3 2 1]]
```

The code, using the `poly_expansion` function, iterates the X matrix (the dataset of two examples and three features), expanding the number of features to nine. At this point, you can compute the dot-product.

```
np.dot(poly_X[0],poly_X[1])

100
```

By using a kernel function instead, you can simply call the function after providing the two features vectors of the examples. You obtain the result without having to create a new dataset.

```
def poly_kernel(a, b):
    return np.sum(a*b)**2

poly_kernel(X[0], X[1])

100
```


REMEMBER

Kernel functions are convenient mapping functions that allow SVMs to obtain a transformed dataset of limited size, which is equivalent to a more complicated and data-intensive nonlinear transformation. Computationally accessible by most computers in terms of processing and memory, kernel functions allow you to automatically try solving a data problem using a nonlinear separating hyperplane without involving any particular human intervention in feature creation.

Discovering the different kernels

In its implementations in both R and Python, SVM offers quite a large range of nonlinear kernels. Here is a list of the kernels and their parameters:

- **Linear:** No extra parameters
- **Radial Basis Function:** Shape parameters: `gamma`
- **Polynomial:** Shape parameters: `gamma`, `degree`, and `coef0`
- **Sigmoid:** Shape parameters: `gamma` and `coef0`
- **Custom-made kernels:** Depends upon the kernel

Even though the choice is large and possibly larger if you start designing your own custom kernel, in practice, using just one of them, the Radial Basis Function (RBF), is common because it's faster than other kernels. In addition, it can map and approximate almost any nonlinear function if you tweak its shape parameter, `gamma`.

The RBF works in a simple but smart way. It creates a margin around every support vector — drawing bubbles in the feature space, as shown in Figure 17-3. Then, according to the `gamma` hyper-parameter value, it expands or restricts the volume of the bubbles so that they fuse with each other and shape classification areas. The `gamma` value plays the role of the radius of all the bubbles that RBF created. The resulting margin and the hyperplane passing through it will consequently show very curvy boundaries, demonstrating that it can be quite flexible, as in the examples provided by Figure 17-3.

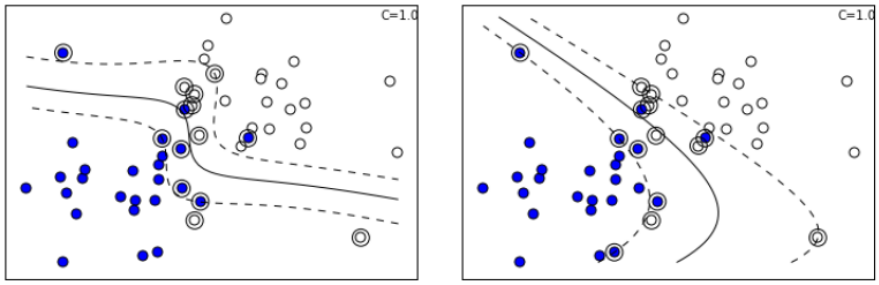

FIGURE 17-3: An RBF kernel that uses diverse hyper-parameters to create unique SVM solutions.

The RBF kernel can adapt itself to different learning strategies, resulting in a bended hyperplane when C, the error cost, is high, and creating a smoother curve line when C is low. It can also fit complex shapes, such as the bull's eye, when a class is placed inside another. All this flexibility comes at the expense of a larger estimate variance, but it also detects complex classification rules that other algorithms may fail to find.

REMEMBER

When tuning an RBF kernel, first fix C, the error cost, to define a bended separating hyperplane; then tune `gamma` to make the margin shape rough and broken when the hyper-parameter `gamma` is low or regular and fused into large bubble-shaped areas when it is high.

The polynomial and sigmoid kernels aren't as adaptable as RBF, thus showing more bias, yet they are both nonlinear transformations of the separating hyperplane. If the sigmoid features a single bend, the polynomial function can have as many bends or separated hyperplanes as its set degree. The higher the degree, the longer it takes to compute.

REMEMBER

You have many shape values to fix when using the sigmoid and polynomial kernels: `gamma` and `coef0` for both and `degree` for polynomial. Because determining the effects that different values have on these parameters is difficult, you need to test different value combinations, using a grid search, for instance, and evaluate the results on a practical basis. Figure 17-4 provides visual representations of what you can do with the polynomial and the sigmoid kernels.

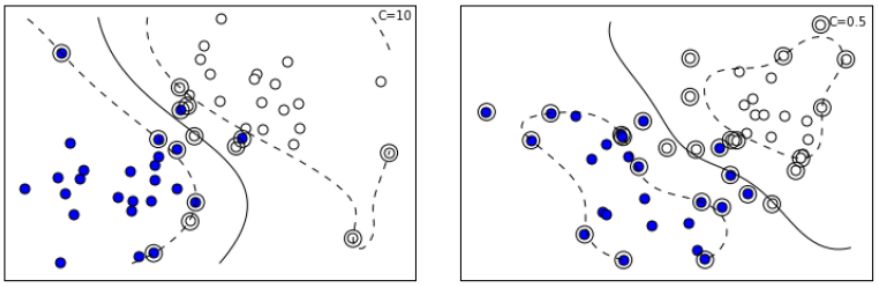

FIGURE 17-4: A sigmoid (left) and a polynomial kernel (right) applied to the same data.

TIP

In spite of the possibility of creating your own kernel function, most data problems are easily solved using the RBF. Simply look for the right combination C and `gamma` by trying different values systematically until you achieve a best result for your validation set or cross-validation procedure.

Illustrating Hyper-Parameters

Although SVMs are complex learning algorithms, they are versatile and easy to use when implemented by a Python class or an R function. Interestingly, both Scikit-learn and the R e1071 libraries (developed by the TU Wien E1071 group on probability theory) rely on the same external C++ library (with a C API to interface with other languages) developed at the National Taiwan University. You can find more on the shared LIBSVM for SVM classification and regression at `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`. The Python implementation also uses the LIBLINEAR C library from the same authors at Taiwan University, which specializes in classification problems using linear methods on large and sparse datasets (see more at `http://www.csie.ntu.edu.tw/~cjlin/liblinear/`).

Because both Python and R wrap the same C library, they offer the same functionality with the same hyper-parameters. Tables 17-1 and 17-2 show a complete overview of both classification and regression SVM across the two languages.

TABLE 17-1 Software Implementations for Classification

Python/R implementation	Purpose	Hyper-parameters
Python: sklearn.svm.SVC R: svm(type= "C-classification")	The LIBSVM implementation for binary and multiclass linear and kernel classification	C (cost in R), kernel, degree, gamma, coef0
Python: sklearn.svm.NuSVC R: svm(type="nu-classification")	Same as above	nu, kernel, degree, gamma. coef0

Python/R implementation	Purpose	Hyper-parameters
Python: sklearn.svm.OneClassSVM R: svm(type="one-classification")	Unsupervised detection of outliers	nu, kernel, degree, gamma, coef0
Python: sklearn.svm.LinearSVC	Based on LIBLINEAR, it is a binary and multiclass linear classifier	Penalty, loss, C (cost in R)

TABLE 17-2 Regression Implementations

Class	Purpose	Hyper-parameters
Python: sklearn.svm.SVR R: svm(type="eps-regression")	The LIBSVM implementation for regression	C (cost in R), kernel, degree, gamma, epsilon, coef0
Python: sklearn.svm.NuSVR R: svm(type="nu-regression")	Same as preceding item	nu, C (cost in R), kernel, degree, gamma, coef0

In previous sections, the chapter discusses most of the hyper-parameters, such as `C`, `kernel`, `gamma`, and `degree`, when describing how an SVM algorithm works in its basic formulation and with kernels. You may be surprised to find that two new SVM versions exist for both classification and regression algorithms; one of them is based on the `Nu` hyper-parameter. The only differences with the `Nu` version are the parameters it takes and the use of a slightly different algorithm. Using either SVM version gets the same results, so you normally choose the non-`Nu` version.

Python also offers a linear version of SVM, `LinearSVC`, which is more like the linear regression model than an SVC. It also permits regularization and provides a fast SVM classification that works well with sparse textual matrices. You see this implementation again in Part 5 when dealing with text classification.

Classifying and Estimating with SVM

As an example of how you can use an SVM to work out a complex problem, this section demonstrates a handwritten recognition task and solves it using a nonlinear kernel, the RBF. The SVM algorithm learns from the digits dataset available from the module datasets in the Scikit-learn package. The digits dataset contains a series of 8-x-8 grayscale pixel images of handwritten numbers ranging from 0 to 9. The problem is quite simple when compared to many problems that image recognition engines solve today, but it helps you grasp the potential of the