6 Support vector machines

This chapter covers

- Introducing support vector machines
- Using the SMO algorithm for optimization
- Using kernels to "transform" data
- Comparing support vector machines with other classifiers

I've seen more than one book follow this pattern when discussing support vector machines (SVMs): "Here's a little theory. Now SVMs are too hard for you. Just download libsvm and use that." I'm not going to follow that pattern. I think if you just read a little bit of the theory and then look at production C++ SVM code, you're going to have trouble understanding it. But if we strip out the production code and the speed improvements, the code becomes manageable, perhaps understandable.

Support vector machines are considered by some people to be the best stock classifier. By *stock*, I mean not modified. This means you can take the classifier in its basic form and run it on the data, and the results will have low error rates. Support vector machines make good decisions for data points that are outside the training set.

In this chapter you're going to learn what support vector machines are, and I'll introduce some key terminology. There are many implementations of support vector

machines, but we'll focus on one of the most popular implementations: the sequential minimal optimization (SMO) algorithm. After that, you'll see how to use something called *kernels* to extend SVMs to a larger number of datasets. Finally, we'll revisit the handwriting example from chapter 1 to see if we can do a better job with SVMs.

Separating data with the maximum margin

> **Support vector machines**
>
> Pros: Low generalization error, computationally inexpensive, easy to interpret results
>
> Cons: Sensitive to tuning parameters and kernel choice; natively only handles binary classification
>
> Works with: Numeric values, nominal values

To introduce the subject of support vector machines I need to explain a few concepts. Consider the data in frames A–D in figure 6.1; could you draw a straight line to put all of the circles on one side and all of the squares on another side? Now consider the data in figure 6.2, frame A. There are two groups of data, and the data points are separated enough that you could draw a straight line on the figure with all the points of one class on one side of the line and all the points of the other class on the other side of the line. If such a situation exists, we say the data is *linearly separable*. Don't worry if this assumption seems too perfect. We'll later make some changes where the data points can spill over the line.

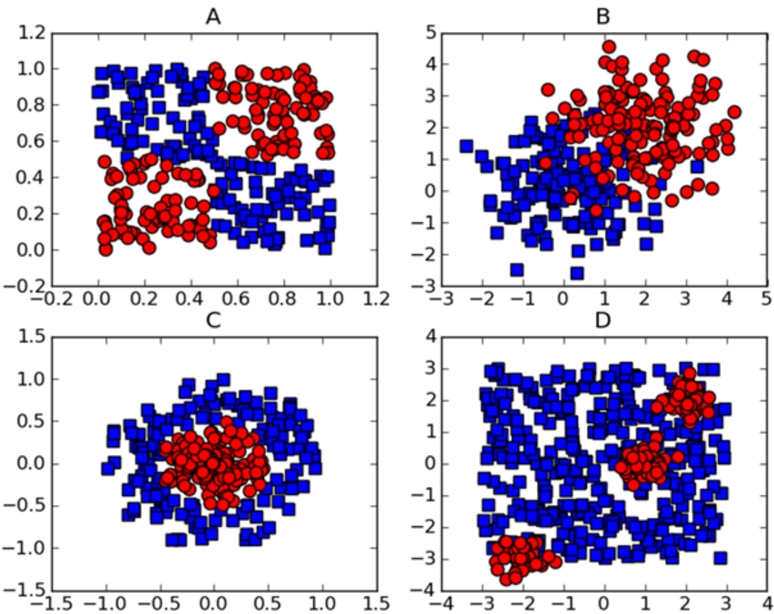

Figure 6.1 Four examples of datasets that aren't linearly separable

The line used to separate the dataset is called a *separating hyperplane*. In our simple 2D plots, it's just a line. But, if we have a dataset with three dimensions, we need a plane to separate the data; and if we have data with 1024 dimensions, we need something with 1023 dimensions to separate the data. What do you call something with 1023 dimensions? How about N-1 dimensions? It's called a *hyperplane*. The hyperplane is our decision boundary. Everything on one side belongs to one class, and everything on the other side belongs to a different class.

We'd like to make our classifier in such a way that the farther a data point is from the decision boundary, the more confident we are about the prediction we've made. Consider the plots in figure 6.2, frames B–D. They all separate the data, but which one does it best? Should we minimize the average distance to the separating hyperplane? In that case, are frames B and C any better than frame D in figure 6.2? Isn't something like that done with best-fit lines? Yes, but it's not the best idea here. We'd like to find the point closest to the separating hyperplane and make sure this is as far away from the separating line as possible. This is known as *margin*. We want to have the greatest possible margin, because if we made a mistake or trained our classifier on limited data, we'd want it to be as robust as possible.

The points closest to the separating hyperplane are known as *support vectors*. Now that we know that we're trying to maximize the distance from the separating line to the support vectors, we need to find a way to optimize this problem.

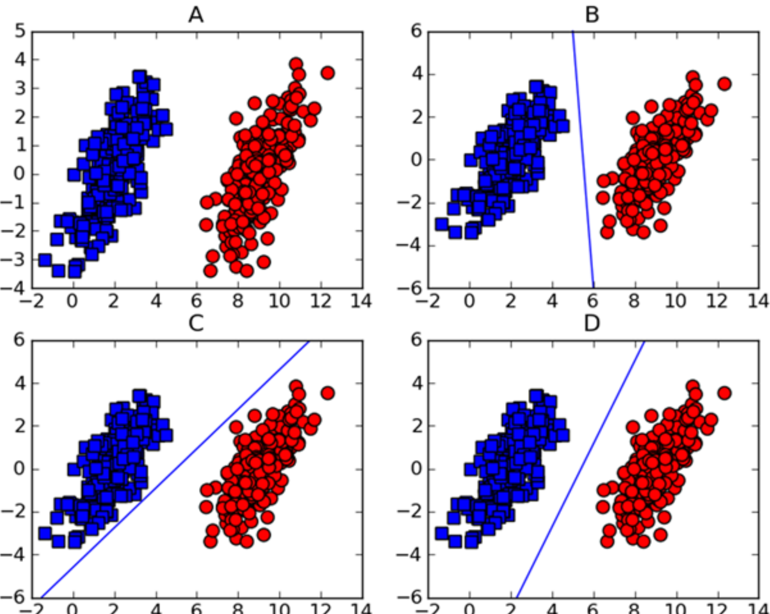

Figure 6.2 Linearly separable data is shown in frame A. Frames B, C, and D show possible valid lines separating the two classes of data.

Finding the maximum margin

How can we measure the line that best separates the data? To start with, look at figure 6.3. Our separating hyperplane has the form $\mathbf{w}^T\mathbf{x}+b$. If we want to find the distance from A to the separating plane, we must measure normal or perpendicular to the line. This is given by $|\mathbf{w}^T\mathbf{x}+b|/||\mathbf{w}||$. The constant b is just an offset like w_0 in logistic regression. All this w and b stuff describes the separating line, or hyperplane, for our data. Now, let's talk about the classifier.

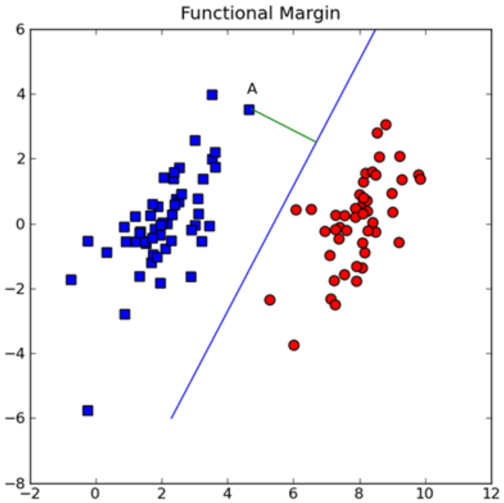

Figure 6.3 The distance from point A to the separating plane is measured by a line normal to the separating plane.

Framing the optimization problem in terms of our classifier

I've talked about the classifier but haven't mentioned how it works. Understanding how the classifier works will help you to understand the optimization problem. We'll have a simple equation like the sigmoid where we can enter our data values and get a class label out. We're going to use something like the Heaviside step function, $f(\mathbf{w}^T\mathbf{x}+b)$, where the function $f(u)$ gives us -1 if $u<0$, and 1 otherwise. This is different from logistic regression in the previous chapter where the class labels were 0 or 1.

Why did we switch from class labels of 0 and 1 to -1 and 1? This makes the math manageable, because -1 and 1 are only different by the sign. We can write a single equation to describe the margin or how close a data point is to our separating hyperplane and not have to worry if the data is in the -1 or +1 class.

When we're doing this and deciding where to place the separating line, this margin is calculated by $label*(\mathbf{w}^T\mathbf{x}+b)$. This is where the -1 and 1 class labels help out. If a point is far away from the separating plane on the positive side, then $\mathbf{w}^T\mathbf{x}+b$ will be a large positive number, and $label*(\mathbf{w}^T\mathbf{x}+b)$ will give us a large number. If it's far from the negative side and has a negative label, $label*(\mathbf{w}^T\mathbf{x}+b)$ will also give us a large positive number.

The goal now is to find the **w** and b values that will define our classifier. To do this, we must find the points with the smallest margin. These are the support vectors briefly mentioned earlier. Then, when we find the points with the smallest margin, we must maximize that margin. This can be written as

$$\arg\max_{w,b}\left\{\min_{n}\left(label\cdot(\boldsymbol{w}^T\boldsymbol{x}+b)\right)\cdot\frac{1}{\|\boldsymbol{w}\|}\right\}$$

Solving this problem directly is pretty difficult, so we can convert it into another form that we can solve more easily. Let's look at the inside of the previous equation, the part inside the curly braces. Optimizing multiplications can be nasty, so what we do is hold one part fixed and then maximize the other part. If we set `label*(`$\mathbf{w}^T\mathbf{x}$`+b)` to be 1 for the support vectors, then we can maximize $||\mathbf{w}||^{-1}$ and we'll have a solution. Not all of the `label*(`$\mathbf{w}^T\mathbf{x}$`+b)` will be equal to 1, only the closest values to the separating hyperplane. For values farther away from the hyperplane, this product will be larger.

The optimization problem we now have is a constrained optimization problem because we must find the best values, provided they meet some constraints. Here, our constraint is that `label*(`$\mathbf{w}^T\mathbf{x}$`+b)` will be 1.0 or greater. There's a well-known method for solving these types of constrained optimization problems, using something called Lagrange multipliers. Using Lagrange multipliers, we can write the problem in terms of our constraints. Because our constraints are our data points, we can write the values of our hyperplane in terms of our data points. The optimization function turns out to be

$$\max_{\alpha}\left[\sum_{i=1}^{m}\alpha-\frac{1}{2}\sum_{i,j=1}^{m} label^{(i)}\cdot label^{(j)}\cdot a_i\cdot a_j\langle x^{(i)},x^{(j)}\rangle\right]$$

subject to the following constraints:

$$\alpha\geq 0, and \sum_{i-1}^{m}\alpha_i\cdot label^{(i)}=0$$

This is great, but it makes one assumption: the data is 100% linearly separable. We know by now that our data is hardly ever that clean. With the introduction of something called *slack variables*, we can allow examples to be on the wrong side of the decision boundary. Our optimization goal stays the same, but we now have a new set of constraints:

$$c\geq\alpha\geq 0, and \sum_{i-1}^{m}\alpha_i\cdot label^{(i)}=0$$

The constant `C` controls weighting between our goal of making the margin large and ensuring that most of the examples have a functional margin of at least 1.0. The constant `C` is an argument to our optimization code that we can tune and get different results. Once we solve for our alphas, we can write the separating hyperplane in terms of these alphas. That part is straightforward. The majority of the work in SVMs is finding the alphas.

There have been some large steps taken in coming up with these equations here. I encourage you to seek a textbook to see a more detailed derivation if you're interested.[1,2]

Christopher M. Bishop, *Pattern Recognition and Machine Learning* (Springer, 2006).

Bernhard Schlkopf and Alexander J. Smola, *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond* (MIT Press, 2001).

2 Approaching SVMs with our general framework

In chapter 1, we defined common steps for building machine learning–based applications. These steps may change from one machine learning task to another and from one algorithm to another. It's worth taking a few minutes to see how these will apply to the algorithm we're looking at in this chapter.

> **General approach to SVMs**
>
> 1. Collect: Any method.
> 2. Prepare: Numeric values are needed.
> 3. Analyze: It helps to visualize the separating hyperplane.
> 4. Train: The majority of the time will be spent here. Two parameters can be adjusted during this phase.
> 5. Test: Very simple calculation.
> 6. Use: You can use an SVM in almost any classification problem. One thing to note is that SVMs are binary classifiers. You'll need to write a little more code to use an SVM on a problem with more than two classes.

Now that we have a little bit of the theory behind us, we'd like to be able to program this problem so that we can use it on our data. The next section will introduce a simple yet powerful algorithm for doing so.

Efficient optimization with the SMO algorithm

The last two equations in section 6.2.1 are what we're going to minimize and some constraints that we have to follow while minimizing. A while back, people were using quadratic solvers to solve this optimization problem. (A *quadratic solver* is a piece of software that optimizes a quadratic function of several variables, subject to linear constraints on the variables.) These quadratic solvers take a lot of computing power and are complex. All of this messing around with optimization is to train our classifier. When we find the optimal values of α, we can get our separating hyperplane or line in 2D and then easily classify data.

We'll now discuss the SMO algorithm, and then we'll write a simplified version of it so that you can properly understand how it works. The simplified version works on small datasets. In the next section we'll move from the simplified to the full version, which works much faster than the simplified version.

1 Platt's SMO algorithm

In 1996 John Platt published a powerful algorithm he called SMO[3] for training the support vector machines. Platt's SMO stands for Sequential Minimal Optimization,

hn C. Platt, "Using Analytic QP and Sparseness to Speed Training of Support Vector Machines" in *Advances Neural Information Processing Systems* 11, M. S. Kearns, S. A. Solla, D. A. Cohn, eds (MIT Press, 1999), 557–63.

and it takes the large optimization problem and breaks it into many small problems. The small problems can easily be solved, and solving them sequentially will give you the same answer as trying to solve everything together. In addition to getting the same answer, the amount of time is greatly reduced.

The SMO algorithm works to find a set of alphas and `b`. Once we have a set of alphas, we can easily compute our weights **w** and get the separating hyperplane.

Here's how the SMO algorithm works: it chooses two alphas to optimize on each cycle. Once a suitable pair of alphas is found, one is increased and one is decreased. To be suitable, a set of alphas must meet certain criteria. One criterion a pair must meet is that both of the alphas have to be outside their margin boundary. The second criterion is that the alphas aren't already clamped or bounded.

3.2 Solving small datasets with the simplified SMO

Implementing the full Platt SMO algorithm can take a lot of code. We'll simplify it in our first example to get an idea of how it works. After we get the simplified version working, we'll build on it to see the full version. The simplification uses less code but takes longer at runtime. The outer loops of the Platt SMO algorithm determine the best alphas to optimize. We'll skip that for this simplified version and select pairs of alphas by first going over every alpha in our dataset. Then, we'll choose the second alpha randomly from the remaining alphas. It's important to note here that we change two alphas at the same time. We need to do this because we have a constraint:

$$\sum a_i \cdot label^{(i)} = 0$$

Changing one alpha may cause this constraint to be violated, so we always change two at a time.

To do this we're going to create a helper function that randomly selects one integer from a range. We also need a helper function to clip values if they get too big. These two functions are given in the following listing. Open a text editor and add the code to svmMLiA.py.

Listing 6.1 Helper functions for the SMO algorithm

```
def loadDataSet(fileName):
    dataMat = []; labelMat = []
    fr = open(fileName)
    for line in fr.readlines():
        lineArr = line.strip().split('\t')
        dataMat.append([float(lineArr[0]), float(lineArr[1])])
        labelMat.append(float(lineArr[2]))
    return dataMat,labelMat

def selectJrand(i,m):
    j=i
    while (j==i):
        j = int(random.uniform(0,m))
    return j

def clipAlpha(aj,H,L):
    if aj > H:
```

```
        aj = H
    if L > aj:
        aj = L
    return aj
```

The data that's plotted in figure 6.3 is available in the file testSet.txt. We'll use this dataset to develop the SMO algorithm. The first function in listing 6.1 is our familiar `loadDatSet()`, which opens up the file and parses each line into class labels, and our data matrix.

The next function, `selectJrand()`, takes two values. The first one, `i`, is the index of our first alpha, and `m` is the total number of alphas. A value is randomly chosen and returned as long as it's not equal to the input i.

The last helper function, `clipAlpha()`, clips alpha values that are greater than `H` or less than `L`. These three helper functions don't do much on their own, but they'll be useful in our classifier.

After you've entered the code from listing 6.1 and saved it, you can try these out using the following:

```
>>> import svmMLiA
>>> dataArr,labelArr = svmMLiA.loadDataSet('testSet.txt')
>>> labelArr
[-1.0, -1.0, 1.0, -1.0, 1.0, 1.0, 1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0,
1.0...
```

You can see that the class labels are -1 and 1 rather than 0 and 1.

Now that we have these working, we're ready for our first version of the SMO algorithm.

Pseudocode for this function would look like this:

Create an alphas vector filled with 0s
While the number of iterations is less than MaxIterations:
    For every data vector in the dataset:
        If the data vector can be optimized:
            Select another data vector at random
            Optimize the two vectors together
            If the vectors can't be optimized → break
    If no vectors were optimized → increment the iteration count

The code in listing 6.2 is a working version of the SMO algorithm. In Python, if we end a line with \, the interpreter will assume the statement is continued on the next line. There are a number of long lines in the following code that need to be broken up, so I've used the \ symbol for this. Open the file svmMLiA.py and enter the code from the following listing.

Listing 6.2 The simplified SMO algorithm

```
def smoSimple(dataMatIn, classLabels, C, toler, maxIter):
    dataMatrix = mat(dataMatIn); labelMat = mat(classLabels).transpose()
    b = 0; m,n = shape(dataMatrix)
```

```
    alphas = mat(zeros((m,1)))
    iter = 0
    while (iter < maxIter):
        alphaPairsChanged = 0
        for i in range(m):
            fXi = float(multiply(alphas,labelMat).T*\
                        (dataMatrix*dataMatrix[i,:].T)) + b
            Ei = fXi - float(labelMat[i])
            if ((labelMat[i]*Ei < -toler) and (alphas[i] < C)) or \
                ((labelMat[i]*Ei > toler) and \
                (alphas[i] > 0)):
                j = selectJrand(i,m)
                fXj = float(multiply(alphas,labelMat).T*\
                            (dataMatrix*dataMatrix[j,:].T)) + b
                Ej = fXj - float(labelMat[j])
                alphaIold = alphas[i].copy();
 alphaJold = alphas[j].copy();
                if (labelMat[i] != labelMat[j]):
                    L = max(0, alphas[j] - alphas[i])
                    H = min(C, C + alphas[j] - alphas[i])
                else:
                    L = max(0, alphas[j] + alphas[i] - C)
                    H = min(C, alphas[j] + alphas[i])
                if L==H: print "L==H"; continue
                eta = 2.0 * dataMatrix[i,:]*dataMatrix[j,:].T - \
                      dataMatrix[i,:]*dataMatrix[i,:].T - \
                      dataMatrix[j,:]*dataMatrix[j,:].T
                if eta >= 0: print "eta>=0"; continue
                alphas[j] -= labelMat[j]*(Ei - Ej)/eta
                alphas[j] = clipAlpha(alphas[j],H,L)
                if (abs(alphas[j] - alphaJold) < 0.00001): print \
                        "j not moving enough"; continue
                alphas[i] += labelMat[j]*labelMat[i]*\
                          (alphaJold - alphas[j])
                b1 = b - Ei- labelMat[i]*(alphas[i]-alphaIold)*\
                     dataMatrix[i,:]*dataMatrix[i,:].T - \
                     labelMat[j]*(alphas[j]-alphaJold)*\
                     dataMatrix[i,:]*dataMatrix[j,:].T
                b2 = b - Ej- labelMat[i]*(alphas[i]-alphaIold)*\
                     dataMatrix[i,:]*dataMatrix[j,:].T - \
                     labelMat[j]*(alphas[j]-alphaJold)*\
                     dataMatrix[j,:]*dataMatrix[j,:].T
                if (0 < alphas[i]) and (C > alphas[i]): b = b1
                elif (0 < alphas[j]) and (C > alphas[j]): b = b2
                else: b = (b1 + b2)/2.0
                alphaPairsChanged += 1
                print "iter: %d i:%d, pairs changed %d" % \
                                    (iter,i,alphaPairsChanged)
        if (alphaPairsChanged == 0): iter += 1
        else: iter = 0
        print "iteration number: %d" % iter
    return b,alphas
```

1. Enter optimization if alphas can be changed
2. Randomly select second alpha
3. Guarantee alphas stay between 0 and C
4. Update i by same amount as j in opposite direction
5. Set the constant term

This is one big function, I know. It's probably the biggest one you'll see in this book. This function takes five inputs: the dataset, the class labels, a constant C, the tolerance,

and the maximum number of iterations before quitting. We've been building functions in this book with a common interface so you can mix and match algorithms and data sources. This function takes lists and inputs and transforms them into NumPy matrices so that you can simplify many of the math operations. The class labels are transposed so that you have a column vector instead of a list. This makes the row of the class labels correspond to the row of the data matrix. You also get the constants `m` and `n` from the shape of the `dataMatIn`. Finally, you create a column matrix for the alphas, initialize this to zero, and create a variable called `iter`. This variable will hold a count of the number of times you've gone through the dataset without any alphas changing. When this number reaches the value of the input `maxIter`, you exit.

In each iteration, you set `alphaPairsChanged` to 0 and then go through the entire set sequentially. The variable `alphaPairsChanged` is used to record if the attempt to optimize any alphas worked. You'll see this at the end of the loop. First, `fXi` is calculated; this is our prediction of the class. The error `Ei` is next calculated based on the prediction and the real class of this instance. If this error is large, then the alpha corresponding to this data instance can be optimized. This condition is tested ❶. In the `if` statement, both the positive and negative margins are tested. In this `if` statement, you also check to see that the alpha isn't equal to 0 or `C`. Alphas will be clipped at 0 or `C`, so if they're equal to these, they're "bound" and can't be increased or decreased, so it's not worth trying to optimize these alphas.

Next, you randomly select a second alpha, `alpha[j]`, using the helper function described in listing 6.1 ❷. You calculate the "error" for this alpha similar to what you did for the first alpha, `alpha[i]`. The next thing you do is make a copy of `alpha[i]` and `alpha[j]`. You do this with the `copy()` method, so that later you can compare the new alphas and the old ones. Python passes all lists by reference, so you have to explicitly tell Python to give you a new memory location for `alphaIold` and `alphaJold`. Otherwise, when you later compare the new and old values, we won't see the change. You then calculate `L` and `H` ❸, which are used for clamping `alpha[j]` between 0 and `C`. If `L` and `H` are equal, you can't change anything, so you issue the `continue` statement, which in Python means "quit this loop now, and proceed to the next item in the `for` loop."

`Eta` is the optimal amount to change `alpha[j]`. This is calculated in the long line of algebra. If `eta` is 0, you also quit the current iteration of the `for` loop. This step is a simplification of the real SMO algorithm. If `eta` is 0, there's a messy way to calculate the new `alpha[j]`, but we won't get into that here. You can read Platt's original paper if you really want to know how that works. It turns out this seldom occurs, so it's OK if you skip it. You calculate a new `alpha[j]` and clip it using the helper function from listing 6.1 and our `L` and `H` values.

Next, you check to see if `alpha[j]` has changed by a small amount. If so, you quit the `for` loop. Next, `alpha[i]` is changed by the same amount as `alpha[j]` but in the opposite direction ❹. After you optimize `alpha[i]` and `alpha[j]`, you set the constant term `b` for these two alphas ❺.

Finally, you've finished the optimization, and you need to take care to make sure you exit the loops properly. If you've reached the bottom of the `for` loop without hitting a `continue` statement, then you've successfully changed a pair of alphas and you can increment `alphaPairsChanged`. Outside the `for` loop, you check to see if any alphas have been updated; if so you set `iter` to 0 and continue. You'll only stop and exit the `while` loop when you've gone through the entire dataset `maxIter` number of times without anything changing.

To see this in action, type in the following:

```
>>> b,alphas = svmMLiA.smoSimple(dataArr, labelArr, 0.6, 0.001, 40)
The output should look something like this:
iteration number: 29
j not moving enough
iteration number: 30
iter: 30 i:17, pairs changed 1
j not moving enough
iteration number: 0
j not moving enough
iteration number: 1
```

This will take a few minutes to converge. Once it's done, you can inspect the results:

```
>>> b
matrix([[-3.84064413]])
```

You can look at the alphas matrix by itself, but there'll be a lot of 0 elements inside. To see the number of elements greater than 0, type in the following:

```
>>> alphas[alphas>0]
matrix([[ 0.12735413,  0.24154794,  0.36890208]])
```

Your results may differ from these because of the random nature of the SMO algorithm. The command `alphas[alphas>0]` is an example of *array filtering*, which is specific to NumPy and won't work with a regular list in Python. If you type in `alphas>0`, you'll get a Boolean array with a true in every case where the inequality holds. Then, applying this Boolean array back to the original matrix will give you a NumPy matrix with only the values that are greater than 0.

To get the number of support vectors, type

```
>>> shape(alphas[alphas>0])
To see which points of our dataset are support vectors, type
>>> for i in range(100):
...     if alphas[i]>0.0: print dataArr[i],labelArr[i]
```

You should see something like the following:

```
...
[4.6581910000000004, 3.507396] -1.0
[3.4570959999999999, -0.082215999999999997] -1.0
[6.0805730000000002, 0.41888599999999998] 1.0
```

The original dataset with these points circled is shown in figure 6.4.

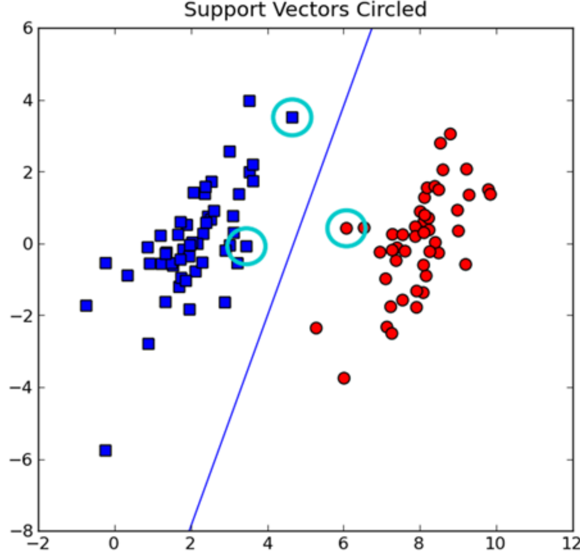

Figure 6.4 SMO sample dataset showing the support vectors circled and the separating hyperplane after the simplified SMO is run on the data

Using the previous settings, I ran this 10 times and took the average time. On my humble laptop this was 14.5 seconds. This wasn't bad, but this is a small dataset with only 100 points. On larger datasets, this would take a long time to converge. In the next section we're going speed this up by building the full SMO algorithm.

Speeding up optimization with the full Platt SMO

The simplified SMO works OK on small datasets with a few hundred points but slows down on larger datasets. Now that we've covered the simplified version, we can move on to the full Platt version of the SMO algorithm. The optimization portion where we change alphas and do all the algebra stays the same. The only difference is how we select which alpha to use in the optimization. The full Platt uses some heuristics that increase the speed. Perhaps in the previous section when executing the example you saw some room for improvement.

The Platt SMO algorithm has an outer loop for choosing the first alpha. This alternates between single passes over the entire dataset and single passes over non-bound alphas. The non-bound alphas are alphas that aren't bound at the limits 0 or C. The pass over the entire dataset is easy, and to loop over the non-bound alphas we'll first create a list of these alphas and then loop over the list. This step skips alphas that we know can't change.

The second alpha is chosen using an inner loop after we've selected the first alpha. This alpha is chosen in a way that will maximize the step size during optimization. In the simplified SMO, we calculated the error `Ej` after choosing `j`. This time, we're going to create a global cache of error values and choose from the alphas that maximize step size, or `Ei-Ej`.

Before we get into the improvements, we're going to need to clean up the code from the previous section. The following listing has a data structure we'll use to clean up the code and three helper functions for caching the E values. Open your text editor and enter the following code.

Listing 6.3 Support functions for full Platt SMO

```
class optStruct:
    def __init__(self,dataMatIn, classLabels, C, toler):
        self.X = dataMatIn
        self.labelMat = classLabels
        self.C = C
        self.tol = toler
        self.m = shape(dataMatIn)[0]
        self.alphas = mat(zeros((self.m,1)))
        self.b = 0
        self.eCache = mat(zeros((self.m,2)))          # 1 Error cache

def calcEk(oS, k):
    fXk = float(multiply(oS.alphas,oS.labelMat).T*\
          (oS.X*oS.X[k,:].T)) + oS.b
    Ek = fXk - float(oS.labelMat[k])
    return Ek

def selectJ(i, oS, Ei):                              # 2 Inner-loop heuristic
    maxK = -1; maxDeltaE = 0; Ej = 0
    oS.eCache[i] = [1,Ei]
    validEcacheList = nonzero(oS.eCache[:,0].A)[0]
    if (len(validEcacheList)) > 1:
        for k in validEcacheList:
            if k == i: continue
            Ek = calcEk(oS, k)
            deltaE = abs(Ei - Ek)
            if (deltaE > maxDeltaE):
                maxK = k; maxDeltaE = deltaE; Ej = Ek   # 3 Choose j for maximum step size
        return maxK, Ej
    else:
        j = selectJrand(i, oS.m)
        Ej = calcEk(oS, j)
    return j, Ej

def updateEk(oS, k):
    Ek = calcEk(oS, k)
    oS.eCache[k] = [1,Ek]
```

The first thing you do is create a data structure to hold all of the important values. This is done with an object. You don't use it for object-oriented programming; it's used as a data structure in this example. I moved all the data into a structure to save typing when you pass values into functions. You can now pass in one object. I could have done this just as easily with a Python dictionary, but that takes more work trying to access member variables; compare `myObject.X` to `myObject['X']`. To accomplish this, you create the class `optStruct`, which only has the `init` method. In this method, you populate the member variables. All of these are the same as in the simplified SMO

code, but you've added the member variable eCache, which is an mx2 matrix (1). The first column is a flag bit stating whether the eCache is valid, and the second column is the actual E value.

The first helper function, calcEk(), calculates an E value for a given alpha and returns the E value. This was previously done inline, but you must take it out because it occurs more frequently in this version of the SMO algorithm.

The next function, selectJ(), selects the second alpha, or the inner loop alpha (2). Recall that the goal is to choose the second alpha so that we'll take the maximum step during each optimization. This function takes the error value associated with the first choice alpha (Ei) and the index i. You first set the input Ei to valid in the cache. *Valid* means that it has been calculated. The code nonzero(oS.eCache[:,0].A)[0] creates a list of nonzero values in the eCache. The NumPy function nonzero() returns a list containing indices of the input list that are—you guessed it—not zero. The nonzero() statement returns the alphas corresponding to non-zero E values, not the E values. You loop through all of these values and choose the value that gives you a maximum change (3). If this is your first time through the loop, you randomly select an alpha. There are more sophisticated ways of handling the first-time case, but this works for our purposes.

The last helper function in listing 6.3 is updateEk(). This calculates the error and puts it in the cache. You'll use this after you optimize alpha values.

The code in listing 6.3 doesn't do much on its own. But when combined with the optimization and the outer loop, it forms the powerful SMO algorithm.

Next, I'll briefly present the optimization routine, to find our decision boundary. Open your text editor and add the code from the next listing. You've already seen this code in a different format.

Listing 6.4 Full Platt SMO optimization routine

```
def innerL(i, oS):                                    Second-choice heuristic (1)
    Ei = calcEk(oS, i)
    if ((oS.labelMat[i]*Ei < -oS.tol) and (oS.alphas[i] < oS.C)) or\
        ((oS.labelMat[i]*Ei > oS.tol) and (oS.alphas[i] > 0)):
        j,Ej = selectJ(i, oS, Ei)
        alphaIold = oS.alphas[i].copy(); alphaJold = oS.alphas[j].copy();
        if (oS.labelMat[i] != oS.labelMat[j]):
            L = max(0, oS.alphas[j] - oS.alphas[i])
            H = min(oS.C, oS.C + oS.alphas[j] - oS.alphas[i])
        else:
            L = max(0, oS.alphas[j] + oS.alphas[i] - oS.C)
            H = min(oS.C, oS.alphas[j] + oS.alphas[i])
        if L==H: print "L==H"; return 0
        eta = 2.0 * oS.X[i,:]*oS.X[j,:].T - oS.X[i,:]*oS.X[i,:].T - \
              oS.X[j,:]*oS.X[j,:].T
        if eta >= 0: print "eta>=0"; return 0
        oS.alphas[j] -= oS.labelMat[j]*(Ei - Ej)/eta
        oS.alphas[j] = clipAlpha(oS.alphas[j],H,L)           (2) Updates
        updateEk(oS, j)                                          Ecache
```

```
        if (abs(oS.alphas[j] - alphaJold) < 0.00001):
            print "j not moving enough"; return 0
        oS.alphas[i] += oS.labelMat[j]*oS.labelMat[i]*\
                        (alphaJold - oS.alphas[j])
        updateEk(oS, i)                                              ❷ Updates Ecache
        b1 = oS.b - Ei- oS.labelMat[i]*(oS.alphas[i]-alphaIold)*\
             oS.X[i,:]*oS.X[i,:].T - oS.labelMat[j]*\
             (oS.alphas[j]-alphaJold)*oS.X[i,:]*oS.X[j,:].T
        b2 = oS.b - Ej- oS.labelMat[i]*(oS.alphas[i]-alphaIold)*\
             oS.X[i,:]*oS.X[j,:].T - oS.labelMat[j]*\
             (oS.alphas[j]-alphaJold)*oS.X[j,:]*oS.X[j,:].T
        if (0 < oS.alphas[i]) and (oS.C > oS.alphas[i]): oS.b = b1
        elif (0 < oS.alphas[j]) and (oS.C > oS.alphas[j]): oS.b = b2
        else: oS.b = (b1 + b2)/2.0
        return 1
    else: return 0
```

The code in listing 6.4 is almost the same as the smoSimple() function given in listing 6.2. But it has been written to use our data structure. The structure is passed in as the parameter oS. The second important change is that selectJ() from listing 6.3 is used to select the second alpha rather than selectJrand() ❶. Lastly ❷, you update the Ecache after alpha values change. The final piece of code that wraps all of this up is shown in the following listing. This is the outer loop where you select the first alpha. Open your text editor and add the code from this listing to svmMLiA.py.

Listing 6.5 Full Platt SMO outer loop

```
def smoP(dataMatIn, classLabels, C, toler, maxIter, kTup=('lin', 0)):
    oS = optStruct(mat(dataMatIn),mat(classLabels).transpose(),C,toler)
    iter = 0
    entireSet = True; alphaPairsChanged = 0
    while (iter < maxIter) and ((alphaPairsChanged > 0) or (entireSet)):
        alphaPairsChanged = 0
        if entireSet:                                                ❶ Go over all values
            for i in range(oS.m):
                alphaPairsChanged += innerL(i,oS)
            print "fullSet, iter: %d i:%d, pairs changed %d" %\
    (iter,i,alphaPairsChanged)
            iter += 1
        else:                                                        ❷ Go over non-bound values
            nonBoundIs = nonzero((oS.alphas.A > 0) * (oS.alphas.A < C))[0]
            for i in nonBoundIs:
                alphaPairsChanged += innerL(i,oS)
                print "non-bound, iter: %d i:%d, pairs changed %d" % \
                (iter,i,alphaPairsChanged)
            iter += 1
        if entireSet: entireSet = False
        elif (alphaPairsChanged == 0): entireSet = True
        print "iteration number: %d" % iter
    return oS.b,oS.alphas
```

The code in listing 6.5 is the full Platt SMO algorithm. The inputs are the same as the function smoSimple(). Initially you create the data structure that will be used to hold

all of your data. Next, you initialize some variables you'll use to control when you exit the function. The majority of the code is in the while loop, similar to smoSimple() but with a few more exit conditions. You'll exit from the loop whenever the number of iterations exceeds your specified maximum or you pass through the entire set without changing any alpha pairs. The maxIter variable has a different use from smoSimple() because in that function you counted an iteration as a pass through the entire set when no alphas were changed. In this function an iteration is defined as one pass through the loop regardless of what was done. This method is superior to the counting used in smoSimple() because it will stop if there are any oscillations in the optimization.

Inside the while loop is different from smoSimple(). The first for loop goes over any alphas in the dataset ❶. We call innerL() to choose a second alpha and do optimization if possible. A 1 will be returned if any pairs get changed. The second for loop goes over all the non-bound alphas, the values that aren't bound at 0 or C. ❷

You next toggle the for loop to switch between the non-bound loop and the full pass, and print out the iteration number. Finally, the constant b and the alphas are returned.

To see this in action, type the following in your Python shell:

```
>>> dataArr,labelArr = svmMLiA.loadDataSet('testSet.txt')
>>> b,alphas = svmMLiA.smoP(dataArr, labelArr, 0.6, 0.001, 40)
non-bound, iter: 2 i:54, pairs changed 0
non-bound, iter: 2 i:55, pairs changed 0
iteration number: 3
fullSet, iter: 3 i:0, pairs changed 0
fullSet, iter: 3 i:1, pairs changed 0
fullSet, iter: 3 i:2, pairs changed 0
```

You can inspect b and alphas similarly to what you did here. Was this method faster? On my humble laptop I did this algorithm with the settings listed previously 10 times and took the average. The average time on my machine was 0.78 seconds. Compare this to smoSimple()on the same dataset, which took an average of 14.5 seconds. The results will be even better on larger datasets, and there are many ways to make this even faster.

What happens if you change the tolerance value? How about if you change the value of C? I mentioned briefly at the end of section 6.2 that the constant C gives weight to different parts of the optimization problem. C controls the balance between making sure all of the examples have a margin of at least 1.0 and making the margin as wide as possible. If C is large, the classifier will try to make all of the examples properly classified by the separating hyperplane. The results from this optimization run are shown in figure 6.5. Comparing figure 6.5 to 6.4 you see that there are more support vectors in figure 6.5. If you recall, figure 6.4 was generated by our simplified algorithm, which randomly picked pairs of alphas. This method worked, but it wasn't as good as the full version of the algorithm, which covered the entire dataset. You may also think that the support vectors chosen should always be closest to the separating hyperplane. Given the settings we have for C, the support vectors circled give us a

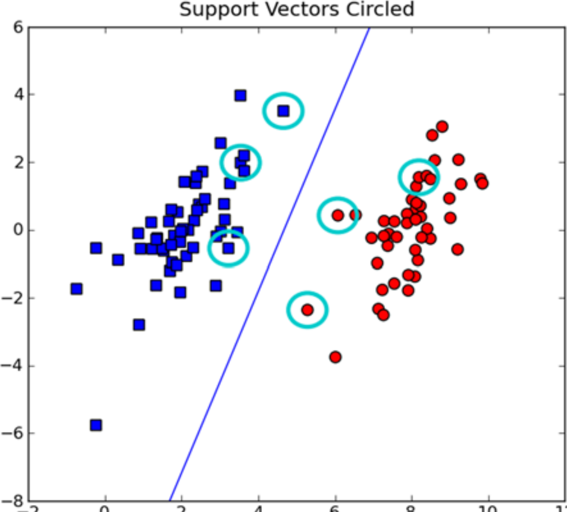

Figure 6.5 Support vectors shown after the full SMO algorithm is run on the dataset. The results are slightly different from those in figure 6.4.

solution that satisfies the algorithm. When you have a dataset that isn't linearly separable, you'll see the support vectors bunch up closer to the hyperplane.

You might be thinking, "We just spent a lot of time figuring out the alphas, but how do we use this to classify things?" That's not a problem. You first need to get the hyperplane from the alphas. This involves calculating ws. The small function listed here will do that for you:

```
def calcWs(alphas,dataArr,classLabels):
    X = mat(dataArr); labelMat = mat(classLabels).transpose()
    m,n = shape(X)
    w = zeros((n,1))
    for i in range(m):
        w += multiply(alphas[i]*labelMat[i],X[i,:].T)
    return w
```

The most important part of the code is the `for` loop, which just multiplies some things together. If you looked at any of the alphas we calculated earlier, remember that most of the alphas are 0s. The non-zero alphas are our support vectors. This `for` loop goes over all the pieces of data in our dataset, but only the support vectors matter. You could just as easily throw out those other data points because they don't contribute to the w calculations.

To use the function listed previously, type in the following:

```
>>> ws=svmMLiA.calcWs(alphas,dataArr,labelArr)
>>> ws
array([[ 0.65307162],
       [-0.17196128]])
```

Now to classify something, say the first data point, type in this:

```
>>> datMat=mat(dataArr)
>>> datMat[0]*mat(ws)+b
matrix([[-0.92555695]])
```

If this value is greater than 0, then its class is a 1, and the class is -1 if it's less than 0. For point 0 you should then have a label of -1. Check to make sure this is true:

```
>>> labelArr[0]
-1.0
```

Now check to make sure other pieces of data are properly classified:

```
>>> datMat[2]*mat(ws)+b
matrix([[ 2.30436336]])
>>> labelArr[2]
1.0
>>> datMat[1]*mat(ws)+b
matrix([[-1.36706674]])
>>> labelArr[1]
-1.0
```

Compare these results to figure 6.5 to make sure it makes sense.

Now that we can successfully train our classifier, I'd like to point out that the two classes fit on either side of a straight line. If you look at figure 6.1, you can probably find shapes that would separate the two classes. What if you want your classes to be inside a circle or outside a circle? We'll next talk about a way you can change the classifier to account for different shapes of regions separating your data.

Using kernels for more complex data

Consider the data in figure 6.6. This is similar to the data in figure 6.1, frame C. Earlier, this was used to describe data that isn't linearly separable. Clearly there's some pattern in this data that we can recognize. Is there a way we can use our powerful tools to capture this pattern in the same way we did for the linear data? Yes, there is. We're going to use something called a *kernel* to transform our data into a form that's easily understood by our classifier. This section will explain kernels and how we can use them to support vector machines. Next, you'll see one popular type of kernel called the *radial bias function,* and finally we'll apply this to our existing classifier.

1 Mapping data to higher dimensions with kernels

The points in figure 6.1 are in a circle. The human brain can recognize that. Our classifier, on the other hand, can only recognize greater than or less than 0. If we just plugged in our X and Y coordinates, we wouldn't get good results. You can probably think of some ways to change the circle data so that instead of X and Y, you'd have some new variables that would be better on the greater-than- or less-than-0 test. This is an example of transforming the data from one feature space to another so that you can deal with it easily with your existing tools. Mathematicians like to call this *mapping from one feature space to another feature space.* Usually, this mapping goes from a lower-dimensional feature space to a higher-dimensional space.

This mapping from one feature space to another is done by a *kernel.* You can think of the kernel as a wrapper or interface for the data to translate it from a difficult formatting to an easier formatting. If this mapping from a feature space to another feature

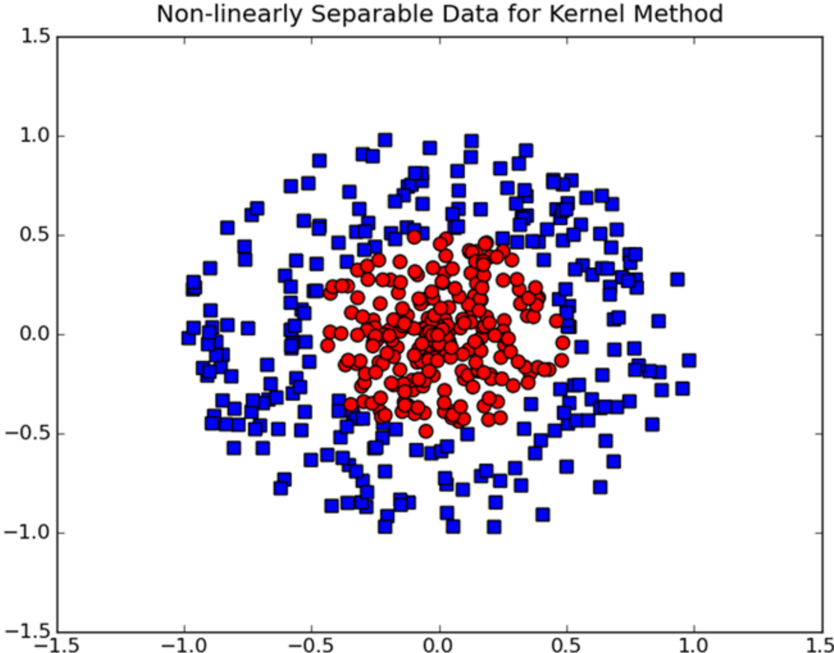

Figure 6.6 **This data can't be easily separated with a straight line in two dimensions, but it's obvious that some pattern exists separating the squares and the circles.**

space sounds confusing, you can think of it as another distance metric. Earlier we encountered distance metrics. There were many different ways to measure the distance, and the same is true with kernels, as you'll see soon. After making the substitution, we can go about solving this linear problem in high-dimensional space, which is equivalent to solving a nonlinear problem in low-dimensional space.

One great thing about the SVM optimization is that all operations can be written in terms of *inner products*. Inner products are two vectors multiplied together to yield a scalar or single number. We can replace the inner products with our kernel functions without making simplifications. Replacing the inner product with a kernel is known as the *kernel trick* or *kernel substation.*

Kernels aren't unique to support vector machines. A number of other machine-learning algorithms can use kernels. A popular kernel is the radial bias function, which we'll introduce next.

5.2 *The radial bias function as a kernel*

The *radial bias function* is a kernel that's often used with support vector machines. A radial bias function is a function that takes a vector and outputs a scalar based on the vector's distance. This distance can be either from 0,0 or from another vector. We'll use the Gaussian version, which can be written as

$$k(x,y) = exp\left(\frac{-\|x-y\|^2}{2\sigma^2}\right)$$

where σ is a user-defined parameter that determines the "reach," or how quickly this falls off to 0.

This Gaussian version maps the data from its feature space to a higher feature space, infinite dimensional to be specific, but don't worry about that for now. This is a common kernel to use because you don't have to figure out exactly how your data behaves, and you'll get good results with this kernel. In our example we have data that's basically in a circle; we could have looked over the data and realized we only needed to measure the distance to the origin; however, if we encounter a new dataset that isn't in that format, then we're in big trouble. We'll get great results with this Gaussian kernel, and we can use it on many other datasets and get low error rates there too.

If you add one function to our svmMLiA.py file and make a few modifications, you'll be able to use kernels with our existing code. Open your svmMLiA.py code and enter the function `kernelTrans()`. Also, modify our class, `optStruct`, so that it looks like the code given in the following listing.

Listing 6.6 Kernel transformation function

```
def kernelTrans(X, A, kTup):
    m,n = shape(X)
    K = mat(zeros((m,1)))
    if kTup[0]=='lin': K = X * A.T
    elif kTup[0]=='rbf':
        for j in range(m):
            deltaRow = X[j,:] - A
            K[j] = deltaRow*deltaRow.T
        K = exp(K /(-1*kTup[1]**2))            <- 1 Element-wise division
    else: raise NameError('Houston We Have a Problem -- \
    That Kernel is not recognized')
    return K

class optStruct:
    def __init__(self,dataMatIn, classLabels, C, toler, kTup):
        self.X = dataMatIn
        self.labelMat = classLabels
        self.C = C
        self.tol = toler
        self.m = shape(dataMatIn)[0]
        self.alphas = mat(zeros((self.m,1)))
        self.b = 0
        self.eCache = mat(zeros((self.m,2)))
        self.K = mat(zeros((self.m,self.m)))
        for i in range(self.m):
            self.K[:,i] = kernelTrans(self.X, self.X[i,:], kTup)
```

I think it's best to look at our new version of `optStruct`. This has everything the same as the previous `optStruct` with one new input: `kTup`. This `kTup` is a generic tuple that

contains the information about the kernel. You'll see this in action in a little bit. At the end of the initialization method a matrix `K` gets created and then populated by calling a function `kernelTrans()`. This global `K` gets calculated once. Then, when you want to use the kernel, you call it. This saves some redundant computations as well.

When the matrix `K` is being computed, the function `kernelTrans()` is called multiple times. This takes three inputs: two numeric types and a tuple. The tuple `kTup` holds information about the kernel. The first argument in the tuple is a string describing what type of kernel should be used. The other arguments are optional arguments that may be needed for a kernel. The function first creates a column vector and then checks the tuple to see which type of kernel is being evaluated. Here, only two choices are given, but you can expand this to many more by adding in other `elif` statements.

In the case of the linear kernel, a dot product is taken between the two inputs, which are the full dataset and a row of the dataset. In the case of the radial bias function, the Gaussian function is evaluated for every element in the matrix in the `for` loop. After the `for` loop is finished, you apply the calculations over the entire vector. It's worth mentioning that in NumPy matrices the division symbol means element-wise rather than taking the inverse of a matrix, as would happen in MATLAB. ❶

Lastly, you raise an exception if you encounter a tuple you don't recognize. This is important because you don't want the program to continue in this case.

Code was changed to use the kernel functions in two preexisting functions: `innerL()` and `calcEk()`. The changes are shown in listing 6.7. I hate to list them out like this. But relisting the entire functions would take over 90 lines, and I don't think anyone would be happy with that. You can copy the code from the source code download to get these changes without manually adding them. Here are the changes:

Listing 6.7 Changes to innerL() and calcEk() needed to user kernels

```
innerL():
                              .
                              .
                              .
eta = 2.0 * oS.K[i,j] - oS.K[i,i] - oS.K[j,j]
                              .
                              .
                              .
b1 = oS.b - Ei- oS.labelMat[i]*(oS.alphas[i]-alphaIold)*oS.K[i,i] -\
                   oS.labelMat[j]*(oS.alphas[j]-alphaJold)*oS.K[i,j]
b2 = oS.b - Ej- oS.labelMat[i]*(oS.alphas[i]-alphaIold)*oS.K[i,j]-\
                   oS.labelMat[j]*(oS.alphas[j]-alphaJold)*oS.K[j,j]
                             .
                             .
                             .

def calcEk(oS, k):
    fXk = float(multiply(oS.alphas,oS.labelMat).T*oS.K[:,k] + oS.b)
    Ek = fXk - float(oS.labelMat[k])
    return Ek
```

Now that you see how to apply a kernel during training, let's see how you'd use it during testing.

3 Using a kernel for testing

We're going to create a classifier that can properly classify the data points in figure 6.6. We'll create a classifier that uses the radial bias kernel. The function earlier had one user-defined input: σ. We need to figure out how big to make this. We'll create a function to train and test the classifier using the kernel. The function is shown in the following listing. Open your text editor and add in the function `testRbf()`.

Listing 6.8 Radial bias test function for classifying with a kernel

```
def testRbf(k1=1.3):
    dataArr,labelArr = loadDataSet('testSetRBF.txt')
    b,alphas = smoP(dataArr, labelArr, 200, 0.0001, 10000, ('rbf', k1))
    datMat=mat(dataArr); labelMat = mat(labelArr).transpose()
    svInd=nonzero(alphas.A>0)[0]
    sVs=datMat[svInd]                                          <- Create matrix of
    labelSV = labelMat[svInd];                                    support vectors
    print "there are %d Support Vectors" % shape(sVs)[0]
    m,n = shape(datMat)
    errorCount = 0
    for i in range(m):
        kernelEval = kernelTrans(sVs,datMat[i,:],('rbf', k1))
        predict=kernelEval.T * multiply(labelSV,alphas[svInd]) + b
        if sign(predict)!=sign(labelArr[i]): errorCount += 1
    print "the training error rate is: %f" % (float(errorCount)/m)
    dataArr,labelArr = loadDataSet('testSetRBF2.txt')
    errorCount = 0
    datMat=mat(dataArr); labelMat = mat(labelArr).transpose()
    m,n = shape(datMat)
    for i in range(m):
        kernelEval = kernelTrans(sVs,datMat[i,:],('rbf', k1))
        predict=kernelEval.T * multiply(labelSV,alphas[svInd]) + b
        if sign(predict)!=sign(labelArr[i]): errorCount += 1
    print "the test error rate is: %f" % (float(errorCount)/m)
```

This code only has one input, and that's optional. The input is the user-defined variable for the Gaussian radial bias function. The code is mostly a collection of stuff you've done before. The dataset is loaded from a file. Then, you run the Platt SMO algorithm on this, with the option `'rbf'` for a kernel.

After the optimization finishes, you make matrix copies of the data to use in matrix math later, and you find the non-zero alphas, which are our support vectors. You also take the labels corresponding to the support vectors and the alphas. Those are the only values you'll need to do classification.

The most important lines in this whole listing are the first two lines in the `for` loops. These show how to classify with a kernel. You first use the `kernelTrans()` function you used in the structure initialization method. After you get the transformed data, you do a multiplication with the alphas and the labels. The other important

thing to note in these lines is how you use only the data for the support vectors. The rest of the data can be tossed out.

The second `for loop` is a repeat of the first one but with a different dataset—the test dataset. You now can compare how different settings perform on the test set and the training set.

To test out the code from listing 6.8, enter the following at the Python shell:

```
>>> reload(svmMLiA)
<module 'svmMLiA' from 'svmMLiA.pyc'>
>>> svmMLiA.testRbf()
                    .
                    .
fullSet, iter: 11 i:497, pairs changed 0
fullSet, iter: 11 i:498, pairs changed 0
fullSet, iter: 11 i:499, pairs changed 0
iteration number: 12
there are 27 Support Vectors
the training error rate is: 0.030000
the test error rate is: 0.040000
```

You can play around with the `k1` parameter to see how the test error, training error, and number of support vectors change with `k1`. The first example with sigma very small (0.1) is shown in figure 6.7.

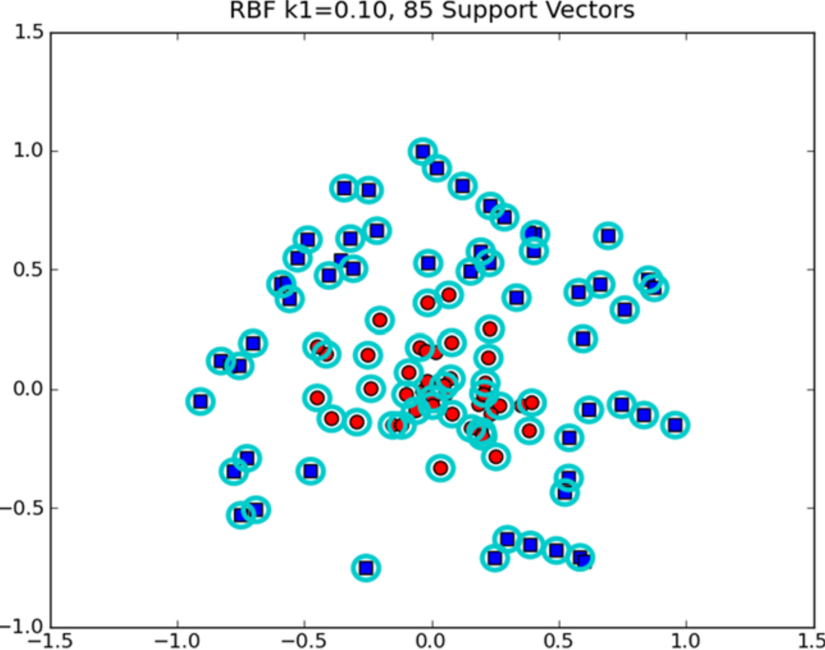

Figure 6.7 Radial bias function with the user-defined parameter `k1=0.1`. The user-defined parameter reduces the influence of each support vector, so you need more support vectors.

In figure 6.7 we have 100 data points, and 85 of them are support vectors. The optimization algorithm found it needed these points in order to properly classify the data. This should give you the intuition that the reach of the radial bias is too small. You can increase sigma and see how the error rate changes. I increased sigma and made another plot, shown in figure 6.8.

Compare figure 6.8 with figure 6.7. Now we have only 27 support vectors. This is much smaller. If you watch the output of the function `testRbf()`, you'll see that the test error has gone down too. This dataset has an optimum somewhere around this setting. If you make the sigma smaller, you'll get a lower training error but a higher testing error.

There is an optimum number of support vectors. The beauty of SVMs is that they classify things efficiently. If you have too few support vectors, you may have a poor decision boundary (this will be demonstrated in the next example). If you have too many support vectors, you're using the whole dataset every time you classify something—that's called k-Nearest Neighbors.

Feel free to play around with other settings in the SMO algorithm or to create new kernels. We're now going to put our support vector machines to use with some larger data and compare it with a classifier you saw earlier.

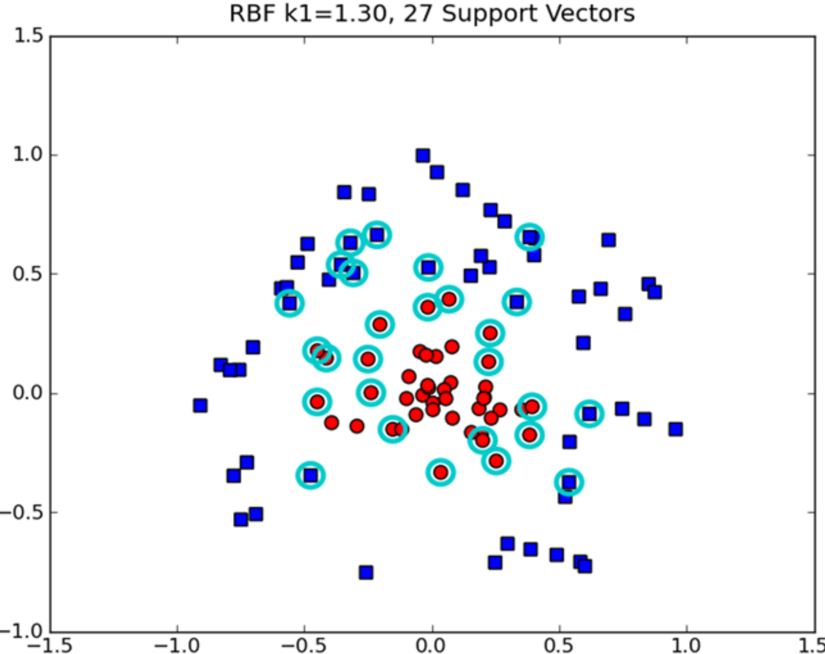

Figure 6.8 Radial bias kernel function with user parameter `k1=1.3`. Here we have fewer support vectors than in figure 6.7. The support vectors are bunching up around the decision boundary.

6.6 Example: revisiting handwriting classification

Consider the following hypothetical situation. Your manager comes to you and says, "That handwriting recognition program you made is great, but it takes up too much memory and customers can't download our application over the air. (At the time of writing there's a 10 MB limit on certain applications downloaded over the air. I'm sure this will be laughable at some point in the future.) We need you to keep the same performance with less memory used. I told the CEO you'd have this ready in a week. How long will it take?" I'm not sure how you'd respond, but if you wanted to comply with their request, you could consider using support vector machines. The k-Nearest Neighbors algorithm used in chapter 2 works well, but you have to carry around all the training examples. With support vector machines, you can carry around far fewer examples (only your support vectors) and achieve comparable performance.

Example: digit recognition with SVMs

1. Collect: Text file provided.
2. Prepare: Create vectors from the binary images.
3. Analyze: Visually inspect the image vectors.
4. Train: Run the SMO algorithm with two different kernels and different settings for the radial bias kernel.
5. Test: Write a function to test the different kernels and calculate the error rate.
6. Use: A full application of image recognition requires some image processing, which we won't get into.

Using some of the code from chapter 2 and the SMO algorithm, let's build a system to test a classifier on the handwritten digits. Open svmMLiA.py and copy over the function `img2vector()` from knn.py in chapter 2. Then, add the code in the following listing.

Listing 6.9 Support vector machine handwriting recognition

```
def loadImages(dirName):
    from os import listdir
    hwLabels = []
    trainingFileList = listdir(dirName)
    m = len(trainingFileList)
    trainingMat = zeros((m,1024))
    for i in range(m):
        fileNameStr = trainingFileList[i]
        fileStr = fileNameStr.split('.')[0]
        classNumStr = int(fileStr.split('_')[0])
        if classNumStr == 9: hwLabels.append(-1)
        else: hwLabels.append(1)
        trainingMat[i,:] = img2vector('%s/%s' % (dirName, fileNameStr))
    return trainingMat, hwLabels
```

```
def testDigits(kTup=('rbf', 10)):
    dataArr,labelArr = loadImages('trainingDigits')
    b,alphas = smoP(dataArr, labelArr, 200, 0.0001, 10000, kTup)
    datMat=mat(dataArr); labelMat = mat(labelArr).transpose()
    svInd=nonzero(alphas.A>0)[0]
    sVs=datMat[svInd]
    labelSV = labelMat[svInd];
    print "there are %d Support Vectors" % shape(sVs)[0]
    m,n = shape(datMat)
    errorCount = 0
    for i in range(m):
        kernelEval = kernelTrans(sVs,datMat[i,:],kTup)
        predict=kernelEval.T * multiply(labelSV,alphas[svInd]) + b
        if sign(predict)!=sign(labelArr[i]): errorCount += 1
    print "the training error rate is: %f" % (float(errorCount)/m)
    dataArr,labelArr = loadImages('testDigits')
    errorCount = 0
    datMat=mat(dataArr); labelMat = mat(labelArr).transpose()
    m,n = shape(datMat)
    for i in range(m):
        kernelEval = kernelTrans(sVs,datMat[i,:],kTup)
        predict=kernelEval.T * multiply(labelSV,alphas[svInd]) + b
        if sign(predict)!=sign(labelArr[i]): errorCount += 1
    print "the test error rate is: %f" % (float(errorCount)/m)
```

The function `loadImages()` appeared as part of `handwritingClassTest()` earlier in kNN.py. It has been refactored into its own function. The only big difference is that in kNN.py this code directly applied the class label. But with support vector machines, you need a class label of -1 or +1, so if you encounter a 9 it becomes -1; otherwise, the label is +1. Actually, support vector machines are only a binary classifier. They can only choose between +1 and -1. Creating a multiclass classifier with SVMs has been studied and compared. If you're interested, I suggest you read a paper called "A Comparison of Methods for Multiclass Support Vector Machines" by C. W. Hus et al.[4] Because we're doing binary classification, I've taken out all of the data except the 1 and 9 digits.

The next function, `testDigits()`, isn't super new. It's almost the exact same code as `testRbf()`, except it calls `loadImages()` to get the class labels and data. The other small difference is that the kernel tuple `kTup` is now an input, whereas it was assumed that you were using the rbf kernel in `testRbf()`. If you don't add any input arguments to `testDigits()`, it will use the default of ('rbf', 10) for `kTup`.

After you've entered the code from listing 6.9, save svmMLiA.py and type in the following:

```
>>> svmMLiA.testDigits(('rbf', 20))
                    .
                    .
L==H
fullSet, iter: 3 i:401, pairs changed 0
iteration number: 4
```

W. Hus, and C. J. Lin, "A Comparison of Methods for Multiclass Support Vector Machines," *IEEE Transac-*
ns on Neural Networks 13, no. 2 (March 2002), 415–25.

```
there are 43 Support Vectors
the training error rate is: 0.017413
the test error rate is: 0.032258
```

I tried different values for sigma as well as trying the linear kernel and summarized them in table 6.1.

Table 6.1 Handwritten digit performance for different kernel settings

Kernel, settings	Training error (%)	Test error (%)	# Support vectors
RBF, 0.1	0	52	402
RBF, 5	0	3.2	402
RBF, 10	0	0.5	99
RBF, 50	0.2	2.2	41
RBF, 100	4.5	4.3	26
Linear	2.7	2.2	38

The results in table 6.1 show that we achieve a minimum test error with the radial bias function kernel somewhere around 10. This is much larger than our previous example, where our minimum test error was roughly 1.3. Why is there such a huge difference? The data is different. In the handwriting data, we have 1,024 features that could be as high as 1.0. In the example in section 6.5, our data varied from -1 to 1, but we had only two features. How can you tell what settings to use? To be honest, I didn't know when I was writing this example. I just tried some different settings. The answer is also sensitive to the settings of C. There are other formulations of the SVM that bring C into the optimization procedure, such as v-SVM. A good discussion about v-SVM can be found in chapter 3 of *Pattern Recognition,* by Sergios Theodoridis and Konstantinos Koutroumbas.[5]

It's interesting to note that the minimum training error doesn't correspond to a minimum number of support vectors. Also note that the linear kernel doesn't have terrible performance. It may be acceptable to trade the linear kernel's error rate for increased speed of classification, but that depends on your application.

.7 Summary

Support vector machines are a type of classifier. They're called machines because they generate a binary decision; they're decision machines. Support vectors have good generalization error: they do a good job of learning and generalizing on what they've learned. These benefits have made support vector machines popular, and they're considered by some to be the best stock algorithm in unsupervised learning.

Sergios Theodoridis and Konstantinos Koutroumbas, *Pattern Recognition,* 4th ed. (Academic Press, 2009), 133.

Support vector machines try to maximize margin by solving a quadratic optimization problem. In the past, complex, slow quadratic solvers were used to train support vector machines. John Platt introduced the SMO algorithm, which allowed fast training of SVMs by optimizing only two alphas at one time. We discussed the SMO optimization procedure first in a simplified version. We sped up the SMO algorithm a lot by using the full Platt version over the simplified version. There are many further improvements that you could make to speed it up even further. A commonly cited reference for further speed-up is the paper titled "Improvements to Platt's SMO Algorithm for SVM Classifier Design."[6]

Kernel methods, or the kernel trick, map data (sometimes nonlinear data) from a low-dimensional space to a high-dimensional space. In a higher dimension, you can solve a linear problem that's nonlinear in lower-dimensional space. Kernel methods can be used in other algorithms than just SVM. The radial-bias function is a popular kernel that measures the distance between two vectors.

Support vector machines are a binary classifier and additional methods can be extended to classification of classes greater than two. The performance of an SVM is also sensitive to optimization parameters and parameters of the kernel used.

Our next chapter will wrap up our coverage of classification by focusing on something called *boosting*. A number of similarities can be drawn between boosting and support vector machines, as you'll soon see.

S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy, "Improvements to Platt's SMO Algorithm r SVM Classifier Design," *Neural Computation* 13, no. 3,(2001), 637–49.