


```

alphas = mat(zeros((m,1)))
iter = 0
while (iter < maxIter):
    alphaPairsChanged = 0
    for i in range(m):
        fx_i = float(multiply(alphas,labelMat).T*\
                     (dataMatrix*dataMatrix[i,:].T)) + b
        Ei = fx_i - 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)
            fx_j = float(multiply(alphas,labelMat).T*\
                         (dataMatrix*dataMatrix[j,:].T)) + b
            Ej = fx_j - float(labelMat[j])
            alphaOld = alphas[i].copy();
            alphaOld = 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] - alphaOld) < 0.00001): print \
                "j not moving enough"; continue
            alphas[i] += labelMat[j]*labelMat[i]*\
                (alphaOld - alphas[j])
            b1 = b - Ei - labelMat[i]*(alphas[i]-alphaOld)*\
                  dataMatrix[i,:]*dataMatrix[i,:].T - \
                  labelMat[j]*(alphas[j]-alphaOld)*\
                  dataMatrix[i,:]*dataMatrix[j,:].T
            b2 = b - Ej - labelMat[i]*(alphas[i]-alphaOld)*\
                  dataMatrix[i,:]*dataMatrix[j,:].T - \
                  labelMat[j]*(alphas[j]-alphaOld)*\
                  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

```

The diagram illustrates the five steps of the SMO algorithm:

- 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 E_i 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 ①. 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 ②. 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 `alphaOld` and `alphaJOld`. Otherwise, when you later compare the new and old values, we won't see the change. You then calculate L and H ③, 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."

η is the optimal amount to change `alpha[j]`. This is calculated in the long line of algebra. If η is 0, you also quit the current iteration of the `for` loop. This step is a simplification of the real SMO algorithm. If η 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 ④. After you optimize `alpha[i]` and `alpha[j]`, you set the constant term `b` for these two alphas ⑤.

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
[3.457095999999999, -0.0822159999999997] -1.0
[6.0805730000000002, 0.4188859999999998] 1.0
```

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

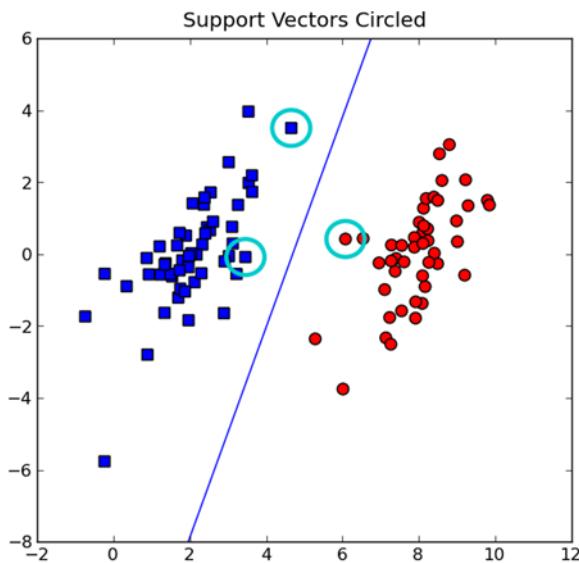


Figure 6.4 SMO sample dataset showing the support vectors circled and the separating hyperplane after the simplified SMO is run on the data

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.

6.4 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 E_j 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 $E_i - E_j$.

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)))  

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):
    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
    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 diagram consists of three numbered callouts pointing to specific parts of the code. Callout 1, labeled 'Error cache', points to the line where the `eCache` matrix is initialized. Callout 2, labeled 'Inner-loop heuristic', points to the `selectJ` function. Callout 3, labeled 'Choose j for maximum step size', points to the inner loop within the `selectJ` function where it iterates over valid cache entries to find the one with the maximum change.

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 $m \times 2$ 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 (E_i) and the index i . You first set the input E_i 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):
    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)
        updateEk(os, j)
```

Second-choice heuristic

①

② **Updates
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)
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

```



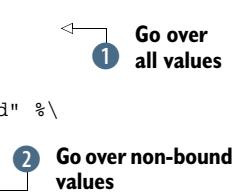
The code in listing 6.4 is almost the same as the `smoSimple()` function given in listing 6.2. But it has been written to use our data structure. The structure is passed in as the parameter `os`. The second important change is that `selectJ()` from listing 6.3 is used to select the second alpha rather than `selectJrand()` ①. Lastly ②, 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:
            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:
            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

