Introduction to support vector machine (SVM)

Introduction

Support vector machines (SVMs) are a type of classifier. They're called machines because they generate a binary decision; they're decision machines. SVMs do a good job of learning and generalizing on what they've learned.

Background

Like previous parts of my series, I have also learned from many sources and you can refer to them in the References section. This is summary of my learning.

Basic concepts (from [8], [9])

There are two groups of data, and data is linearly separable when:

- In one dimension, you can find a point separating the data.
- In two dimensions, you can find a line separating the data.
- In three dimensions, you can find a plane separating the data.

What do we use to separate the data when there are more than three dimensions? We use what is called a **hyperplane**. 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.

The optimization problem

In this article, one single SVM model is for two labels classification, whose label is y ϵ {-1, 1}. And the hyperplane we want to find to separate the two classes dataset is h, for which classifier, we use parameters w, b and we write our classifier as

$$h_{w,b}(x) = g(w^T x + b)$$

Here, $g(z) = 1$ if $z \geq 0$, and $g(z) = -1$ otherwise.

Lets now formalize the margin intuition into notions of the functional and geometric margins. Given a training example $(x^{(i)},y^{(i)})$, we define the **functional margin** of (w, b) with respect to the training example:

$$\hat{\gamma}^{(i)} = y^{(i)}(w^T x + b)$$

Note that if $y^{(i)} = 1$, then for the functional margin to be large (i.e., for our prediction to be confident and correct), we need $w^T x + b$ to be a large positive number. Conversely, if $y^{(i)} = -1$, then for the functional margin to be large, we need $w^T x + b$ to be a large negative 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. Then, when we find the points with the smallest margin, we must maximize that margin.

Given a training set $S = \{(x^{(i)},y^{(i)}); i = 1, \ldots, m\}$, we also define the function margin of (w, b) with respect to S to be the smallest of the functional margins of the individual training examples. This can be written:

$$\hat{\gamma} = \min_{i=1,\ldots,m} \hat{\gamma}^{(i)}.$$

Now we define the **geometric margin** of (w, b) with respect to a training example $(x^{(i)}, y^{(i)})$ to be

$$\gamma^{(i)} = y^{(i)} \left(\left(\frac{w}{||w||} \right)^T x^{(i)} + \frac{b}{||w||} \right)$$

Given a training set $S = \{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$, we also define the geometric margin of (w, b) with respect to S to be the smallest of the geometric margins on the individual training examples:

$$\gamma = \min_{i=1,\ldots,m} \gamma^{(i)}.$$

The optimization problem we now have is a constrained optimization problem because we must find the best values, provided they meet some constraints. 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. When we construct the Lagrangian for our optimization problem we have:

$$\mathcal{L}(w,b,\alpha)=\frac{1}{2}||w||^2-\sum_{i=1}^{m}\alpha_i\left[y^{(i)}(w^Tx^{(i)}+b)-1\right]$$

Here, the α_i's are the Lagrange multipliers. Let's find the dual form of the problem:

$$\mathcal{L}(w,b,\alpha)=\sum_{i=1}^{m}\alpha_i-\frac{1}{2}\sum_{i,j=1}^{m}y^{(i)}y^{(j)}\alpha_i\alpha_j(x^{(i)})^Tx^{(j)}$$

Recall that we got to the equation above by minimizing L with respect to w and b. Putting this together with the constraints, we obtain the following dual optimization problem:

$$\begin{aligned}\max_\alpha \quad & W(\alpha)=\sum_{i=1}^{m}\alpha_i-\frac{1}{2}\sum_{i,j=1}^{m}y^{(i)}y^{(j)}\alpha_i\alpha_j\langle x^{(i)},x^{(j)}\rangle\\ \text{s.t.} \quad & \alpha_i\geq 0,\quad i=1,\ldots,m\\ & \sum_{i=1}^{m}\alpha_i y^{(i)}=0,\end{aligned}$$

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:

$$\max_\alpha \quad W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle$$

$$\text{s.t.} \quad 0 \leq \alpha_i \leq C, \quad i = 1, \ldots, m$$

$$\sum_{i=1}^{m} \alpha_i y^{(i)} = 0,$$

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

Using the code

We'll now discuss the SMO algorithm. 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. We will solve small dataset with the simplified SMO and our data points will be stored in the **mySampleSVM.txt** file, its content can look like this:

X1	X2	Label
-3	-2	-1
-2	3	-1
-1	-4	-1
2	3	-1
3	4	-1
-1	9	1
2	14	1
1	17	1
3	12	1
0	8	1

The following loadDataSet() will load data from the mySampleSVM.txt:

```
def loadDataSet(fileName):

      dataMat = []
```

```
    labelMat = []
    fr = open(fileName)
    for line in fr.readlines():
        lineArr = line.strip().split()
        dataMat.append([float(lineArr[0]), float(lineArr[1])])
        labelMat.append(float(lineArr[2]))
    return dataMat,labelMat
```

We also need to create two helper functions: the first one randomly selects one integer from a range

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

This function takes two parameters: **i** is the index of our first alpha and **m** is the total number of alphas.

The second one clips values if they get too big

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

this function clips alpha values that are greater than H or less than L.

Now that we have these working, we're ready for the simplified SMO algorithm ([1], [2]):

```
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
```

Once we have a set of alphas and b, we can easily compute our weights w

```
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
```

and we can also easily get the separating hyperplane

```
def plotBestFit(point, w, alphas, b, dataMat, labelMat):
```

```
shape(alphas[alphas>0])
svmMat = []
alphaMat = []
for i in range(10):
      alphaMat.append(alphas[i])
      if alphas[i]>0.0:
            svmMat.append(dataMat[i])



svmArr = np.array(svmMat)
alphaArr = np.array(alphaMat)
labelArr = np.array(labelMat)
dataArr = np.array(dataMat)



numofSVMs = shape(svmArr)[0]
print("Number of SVMs: %d" % numofSVMs)



xSVM = []; ySVM = []
for i in range(numofSVMs):
      xSVM.append(svmArr[i,0]); ySVM.append(svmArr[i,1])



dataArr = np.array(dataMat)
n = shape(dataArr)[0]
xcord1 = []; ycord1 = []
xcord2 = []; ycord2 = []

for i in range(n):
      if int(labelMat[i])== 1:
```

```
                    xcord1.append(dataArr[i,0]); ycord1.append(dataArr[i,1])

            else:
                    xcord2.append(dataArr[i,0]); ycord2.append(dataArr[i,1])



    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.scatter(xcord1, ycord1, s=30, c='red', marker='s')
    for j in range(0,len(xcord1)):
            for l in range(numofSVMs):
                    if (xcord1[j]== xSVM[l]) and (ycord1[j]== ySVM[l]):
                            ax.annotate("SVM", (xcord1[j],ycord1[j]),
(xcord1[j]+1,ycord1[j]+2),arrowprops=dict(facecolor='black', shrink=0.005))


    ax.scatter(xcord2, ycord2, s=30, c='green')
    for k in range(0,len(xcord2)):
            for l in range(numofSVMs):
                    if (xcord2[k]== xSVM[l]) and (ycord2[k]== ySVM[l]):
                            ax.annotate("SVM", (xcord2[k],ycord2[k]),(xcord2[k]-
1,ycord2[k]+1),arrowprops=dict(facecolor='black', shrink=0.005))


    red_patch = mpatches.Patch(color='red', label='Class 1')
    green_patch = mpatches.Patch(color='green', label='Class -1')
    plt.legend(handles=[red_patch,green_patch])


    x = []
    y = []
    for xfit in np.linspace(-3.0, 3.0):
            x.append(xfit)
            y.append(float((-w[0]/w[1])*xfit - b[0,0])/w[1])


    ax.plot(x,y)
    predictedClass(point,w,b)
    p = mat(point)
    ax.scatter(p[0,0], p[0,1], s=30, c='black', marker='s')
    circle1=plt.Circle((p[0,0],p[0,1]),0.6, color='b', fill=False)
    plt.gcf().gca().add_artist(circle1)


    plt.show()
```

We can try to predict with (3,4) point

```
dataArr,labelArr = loadDataSet('mySampleSVM.txt')
b,alphas = smoSimple(dataArr, labelArr, 0.6, 0.001, 40)
w = calcWs(alphas,dataArr,labelArr)
plotBestFit([3,4], w, alphas, b, dataArr, labelArr)
```

Our result:

```
L==H
L==H
iter: 0 i:2, pairs changed 1
iteration number: 0
L==H
iter: 0 i:4, pairs changed 2
iteration number: 0
L==H
iter: 0 i:6, pairs changed 3
iteration number: 0
j not moving enough
L==H
iter: 0 i:9, pairs changed 4
iteration number: 0
j not moving enough
```

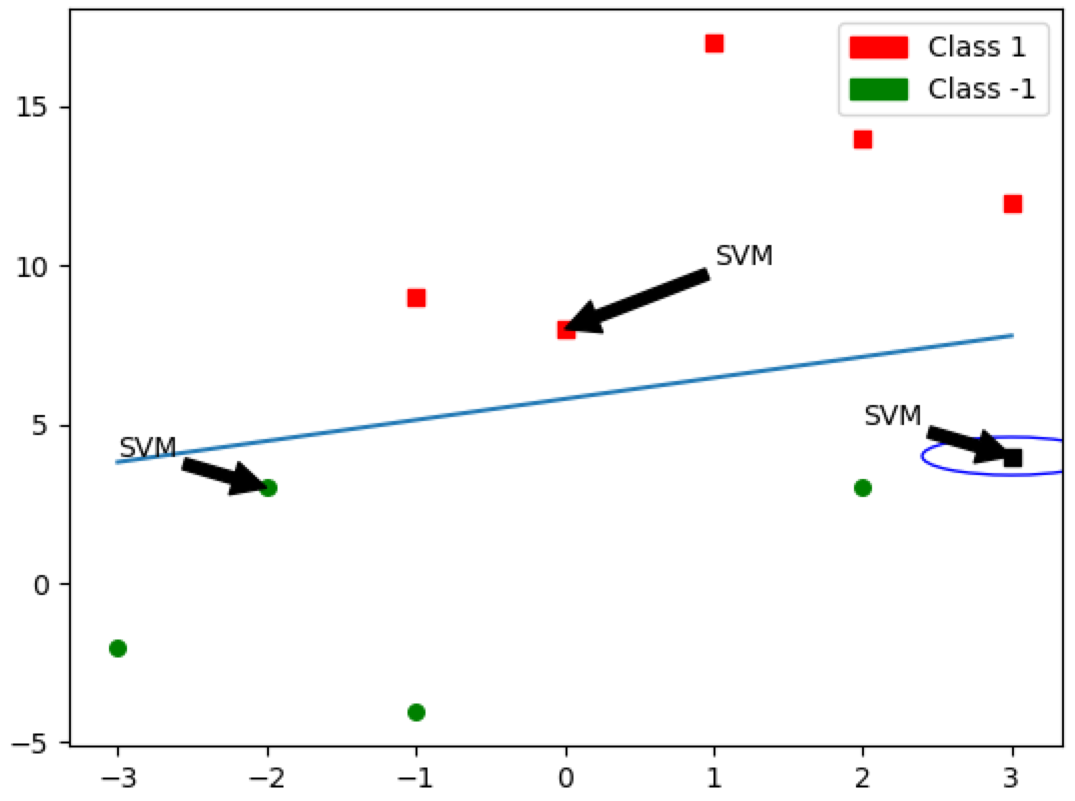
```
iter: 0 i:1, pairs changed 1
iteration number: 0
iteration number: 0
iteration number: 0
iteration number: 0
j not moving enough
iter: 0 i:6, pairs changed 2
iteration number: 0
iter: 0 i:7, pairs changed 3
iteration number: 0
L==H
iter: 0 i:9, pairs changed 4
iteration number: 0
L==H
iter: 0 i:1, pairs changed 1
iteration number: 0
iteration number: 0
iter: 0 i:3, pairs changed 2
iteration number: 0
iter: 0 i:4, pairs changed 3
iteration number: 0
j not moving enough
iteration number: 0
iteration number: 0
j not moving enough
iter: 0 i:9, pairs changed 4
iteration number: 0
iteration number: 1
iter: 1 i:1, pairs changed 1
```

```
iteration number: 0
iteration number: 0
j not moving enough
j not moving enough
iteration number: 0
j not moving enough
iteration number: 0
iteration number: 0
iteration number: 0
iteration number: 1
iteration number: 2
iteration number: 3
j not moving enough
iter: 3 i:4, pairs changed 1
iteration number: 0
j not moving enough
iteration number: 0
iteration number: 0
j not moving enough
j not moving enough
iteration number: 1
j not moving enough
iteration number: 2
j not moving enough
iteration number: 3
j not moving enough
iteration number: 4
iteration number: 5
j not moving enough
```

```
iter: 5 i:9, pairs changed 1

iteration number: 0

iteration number: 1

iter: 1 i:1, pairs changed 1

iteration number: 0

iteration number: 0

j not moving enough

L==H

iteration number: 0

iteration number: 0

iteration number: 0

iteration number: 0

iter: 0 i:9, pairs changed 2

iteration number: 0

iteration number: 1

j not moving enough

iteration number: 2

iteration number: 3

iter: 3 i:4, pairs changed 1

iteration number: 0

iteration number: 0

iteration number: 0

iteration number: 0

iteration number: 0

iter: 0 i:9, pairs changed 2

iteration number: 0

iteration number: 1

j not moving enough

iteration number: 2
iteration number: 3
iteration number: 4
iteration number: 5
iteration number: 6
iteration number: 7
iteration number: 8
iteration number: 9
iteration number: 10
j not moving enough
iteration number: 11
iteration number: 12
iteration number: 13
iteration number: 14
iteration number: 15
iteration number: 16
iteration number: 17
iteration number: 18
iteration number: 19
j not moving enough
iteration number: 20
iteration number: 21
iteration number: 22
iteration number: 23
iteration number: 24
iteration number: 25
iteration number: 26
iteration number: 27
iteration number: 28
j not moving enough
```

```
iteration number: 29

iteration number: 30

iteration number: 31

iteration number: 32

iteration number: 33

iteration number: 34

iteration number: 35

iteration number: 36

iteration number: 37

iter: 37 i:1, pairs changed 1

iteration number: 0

iteration number: 0

iteration number: 0

j not moving enough

iteration number: 0

iteration number: 0

iteration number: 0

iteration number: 0

iteration number: 0

iteration number: 1

iteration number: 2

iteration number: 3

iteration number: 4

j not moving enough

iteration number: 5

iteration number: 6

iteration number: 7

iteration number: 8

iteration number: 9
iteration number: 10
iteration number: 11
iteration number: 12
iteration number: 13
j not moving enough
iteration number: 14
iteration number: 15
iteration number: 16
iteration number: 17
iteration number: 18
iteration number: 19
iteration number: 20
iteration number: 21
iteration number: 22
j not moving enough
iteration number: 23
iteration number: 24
iteration number: 25
iteration number: 26
iteration number: 27
iteration number: 28
iteration number: 29
iteration number: 30
iteration number: 31
j not moving enough
iteration number: 32
iteration number: 33
iteration number: 34
iteration number: 35
iteration number: 36
iteration number: 37
iteration number: 38
iteration number: 39
iteration number: 40
j not moving enough
iteration number: 41
iteration number: 42
iteration number: 43
iteration number: 44
iteration number: 45
Number of SVMs: 3
Class -1
```

And:



The simplified SMO works OK on small datasets with a few hundred points but slows down on larger datasets. We can move on to the full version of the SMO algorithm. The full 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 full version of the SMO algorithm can look like this:

```
def smoFull(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
```

You can refer all of source Python code of the full SMO here. We can try to predict with (3,4) point again:

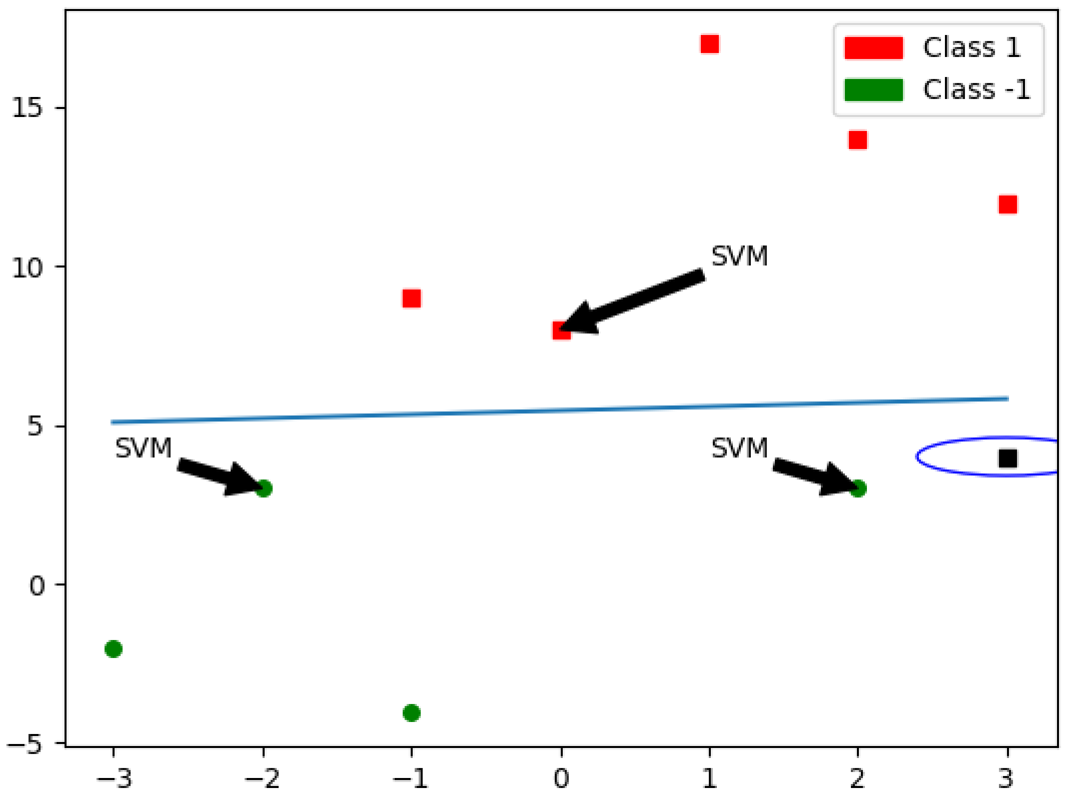
```
dataArr,labelArr = loadDataSet('mySampleSVM.txt')

b,alphas = smoFull(dataArr, labelArr, 0.6, 0.001, 40)

w = calcWs(alphas,dataArr,labelArr)

plotBestFit([3,4], w, alphas, b, dataArr, labelArr)
```

Our result:

```
L==H

fullSet, iter: 0 i:0, pairs changed 0

fullSet, iter: 0 i:1, pairs changed 1

fullSet, iter: 0 i:2, pairs changed 1

fullSet, iter: 0 i:3, pairs changed 2

L==H

fullSet, iter: 0 i:4, pairs changed 2

fullSet, iter: 0 i:5, pairs changed 2

fullSet, iter: 0 i:6, pairs changed 2

fullSet, iter: 0 i:7, pairs changed 2

fullSet, iter: 0 i:8, pairs changed 2

fullSet, iter: 0 i:9, pairs changed 2

iteration number: 1

j not moving enough

non-bound, iter: 1 i:1, pairs changed 0

non-bound, iter: 1 i:3, pairs changed 0

non-bound, iter: 1 i:9, pairs changed 0

iteration number: 2

fullSet, iter: 2 i:0, pairs changed 0

j not moving enough

fullSet, iter: 2 i:1, pairs changed 0

fullSet, iter: 2 i:2, pairs changed 0

fullSet, iter: 2 i:3, pairs changed 0

L==H
```

```
fullSet, iter: 2 i:4, pairs changed 0
fullSet, iter: 2 i:5, pairs changed 0
fullSet, iter: 2 i:6, pairs changed 0
fullSet, iter: 2 i:7, pairs changed 0
fullSet, iter: 2 i:8, pairs changed 0
fullSet, iter: 2 i:9, pairs changed 0
iteration number: 3
Number of SVMs: 3
Class -1
```

And:



Imagine you have some complex data that is not separable, we’re going to use something called a **kernel** to transform our data into a form that’s easily understood by our classifier. This topic is outside the scope of this article.

Points of Interest

In this article, I introduced about support vector machines. There are many implementations of support vector machines, but I only focused on one of the most popular implementations: the sequential minimal optimization (SMO) algorithm. You can view all of source code in this article here.

References

[1] http://cs229.stanford.edu/notes/cs229-notes3.pdf

[2] http://cs229.stanford.edu/materials/smo.pdf

[3] https://pythonprogramming.net/soft-margin-kernel-cvxopt-svm-machine-learning-tutorial/

[4] http://www.robots.ox.ac.uk/~az/lectures/ml/lect2.pdf

[5] http://eric.univ-lyon2.fr/~ricco/cours/slides/en/svm.pdf

[6] http://www.ccs.neu.edu/home/vip/teach/MLcourse/6_SVM_kernels/lecture_notes/svm/svm.pdf

[7] Support Vector Machine Succintly

[8] Machine Learning in Action