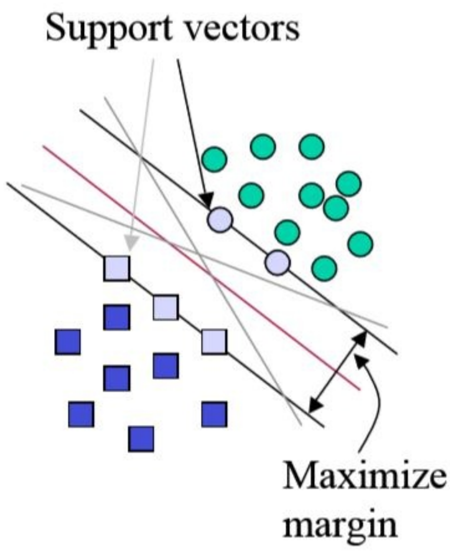
Support Vector Machine (Lagrangian multipler Method)

SVMs maximize the margin (Winston terminology: the ‘street’) around the separating line (or hyperplane).

Hypothesis

We start with assumpution equation (Called hypothesis) which can separte data in two classes.



$h(x) =b + w_0x_0+w_1x_1$

OR

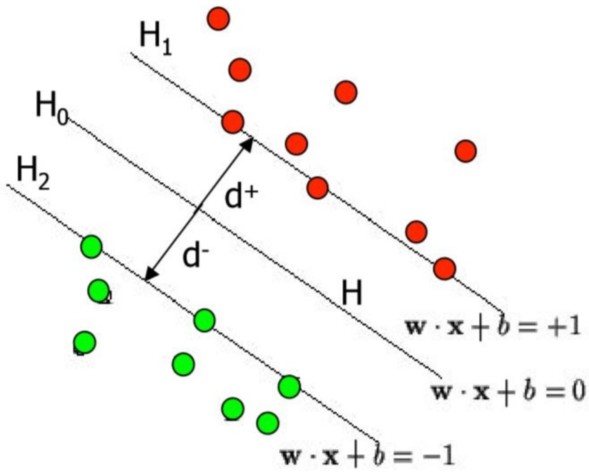
$h(x)= W^TX$ where $W=\left[\begin{matrix} b \\ w_0 \\ w_1 \\ \end{matrix}\right]$ and $X=\left[\begin{matrix} 1 \\ x_0 \\ x_1 \\ \end{matrix}\right]$

OR

$h(x)= b+wx$ where $w=\left[\begin{matrix} w_0 & w_1 \end{matrix}\right]$ and $x=\left[\begin{matrix} x_0 \\ x_1 \\ \end{matrix}\right]$

Margins

Define the equation H such that: $W^TX ≥ +1$ when Y =1 $W^TX ≤ -1$ when Y = 0



d^+ is the shortest distance to the closest positive point

d^- is the shortest distance to the closest negative point

The margin of a separating hyperplane is $M=d^+ + d^–$.

H_0, H_1 and H_2 are the planes: $H_1: W^TX = +1$

$H_2: W^TX = –1$

The points on the planes H_1 and H_2 are the tips of the Support Vectors The plane H_0 is the median in between

Maximizing the margin(d^+ and d^-)

We want a classifier (linear separator) with as big a margin as possible. Recall the distance from a point(x_0,y_0) to a line: $A.x+B.y+c = 0$ is:

$||D||=\displaystyle \frac{|A.x_0 +B.y_0 +c|}{\sqrt{(A^2+B^2)}}$

Therefore $||D||=\displaystyle \frac{|w_0.x_0 +w_1.x_1 +b|}{\sqrt{(w_0^2+w_1^2)}}$, so,

The distance between H_0 and H_1 and between H_0 and H_2 is then:

$M=|d^+| + |d^–|=\frac{|W^TX|^++|W^TX|^-}{||w||}=\frac{2}{||w||}$,

Optimization Objective

In order to maximize the margin,

we need to minimize $\frac{1}{2}||w||^2$ with the condition that:

$W^TX ≥ +1$ when Y =1

$W^TX ≤ –1$ when Y =0

This is a constrained optimization problem.

Loss Function

We predict $\hat{Y}= 1$ if $h(x)>=1$ i.e. $W^TX>=1$

We predict $\hat{Y}= 0$ if $h(x)<1$ i.e. $W^TX<-1$

Our objective is to minimize Error in predicted values.

$ Error= \hat{Y}-Y$ Where $\hat{Y}=h(X)$

we define Loss/Cost function as follows
We calculate loss,

$Loss= \begin{cases} 0 & Y=1 & h(X)>=1 \\ 1-h(X) & Y=1 & h(X)<1\\ \\ h(X)+1 & Y=0 & h(X)>-1 \\ 0 & Y=0 & h(X)<=-1 \end{cases}$

It is difficult to represent above equation Therefore we can change Y=0 to Y=-1 and $h(x)$ to $Y.h(x)$ then equation becomes

$Loss= \begin{cases} 0 & Y=1 & Y.h(X)>=1 \\ 1-Y.h(X) & Y=1 & Y.h(X)<1\\ \\ -Y.h(X)+1 & Y=-1 & -Y.h(X)>-1 \\ 0 & Y=-1 & -Y.h(X)<=-1 \end{cases}$

$\implies$

$Loss= \begin{cases} 0 & Y=1 & Y.h(X)>=1 \\ 1-Y.h(X) & Y=1 & Y.h(X)<1\\ \\ -Y.h(X)+1 & Y=-1 & Y.h(X)<1 \\ 0 & Y=-1 & Y.h(X)>=1 \end{cases}$

$\implies$

$Loss= \begin{cases} 0 & Y.h(X)>=1 \\ 1-Y.h(X) & Y.h(X)<1 \end{cases}$

$\implies$

$Loss= \begin{cases} 0 & 1-Y.h(X)<=0 \\ 1-Y.h(X) & 1-Y.h(X)>0 \end{cases}$

$\implies$

$L(W)=\frac{1}{n} \displaystyle \sum_{i=1}^nmax[0,(1-Y.h(X))]$

Final Loss Fuction

Now Objective is mimimize the $L(W)$ as well as maximize the margin (by minimize $\frac{1}{2}||w||^2$)

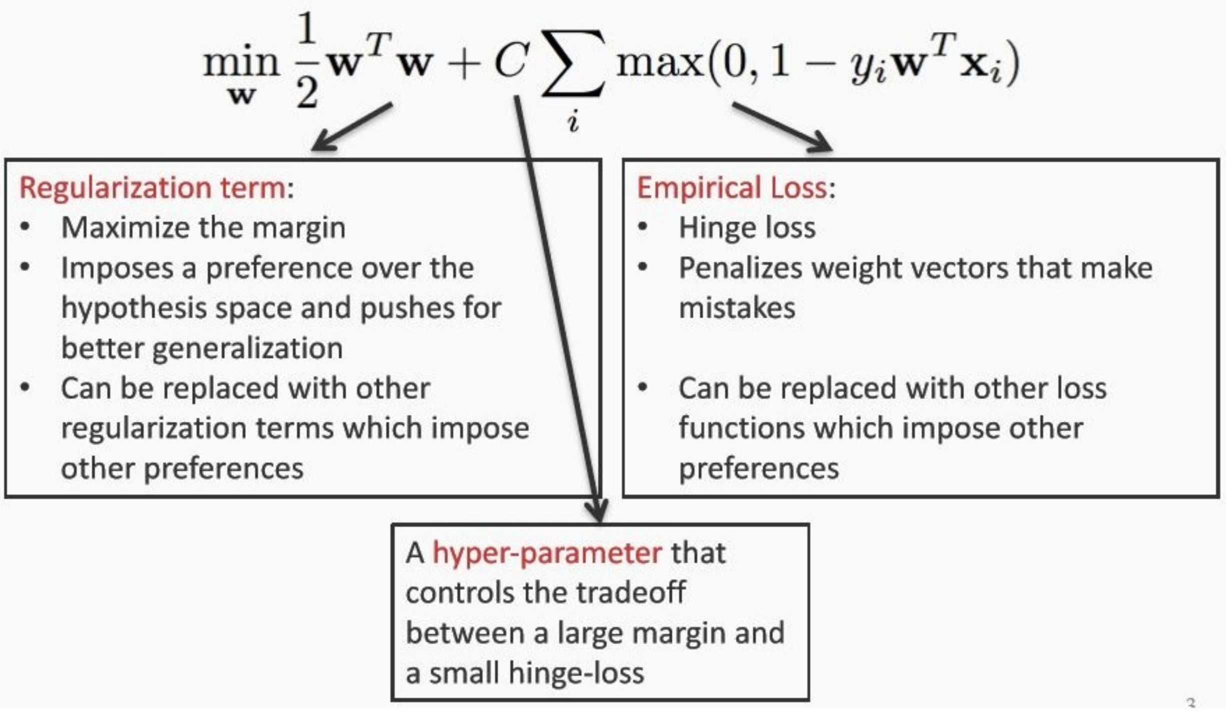
$L(W)=C\frac{1}{n} \displaystyle \sum_{i=1}^nmax[0,(1-Y.h(X))] +\frac{1}{2}||w||^2 $

C is tuning parameter which will affect the Margin.

When C is large, Margin term will reduce.

When C is small, Margin term will increase.

Solving Optimization problem



Methods

- Older methods: Used techniques from Quadratic Programming
    - **Lagrangian multiplier** method
- Gradient Descent
    - Batch Gradient Descent
    - Stochastic Gradient Descent

  Solving Lagrangian multiplier requires understanding of Linear Programing (Dual Problems). Objective of this document is to understand how SVM works and tuning parameter C affects the margins.

  Read More about dual problems and solving quadratic problems

In [ ]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
import itertools
```

Generate Data

In [ ]:

```
X1=[]
X2=[]
Y1=[]

for i,j in itertools.product(range(50),range(50)):
    if abs(i-j)>5 and abs(i-j)<40 and np.random.randint(5,size=1) >0:
        X1=X1+[i/2]
        X2=X2+[j/2]
        if (i>j):
            Y1=Y1+[1]
        else:
            Y1=Y1+[0]
### Add few misclassified Data
for i,j in itertools.product(range(50),range(50)):
    if abs(i-j)<2  and abs(i-25)>13 and  np.random.randint(10,size=1) >7:
        if (i<25):
            X1=X1+[i/2]
            X2=X2+[j/2-1.5]
            Y1=Y1+[0]
        else:
            X1=X1+[i/2]
            X2=X2+[j/2+1]
            Y1=Y1+[1]

X=np.array([X1,X2]).T
Y=np.array([Y1]).T
```

Visualize Data

In [ ]:
```
cmap = ListedColormap(['blue', 'red'])
plt.scatter(X1,X2, c=Y1,marker='.', cmap=cmap)
plt.show()
```

SVM - Using Lagrangian Multipler Method

Linear Kernel

$K(X^i,X^j)= (X^i)^TX^j $ is dot product.

In [ ]:
```
def SVM_Train(X, Y, C,  tol=0.001, max_passes= 50):
    m,n = X.shape
    y=np.array(Y)    # not to overwrite Y referenced here
    y[y==0] = -1
    alphas = np.zeros((m, 1))
    b = 0
    E = np.zeros((m, 1))
    passes = 0
    eta = 0
    L = 0
    H = 0
    y=y.flatten()
    E=E.flatten()
    alphas=alphas.flatten()

    K = np.matmul(X,X.T) # Linear Kernel


    while (passes < max_passes):
        num_changed_alphas = 0
        for i in range(m):
            E[i] = b + np.sum(alphas*y*K[:,i]) - y[i]
            if ((y[i]*E[i] < -tol and alphas[i] < C) or (y[i]*E[i] > tol and alphas[i] > 0)):
                j= np.random.randint(0,m)
                while (i==j):
                    j= np.random.randint(0,m)
                E[j] = b + np.sum(alphas*y*K[:,j]) - y[j]
                alpha_i_old = alphas[i]
                alpha_j_old = alphas[j]
                if (y[i] == y[j]):
                    L = np.max([0, alphas[j] + alphas[i] - C])
                    H = np.min([C, alphas[j] + alphas[i]])
                else:
                    L =np.max([0, alphas[j] - alphas[i]])
                    H = np.min([C, C + alphas[j] - alphas[i]])
                if (L == H):
                    continue
                eta = 2.0 * K[i,j] - K[i,i] - K[j,j]
                if (eta >= 0):
                    continue
                alphas[j] = alphas[j] -(y[j] * (E[i] - E[j])) / eta
                alphas[j] = np.min ([H, alphas[j]])
                alphas[j] = np.max ([L, alphas[j]])
                if (np.abs(alphas[j] - alpha_j_old) < tol):
                    alphas[j] = alpha_j_old
                    continue
                alphas[i] = alphas[i] + y[i]*y[j]*(alpha_j_old - alphas[j])
                b1 = b - E[i] - y[i] * (alphas[i] - alpha_i_old) *  K[i,j] - y[j] * (alphas[j] - alpha_j_old) *
                b2 = b - E[j] - y[i] * (alphas[i] - alpha_i_old) *  K[i,j] - y[j] * (alphas[j] - alpha_j_old) *
                if (0 < alphas[i] and alphas[i] < C):
                    b = b1
                elif (0 < alphas[j] and alphas[j] < C):
                    b = b2
                else:
                    b = (b1+b2)/2
                num_changed_alphas += 1
            #END IF
        #END FOR
        if (num_changed_alphas == 0):
            passes = passes + 1
        else:
            passes = 0
    #end while
    W=np.matmul((alphas*y).reshape(1,m),X).T
    weights=np.row_stack(([[b]],W))
    return weights
```

In [ ]:
```
def predict(X,weights):
    fx=weights[0,0]+np.matmul(X, weights[1:,:])
    fx[fx>0]=1
    fx[fx<=0]=0
    PY=fx
    return PY
```

In [ ]:
```
def accurracy(Y1,Y2):
    m=np.mean(np.where(Y1==Y2,1,0))
    return m*100
```

In [ ]:
```
def plot_Decision_Boundry(X,Y,weights):
    plt.figure(figsize=(8,8))
    plt.scatter(X[:,0],X[:,1], c=Y[:,0],marker='.', cmap=cmap)

    #Predict for each X1 and X2 in Grid
    x_min, x_max = X[:, 0].min() , X[:, 0].max()
    y_min, y_max = X[:, 1].min() , X[:, 1].max()
    u = np.linspace(x_min, x_max, 100)
    v = np.linspace(y_min, y_max, 100)

    U,V=np.meshgrid(u,v)
    UV=np.column_stack((U.flatten(),V.flatten()))
    W=predict(UV, weights)
    plt.scatter(U.flatten(), V.flatten(),  c=W.flatten(), cmap=cmap,marker='.', alpha=0.1)


    # w1.x1+ w2.x2 +b=0
    # x2= -b/w2 - w1/w2*x1

    #Exact Decision Boundry

    for i in range(len(u)):
        v[i]=-weights[0,0]/weights[2,0]  -weights[1,0]* u[i]/weights[2,0]
    plt.plot(u, v, color='k')
    ############################################################################

    #Margins
    M= (2/np.sqrt(weights[1,0]**2+weights[2,0]**2))



    for i in range(len(u)):
        v[i]=M-weights[0,0]/weights[2,0]  -weights[1,0]* u[i]/weights[2,0]
    plt.plot(u, v, color='gray')

    for i in range(len(u)):
        v[i]=-M-weights[0,0]/weights[2,0]  -weights[1,0]* u[i]/weights[2,0]
    plt.plot(u, v, color='gray')


    plt.show()
```

Training with Large Margin (C = 0.01)

In [ ]:
```
C=0.01
weights=SVM_Train(X, Y, C=C)

pY=predict(X, weights)
print("Accurracy=",accurracy(Y, pY))

plot_Decision_Boundry(X,Y,weights)
```

Training with Small Margin (C =5)

In [ ]:
```
C=5
weights=SVM_Train(X, Y, C=C)

pY=predict(X, weights)
print("Accurracy=",accurracy(Y, pY))

plot_Decision_Boundry(X,Y,weights)
```

References

LIBSVM Paper

Towards Data Science

SVM-Tutorial

Idiot’s guide to Support vector machines

In [ ]: