**1. Perceptrons**

**Answer 1.1**

*Importing libraries:*

**import** numpy **as** np

**from** numpy **import** linalg **as** LA

**import** matplotlib.pyplot **as** plt

**import** pandas **as** pd

**from** timeit **import** default\_timer **as** timer

**import** sys

sys.setrecursionlimit(10000)

%matplotlib inline

*Helper functions::*

# normalizing each vector i.e. each Z\_i in Z

**def** normalize(Z):

X=np.empty((np.shape(Z)[0],np.shape(Z)[1]))

**for** i **in** **range**(np.shape(Z)[0]):

norm = LA.norm(Z[i])

**if**(norm==0):

X[i]=Z[i]

**print**("here at i: ",i)

**else**:

X[i]=Z[i]/norm

# to check if norm/length == 1

# print(LA.norm(X[i]))

**return** X

# to normalize a single point i.e. 1D array

**def** normalize\_single(vec):

norm = LA.norm(vec)

**if** norm == 0:

**return** vec

**return** vec / norm

# discard the vector/point and replace with new point which satisfies abs(vec[k-1])>=ep

**def** X\_i\_replace(vec, ep):

# given mean and variance

mu=0

sigma=1

k=**len**(vec)

new\_vec=np.random.normal(mu,sigma,k)

new\_vec=normalize\_single(new\_vec)

**if**(**abs**(new\_vec[k-1])<ep): # k-1 is th kth index

# recursively call itself until abs(vec[k-1])>=ep

**return** X\_i\_replace(new\_vec, ep)

# print("new vec")

**else**:

# just checking if norm of new point == 1

# print("new point created with norm: ",new\_vec," --- ",LA.norm(new\_vec))

**return** new\_vec

# normalizing/scaling weights and bias such that |w|^2 + b\*b = 1

**def** normalize\_w\_and\_b(w,b):

#w2=np.dot(w,np.transpose(w))

w2=np.dot(w,w)

b2=b\*b

**sum**=w2+b2

w\_=w/**pow**(**sum**,0.5)

b\_=b/**pow**(**sum**,0.5)

**return** w\_,b\_

*Function to generate X and Y::*

# gen X i.e. m datapoints with k features (k dimensions)

**def** gen\_X(m,k,ep):

**pass**

# given mean and variance

mu=0

sigma=1

# i.i.d standard normal data

Z=np.random.normal(mu,sigma,size=(m,k))

# normalizing each vector i.e. each Z\_i in Z

X=normalize(Z)

# check if absolute value of X\_i\_k < epsilon for all X\_i

**for** i **in** **range**(m):

**if**(**abs**(X[i][k-1])<ep): # k-1 is th kth index

# discard the vector/point and replace with new point which satisfies the above condition

# print(i)

X[i]=X\_i\_replace(X[i],ep)

**return** X

# gen Y/target/labels/output

**def** gen\_Y(X,ep):

**pass**

m=np.shape(X)[0]

k=np.shape(X)[1]

Y=np.empty((m)).astype(**int**)

**for** i **in** **range**(m):

**if**(X[i][k-1]>=ep):

Y[i]=1

**elif**(X[i][k-1]<=-ep):

Y[i]=-1

**return** Y

*Function - Perceptron Learning Algorithm (PLA):*

# returns steps to converge, "typical" weights and "typical" bias

**def** PLA(X,Y,MAX\_ITER=10000):

# init wts and bias

wt=np.zeros((np.shape(X)[1]))

b=0

steps\_to\_converge=0

**while**(True):

steps\_to\_converge+=1

misclassified=0

**for** X\_i, Y\_i **in** **zip**(X,Y):

linear\_op=np.dot(X\_i,wt)+b

**if**(linear\_op>0):

f\_x=1.0

**else**:

f\_x=-1.0

# if misclassified

**if**(f\_x!=Y\_i):

misclassified+=1

# update step

wt+=np.dot(X\_i,Y\_i)

b+=Y\_i

**if**(misclassified==0):

**return** steps\_to\_converge, wt, b

**if**(steps\_to\_converge>MAX\_ITER):

#breaking

**return** steps\_to\_converge, wt, b

**Answer 1:**

start=timer()

# init parameters

k=10

ep=0.1

m=**list**(**range**(10,10011,20))

# number of iterations to calculate the average

avg\_iter=10

**print**("m values (List) =",m)

avg\_steps\_=[]

# running the sim

**for** m\_i **in** m:

# init avg number of steps to converge

avg\_steps=0

**for** i **in** **range**(avg\_iter):

# generate X and Y

X=gen\_X(m\_i,k,ep)

Y=gen\_Y(X,ep)

# get number of steps to converge from PLA function

steps\_to\_converge, \_, \_ = PLA(X,Y)

# summing up number of steps to converge

avg\_steps+=steps\_to\_converge

# averaging number of steps to converge

avg\_steps/=avg\_iter

# add to a list to plot

avg\_steps\_.append(avg\_steps)

# converting to pandas dataframe to get the

# rolling mean (moving average) for better analysis

df\_avg\_steps = pd.DataFrame(avg\_steps\_)

rolling\_mean\_steps = df\_avg\_steps.rolling(window=50).mean()

# plotting

plt.plot(m, avg\_steps\_, label="Original", color="limegreen")

plt.plot(m, rolling\_mean\_steps, label="Moving Average - 50", color="red", linewidth="2")

plt.xlabel("m: No. of datapoints")

plt.ylabel("Avg. number of steps")

plt.legend(loc="upper right")

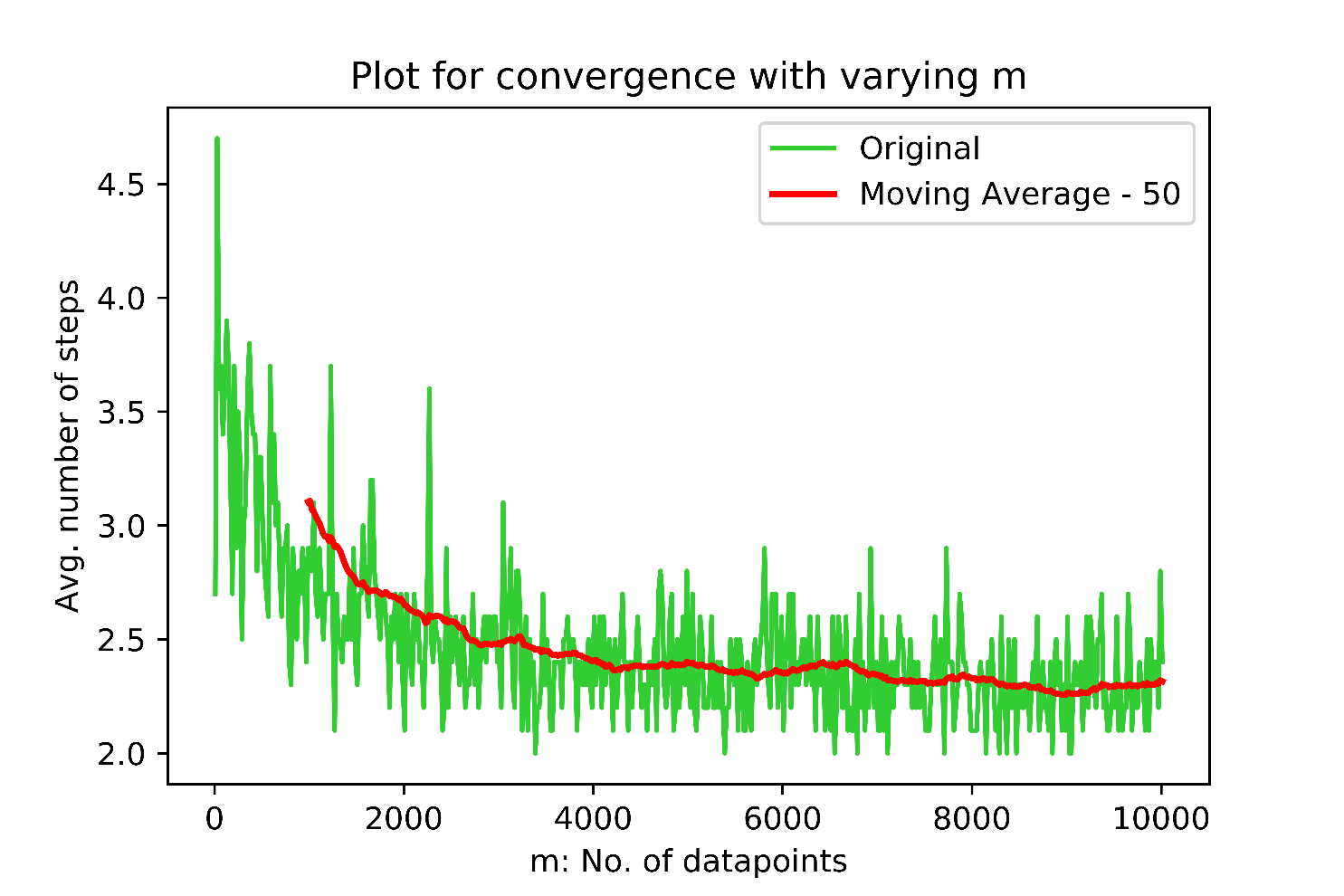
plt.title("Plot for convergence with varying m")

plt.savefig('Q1\_fig.png',dpi=1200)

plt.show()

**print**("Time taken: ", timer()-start)

m values (List) = [10, 30, 50, 70, …, 9930, 9950, 9970, 9990, 10010]



Time taken: 624.5963182

*For the above plot and following plots, the red line indicates a moving average with the mentioned window size.*

From the above plot, we can say that although initially (upto m=4000) it seems that the number of steps to converge is dependent on the number of datapoints, but on a larger scale, the number of steps for convergence does not depend on the number of datapoints (m). Also from the formula (given in the notes), we can confirm the same. There is no dependency on m or k, but only on the margin ().

The results make sense since it agrees with the mathematical formulation.

Larger and larger m values will not produce anything different.

**Answer 2:**

start=timer()

# init parameters

k=**list**(**range**(2,1001,1))

ep=0.05

m=100

# number of iterations to calculate the average

avg\_iter=10

**print**("k values (List) =",k)

avg\_steps\_=[]

# running the sim

**for** k\_i **in** k:

# init avg number of steps to converge

avg\_steps=0

**for** i **in** **range**(avg\_iter):

# generate X and Y

X=gen\_X(m,k\_i,ep)

Y=gen\_Y(X,ep)

# get number of steps to converge from PLA function

steps\_to\_converge, \_, \_ = PLA(X,Y)

# summing up number of steps to converge

avg\_steps+=steps\_to\_converge

# averaging number of steps to converge

avg\_steps/=avg\_iter

# add to a list to plot

avg\_steps\_.append(avg\_steps)

# converting to pandas dataframe to get the

# rolling mean (moving average) for better analysis

df\_avg\_steps = pd.DataFrame(avg\_steps\_)

rolling\_mean\_steps = df\_avg\_steps.rolling(window=40).mean()

# plotting

plt.plot(k, avg\_steps\_, label="Original", color="limegreen")

plt.plot(k, rolling\_mean\_steps, label="Moving Average - 40", color="red", linewidth="2")

plt.xlabel("k: No. of features")

plt.ylabel("Avg. number of steps")

plt.legend(loc="upper right")

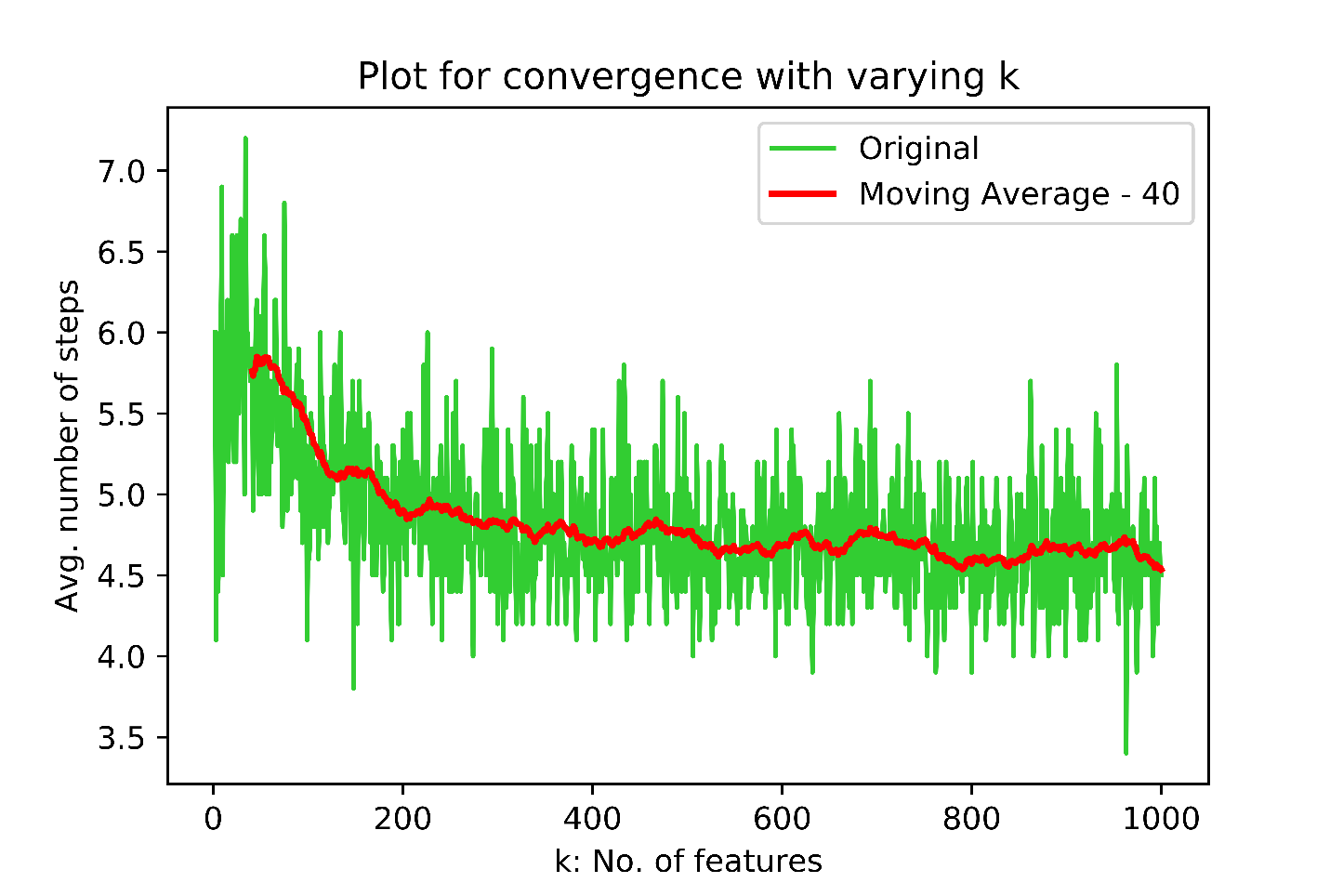
plt.title("Plot for convergence with varying k")

plt.savefig('Q2\_fig.png',dpi=1200)

plt.show()

**print**("Time taken: ", timer()-start)

k values (List) = [2, 3, 4, 5, …, 995, 996, 997, 998, 999, 1000]



Time taken: 173.78885200000002

From the above plot, we can say that although initially (upto k=200) it seems that the number of steps to converge is dependent on the number of dimensions/features, but on a larger scale, the number of steps for convergence does not depend on the number of dimensions /features (k). Also from the formula (given in the notes), we can confirm the same. There is no dependency on m or k, but only on the margin ().

The results make sense since it agrees with the mathematical formulation.

Larger and larger k values will not produce anything different.

**Answer 3:**

start=timer()

# init parameters

k=5

ep=np.arange(0.01,0.96,0.01)

m=100

# number of iterations to calculate the average

avg\_iter=10

**print**("epsilon values (List) =",ep)

avg\_steps\_=[]

# running the sim

**for** ep\_i **in** ep:

# init avg number of steps to converge

avg\_steps=0

**for** i **in** **range**(avg\_iter):

# generate X and Y

X=gen\_X(m,k,ep\_i)

Y=gen\_Y(X,ep\_i)

# get number of steps to converge from PLA function

steps\_to\_converge, \_, \_ = PLA(X,Y)

# summing up number of steps to converge

avg\_steps+=steps\_to\_converge

# averaging number of steps to converge

avg\_steps/=avg\_iter

# add to a list to plot

avg\_steps\_.append(avg\_steps)

#df\_avg\_steps = pd.DataFrame(avg\_steps\_)

#rolling\_mean\_steps = df\_avg\_steps.rolling(window=3).mean()

# plotting

plt.plot(ep, avg\_steps\_, label="Original", color="limegreen")

#plt.plot(ep, rolling\_mean\_steps, label="Moving Average - 3", color="red", linewidth="2", linestyle="dashed")

plt.xlabel("epsilon: Forced Margin? :D")

plt.ylabel("Avg. number of steps")

plt.legend(loc="upper right")

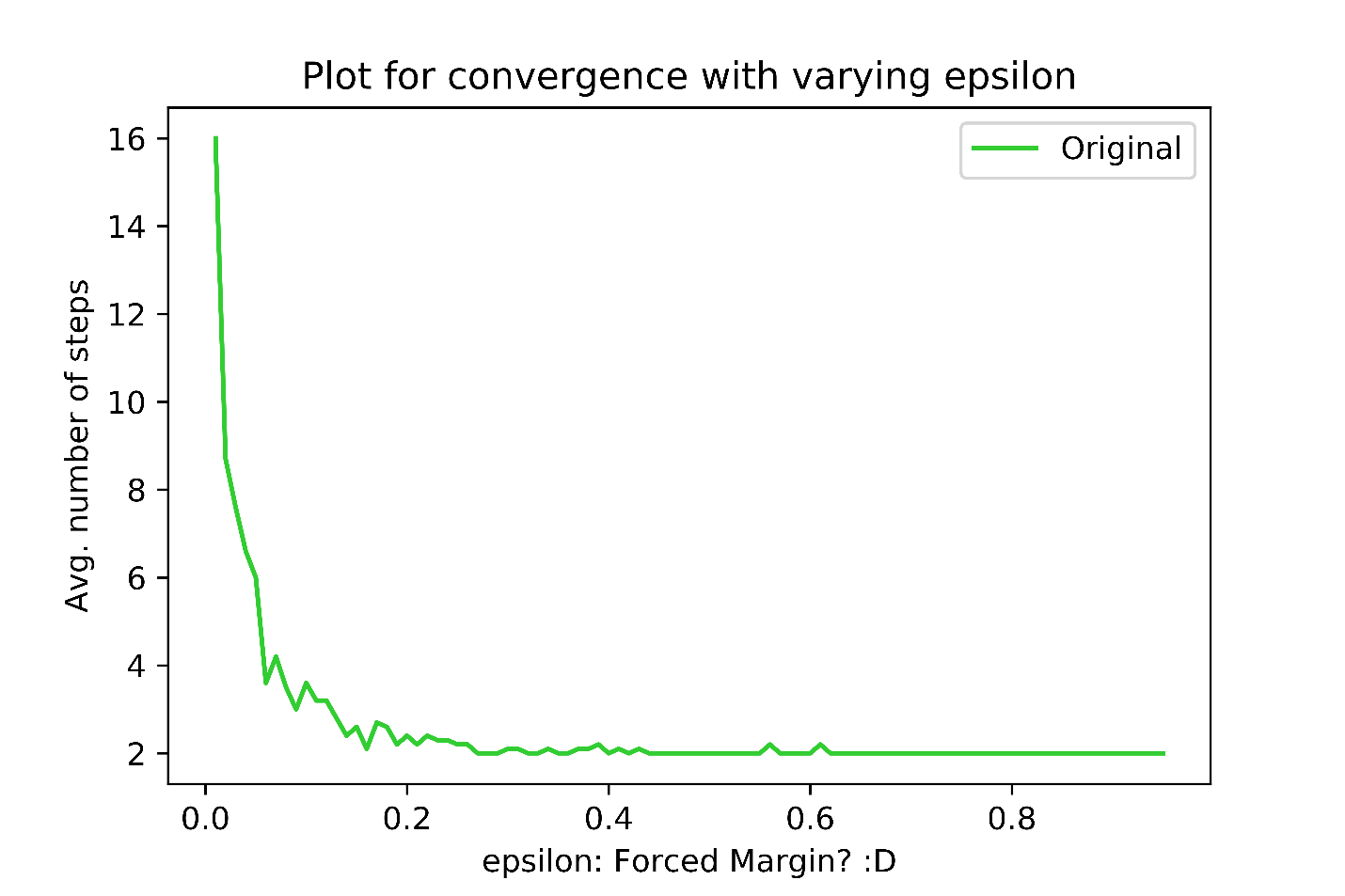
plt.title("Plot for convergence with varying epsilon")

plt.savefig('Q3\_fig.png',dpi=1200)

plt.show()

**print**("Time taken: ", timer()-start)

epsilon values (List) = [0.01 0.02 0.03 0.04 0.05 … 0.92 0.93 0.94 0.95]



Time taken: 27.150961800000005

From the above plot, we can say that the number of steps for convergence depends on the value of epsilon. Epsilon here acts as a kind of forced margin.

Hence from the formula (given in the notes), we can confirm the same. There is no dependency on m or k, but only on the margin ().

The results make sense since from our intuition we can also imagine that more is the margin (forced), easier it is for the linear separator to be found. For smaller values of epsilon, the linear separator only exists within a small region and hence takes more steps to converge.

Looking at different k and m (except for small values of k and m) will not produce anything different.

**Bonus Answer Part 1:**

start=timer()

# init parameters

k=10

ep=0.1

m=**list**(**range**(10,3011,20))

# number of iterations to calculate the average

avg\_iter=10

# init ideal weights and bias

wt\_ideal=np.zeros((k))

wt\_ideal[-1]=1

b\_ideal=0

**print**("m values (List) =",m)

ordinate\_val\_list=[]

# running the sim

**for** m\_i **in** m:

# init typical average wt and bias

avg\_wt\_typ=np.zeros((k))

avg\_b\_typ=0

**for** i **in** **range**(avg\_iter):

# generate X and Y

X=gen\_X(m\_i,k,ep)

Y=gen\_Y(X,ep)

# get typical wt and bias from PLA function

\_, wt\_typical, b\_typical = PLA(X,Y)

# summing up wts and bias

avg\_wt\_typ+=wt\_typical

avg\_b\_typ+=b\_typical

# averaging wts and bias

avg\_wt\_typ/=avg\_iter

avg\_b\_typ/=avg\_iter

# normalizing/scaling weights and bias such that |w|^2 + b^2 = 1

avg\_wt\_typ, avg\_b\_typ = normalize\_w\_and\_b(avg\_wt\_typ,avg\_b\_typ)

wt\_ideal, b\_ideal = normalize\_w\_and\_b(wt\_ideal,b\_ideal)

# calculating the value of the given equation in steps

wt\_diff=wt\_ideal-avg\_wt\_typ

b\_diff=b\_ideal-avg\_b\_typ

val=**pow**(LA.norm(wt\_diff),2)+**pow**(b\_diff,2)

# add to a list to plot

ordinate\_val\_list.append(val)

# converting to pandas dataframe to get the

# rolling mean (moving average) for better analysis

df\_val = pd.DataFrame(ordinate\_val\_list)

rolling\_mean\_steps = df\_val.rolling(window=5).mean()

# plotting

plt.plot(m, ordinate\_val\_list, label="Original", color="limegreen")

plt.plot(m, rolling\_mean\_steps, label="Moving Average - 5", color="red") #, linestyle="dashed")

plt.xlabel("m: No. of datapoints")

plt.ylabel("Distance between ideal and typical perceptron")

plt.legend(loc="upper right")

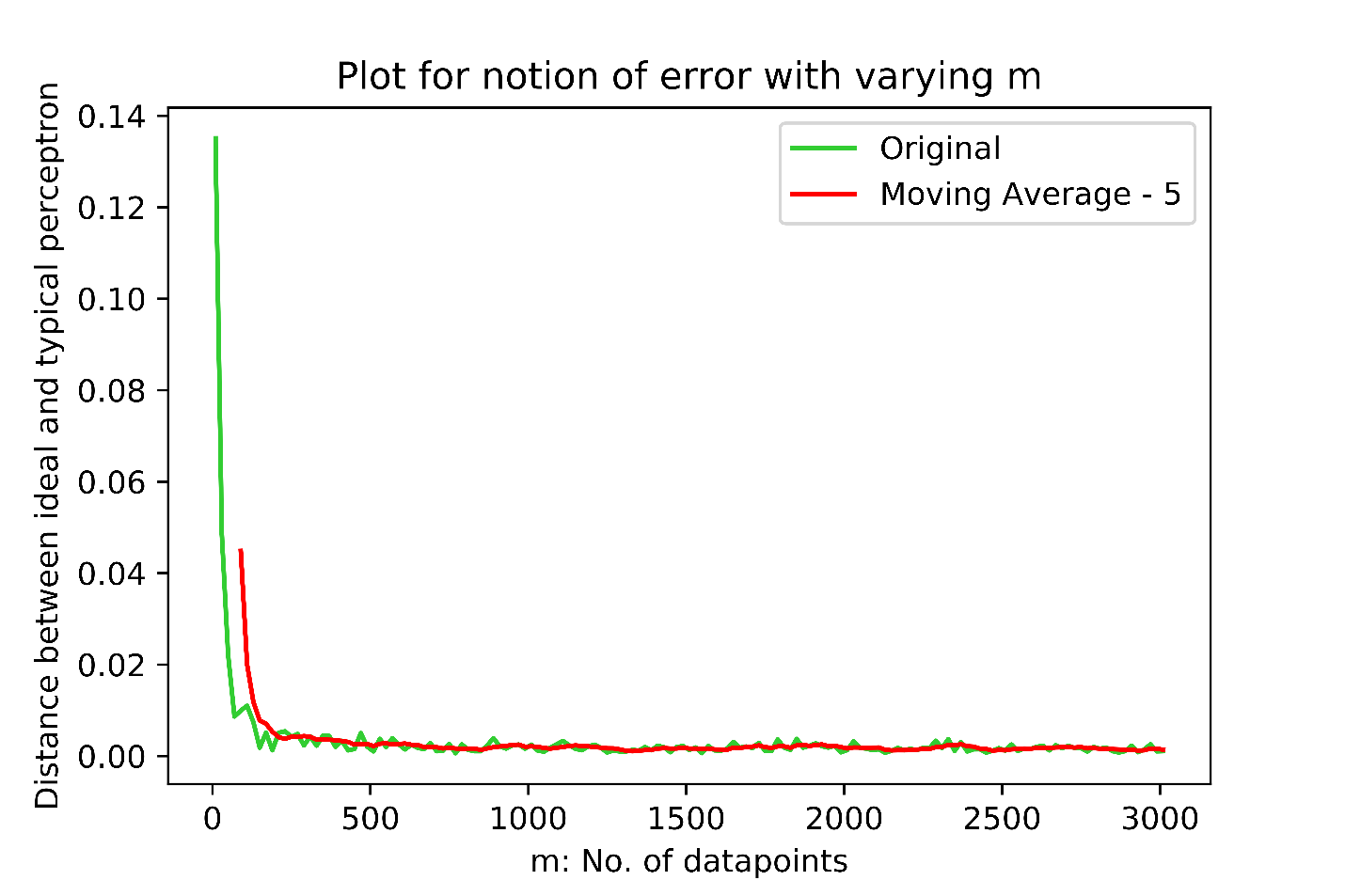
plt.title("Plot for notion of error with varying m")

plt.savefig('Bonus\_1\_fig.png',dpi=1200)

plt.show()

**print**("Time taken: ", timer()-start)

m values (List) = [10, 30, 50, 70, 90, …, 2930, 2950, 2970, 2990, 3010]



Time taken: 57.121473899999955

This plot makes sense since if we add more data points, the new data points will tend to occupy all allowed space (on the sphere's surface except the margin/epsilon) and the perceptron will have less space move around (like being crunched by data points) and we prepared the data in such a way such that there exists a linear separator. So the averaged typical perceptron will get very close to the ideal perceptron as we get in more and more data.

NB: Since epsilon is small it quickly reaches an error close to 0, but I tested with epsilon=0.5, and it didn't converge (at least till m~3000)

**Bonus Answer Part 2:**

start=timer()

# init parameters

k=**list**(**range**(2,1001,1))

ep=0.05

m=100

# number of iterations to calculate the average

avg\_iter=10

# init ideal bias

b\_ideal=0

**print**("k values (List) =",k)

ordinate\_val\_list=[]

# running the sim

**for** k\_i **in** k:

# init typical average wt and bias

avg\_wt\_typ=np.zeros((k\_i))

avg\_b\_typ=0

# init ideal weights

wt\_ideal=np.zeros((k\_i))

wt\_ideal[-1]=1

**for** i **in** **range**(avg\_iter):

# generate X and Y

X=gen\_X(m,k\_i,ep)

Y=gen\_Y(X,ep)

# get typical wt and bias from PLA function

\_, wt\_typical, b\_typical = PLA(X,Y)

# summing up wts and bias

avg\_wt\_typ+=wt\_typical

avg\_b\_typ+=b\_typical

# averaging wts and bias

avg\_wt\_typ/=avg\_iter

avg\_b\_typ/=avg\_iter

# normalizing/scaling weights and bias such that |w|^2 + b^2 = 1

avg\_wt\_typ, avg\_b\_typ = normalize\_w\_and\_b(avg\_wt\_typ,avg\_b\_typ)

wt\_ideal, b\_ideal = normalize\_w\_and\_b(wt\_ideal,b\_ideal)

# calculating the value of the given equation in steps

wt\_diff=wt\_ideal-avg\_wt\_typ

b\_diff=b\_ideal-avg\_b\_typ

val=**pow**(LA.norm(wt\_diff),2)+**pow**(b\_diff,2)

# add to a list to plot

ordinate\_val\_list.append(val)

# converting to pandas dataframe to get the

# rolling mean (moving average) for better analysis

df\_val = pd.DataFrame(ordinate\_val\_list)

rolling\_mean\_steps = df\_val.rolling(window=40).mean()

# plotting

plt.plot(k, ordinate\_val\_list, label="Original", color="limegreen")

plt.plot(k, rolling\_mean\_steps, label="Moving Average - 40", color="red", linewidth="2")#, linestyle="dashed")

plt.xlabel("k: No. of features")

plt.ylabel("Distance between ideal and typical perceptron")

plt.legend(loc="upper left")

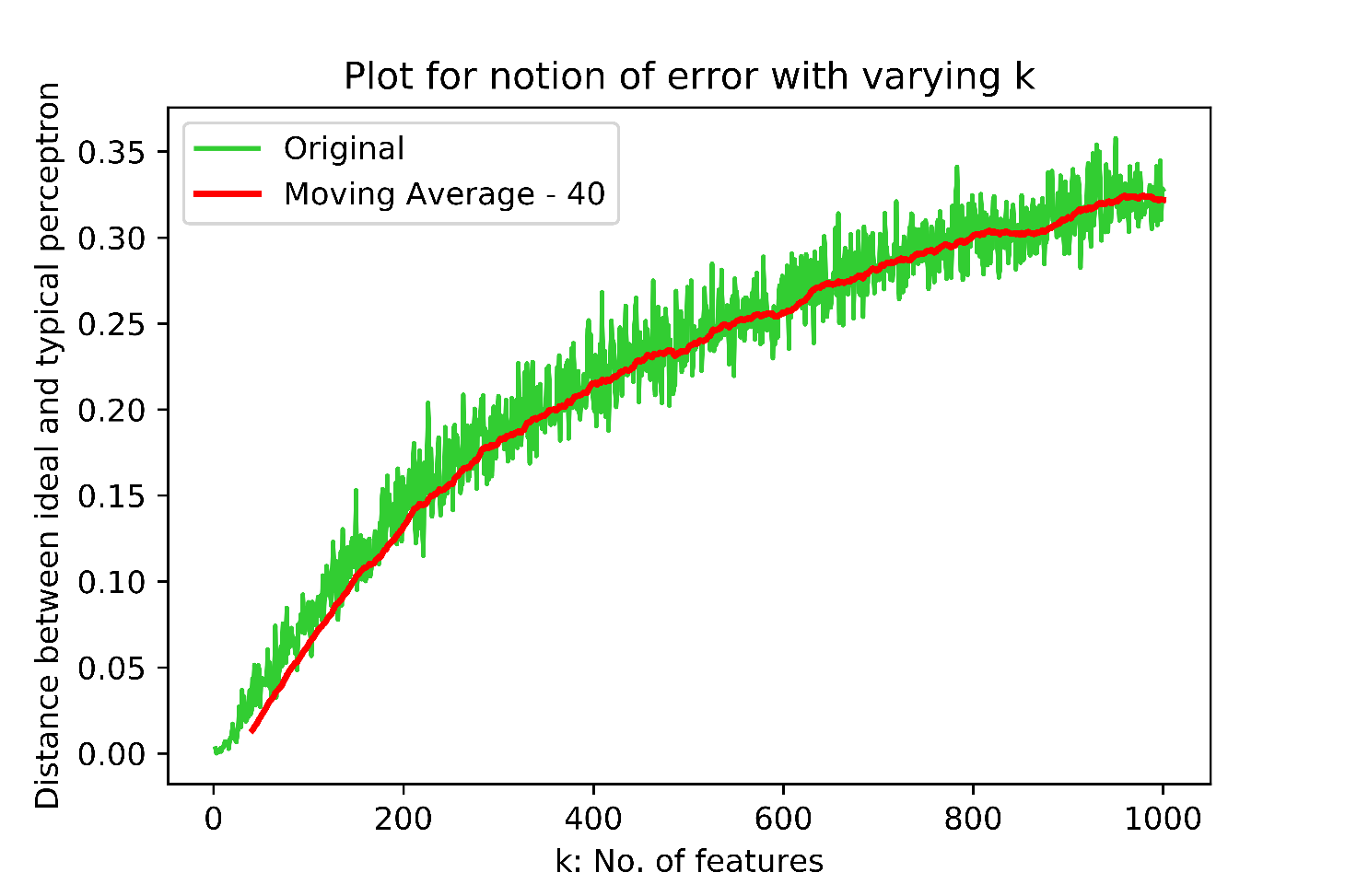
plt.title("Plot for notion of error with varying k")

plt.savefig('Bonus\_2\_fig.png',dpi=1200)

plt.show()

**print**("Time taken: ", timer()-start)

k values (List) = [2, 3, 4, 5, 6, …, 994, 995, 996, 997, 998, 999, 1000]



Time taken: 176.35292900000013

This plot makes sense since if we add more dimensions to the data, there are more possible perceptrons that can exist and hence more space for error.

We can visualize this somewhat: In 2D, the line can rotate in only one dimension, but in 3D, the plane can rotate in 2 dimensions. So as we go up in dimentionality (features), the hyperplane (the linear separator) will have more space to move around.

**Bonus Answer Part 3:**

start=timer()

# init parameters

k=5

ep=np.arange(0.01,0.96,0.01)

m=100

# number of iterations to calculate the average

avg\_iter=10

# init ideal weights and bias

wt\_ideal=np.zeros((k))

wt\_ideal[-1]=1

b\_ideal=0

**print**("epsilon values (List) =",ep)

ordinate\_val\_list=[]

# running the sim

**for** ep\_i **in** ep:

# init typical average wt and bias

avg\_wt\_typ=np.zeros((k))

avg\_b\_typ=0

**for** i **in** **range**(avg\_iter):

# generate X and Y

X=gen\_X(m,k,ep\_i)

Y=gen\_Y(X,ep\_i)

# get typical wt and bias from PLA function

\_, wt\_typical, b\_typical = PLA(X,Y)

# summing up wts and bias

avg\_wt\_typ+=wt\_typical

avg\_b\_typ+=b\_typical

# averaging wts and bias

avg\_wt\_typ/=avg\_iter

avg\_b\_typ/=avg\_iter

# normalizing/scaling weights and bias such that |w|^2 + b^2 = 1

avg\_wt\_typ, avg\_b\_typ = normalize\_w\_and\_b(avg\_wt\_typ,avg\_b\_typ)

wt\_ideal, b\_ideal = normalize\_w\_and\_b(wt\_ideal,b\_ideal)

# calculating the value of the given equation in steps

wt\_diff=wt\_ideal-avg\_wt\_typ

b\_diff=b\_ideal-avg\_b\_typ

val=**pow**(LA.norm(wt\_diff),2)+**pow**(b\_diff,2)

# add to a list to plot

ordinate\_val\_list.append(val)

# converting to pandas dataframe to get the

# rolling mean (moving average) for better analysis

df\_val = pd.DataFrame(ordinate\_val\_list)

rolling\_mean\_steps = df\_val.rolling(window=4).mean()

# plotting

plt.plot(ep, ordinate\_val\_list, label="Original", color="limegreen")

plt.plot(ep, rolling\_mean\_steps, label="Moving Average - 4", color="red") #, linestyle="dashed")

plt.xlabel("epsilon: Forced Margin?")

plt.ylabel("Distance between ideal and typical perceptron")

plt.legend(loc="upper left")

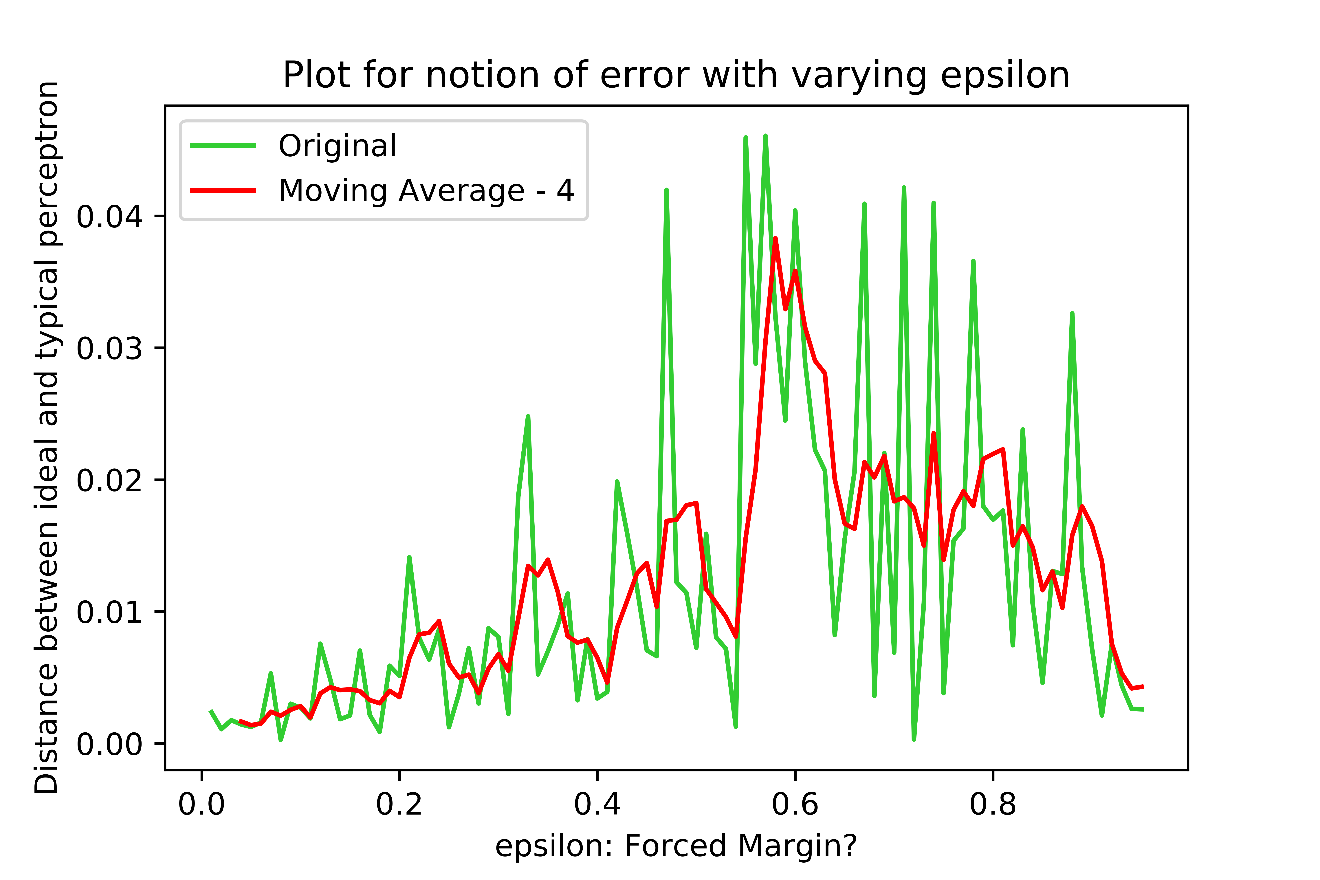
plt.title("Plot for notion of error with varying epsilon")

plt.savefig('Bonus\_3\_fig.png',dpi=1200)

plt.show()

**print**("Time taken: ", timer()-start)

epsilon values (List) = [0.01 0.02 0.03 0.04 0.05 0.06 … 0.91 0.92 0.93 0.94 0.95]



Time taken: 23.876258000000007

This plot doesn't make sense, as my intuition says error should increase with increasing epsilon. The typical perceptron gets more available space as we increase epsilon and more chance to stray away from the ideal perceptron. Although the error increases till epsilon ~ 0.6, but on further increasing epsilon, it tends to decrease which is weird.

One guess is that it has something to do with probability. Kind of like the gaussian/normal distribution where most of the data is near the mean. The perceptron has space to move in the upper as well as the lower "hemisphere" (hyper hemisphere?) but maybe stays near the "equator" with higher probability.

Another guess is that the perceptron has more freedom to rotate, so in a way although it will misclassify a lot compared to the ideal perceptron, but perform better than a perceptron which is horizontal (kind of) and far away from the ideal perceptron.

**2. SVMs**

**Answer 2.1a**

Let the radius be ‘r’

Let the center of the be (a, b) [Given its 2D, so 2 coordinates]

The linear separator would be a circle of radius ‘r’ with center at (a, b).

Equation of the circle would be

(1)

Anything outside the circle would be classified as negative and anything inside the circle would be classified as positive.

So, we can classify any datapoint (vector) as follows:

On expanding (1), we get:

or,

or,

or,

The above is of the form which can be represented by the given feature map:

Thus we see that a linear separator can always be found in the given embedded space.

**Answer 2.1b**

Let the number of dimensions be ‘k’

Let the center of the be (c1, c2, …, ck)

Let the width along each axis be (w1, w2, …, wk)

Equation of the ellipsoid (k dimensional) would be

(2)

Anything outside the ellipsoid would be classified as negative and anything inside the ellipsoid would be classified as positive.

So, we can classify any datapoint (vector) as follows:

On expanding (2), we get:

The math is long, but we can follow the same procedure as in 2.1a and show that the separator lies in the quadratic feature space.

Also, I ignored orientation as those would bring thetas but are real-valued coefficients and can be absorbed.

This is possible because there is no term , so it always remains in the quadratic feature space.

**Answer 2.2**

*Based on the wording in the question, I thought the positive class would be in* ***either one*** *of the two disjoint ellipses, but based on the clarification on the discussion board, I’m considering both ellipses to contain the positive class (and the negative class everywhere else).*

Let the first ellipse be A and the second ellipse be B.

Let the center of A be (hA, kA) and that of B be (hB, kB).

Let the width along each axis of A be (aA, bA) and that of B be (aB, bB).

Equation of ellipse A:

or,

or,

Let

Similarly, the equation of ellipse B:

Let

So, we can classify any datapoint (vector) as follows:

*From Answer 1, we confirmed that be it a circle or an ellipse, the separator follows the same idea/logic (just more coefficient terms in the case of ellipses). So I tested it out on paper by trying different test cases on two disjoint (non-concentric) circles to find the resultant sign.*

or (absorbing/replacing variables which are not as C’s,

or (absorbing/replacing variables which are not as D’s,

or (absorbing/replacing variables which are not as F’s,

or (absorbing/replacing variables which are not as G’s,

The above can be condensed further by taking common terms out of . Then it is of the form which can be represented by the feature map:

or equivalently with the kernel:

Thus, the kernel will recover a separator.

**Answer 2.3**

There are 2 concentric circles.

Let the smaller circle be A and the larger circle be B.

Let both of them be centered at (a,b)

Let the radius of the smaller circle be ‘rA’.

Let the radius of the larger circle be ‘rB’.

Equation of circle A:

or,

Let

Similarly, the equation of circle B:

Let

So, we can classify any datapoint (vector) as follows:

*I tested it out on paper by trying different test cases on two concentric circles to find the resultant sign.*

or (absorbing/replacing variables which are not as C’s,

or (absorbing/replacing variables which are not as D’s,

or (absorbing/replacing variables which are not as F’s,

or (absorbing/replacing variables which are not as G’s,

The above can be condensed further by taking common terms out of . Then it is of the form which can be represented by the feature map:

or equivalently with the kernel:

Thus, the kernel will recover a separator.

**Answer 2.4**

The labels for the datapoints is not given, so I’m assuming them to be the same as was discussed in the class, i.e.

A = ((1, 1), -1)

B = ((-1, 1), 1)

C = ((-1, -1), -1)

D = ((1, -1), 1)

This uses the representation: ((x1,x2), y), where is the datapoint and y is the label/output. We have four datapoints A, B, C and D.

And a correct classifier would be:

We will now solve and prove the above using the dual SVM

The dual SVM problem would be:

Expanding the above:

**Part 1:**

**For the Polynomial Kernel:**

and

Referencing

A = ((1, 1), -1)

B = ((-1, 1), 1)

C = ((-1, -1), -1)

D = ((1, -1), 1)

Now our objective function becomes:

or,

Applying  **[i=A,B,C,D]**, we get:

or (rearranging),

Solving by Inverse Matrix Method:

A.X=B

B

Therefore,

Therefore, all 4 inputs are support vectors, so optimum value of the

Hence,

or,

Referencing

and

A = ((1, 1), -1)

B = ((-1, 1), 1)

C = ((-1, -1), -1)

D = ((1, -1), 1)

or,

The first element of w\* is the bias:

The separating function is

or,

or,

or,

Hence we found the separator using the dual SVM.

The separators are the the axes. A point will be classified as negative in the 1st and 3rd quadrant and a point will be classified as positive in the 2nd and 4th quadrant.

**Part 2:**

**For the Gaussian-like (no sigma) Kernel:**

Referencing

A = ((1, 1), -1)

B = ((-1, 1), 1)

C = ((-1, -1), -1)

D = ((1, -1), 1)

After finding the differenence of the 2 vectors, we find its norm2 (so inner or dot product)

Our initial(general) objective function:

Now our objective function becomes:

or,

Applying  **[i=A,B,C,D]**, we get:

or (rearranging),

Solving by Inverse Matrix Method:

A.X=B

B

Therefore,

Therefore, all 4 inputs are support vectors, so optimum value of the

Hence,

or,

Referencing

and

A = ((1, 1), -1)

B = ((-1, 1), 1)

C = ((-1, -1), -1)

D = ((1, -1), 1)

or,

The first element of w\* is the bias:

The separating function is

or,

or,

or,

Hence we found the separator using the dual SVM.

**References:**

1. Lecture Notes and Videos.
2. <https://tohtml.com/python/>
3. <https://www.mathsisfun.com/algebra/circle-equations.html>
4. <https://www.mathwarehouse.com/ellipse/equation-of-ellipse.php>
5. <https://mathworld.wolfram.com/Ellipsoid.html>
6. <https://math.libretexts.org/Bookshelves/Algebra/Map%3A_College_Algebra_(OpenStax)/08%3A_Analytic_Geometry/8.02%3A_The_Ellipse#:~:text=Writing%20Equations%20of%20Ellipses%20Not%20Centered%20at%20the%20Origin,-Like%20the%20graphs&text=If%20an%20ellipse%20is%20translated,by%20(y%E2%88%92k)>

\*\*\*