a-i) Visualization of given data set (**Dataset id: 9-18-9**):

Feature 1 and Feature 2 of the given dataset are plotted against X-axis and Y-axis respectively. Colour *'red'* is used to mark -1 class while colour *'green'* is used to mark +1 class from the dataset. We can see that the data is divided into different classes by a curve.

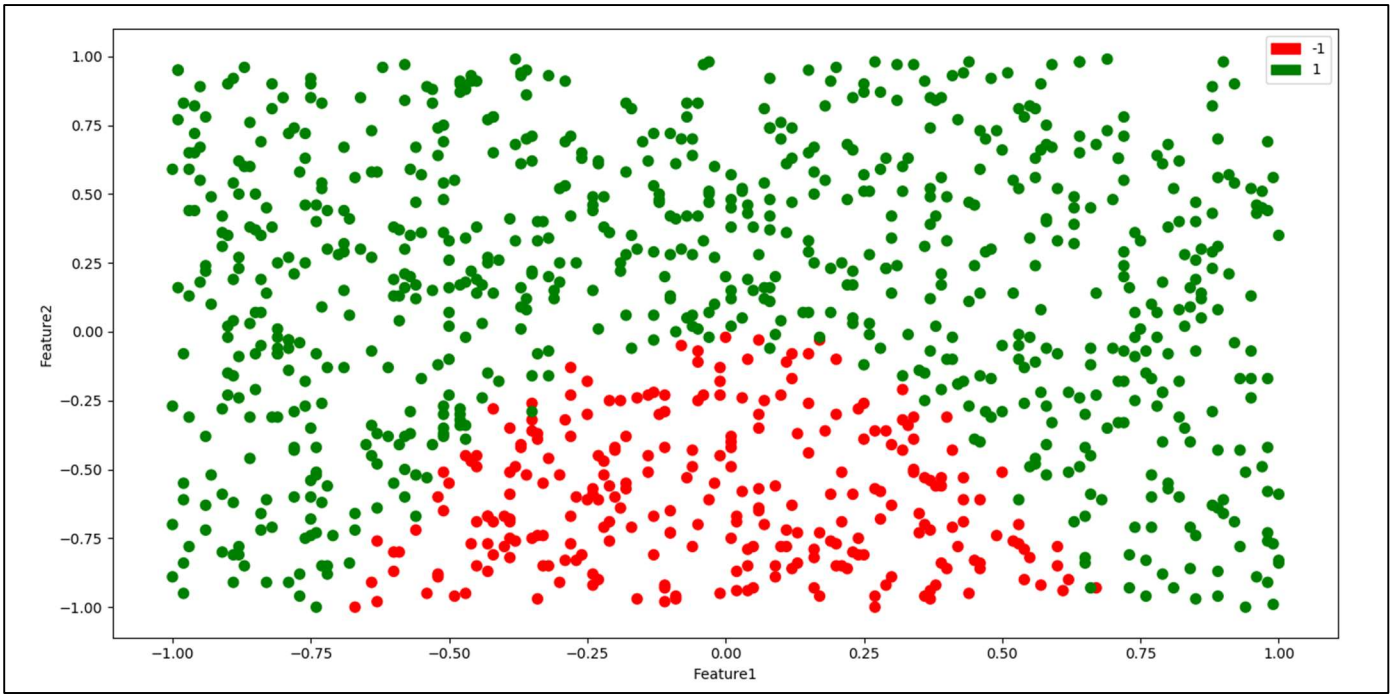

Figure 1: Raw data visualization

a-ii) Logistic Regression:

```
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression()
lr.fit(X,y)

# Model Parameters and score

print("Logistic Regression intercept: "+str(lr.intercept_))
print("Logistic Regression coefficients: "+str(lr.coef_))
print("Logistic Regression score: "+str(lr.score(X,y)))
```

Logistic Regression intercept: [2.06877543]
Logistic Regression coefficients: [[0.03936939 3.74296068]]
Logistic Regression score: 0.8118118118118118

We are using LogisticRegression model from the Sklearn libray and training the model with given dataset. Since we have 2 parameters, our model should be equivalent to $\mathbf{y = \theta_1 x_1 + \theta_2 x_2 + Intercept.}$ So as per the outputs $\mathbf{\theta_1}$ = 0.03936939 ; $\mathbf{\theta_2}$ = 3.74296068 ; **Intercept** = 2.06877543. Score 0.81 represents accuracy of the current model, i.e. the model predicts correct output for 81% of the datapoints given to it.

From the output we can clearly see $\mathbf{\theta_2 >> \theta_1}$ This means feature 2 is a bigger deciding factor when compared to feature 1.

a-iii) Prediction visualization:

The given dataset has been plotted with *'red'* colour for -1 class and *'green'* colour for +1 class. On top of it, predictions made by the model are also plotted with *'blue'* colour for -1 predicted class and *'yellow'* for +1 predicted class. The decision boundary is plotted in *'cyan'* colour.

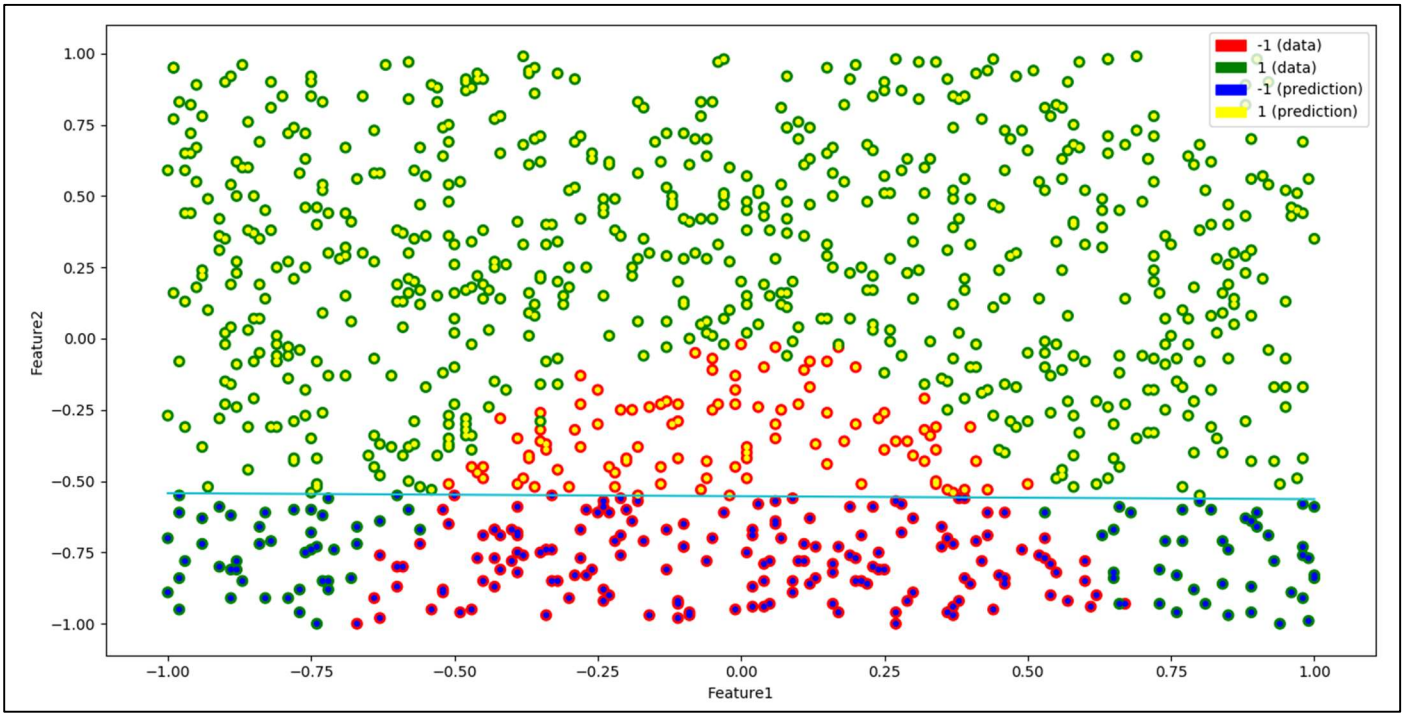

Figure 2: Visualization of predicted values

To obtain the line we can simply solve this equation **'$y = \theta_1 x_1 + \theta_2 x_2$ + Intercept'** mentioned above for x1= +1 and x1= -1 and get the values of x2. Then we can just plot a line passing through these two points.

```
point1= -(lr.intercept_[0]+(lr.coef_[0][0]))/(lr.coef_[0][1]) #x2 value when x1 is +1
point2= -(lr.intercept_[0]-(lr.coef_[0][0]))/(lr.coef_[0][1]) #x2 value when x1 is -1

plt.plot([1,-1],[point1,point2],'tab:cyan')
plt.show()
```

a-iv) Comparison with training data:

From the visualization we can see that the training data is divided into two classes by a quadratic curve but the model has simply divided the data into two parts by a straight line at x2 ≈ -0.5 and all data items above this point are classified as +1 and all those which are below are classified as -1.

b-i) SVM with different values of C:

In SVM we use hinge loss function ***max(0, 1 – y θ^T x)***. But this loss function can always be forced to 0 if **θ** is large enough, so penalty (**θ^T θ**) is introduced to get a proper behaviour from the model. The value of C is defined to increase or decrease the importance of penalty in the SVM cost function. So the Final SVM cost function = HingeLossFunction + (Penalty/C). From the equation we can see that a very large value of C will decrease the penalty factor and vice-versa.

After training Linear SVM model for the given dataset on different values of C, following parameters were obtained:

Table 1: SVM parameters for different values of C

C	θ_1	θ_2	Intercept	Score
0.001	-0.01	0.35	0.33	0.75
0.1	0.02	1.29	0.69	0.81
1	0.02	1.38	0.73	0.81
100	0.03	1.40	0.69	0.82
1000	1.40	0.92	0.24	0.65

Analysis of this table and the impact of C on the model is discussed in part b-iii)

b-ii) Prediction visualization for all models:

Similar to above visualizations baseline data and prediction data are plotted on the same chart along with the decision boundary line in 'cyan' colour.

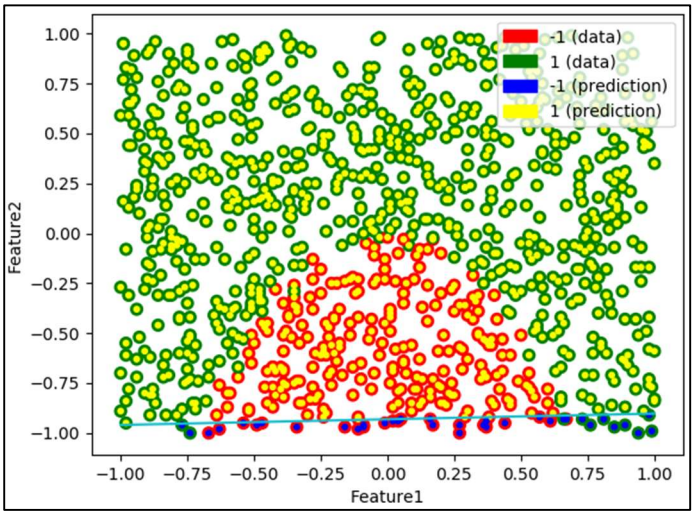

Figure 3: SVM with C=0.001

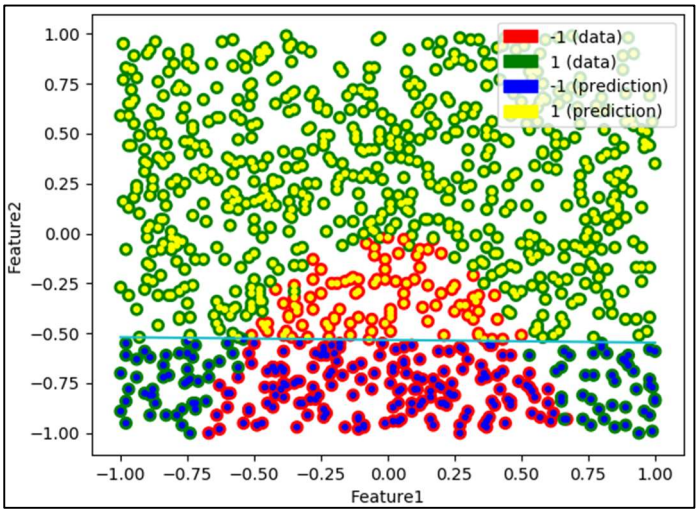

Figure 4: SVM with C=0.1

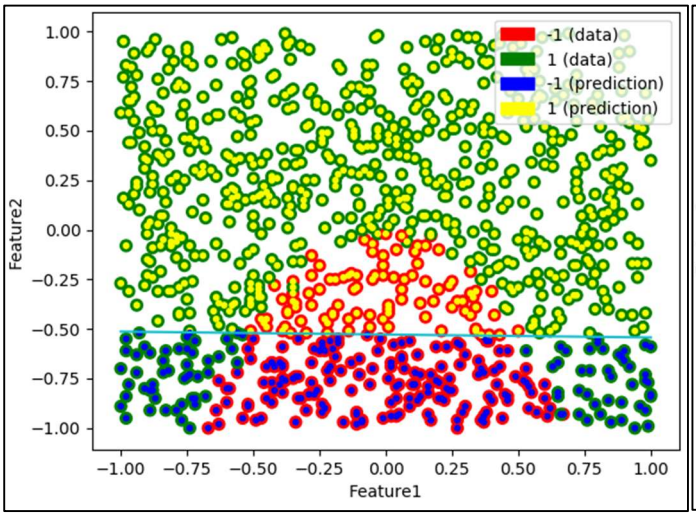

Figure 5: SVM with C=1

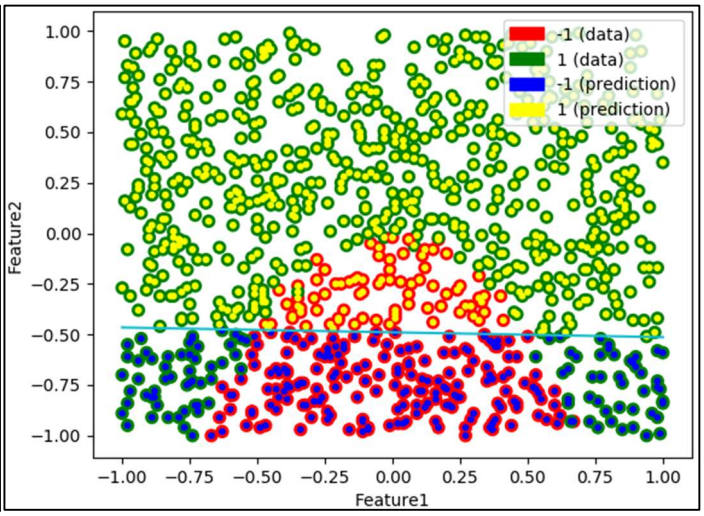

Figure 6: SVM with C=100

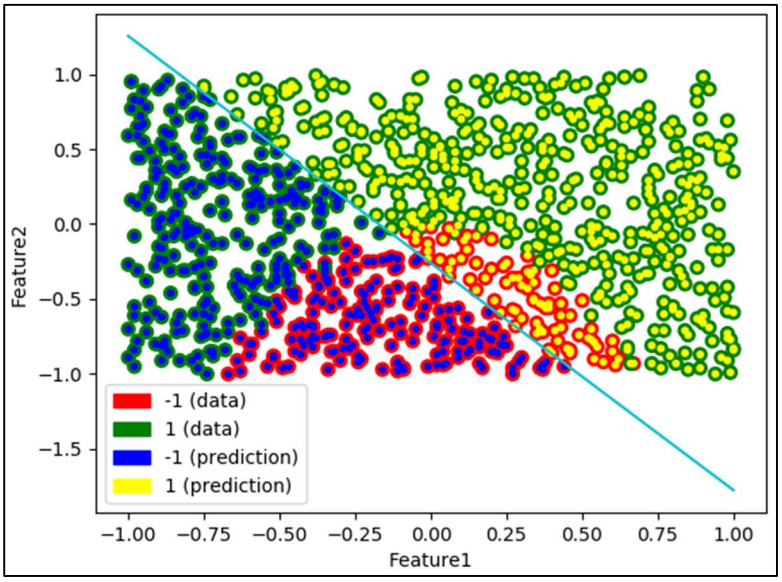

Figure 7: SVM with C=1000

b-iii) Impact of C

Based on the values provided in Table 1 and figures above, we can infer that the accuracy of the model is good when values of C is close to 1. For C=0.001 the penalty carries too much importance and values of **θ** are not able to scale up properly resulting in poor accuracy score. We can see this in Figure 3, almost all data points are predicted in class +1 and model is not able to predict properly. On the other hand for C=1000 , the penalty has too less importance and we can see **θ** values become too large but accuracy is decreased. The Figure 7 shows the abnormal behaviour of the model due to high value of C.

b-iv) Comparison with Logistic Regression in part a

The SVM model is fairly similar to the Logistic Regression model in part a when the values of C are close to 1. We can compare them based on accuracy score. For both the models, accuracy is approximately close to 81% for the right value of C, which can be confirmed by looking at the decision boundaries of both the models provided in the plots above. But for very small / very large values of C the SVM model drops accuracy and loses the underlying pattern in data, so it is important to have correct values of C when using SVM.

c-i) Logistic Regression with additional squared features

```
X1_sq=np.square(df.iloc[:,0]) # Square of first parameter
X2_sq=np.square(df.iloc[:,1]) # Square of second parameter

X_inputs=np.column_stack((X1,X2,X1_sq,X2_sq))

# Train the Logistic Regression
lr_sq = LogisticRegression(penalty='none')
lr_sq.fit(X_inputs,y)

print("Squared Logistic Regression intercept: "+str(lr_sq.intercept_))
print("Squared Logistic Regression coefficients: "+str(lr_sq.coef_))
print("Squared Logistic Regression score: "+str(lr_sq.score(X_inputs,y)))
```

Squared Logistic Regression intercept: [0.79755297]
Squared Logistic Regression coefficients: [[1.2249378 44.61868094 77.33647427 6.91293073]]
Squared Logistic Regression score: 0.986986986986987

After using the LogisticRegression with additional squared features our model should be equivalent to **$y = \theta_1 x_1 + \theta_2 x_2 + \theta_3 (x_1)^2 + \theta_4 (x_2)^2$ + Intercept.** So as per the outputs θ_1 = *1.2249378* ; θ_2 = *44.61868094* ; θ_3 = *77.33647427* ; θ_4 = *6.91293073*; **Intercept** = *0.79755297*. Score 0.98 represents accuracy of the current model, i.e. the model predicts correct output for 98% of the datapoints given to it.

c-ii) Visualization and comparison

Similar to above plots we have depicted baseline data and the prediction data on the same graph along with the decision boundary.

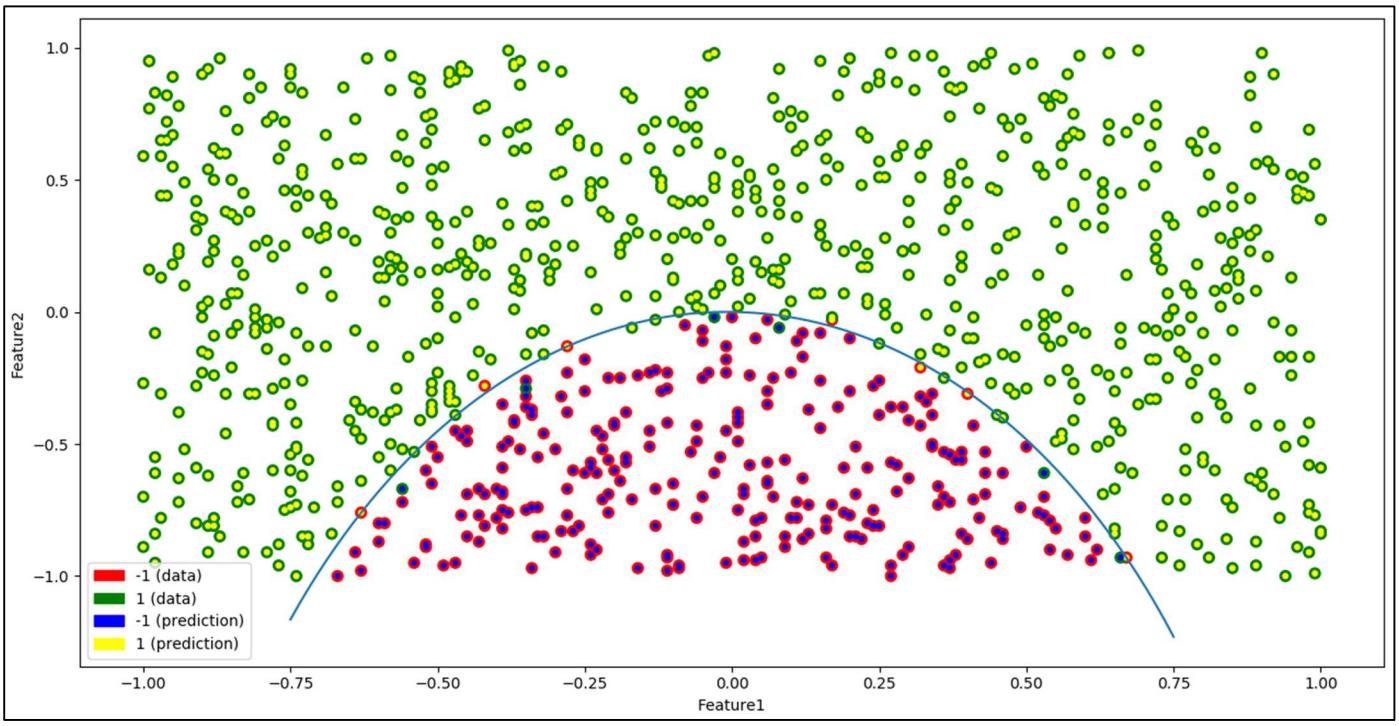

Figure 8: Logistic Regression with squared inputs

Comparing with the models in part a and part b we can infer that addition of the squared inputs to our model training has improved the results drastically. We can see θ_2 = *44.61* ; θ_3 = *77.33* are significantly higher than θ_1 = *1.22* and θ_4 = *6.91* . This suggests that (x_2) and $(x_1)^2$ are co-related to each other in such a way that the data points can be classified into +1 and -1 in a linearly separable way. This why the accuracy has jumped up to 98% as compared to 80% in the models in part a and part b, as these models did not have the squared features (or linearly separable features) as their inputs.

In the figure below we have plotted baseline data with $(x_1)^2$ on the X axis and (x_2) on the Y axis. Class -1 is represented in *'red'* and class +1 is represented in *'green'*. From the figure we can clearly see that we can draw a straight line to separate +1 and -1 classes from the dataset. That's why these two inputs are the deciding factor for the new model.

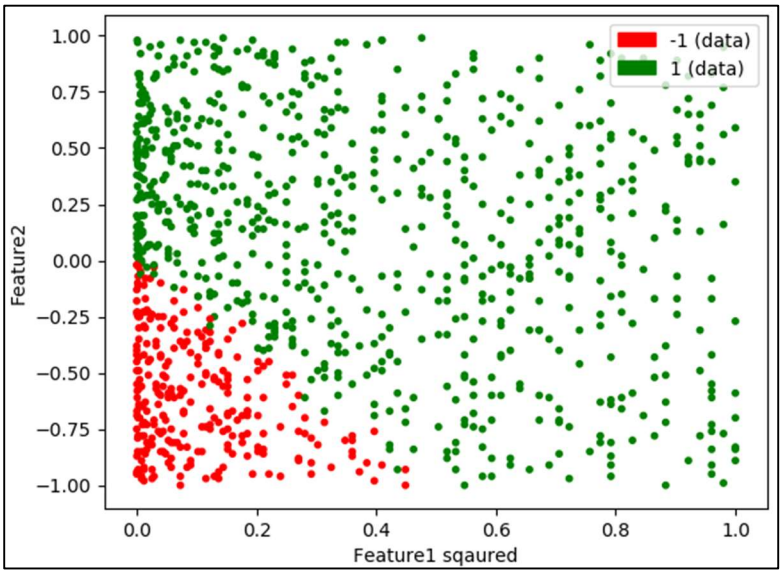

Figure 9: Plot of (feature1)squared against feature2

c-iii) Comparison with a baseline model:

If we create a baseline model which always predict most common class, the accuracy of that model for the given data set would be (count of common class) / (total count). For the given data set this turns out to be 737/999=0.73 This means even a dumb model has an accuracy of **73%**. When we compare this with models in part a and part b which have accuracy close to 81% this does not seem to be a significant increase from the baseline model. But comparing it with accuracy of **98% for model in part c** we can say model c is much more effective than the baseline model.

c-iv) Decision boundary:

The model represents equation $\mathbf{y = \theta_1 x_1 + \theta_2 x_2 + \theta_3 (x_1)^2 + \theta_4 (x_2)^2 + Intercept}$

If we substitute some values of x1 we can solve the quadratic equation to find the corresponding values of x2. We can compare this with $\mathbf{ax^2+bx+c = 0}$ and find the root by using $\mathbf{(-b + sqrt\ (b^2 – 4ac))/2a}$ and then plot the curve.

```
x1a = np.linspace(-0.75,0.75,100)   # Random x1 values from -0.75 to 0.75

# comparing with a*x*x + b*x + c = 0
a=lr_sq.coef_[0][3]
b=lr_sq.coef_[0][1]

x2a = []
x2b = []

# find values of c and solve for x2
for k in x1a:
    c=( (lr_sq.coef_[0][0]*k) + (lr_sq.coef_[0][2]*k*k))
    tt =np.absolute(((b*b) - (4*a*c)))
    root1 = (-b + np.sqrt(tt))/(2*a)
    x2a.append(root1)

plt.plot(x1a,x2a)
plt.show()
```

APPENDIX

- Code referred from lecture slides and sklearn, matplotlib, numpy api documentation.

Python code:

```
# Name: Omkar Pramod Padir
# Student Id: 20310203
# Dataset id:9-18-9
# Course: Machine Learning CS7CS4


import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import matplotlib.patches as mpatches

# function to plot given data
def PlotBaselineData():
    plt.xlabel('Feature1')
    plt.ylabel('Feature2')

    plt.scatter(X1, X2, 50, y, cmap=cmap)


    plt.legend(handles=[red_patch, green_patch])

cmap, norm = mcolors.from_levels_and_colors([-1, 0, 1], ['red', 'green'])
red_patch = mpatches.Patch(color='red', label='-1 (data)')
green_patch = mpatches.Patch(color='green', label='1 (data)')

# Part A starts here

# Load data and create arrays of input and output

df = pd.read_csv("ML_W2_DATA.csv")

X1=df.iloc[:,0]
X2=df.iloc[:,1]
X=np.column_stack((X1,X2))
y=df.iloc[:,2]

# Plot the baseline data

PlotBaselineData()
plt.show()

# Create and fit Logistic Regression Model

from sklearn.linear_model import LogisticRegression

lr = LogisticRegression()
lr.fit(X,y)

# Model Parameters and score

print("Logistic Regression intercept: "+str(lr.intercept_))
print("Logistic Regression coefficients: "+str(lr.coef_))
print("Logistic Regression score: "+str(lr.score(X,y)))

PlotBaselineData()
cmap2, norm = mcolors.from_levels_and_colors([-1,0,1], ['blue', 'yellow'])
blue_patch = mpatches.Patch(color='blue', label='-1 (prediction)')
yellow_patch = mpatches.Patch(color='yellow', label='1 (prediction)')
plt.legend(handles=[red_patch,green_patch,blue_patch,yellow_patch])
plt.scatter(X1, X2, 10, lr.predict(X), cmap=cmap2, marker="o")

# Solve the equation to get x2 values by using coefficient and intercept of the model

point1= -(lr.intercept_[0]+(lr.coef_[0][0]))/(lr.coef_[0][1]) #x2 value when x1 is +1
point2= -(lr.intercept_[0]-(lr.coef_[0][0]))/(lr.coef_[0][1]) #x2 value when x1 is -1

plt.plot([1,-1],[point1,point2],'tab:cyan')
plt.show()


# Part B starts here


from sklearn.svm import LinearSVC

# Train SVM models for different values of C

lsvc_001=LinearSVC(C=0.001).fit(X,y)
```

```
lsvc_point1=LinearSVC(C=0.1).fit(X,y)
lsvc_1=LinearSVC(C=1).fit(X,y)
lsvc_100=LinearSVC(C=100).fit(X,y)
lsvc_1000=LinearSVC(C=1000).fit(X,y)

svc_arr=[lsvc_001,lsvc_point1,lsvc_1,lsvc_100,lsvc_1000]
svc_name=['SVM, C=0.001','SVM, C=0.1','SVM, C=1','SVM, C=100','SVM, C=1000']

for i in range(5):
    print("For model "+svc_name[i])
    print("Intercept: " + str(svc_arr[i].intercept_))
    print("Coefficients: " + str(svc_arr[i].coef_))
    print("Score: " + str(svc_arr[i].score(X,y)))

# function to plot prediction data points of the model
def PlotSVMData(model):
    plt.scatter(X1, X2, 10, model.predict(X), cmap=cmap2, marker="o")
    plt.legend(handles=[red_patch,green_patch,blue_patch, yellow_patch])

# function to plot decision boundary of the model
def PlotLine(lr):
    point1 = -(lr.intercept_[0] + (lr.coef_[0][0])) / (lr.coef_[0][1])  # x2 value when x1 is +1
    point2 = -(lr.intercept_[0] - (lr.coef_[0][0])) / (lr.coef_[0][1])  # x2 value when x1 is -1

    plt.plot([1, -1], [point1, point2], 'tab:cyan')

# plot all svm models
for m in svc_arr:
    PlotBaselineData()
    PlotSVMData(m)
    PlotLine(m)
    plt.show()


# Part C starts here

X1_sq=np.square(df.iloc[:,0]) # Square of first parameter
X2_sq=np.square(df.iloc[:,1]) # Square of second parameter

X_inputs=np.column_stack((X1,X2,X1_sq,X2_sq))

# Train the Logistic Regression
lr_sq = LogisticRegression(penalty='none')
lr_sq.fit(X_inputs,y)

print("Squared Logistic Regression intercept: "+str(lr_sq.intercept_))
print("Squared Logistic Regression coefficients: "+str(lr_sq.coef_))
print("Squared Logistic Regression score: "+str(lr_sq.score(X_inputs,y)))

PlotBaselineData()
plt.scatter(X1, X2, 10, lr_sq.predict(X_inputs), cmap=cmap2, marker="o")
plt.legend(handles=[ red_patch,green_patch,blue_patch, yellow_patch])

# plot decision boundary
x1a = np.linspace(-0.75,0.75,100)   # Random x1 values from -0.75 to 0.75

# comparing with a*x*x + b*x + c = 0
a=lr_sq.coef_[0][3]
b=lr_sq.coef_[0][1]

x2a = []
x2b = []

# find values of c and solve for x2
for k in x1a:
    c=( (lr_sq.coef_[0][0]*k) + (lr_sq.coef_[0][2]*k*k))
    tt =np.absolute(((b*b) - (4*a*c)))
    root1 = (-b + np.sqrt(tt))/(2*a)
    x2a.append(root1)

plt.plot(x1a,x2a)
plt.show()

# Plot of x1*x1 against x2. Just for reference
plt.xlabel('Feature1 sqaured')
plt.ylabel('Feature2')
plt.scatter(X1_sq, X2, 10, y, cmap=cmap)
plt.legend(handles=[red_patch,green_patch])
plt.show()

# Comparison with baseline predictor
count_p1=0
count_m1=0
total=0
for t in y:

    if(t==-1):
        count_m1=count_m1+1
```

```
    else:
        count_p1=count_p1+1
    total=total+1

print(count_p1 , count_m1)
if count_p1 > count_m1:
    print("Accuracy of baseline predictor: "+str(count_p1/total))
else:
    print("Accuracy of baseline predictor: " + str(count_m1/total))
```