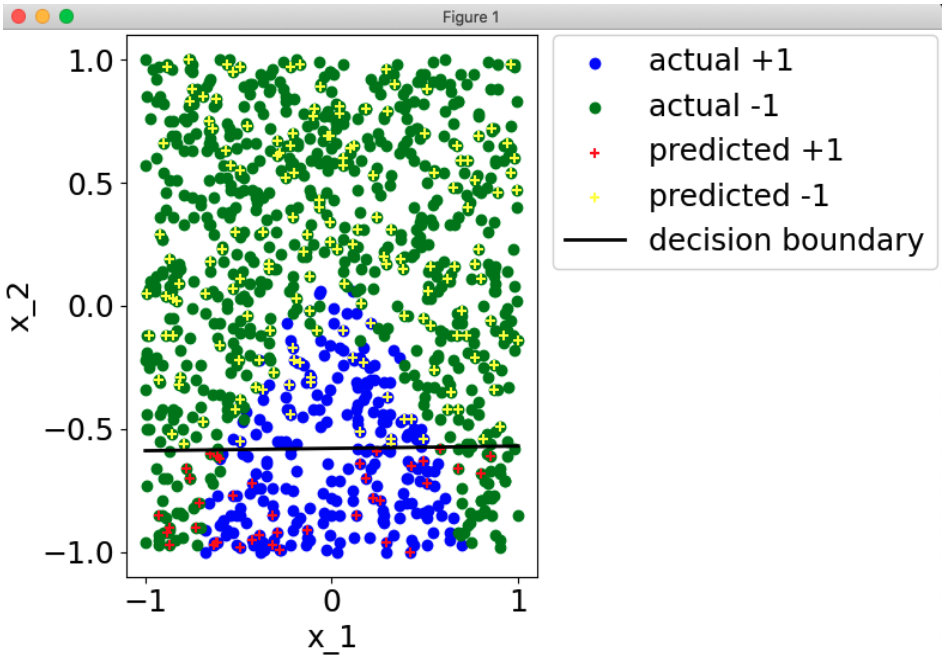
Machine Learning Assignment Week 2

Name: Masanari Doi

Student number: 19313167

Q (a)

(i)



(Figure 1)

```
for row in range(len(df)):
   if df.values[row][2] == 1:
       x1RealPlus.append(df.values[row][0])
       x2RealPlus.append(df.values[row][1])
   else:
       x1RealMinus.append(df.values[row][0])
       x2RealMinus.append(df.values[row][1])
```

In order to visualise the data, a data file is obtained as df. Then, in this for-loop, it checks whether the x1 and x2 have a value of +1 or -1. If the 2 features have +1, they are added to matrices, x1RealPlus and x2RealPlus, but if they do not, x1RealMinus and x2RealMinus added these 2 features. For example, if the x1 is -0.13, x2 is -0.19 and the target value is +1, x1 is added to x1Realplus and x2 goes to x2RealPlus. This for-loop checks it for all rows.

```
plt.scatter(x1RealPlus, x2RealPlus, color="blue", label="actual +1")
plt.scatter(x1RealMinus, x2RealMinus, color="green", label="actual -1")
```

After this for-loop, these arrays can be plotted in a graph in a way that x1RealPlus and x1RealMinus are set to x values and, x2RealPlus and x2RealMinus are set to y values as figure 1 shows. If the target value is +1, the colour is blue and if not it is green. (This graph already has multiple plots to reduce the number of figures and save spaces.)

(ii)

```
xData = df.iloc[:,0:2]
resultData = df.iloc[:,2]
```

xData = df.iloc[:,0:2] is used to construct a 999x2 matrix which contains only features, x1 and x2. In the same way, by resultData = df.iloc[:,2] is for 999x2 matrix containing target values, +1 or -1.

```
xTrain, xTest, resultTrain, resultTest = train_test_split(xData, resultData,
test_size=0.2)
```

Then, the code above shows that the data is split for training and testing. test_size=0.2 means that 80% of the data is used for training and the rest is for testing.

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression()
model.fit(xTrain, resultTrain)                          # train data
print("slope = ", model.coef_)                          # get a slope here
print("intercept = ", model.intercept_)
```

After this, model = LogisticRegression() and model.fit(xTrain, resultTrain) are used to train a logistic regression classifier on the data. The parameter values are as follows.

```
Q(a) (ii)
slope =  [[ 0.03244979 -3.42049534]]
intercept =  [-1.98299892]
```

It can be said that the parameter values of the trained model $y = \theta0 + \theta1x1 + \theta2x2$ are $\theta0 = -1.98299892$, $\theta1 = 0.03244979$ and $\theta2 = -3.42049534$. Since the absolute value of $\theta2$ is larger than $\theta1$, feature x2 has greater weight than x1. Moreover, $\theta2$ has a negative value so if if x2 is increased, the possibility that the target value becomes -1 increases. If decreased, the value is more likely to be +1.

(iii)

```
predData = np.array(model.predict(xTestArray))
```

predData = np.array(model.predict(xTestArray)) is used to get a prediction data.(xTestArray is an array of x test data.) In the same way as (i), the matrices for x values and y values are generated, then the predictions can be plotted to the graph. As figure 1, the predictions are illustrated as red or yellow with + marker.

```
def get_a_decisionBoundary(coefficient, intercept):
coef = np.array(coefficient)
coef = coef.reshape(-1,1)
x = np.linspace(-1, 1, 5)
y = -((coef[0]*x)+intercept)/coef[1]
return x, y
```

Model: $h\theta(x) = \text{sign}(\theta Tx)$
In order to get a decision boundary, it is necessary to get $\theta Tx = 0$.
In this case, $\theta Tx = \theta0 + \theta1x1 + \theta2x2$.
Therefore, we can get an equation $\theta0 + \theta1x1 + \theta2x2 = 0 \rightarrow x2 = -(\theta1x1 + \theta0)/\theta2$.
In my code, y = -((coef[0]*x)+intercept)/coef[1] represents $x2 = -(\theta1x1 + \theta0)/\theta1$. i.e.
y = x2, x = x1, coef[0] = $\theta1$, coef[1] = $\theta2$ and intercept = $\theta0$. In this situation, according to the output, slope is [[0.0099641 -1.26770045]] and intercept is [-0.70349456].
Hence, my code gets the decision boundary as
x2 = - (0.0099641x1 - 0.70349456)/(-1.26770045), and plots it as Figure 1 shows by a black line.

(iv)

It can be seen that there are some wrong predictions because it has a linear decision boundary. As parameter values are calculated on (ii), only x2 values look an important cause for target values although the actual target values seem to have a quadratic decision boundary.

Q (b)

(i)

```
from sklearn.svm import LinearSVC
model = LinearSVC(C=c).fit(xTrain, resultTrain)
print("when C =", c)
print("slope = ", model.coef_)
print("intercept = ", model.intercept_)
```

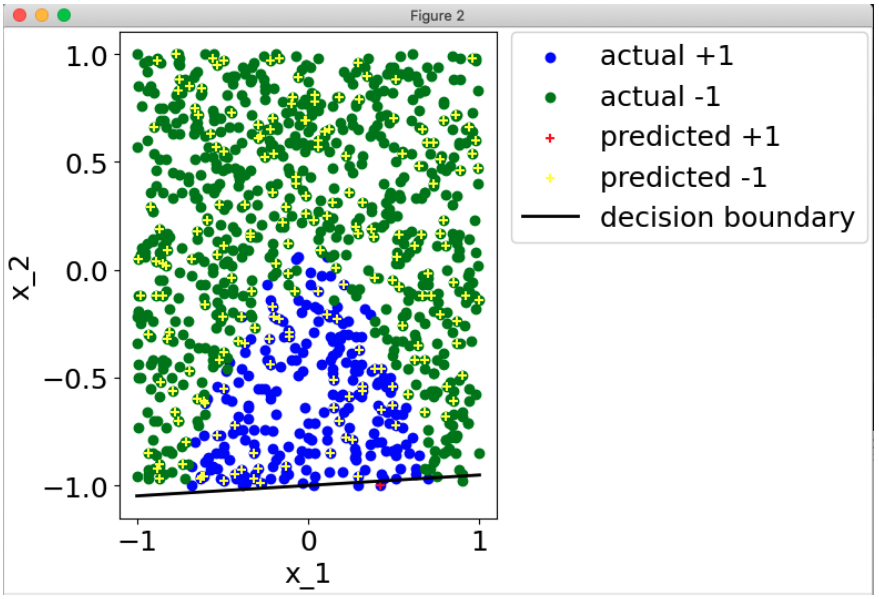
model = LinearSVC(C=c).fit(xTrain, resultTrain) is used to train linear SVM classifiers with input c. Outputs of this code are as follows.

when C = 0.001
slope = [[0.01500518 -0.30962561]]
intercept = [-0.30978538]

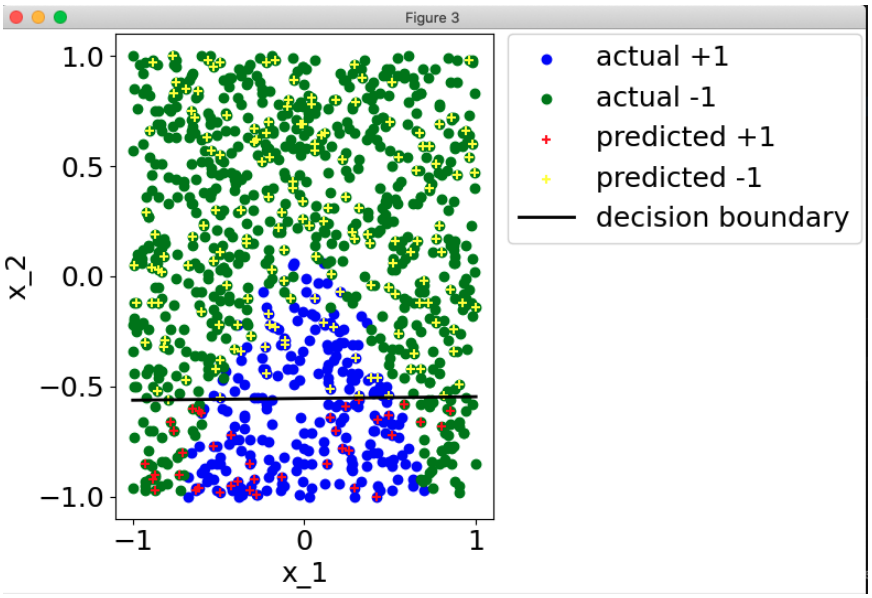
when C = 1.0
slope = [[0.0099641 -1.26770045]]
intercept = [-0.70349456]

when C = 100
slope = [[0.06429792 -1.23879906]]
intercept = [-0.72382737]
For this SVM model, model is $h\theta(x) = sign(\theta Tx)$
It can be said that by applying these outputs to θTx = y = θ0 + θ1x1 + θ2x2, each equations are,
y = -0.30978538 + 0.01500518x1 -0.30962561x2 (C = 0.001)
y = -0.70349456 + 0.0099641x1 -1.26770045x2 (C = 1.0)
y = -0.72382737 + 0.06429792x1 -1.23879906x2 (C = 100)

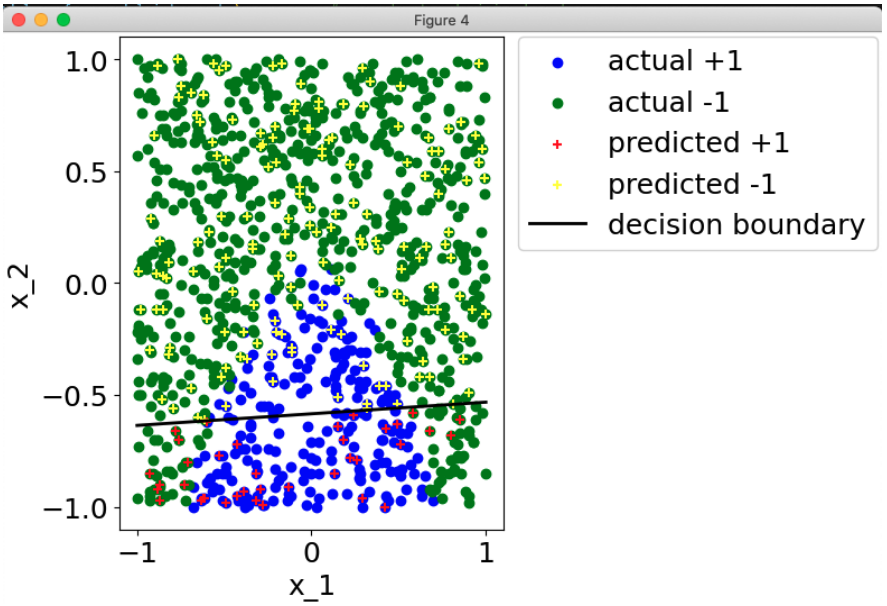
(ii)
Figure 2 for C = 0.001, Figure 3 for C = 1.0 and Figure 4 for C = 100.



(Figure 2)



(Figure 3)



(Figure 4)

(iii)

Large value of C makes penalty less important because making the $\theta T\theta/C$ penalty too weak. In the same way, choosing a too small value for C can be a cause of the prediction error because it makes the $\theta T\theta/C$ penalty too strong.

(iv)

Compared to (a), especially when C = 1.0 and C = 100, predictions and decision boundaries are similar. It can be because the only difference between logistic regression and SVM is cost function.

Q (c)

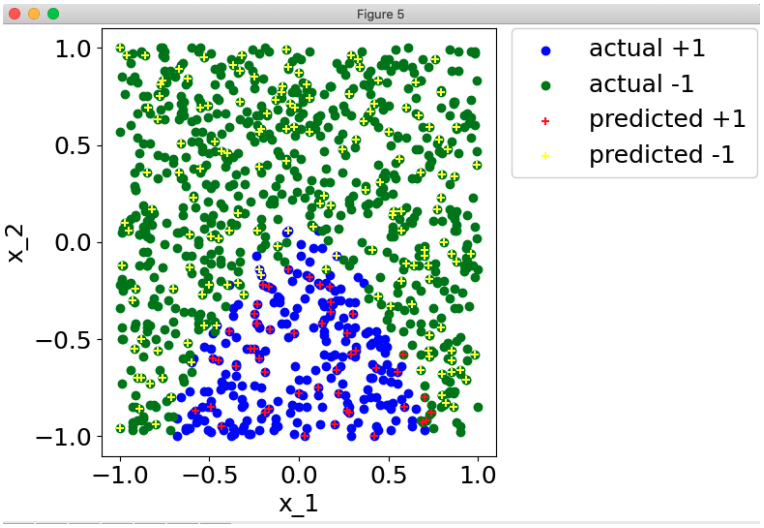
(i)

```
    for row in range(len(xData)):
        newFeature1 = tempXData[row][0]*tempXData[row][0]
        newFeature2 = tempXData[row][1]*tempXData[row][1]
        xData = np.insert(xData, fCul, newFeature1)
        xData = np.insert(xData, secCul, newFeature2)
        fCul = fCul +4
        secCul = secCul + 4
xData = xData.reshape(-1, 4)
```

The code above adds squared original values to the data set by using np.insert().
Then xData = xData.reshape(-1, 4) is used to make a 999x4 array of x data that includes new squared features.

(ii)

Compared to (a) and (b), the predictions are more accurate in the way that x1 features now seem to be considered as an important factor of deciding target values.



(Figure 5)

(iii)

Adding new features as squared values is more accurate. For example, always predicting the most common class can be a cause of an unclear prediction. For instance, if the weather forecast had said tomorrow's weather would be rainy and 55% of the day it rained, it can be said that it is a baseline to fulfil.

Appendix

```
from cProfile import label
from re import X
# from statistics import LinearRegression
from tkinter import Y
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.svm import LinearSVC
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

df = pd.read_csv('/Users/doimasanari/Desktop/datasetMasanariDoi.csv')

x1RealPlus = []
x2RealPlus = []

x1RealMinus = []
x2RealMinus = []

xData = df.iloc[:,0:2]     # construct a 999x2 matrix which consists only x1 and x2
resultData = df.iloc[:,2]  # construct a 999x1 matrix which consists only values 1 or -1

for row in range(len(df)):    # this for-loop distinguishes x1 and x2 according to its value, 1 or
-1.

  if df.values[row][2] == 1:
     x1RealPlus.append(df.values[row][0])
     x2RealPlus.append(df.values[row][1])
  else:
     x1RealMinus.append(df.values[row][0])
     x2RealMinus.append(df.values[row][1])


xTrain, xTest, resultTrain, resultTest = train_test_split(xData, resultData, test_size=0.2) #
split the data for training and testing.

xTestArray = np.array(xTest) #make an array of x test data

def get_a_decisionBoundary(coefficient, intercept):
  coef = np.array(coefficient)           #
  coef = coef.reshape(-1,1)              # make a coefficient array readable
  x = np.linspace(-1, 1, 5)              #
  y = -((coef[0]*x)+intercept)/coef[1]    # x2 = - (θ1x1+ θ0)/θ1
  return x, y

def get_a_predData(model, xTestArray):              # get a prediction data in this function
  predData = np.array(model.predict(xTestArray))
```

```
  predData = predData.reshape(-1,1)              # make a tidy 999x1 array of prediction data
which contains values, 1 or -1
  return predData

def get_a_prediction_array(xTestArray, predData):  # distinguish x1 and x2 of test data
according to its value, 1 or -1,
                                    # in order to make an array of x predicted data
  x1PredPlus = []
  x2PredPlus = []
  x1PredMinus = []
  x2PredMinus = []

  for row in range(len(xTestArray)):

     if predData[row][0] == 1:                    #if x1 and x2 are predicted to have +1
        x1PredPlus.append(xTestArray[row][0])
        x2PredPlus.append(xTestArray[row][1])
     else:
        x1PredMinus.append(xTestArray[row][0])     #if x1 and x2 are predicted to have -1
        x2PredMinus.append(xTestArray[row][1])

  return x1PredPlus, x2PredPlus, x1PredMinus, x2PredMinus

def get_a_graph(x, y, x1PredPlus, x2PredPlus, x1PredMinus, x2PredMinus, num): # make a
graph here

  plt.figure(num)
  plt.rc('font', size=18)
  plt.rcParams["figure.constrained_layout.use"] = True
  plt.scatter(x1RealPlus, x2RealPlus, color="blue", label="actual +1")
  plt.scatter(x1RealMinus, x2RealMinus, color="green", label="actual -1")
  plt.scatter(x1PredPlus, x2PredPlus, color="red", marker="+", label = "predicted +1")
  plt.scatter(x1PredMinus, x2PredMinus, color="yellow", marker="+", label = "predicted
-1")
  if(num != 5):
     plt.plot(x, y, color="black", linewidth = 2, label = "decision boundary")
  plt.xlabel("x_1")
  plt.ylabel("x_2")
  plt.legend(bbox_to_anchor=(1.04, 1), borderaxespad=0)

def logisticRegression(xTrain, resultTrain, xTestArray, num, questionNum): # train data by
logistic Regression

  model = LogisticRegression()
  model.fit(xTrain, resultTrain)                              # train data
  print("slope = ", model.coef_)                  # get a slope here
  print("intercept = ", model.intercept_)            # get an intercept here
  # print("train score = ", format(model.score(xTrain, resultTrain)))
  if(questionNum == 1):
```

```
      x, y = get_a_decisionBoundary(model.coef_ , model.intercept_)      # go and get a
decision boundary
   else :
      x = 0
      y = 0
   predData = get_a_predData(model, xTestArray)                          # go and get a prediction
data
   x1PredPlus, x2PredPlus, x1PredMinus, x2PredMinus = get_a_prediction_array(xTestArray,
predData) # go and get arrays of predited x data
   get_a_graph(x, y, x1PredPlus, x2PredPlus, x1PredMinus, x2PredMinus, num) # go and get
a graph


def liner_SVC (c, xTrain, resultTrain, num):                             # train data by
logisticRegression

   model = LinearSVC(C=c).fit(xTrain, resultTrain)                       # train a data
   print("when C =", c)
   print("slope = ", model.coef_)                                    # get a slope here
   print("intercept = ", model.intercept_)                             # get an intercept here
   x, y = get_a_decisionBoundary(model.coef_ , model.intercept_)          # go and get a
decision boundary
   predData = get_a_predData(model, xTestArray)                          # go and get a prediction
data
   x1PredPlus, x2PredPlus, x1PredMinus, x2PredMinus = get_a_prediction_array(xTestArray,
predData)  # go and get arrays of predited x data
   get_a_graph(x, y, x1PredPlus, x2PredPlus, x1PredMinus, x2PredMinus, num)  # go and get
a graph

def dummy_data ():    # dummy graph data. close the graph window then check an actual
graph

   plt.figure(0)
   plt.rc('font', size=18)
   plt.rcParams["figure.constrained_layout.use"] = True
   plt.plot(0, 0, color="black", linewidth = 2, label = "this is dummy data please close this
window and see actual graphs")
   plt.xlabel("x_1")
   plt.ylabel("x_2")
   plt.legend(bbox_to_anchor=(1.04, 1), borderaxespad=0)


dummy_data() #this is dummy data. plese close this graph and see an actual graph

# for question (a) and (b)
print("Q(a) (ii)")
logisticRegression(xTrain, resultTrain, xTestArray, 1, 1) #this is for question (a)

print("Q(b) (i)")
liner_SVC (0.001, xTrain, resultTrain, 2) # these three linear_SVC are answering question (b)
```

```
liner_SVC (1.0, xTrain, resultTrain, 3)
liner_SVC (100, xTrain, resultTrain, 4)

# below is for question (c)
def new_training(xData): # this function makes new training data, i.e. adding the square of each feature.

  xData = np.array(xData)
  tempXData = xData
  fCul = 2
  secCul = 3

  for row in range(len(xData)):
      newFeature1 = tempXData[row][0]*tempXData[row][0] # making square of original two features
      newFeature2 = tempXData[row][1]*tempXData[row][1]
      xData = np.insert(xData, fCul, newFeature1)       # adding new squared features
      xData = np.insert(xData, secCul, newFeature2)
      fCul = fCul +4
      secCul = secCul + 4

  xData = xData.reshape(-1, 4)                          # make a 999x4 array of x data that includes new squared features.
  xTrain, xTest, resultTrain, resultTest = train_test_split(xData, resultData, test_size=0.2) # split new data for training and testing

  return xData, xTrain, xTest, resultTrain, resultTest

xData, xTrain, xTest, resultTrain, resultTest = new_training(xData)
xTestArray = np.array(xTest)
print("Q(c) (i)")
logisticRegression(xTrain, resultTrain, xTestArray, 5, 2) # answering question (c)

plt.show()
```