SML Assignment 3

October 27, 2019

[87]:
```python
import pandas as pd
import numpy as np
from sklearn.model_selection import KFold
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import metrics
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.preprocessing import StandardScaler
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.tree import DecisionTreeClassifier
import math
import operator
import random
from sklearn.model_selection import KFold
import warnings
warnings.filterwarnings('ignore')
```

[3]:
```python
data_spam = pd.read_csv("C:/Users/mouni/Downloads/SML/HW3/spambase/spambase.
↪data")
```

[4]:
```python
data_spam = data_spam.rename(columns={'0':'word_freq_make','0.64':
↪'word_freq_address',
                                      '0.64.1':'word_freq_all','0.1':
↪'word_freq_3d',
                                      '0.32':'word_freq_our','0.2':
↪'word_freq_over',
                                      '0.3':'word_freq_remove','0.4':
↪'word_freq_internet',
                                      '0.5':'word_freq_order','0.6':
↪'word_freq_mail',
                                      '0.7':'word_freq_receive','0.64.2':
↪'word_freq_will','0.8':'word_freq_people',
```

```
                                        '0.9':'word_freq_report','0.10':
↪'word_freq_addresses','0.32.1':'word_freq_free',
                                        '0.11':'word_freq_business','1.29':
↪'word_freq_email',
                                        '1.93':'word_freq_you','0.12':
↪'word_freq_credit','0.96':'word_freq_your',
                                        '0.13':'word_freq_font','0.14':
↪'word_freq_000','0.15':'word_freq_money',
                                        '0.16':'word_freq_hp','0.17':
↪'word_freq_hpl','0.18':'word_freq_george',
                                        '0.19':'word_freq_650','0.20':
↪'word_freq_lab','0.21':'word_freq_labs',
                                        '0.22':'word_freq_telnet','0.23':
↪'word_freq_857','0.24':'word_freq_data',
                                        '0.25':'word_freq_415','0.26':
↪'word_freq_85','0.27':'word_freq_technology',
                                        '0.28':'word_freq_1999','0.29':
↪'word_freq_parts','0.30':'word_freq_pm',
                                        '0.31':'word_freq_direct','0.32.2':
↪'word_freq_cs','0.33':'word_freq_meeting',
                                        '0.34':'word_freq_original','0.35':
↪'word_freq_project','0.36':'word_freq_re',
                                        '0.37':'word_freq_edu','0.38':
↪'word_freq_table','0.39':'word_freq_conference',
                                        '0.40':'char_freq_semicolon','0.41':
↪'char_freq_leftparen',
                                        '0.42':'char_freq_leftsquare','0.778':
↪'char_freq_bang',
                                        '0.43':'char_freq_dollar','0.44':
↪'char_freq_hash',
                                        '3.756':'capital_run_length_average','61':
↪'capital_run_length_longest',
                                        '278':'capital_run_length_total','1':
↪'is_spam'})
```

[5]:
```
data_spam.head()
```

[5]:
```
   word_freq_make  word_freq_address  word_freq_all  word_freq_3d  \
0            0.21               0.28           0.50           0.0
1            0.06               0.00           0.71           0.0
2            0.00               0.00           0.00           0.0
3            0.00               0.00           0.00           0.0
4            0.00               0.00           0.00           0.0

   word_freq_our  word_freq_over  word_freq_remove  word_freq_internet  \
0           0.14            0.28              0.21                0.07
1           1.23            0.19              0.19                0.12
```

```
2           0.63             0.00               0.31                 0.63
3           0.63             0.00               0.31                 0.63
4           1.85             0.00               0.00                 1.85

   word_freq_order  word_freq_mail  ...  char_freq_semicolon  \
0             0.00            0.94  ...                 0.00
1             0.64            0.25  ...                 0.01
2             0.31            0.63  ...                 0.00
3             0.31            0.63  ...                 0.00
4             0.00            0.00  ...                 0.00

   char_freq_leftparen  char_freq_leftsquare  char_freq_bang  \
0                0.132                   0.0           0.372
1                0.143                   0.0           0.276
2                0.137                   0.0           0.137
3                0.135                   0.0           0.135
4                0.223                   0.0           0.000

   char_freq_dollar  char_freq_hash  capital_run_length_average  \
0             0.180           0.048                       5.114
1             0.184           0.010                       9.821
2             0.000           0.000                       3.537
3             0.000           0.000                       3.537
4             0.000           0.000                       3.000

   capital_run_length_longest  capital_run_length_total  is_spam
0                         101                      1028        1
1                         485                      2259        1
2                          40                       191        1
3                          40                       191        1
4                          15                        54        1

[5 rows x 58 columns]
```

[6]:
```
data_y = data_spam['is_spam']
data_x = data_spam.drop(columns = ['is_spam'])
data_x.shape
```

[6]:
```
(4600, 57)
```

[7]:
```
scaler = StandardScaler()
data_scaled = scaler.fit_transform(data_x)
data_scaled.shape
```

[7]:
```
(4600, 57)
```

[8]:
```
df = pd.DataFrame(data=data_scaled[:,0:],
                  index=data_scaled[:,0],
```

```
                    columns=['word_freq_make',␣
↪'word_freq_address','word_freq_all', 'word_freq_3d', 'word_freq_our',
                              'word_freq_over', 'word_freq_remove',␣
↪'word_freq_internet','word_freq_order', 'word_freq_mail',
                              'word_freq_receive','word_freq_will',␣
↪'word_freq_people', 'word_freq_report',
                              'word_freq_addresses', 'word_freq_free',␣
↪'word_freq_business','word_freq_email',
                              'word_freq_you', 'word_freq_credit',␣
↪'word_freq_your','word_freq_font', 'word_freq_000',
                              'word_freq_money', 'word_freq_hp','word_freq_hpl',␣
↪'word_freq_george', 'word_freq_650',
                              'word_freq_lab','word_freq_labs',␣
↪'word_freq_telnet', 'word_freq_857', 'word_freq_data',
                              'word_freq_415', 'word_freq_85',␣
↪'word_freq_technology', 'word_freq_1999','word_freq_parts',
                              'word_freq_pm', 'word_freq_direct',␣
↪'word_freq_cs','word_freq_meeting', 'word_freq_original',
                              'word_freq_project', 'word_freq_re',␣
↪'word_freq_edu', 'word_freq_table',
                              'word_freq_conference', 'char_freq_semicolon',␣
↪'char_freq_leftparen', 'char_freq_leftsquare',
                              'char_freq_bang', 'char_freq_dollar',␣
↪'char_freq_hash','capital_run_length_average',
                              'capital_run_length_longest',␣
↪'capital_run_length_total'])
```

[9]:
```
df.head()
```

[9]:
```
           word_freq_make  word_freq_address  word_freq_all  word_freq_3d  \
 0.345252        0.345252           0.051976       0.435261     -0.046905
-0.145982       -0.145982          -0.164984       0.851833     -0.046905
-0.342475       -0.342475          -0.164984      -0.556576     -0.046905
-0.342475       -0.342475          -0.164984      -0.556576     -0.046905
-0.342475       -0.342475          -0.164984      -0.556576     -0.046905

           word_freq_our  word_freq_over  word_freq_remove  \
 0.345252      -0.256087        0.672259          0.244655
-0.145982       1.364700        0.343576          0.193562
-0.342475       0.472524       -0.350309          0.500124
-0.342475       0.472524       -0.350309          0.500124
-0.342475       2.286616       -0.350309         -0.291828

           word_freq_internet  word_freq_order  word_freq_mail  ...  \
 0.345252           -0.088058        -0.323341        1.086529  ...
-0.145982            0.036609         1.973754        0.016339  ...
-0.342475            1.308212         0.789315        0.605719  ...
```

```
-0.342475             1.308212         0.789315         0.605719  ...
-0.342475             4.350087        -0.323341        -0.371410  ...

           word_freq_conference  char_freq_semicolon  char_freq_leftparen  \
 0.345252             -0.111559            -0.158471            -0.026117
-0.145982             -0.111559            -0.117398             0.014571
-0.342475             -0.111559            -0.158471            -0.007622
-0.342475             -0.111559            -0.158471            -0.015020
-0.342475             -0.111559            -0.158471             0.310487

           char_freq_leftsquare  char_freq_bang  char_freq_dollar  \
 0.345252             -0.155215        0.126330          0.423674
-0.145982             -0.155215        0.008631          0.439942
-0.342475             -0.155215       -0.161788         -0.308392
-0.342475             -0.155215       -0.164240         -0.308392
-0.342475             -0.155215       -0.329755         -0.308392

           char_freq_hash  capital_run_length_average  \
 0.345252        0.008739                   -0.002453
-0.145982       -0.079768                    0.145895
-0.342475       -0.103060                   -0.052154
-0.342475       -0.103060                   -0.052154
-0.342475       -0.103060                   -0.069079

           capital_run_length_longest  capital_run_length_total
 0.345252                    0.250546                  1.228189
-0.145982                    2.220875                  3.258376
-0.342475                   -0.062450                 -0.152207
-0.342475                   -0.062450                 -0.152207
-0.342475                   -0.190726                 -0.378150

[5 rows x 57 columns]
```

0.0.1 Problem 1 [Logistic regression]

1 (a) Split the original data into 75% for training and 25% for testing. Choose the training set at random and ensure that the fraction of SPAM examples in the training set is close to the fraction of 39.4% SPAM examples in the entire dataset. Train a logistic regression model on the training set and output the following on the testing set: 1. Confusion matrix 2. True Positives, False Positives, True Negatives, False Negatives 3. Accuracy, Error 4. Precision, Recall, F1 score

[10]:
```python
xTrain, xTest, yTrain, yTest = train_test_split(df, data_y, test_size = 0.25,␣
 ↪random_state = 0)
```

[11]:
```python
len(yTrain.loc[yTrain == 1])/len(xTrain)
```

[11]:
```
0.392463768115942
```

It is clear that the fraction of SPAM examples in the training set is close to the fraction of 39.4%

SPAM examples in the entire dataset.

Logistic Regression on training data

[96]:
```
logisticRegr = LogisticRegression()
fit1 = logisticRegr.fit(xTrain, yTrain)
fit1
```

[96]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

[97]:
```
predictions = logisticRegr.predict(xTest)
```

Confusion Matrix

[98]:
```
cm = metrics.confusion_matrix(yTest, predictions)
print(cm)
TN = cm[0][0]
FN = cm[1][0]
FP = cm[0][1]
TP = cm[1][1]
```

```
[[655  37]
 [ 45 413]]
```

Accuracy score of the model and evalution metrics

[99]:
```
score = logisticRegr.score(xTest, yTest)
print("Accuracy: ",score)
```

```
Accuracy:  0.928695652173913
```

[100]:
```
print(metrics.classification_report(yTest, predictions))
print("\nCross entrophy: ", metrics.log_loss(yTest,predictions))
```

```
              precision    recall  f1-score   support

           0       0.94      0.95      0.94       692
           1       0.92      0.90      0.91       458

    accuracy                           0.93      1150
   macro avg       0.93      0.92      0.93      1150
weighted avg       0.93      0.93      0.93      1150


Cross entrophy:  2.4627906517283424
```

Calculating accuracy, precision and recall using confusion matrix

[101]:
```
# Overall accuracy
ACC = (TP+TN)/(TP+FP+FN+TN)
print("Accuracy:", ACC)

# Recall
recall = TP/(TP+FN)
print("Recall:", recall)

# Precision
precision = TP/(TP+FP)
print("Precision:", precision)

# Error
print("Error:", 1-ACC)

# F1 Score
f1 = 2*((precision*recall)/(precision+recall))
print("F1 score:", f1)
```

```
Accuracy: 0.928695652173913
Recall: 0.9017467248908297
Precision: 0.9177777777777778
Error: 0.07130434782608697
F1 score: 0.9096916299559471
```

1 (b) Print the coefficients of the features in the model. Which features contribute mostly to the prediction? Which ones are positively correlated and which ones are negatively correlated with the SPAM class?

[110]:
```
cols = xTest.columns
print(type(cols))
coeff = fit1.coef_
print(type(coeff))
coeff_positive = coeff[coeff>0]
coeff_negative = coeff[coeff<0]
```

```
<class 'pandas.core.indexes.base.Index'>
<class 'numpy.ndarray'>
```

[116]:
```
# extract the feature names into list

feature_names = list(cols)
feature_names

# convert list to array

features = np.asarray(feature_names)
```

```python
features

# convert both the array vectors into dataframe
res = pd.DataFrame(coeff)
res = res.T
res1 = pd.DataFrame(features)

# merge dataframes
results = pd.merge(res1, res, left_index=True, right_index=True)
results = results.rename(columns={'0_x':'Features','0_y':'coeffs_CF'})

results
```

[116]:

	Features	coeffs_CF
0	word_freq_make	-0.111037
1	word_freq_address	-0.243277
2	word_freq_all	0.073904
3	word_freq_3d	0.975063
4	word_freq_our	0.325145
5	word_freq_over	0.192316
6	word_freq_remove	0.921651
7	word_freq_internet	0.197067
8	word_freq_order	0.124359
9	word_freq_mail	0.063534
10	word_freq_receive	-0.106708
11	word_freq_will	-0.157030
12	word_freq_people	0.035253
13	word_freq_report	0.027467
14	word_freq_addresses	0.285260
15	word_freq_free	0.956678
16	word_freq_business	0.403099
17	word_freq_email	0.061344
18	word_freq_you	0.154829
19	word_freq_credit	0.380577
20	word_freq_your	0.254695
21	word_freq_font	0.213527
22	word_freq_000	0.785613
23	word_freq_money	0.148142
24	word_freq_hp	-2.303191
25	word_freq_hpl	-0.888267
26	word_freq_george	-4.017833
27	word_freq_650	0.241077
28	word_freq_lab	-1.048815
29	word_freq_labs	-0.073411
30	word_freq_telnet	-1.934493
31	word_freq_857	-0.089125
32	word_freq_data	-0.523009

```
33              word_freq_415   0.011337
34               word_freq_85  -1.084915
35       word_freq_technology   0.348834
36             word_freq_1999   0.033174
37            word_freq_parts   0.052826
38               word_freq_pm  -0.320756
39           word_freq_direct  -0.086698
40               word_freq_cs  -1.775562
41          word_freq_meeting  -1.506114
42         word_freq_original  -0.172644
43          word_freq_project  -0.786283
44               word_freq_re  -0.803914
45              word_freq_edu  -1.329747
46            word_freq_table  -0.189578
47       word_freq_conference  -0.932197
48        char_freq_semicolon  -0.293928
49        char_freq_leftparen  -0.006540
50       char_freq_leftsquare  -0.166719
51             char_freq_bang   0.231042
52           char_freq_dollar   1.147047
53             char_freq_hash   0.703303
54  capital_run_length_average   2.037938
55  capital_run_length_longest   1.760794
56    capital_run_length_total   0.400724
57                        ones  -1.245100
```

The features that contribute mostly to the prediction are

- capital_run_length_average
- capital_run_length_longest
- char_freq_dollar

Positively correlated features with SPAM class

[117]:
```python
coeff_positive = results.loc[results['coeffs_CF'] > 0]

print(coeff_positive)
```

```
                    Features  coeffs_CF
2              word_freq_all   0.073904
3               word_freq_3d   0.975063
4              word_freq_our   0.325145
5             word_freq_over   0.192316
6           word_freq_remove   0.921651
7         word_freq_internet   0.197067
8            word_freq_order   0.124359
9             word_freq_mail   0.063534
12          word_freq_people   0.035253
```

```
13             word_freq_report   0.027467
14          word_freq_addresses   0.285260
15               word_freq_free   0.956678
16           word_freq_business   0.403099
17              word_freq_email   0.061344
18                word_freq_you   0.154829
19             word_freq_credit   0.380577
20               word_freq_your   0.254695
21               word_freq_font   0.213527
22                word_freq_000   0.785613
23              word_freq_money   0.148142
27                word_freq_650   0.241077
33                word_freq_415   0.011337
35         word_freq_technology   0.348834
36               word_freq_1999   0.033174
37              word_freq_parts   0.052826
51               char_freq_bang   0.231042
52             char_freq_dollar   1.147047
53               char_freq_hash   0.703303
54   capital_run_length_average   2.037938
55   capital_run_length_longest   1.760794
56     capital_run_length_total   0.400724
```

Negatively correlated features with SPAM class

[118]:
```python
coeff_negative = results.loc[results['coeffs_CF'] < 0]

print(coeff_negative)
```

```
              Features  coeffs_CF
0       word_freq_make  -0.111037
1    word_freq_address  -0.243277
10   word_freq_receive  -0.106708
11      word_freq_will  -0.157030
24        word_freq_hp  -2.303191
25       word_freq_hpl  -0.888267
26    word_freq_george  -4.017833
28       word_freq_lab  -1.048815
29      word_freq_labs  -0.073411
30    word_freq_telnet  -1.934493
31       word_freq_857  -0.089125
32      word_freq_data  -0.523009
34        word_freq_85  -1.084915
38        word_freq_pm  -0.320756
39    word_freq_direct  -0.086698
40        word_freq_cs  -1.775562
41   word_freq_meeting  -1.506114
42  word_freq_original  -0.172644
```

```
43      word_freq_project  -0.786283
44           word_freq_re  -0.803914
45          word_freq_edu  -1.329747
46        word_freq_table  -0.189578
47   word_freq_conference  -0.932197
48    char_freq_semicolon  -0.293928
49    char_freq_leftparen  -0.006540
50   char_freq_leftsquare  -0.166719
57                   ones  -1.245100
```

1 (c) Vary the decision threshold T {0.25, 0.5, 0.75, 0.9} and report for each value the model accuracy, precision, and recall. Comment on how these metrics vary with the choice of threshold.

- Threshold = 0.5 corresponds to actual logistic regression accuracy and error.
- As the threshold increases the accuracy of the model is degrading little bit.

Threshold = 0.5

[119]:
```
prediction_T = logisticRegr.predict_proba(xTest)>= 0.5

prediction_T = prediction_T[:,0]
cm_05 = metrics.confusion_matrix(yTest, prediction_T)
print(cm_05)
TN_05 = cm_05[1][0]
FN_05 = cm_05[0][0]
FP_05 = cm_05[1][1]
TP_05 = cm_05[0][1]
```

```
[[ 37 655]
 [413  45]]
```

[120]:
```
# Overall accuracy
ACC_05 = (TP_05+TN_05)/(TP_05+FP_05+FN_05+TN_05)
print("Accuracy:", ACC_05)

# Sensitivity, hit rate, recall, or true positive rate
recall_05 = TP_05/(TP_05+FN_05)
print("Recall:", recall_05)

# Precision or positive predictive value
precision_05 = TP_05/(TP_05+FP_05)
print("Precision:", precision_05)

# Error
print("Error:", 1-ACC_05)

# F1 Score
```

```
f1_05 = 2*((precision_05*recall_05)/(precision_05+recall_05))
print("F1 score:", f1_05)
```

```
Accuracy: 0.928695652173913
Recall: 0.9465317919075145
Precision: 0.9357142857142857
Error: 0.07130434782608697
F1 score: 0.9410919540229885
```

Threshold = 0.25

[121]:
```
prediction_T = logisticRegr.predict_proba(xTest)>= 0.25

prediction_T = prediction_T[:,0]
cm_25 = metrics.confusion_matrix(yTest, prediction_T)
print(cm_25)
TN_25 = cm_25[1][0]
FN_25 = cm_25[0][0]
FP_25 = cm_25[1][1]
TP_25 = cm_25[0][1]
```

```
[[ 15 677]
 [363  95]]
```

[122]:
```
# Overall accuracy
ACC_25 = (TP_25+TN_25)/(TP_25+FP_25+FN_25+TN_25)
print("Accuracy:", ACC_25)

# Sensitivity, hit rate, recall, or true positive rate
recall_25 = TP_25/(TP_25 + FN_25)
print("Recall:", recall_25)

# Precision or positive predictive value
precision_25 = TP_25/(TP_25+FP_25)
print("Precision:", precision_25)

# Error
print("Error:", 1-ACC_25)

# F1 Score
f1_25 = 2*((precision_25*recall_25)/(precision_25+recall_25))
print("F1 score:", f1_25)
```

```
Accuracy: 0.9043478260869565
Recall: 0.9783236994219653
Precision: 0.8769430051813472
Error: 0.09565217391304348
F1 score: 0.924863387978142
```

Threshold = 0.75

[123]:
```
prediction_T = logisticRegr.predict_proba(xTest)>= 0.75

prediction_T = prediction_T[:,0]
cm_75 = metrics.confusion_matrix(yTest, prediction_T)
print(cm_75)
TN_75 = cm_75[1][0]
FN_75 = cm_75[0][0]
FP_75 = cm_75[1][1]
TP_75 = cm_75[0][1]
```

```
[[102 590]
 [445  13]]
```

[124]:
```
# Overall accuracy
ACC_75 = (TP_75+TN_75)/(TP_75+FP_75+FN_75+TN_75)
print("Accuracy:", ACC_75)

# Sensitivity, hit rate, recall, or true positive rate
recall_75 = TP_75/(TP_75+FN_75)
print("Recall:", recall_75)

# Precision or positive predictive value
precision_75 = TP_75/(TP_75+FP_75)
print("Precision:", precision_75)

# Error
print("Error:", 1-ACC_75)

# F1 Score
f1_75 = 2*((precision_75*recall_75)/(precision_75+recall_75))
print("F1 score:", f1_75)
```

```
Accuracy: 0.9
Recall: 0.8526011560693642
Precision: 0.9784411276948259
Error: 0.09999999999999998
F1 score: 0.911196911196911
```

Threshold = 0.9

[125]:
```
prediction_T = logisticRegr.predict_proba(xTest)>= 0.9

prediction_T = prediction_T[:,0]
cm_09 = metrics.confusion_matrix(yTest, prediction_T)
print(cm_09)
TN_09 = cm_09[1][0]
FN_09 = cm_09[0][0]
```

```
FP_09 = cm_09[1][1]
TP_09 = cm_09[0][1]
```

```
[[224 468]
 [455   3]]
```

[126]:
```
# Overall accuracy
ACC_09 = (TP_09+TN_09)/(TP_09+FP_09+FN_09+TN_09)
print("Accuracy:", ACC_09)

# Sensitivity, hit rate, recall, or true positive rate
recall_09 = TP_09/(TP_09+FN_09)
print("Recall:", recall_09)

# Precision or positive predictive value
precision_09 = TP_09/(TP_09+FP_09)
print("Precision:", precision_09)

# Error
print("Error:", 1-ACC_09)

# F1 Score
f1_09 = 2*((precision_09*recall_09)/(precision_09+recall_09))
print("F1 score:", f1_09)
```

```
Accuracy: 0.802608695652174
Recall: 0.6763005780346821
Precision: 0.9936305732484076
Error: 0.19739130434782604
F1 score: 0.8048151332760104
```

1 (d) Use your implementation of gradient descent from Homework 2 and adapt it for logistic regression. Take 3 values of the learning rate and report the cross-entropy loss objective after 10, 50, and 100 iterations. At 100 iterations, report the accuracy and F1 score for the 3 learning rates, and compare with the metrics given by the package.

- The cross entropy for three learning rates are hugely varying. For learning rate 0.2, the cross entrophy is less and is close to the package logistic regressoin.
- The cross entrophy for alpha = **0.2** is **3.9** and accuracy is **88**
- The cross entrophy for alpha = **0.1** is **5.7** and accuracy is **83**
- The cross entrophy for package based logistic regression is **2.5** and accuracy is **93**
- However the implementation is fair enough.
- For 100 iterations, all the learning rates are giving out better results.

[155]:
```
# Padding ones to the X variable
X1 = xTrain
X1['ones'] = 1
```

```
X2 = xTest
X2['ones'] = 1
```

[128]:
```
def gradient_descent(X,y,theta,alpha,n):

    m = len(y)

    for i in range(n):
        prediction = np.dot(X,theta)
        h = 1 / (1 + np.exp(-prediction))
        theta = theta - (1/m) * alpha * (X.T.dot(h - y))

    return theta
```

[129]:
```
yTrain_gd = yTrain.to_numpy().reshape(yTrain.shape[0],1)
```

[130]:
```
alpha = 0.01
iter = [10,50,100]
cols = xTrain.columns
theta = np.random.randn(len(cols),1)
theta_df = theta
for i in iter:
    theta_j = gradient_descent(X1,yTrain_gd,theta_df,alpha,i)
    yTest_pred_gd = xTest.dot(theta_j)
    yTest_pred_gd = yTest_pred_gd.to_numpy()

    for i in range(len(yTest_pred_gd)):
        if yTest_pred_gd[i]>0.5:
            yTest_pred_gd[i] = 1
        else:
            yTest_pred_gd[i] = 0

    print("Cross entrophy: ", metrics.log_loss(yTest,yTest_pred_gd))
    print("mean squared error on scaled test data (GD):",metrics.
↪mean_squared_error(yTest,yTest_pred_gd))
    print("\nConfusion matrix \n", metrics.confusion_matrix(yTest,␣
↪yTest_pred_gd))
    print("\nReport: \n",metrics.classification_report(yTest, yTest_pred_gd))
```

```
Cross entrophy:  16.8491866095176
mean squared error on scaled test data (GD): 0.48782608695652174

Confusion matrix
 [[303 389]
 [172 286]]
```

```
Report:
               precision    recall  f1-score   support

           0       0.64      0.44      0.52       692
           1       0.42      0.62      0.50       458

    accuracy                           0.51      1150
   macro avg       0.53      0.53      0.51      1150
weighted avg       0.55      0.51      0.51      1150

Cross entropy:  15.858056513023852
mean squared error on scaled test data (GD): 0.4591304347826087

Confusion matrix
 [[328 364]
 [164 294]]

Report:
               precision    recall  f1-score   support

           0       0.67      0.47      0.55       692
           1       0.45      0.64      0.53       458

    accuracy                           0.54      1150
   macro avg       0.56      0.56      0.54      1150
weighted avg       0.58      0.54      0.54      1150

Cross entropy:  15.017101962572589
mean squared error on scaled test data (GD): 0.43478260869565216

Confusion matrix
 [[343 349]
 [151 307]]

Report:
               precision    recall  f1-score   support

           0       0.69      0.50      0.58       692
           1       0.47      0.67      0.55       458

    accuracy                           0.57      1150
   macro avg       0.58      0.58      0.56      1150
weighted avg       0.60      0.57      0.57      1150
```

[131]:
```
alpha = 0.1
iter = [10,50,100]
```

```
cols = xTrain.columns
theta = np.random.randn(len(cols),1)
theta_df = theta
for i in iter:
    theta_j = gradient_descent(X1,yTrain_gd,theta_df,alpha,i)
    yTest_pred_gd = xTest.dot(theta_j)
    yTest_pred_gd = yTest_pred_gd.to_numpy()

    for i in range(len(yTest_pred_gd)):
        if yTest_pred_gd[i]>0.5:
            yTest_pred_gd[i] = 1
        else:
            yTest_pred_gd[i] = 0
    print("Cross entropy: ", metrics.log_loss(yTest,yTest_pred_gd))
    print("mean squared error on scaled test data (GD):",metrics.
 ↪mean_squared_error(yTest,yTest_pred_gd))
    print("confusion matrix \n", metrics.confusion_matrix(yTest, yTest_pred_gd))
    print("report: \n",metrics.classification_report(yTest, yTest_pred_gd))
```

```
Cross entropy:  12.734449654675453
mean squared error on scaled test data (GD): 0.36869565217391304
confusion matrix
 [[472 220]
 [204 254]]
report:
               precision    recall  f1-score   support

           0       0.70      0.68      0.69       692
           1       0.54      0.55      0.55       458

    accuracy                           0.63      1150
   macro avg       0.62      0.62      0.62      1150
weighted avg       0.63      0.63      0.63      1150

Cross entropy:  8.830007830742824
mean squared error on scaled test data (GD): 0.25565217391304346
confusion matrix
 [[556 136]
 [158 300]]
report:
               precision    recall  f1-score   support

           0       0.78      0.80      0.79       692
           1       0.69      0.66      0.67       458

    accuracy                           0.74      1150
   macro avg       0.73      0.73      0.73      1150
```

```
weighted avg       0.74      0.74      0.74      1150

Cross entrophy:  6.337173030878513
mean squared error on scaled test data (GD): 0.18347826086956523
confusion matrix
 [[608  84]
 [127 331]]
report:
               precision    recall  f1-score   support

           0       0.83      0.88      0.85       692
           1       0.80      0.72      0.76       458

    accuracy                           0.82      1150
   macro avg       0.81      0.80      0.81      1150
weighted avg       0.82      0.82      0.81      1150
```

[132]:
```python
alpha = 0.2
iter = [10,50,100]
cols = xTrain.columns
theta = np.random.randn(len(cols),1)
theta_df = theta
for i in iter:
    theta_j = gradient_descent(X1,yTrain_gd,theta_df,alpha,i)
    yTest_pred_gd = xTest.dot(theta_j)
    yTest_pred_gd = yTest_pred_gd.to_numpy()

    for i in range(len(yTest_pred_gd)):
        if yTest_pred_gd[i]>0.5:
            yTest_pred_gd[i] = 1
        else:
            yTest_pred_gd[i] = 0

    print("Cross entrophy: ", metrics.log_loss(yTest,yTest_pred_gd))
    print("mean squared error on scaled test data (GD):",metrics.
 ↪mean_squared_error(yTest,yTest_pred_gd))
    print("confusion matrix \n", metrics.confusion_matrix(yTest, yTest_pred_gd))
    print("report: \n",metrics.classification_report(yTest, yTest_pred_gd))
```

```
Cross entrophy:  10.932388992105132
mean squared error on scaled test data (GD): 0.3165217391304348
confusion matrix
 [[526 166]
 [198 260]]
report:
               precision    recall  f1-score   support
```

```
           0       0.73      0.76      0.74       692
           1       0.61      0.57      0.59       458

    accuracy                           0.68      1150
   macro avg       0.67      0.66      0.67      1150
weighted avg       0.68      0.68      0.68      1150

Cross entrophy:  6.427275577295552
mean squared error on scaled test data (GD): 0.18608695652173912
confusion matrix
 [[606  86]
 [128 330]]
report:
               precision    recall  f1-score   support

           0       0.83      0.88      0.85       692
           1       0.79      0.72      0.76       458

    accuracy                           0.81      1150
   macro avg       0.81      0.80      0.80      1150
weighted avg       0.81      0.81      0.81      1150

Cross entrophy:  4.895538545924942
mean squared error on scaled test data (GD): 0.14173913043478262
confusion matrix
 [[631  61]
 [102 356]]
report:
               precision    recall  f1-score   support

           0       0.86      0.91      0.89       692
           1       0.85      0.78      0.81       458

    accuracy                           0.86      1150
   macro avg       0.86      0.84      0.85      1150
weighted avg       0.86      0.86      0.86      1150
```

[133]:
```
alpha = [0.01,0.1,0.2]
iter = 100
cols = xTrain.columns
theta = np.random.randn(len(cols),1)
theta_df = theta
for i in alpha:
    theta_j = gradient_descent(X1,yTrain_gd,theta_df,i,iter)
    yTest_pred_gd = xTest.dot(theta_j)
```

```
    yTest_pred_gd = yTest_pred_gd.to_numpy()

    for j in range(len(yTest_pred_gd)):
        if yTest_pred_gd[j]>0.5:
            yTest_pred_gd[j] = 1
        else:
            yTest_pred_gd[j] = 0
    print("Alpha: ", i)
    print("Cross entropy: ", metrics.log_loss(yTest,yTest_pred_gd))
    print("mean squared error on scaled test data (GD):",metrics.
 ↪mean_squared_error(yTest,yTest_pred_gd))
    print("confusion matrix \n", metrics.confusion_matrix(yTest, yTest_pred_gd))
    print("report: \n",metrics.classification_report(yTest, yTest_pred_gd))
```

```
Alpha:  0.01
Cross entropy:  12.854571318352383
mean squared error on scaled test data (GD): 0.37217391304347824
confusion matrix
 [[491 201]
 [227 231]]
report:
               precision    recall  f1-score   support

           0       0.68      0.71      0.70       692
           1       0.53      0.50      0.52       458

    accuracy                           0.63      1150
   macro avg       0.61      0.61      0.61      1150
weighted avg       0.62      0.63      0.63      1150

Alpha:  0.1
Cross entropy:  5.796566096001651
mean squared error on scaled test data (GD): 0.16782608695652174
confusion matrix
 [[608  84]
 [109 349]]
report:
               precision    recall  f1-score   support

           0       0.85      0.88      0.86       692
           1       0.81      0.76      0.78       458

    accuracy                           0.83      1150
   macro avg       0.83      0.82      0.82      1150
weighted avg       0.83      0.83      0.83      1150

Alpha:  0.2
```

```
Cross entropy:  3.934453292869278
mean squared error on scaled test data (GD): 0.11391304347826087
confusion matrix
 [[640  52]
 [ 79 379]]
report:
               precision    recall  f1-score   support

           0       0.89      0.92      0.91       692
           1       0.88      0.83      0.85       458

    accuracy                           0.89      1150
   macro avg       0.88      0.88      0.88      1150
weighted avg       0.89      0.89      0.89      1150
```

0.0.2 Problem 2 [Comparing classifiers]

You can use the same training and testing data as in Problem 1. Train the following classifiers using the training data: 1. Logistic regression 2. LDA 3. kNN 4. Naive Bayes 5. Decision tree

(a) answer is found at the end of (b) part.

(b) Print the accuracy and error metrics for all 5 classifiers on both training and testing data. Which model is performing best? Which one is performing worst? Write down some observations. - Logistic Regression has an accuracy of 92.8 and KNN has an accuracy of 91.1. - Number of true positives and true negatives are similar and high is Logistic regression and KNN classification. - Naive Bayes has the least accuracy score of 82 and the number of true positives and true negatives are less compared to other models.

Logistic Regression

[134]:
```
logisticRegr = LogisticRegression()
fit1 = logisticRegr.fit(xTrain, yTrain)
fit1
```

[134]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

[135]:
```
y_pred_reg = logisticRegr.predict(xTest)
y_pred_reg
```

[135]:
```
array([1, 0, 0, ..., 0, 1, 0], dtype=int64)
```

[136]:
```
print("TESTING:\n")
score = logisticRegr.score(xTest, yTest)
print("Accuracy of the model: ",score)

# Testing with accuracy score
sc = metrics.accuracy_score(yTest, y_pred_reg)
```

```python
print("\nAccuracy: ",sc)

print("Error: ",1-sc)
cm_reg = metrics.confusion_matrix(yTest, y_pred_reg)
TN_reg = cm_reg[0][0]
FN_reg = cm_reg[1][0]
FP_reg = cm_reg[0][1]
TP_reg = cm_reg[1][1]

print("\nconfusion matrix \n", cm_reg)
print("\nReport:\n",metrics.classification_report(yTest, y_pred_reg))

print("\nTRAINING: \n")

# Testing with Logistic regression score
score = logisticRegr.score(xTrain, yTrain)
print("Accuracy of the model: ",score)
print("Error: ",1-score)
```

```
TESTING:

Accuracy of the model:  0.928695652173913

Accuracy:  0.928695652173913
Error:  0.07130434782608697

confusion matrix
 [[655  37]
 [ 45 413]]

Report:
               precision    recall  f1-score   support

           0       0.94      0.95      0.94       692
           1       0.92      0.90      0.91       458

    accuracy                           0.93      1150
   macro avg       0.93      0.92      0.93      1150
weighted avg       0.93      0.93      0.93      1150


TRAINING:

Accuracy of the model:  0.9263768115942029
Error:  0.07362318840579707
```

Naive Bayes

[137]:
```python
nb = GaussianNB()
fit2 = nb.fit(xTrain, yTrain)
fit2
```

[137]:
```
GaussianNB(priors=None, var_smoothing=1e-09)
```

[138]:
```python
y_pred_nb = nb.predict(xTest)
y_pred_nb
```

[138]:
```
array([1, 0, 0, ..., 1, 1, 1], dtype=int64)
```

[139]:
```python
print("TESTING:\n")
score = nb.score(xTest, yTest)
print("Accuracy of the model: ",score)

# Testing with accuracy score
sc = metrics.accuracy_score(yTest, y_pred_nb)

print("\nAccuracy: ",sc)

print("Error: ",1-sc)
cm_nb = metrics.confusion_matrix(yTest, y_pred_nb)
TN_nb = cm_nb[0][0]
FN_nb = cm_nb[1][0]
FP_nb = cm_nb[0][1]
TP_nb = cm_nb[1][1]

print("\nconfusion matrix \n", cm_nb)
print("\nReport:\n",metrics.classification_report(yTest, y_pred_nb))

print("TRAINING: \n")

# Testing with naive bayes score
score = nb.score(xTrain, yTrain)
print("Accuracy of the model: ",score)
print("Error: ",1-score)
```

```
TESTING:

Accuracy of the model:  0.8243478260869566

Accuracy:  0.8243478260869566
Error:  0.17565217391304344

confusion matrix
 [[502 190]
 [ 12 446]]

Report:
```

```
              precision    recall  f1-score   support

           0       0.98      0.73      0.83       692
           1       0.70      0.97      0.82       458

    accuracy                           0.82      1150
   macro avg       0.84      0.85      0.82      1150
weighted avg       0.87      0.82      0.83      1150

TRAINING:

Accuracy of the model:  0.8176811594202898
Error:  0.18231884057971015
```

Linear Discriminant Analysis

[140]:
```python
lda = LDA(n_components=1)
fit3 = lda.fit(xTrain, yTrain)
y_pred_lda = fit3.predict(xTest)
```

[141]:
```python
print("TESTING:\n")
score = lda.score(xTest, yTest)
print("Accuracy of the model: ",score)

# Testing with accuracy score
sc = metrics.accuracy_score(yTest, y_pred_lda)
print("\nAccuracy: ",sc)

print("Error: ",1-sc)

cm_lda = metrics.confusion_matrix(yTest, y_pred_lda)
TN_lda = cm_lda[0][0]
FN_lda = cm_lda[1][0]
FP_lda = cm_lda[0][1]
TP_lda = cm_lda[1][1]

print("\nconfusion matrix \n", cm_nb)
print("\nReport:\n",metrics.classification_report(yTest, y_pred_lda))

print("TRAINING: \n")

# Testing with LDA score
score = lda.score(xTrain, yTrain)
print("Accuracy of the model: ",score)
print("Error: ",1-score)
```

```
TESTING:

Accuracy of the model:  0.8947826086956522
```

```
Accuracy:  0.8947826086956522
Error:  0.10521739130434782

confusion matrix
 [[502 190]
 [ 12 446]]

Report:
               precision    recall  f1-score   support

           0       0.88      0.96      0.92       692
           1       0.93      0.79      0.86       458

    accuracy                           0.89      1150
   macro avg       0.90      0.88      0.89      1150
weighted avg       0.90      0.89      0.89      1150

TRAINING:

Accuracy of the model:  0.8855072463768116
Error:  0.11449275362318845
```

Decision Tree

[142]:
```
classifier_tree = DecisionTreeClassifier()
classifier_tree.fit(xTrain, yTrain)
```

[142]:
```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                       max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort=False,
                       random_state=None, splitter='best')
```

[143]:
```
y_pred_dtree = classifier_tree.predict(xTest)
```

[144]:
```
print("TESTING:\n")
score = classifier_tree.score(xTest, yTest)
print("Accuracy of the model: ",score)

# Testing with accuracy score
sc = metrics.accuracy_score(yTest, y_pred_dtree)

print("\nAccuracy: ",sc)

print("Error: ",1-sc)

cm_tree = metrics.confusion_matrix(yTest, y_pred_dtree)
TN_tree = cm_tree[0][0]
```

```
FN_tree = cm_tree[1][0]
FP_tree = cm_tree[0][1]
TP_tree = cm_tree[1][1]

print("\nconfusion matrix \n", cm_tree)
print("\nReport:\n",metrics.classification_report(yTest, y_pred_dtree))

print("TRAINING:")

# Testing with decision tree score
score = classifier_tree.score(xTrain, yTrain)
print("\nAccuracy of the model: ",score)
print("Error: ",1-score)
```

```
TESTING:

Accuracy of the model:  0.8956521739130435

Accuracy:  0.8956521739130435
Error:  0.10434782608695647

confusion matrix
 [[623  69]
 [ 51 407]]

Report:
               precision    recall  f1-score   support

           0       0.92      0.90      0.91       692
           1       0.86      0.89      0.87       458

    accuracy                           0.90      1150
   macro avg       0.89      0.89      0.89      1150
weighted avg       0.90      0.90      0.90      1150

TRAINING:

Accuracy of the model:  0.9991304347826087
Error:  0.0008695652173913437
```

K Nearest Neighbours

[145]:
```
classifier = KNeighborsClassifier(n_neighbors=10)
classifier.fit(xTrain, yTrain)
```

[145]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=10, p=2,
                     weights='uniform')
```

[146]:
```
y_pred_knn = classifier.predict(xTest)
y_pred_knn
```

[146]:
```
array([1, 0, 0, ..., 0, 0, 0], dtype=int64)
```

[147]:
```
print("TESTING:\n")
score = classifier.score(xTest, yTest)
print("Accuracy of the model: ",score)

# Testing with accuracy score
sc = metrics.accuracy_score(yTest, y_pred_knn)

print("\nAccuracy: ",sc)
print("Error: ",1-sc)

cm_knn = metrics.confusion_matrix(yTest, y_pred_knn)
TN_knn = cm_knn[0][0]
FN_knn = cm_knn[1][0]
FP_knn = cm_knn[0][1]
TP_knn = cm_knn[1][1]

print("\nconfusion matrix \n", cm_knn)
print("\nReport:\n",metrics.classification_report(yTest, y_pred_knn))

print("TRAINING: \n")

# Testing with knn score
score = classifier.score(xTrain, yTrain)
print("Accuracy of the model: ",score)
print("Error: ",1-score)
```

```
TESTING:

Accuracy of the model:  0.9113043478260869

Accuracy:  0.9113043478260869
Error:  0.08869565217391306

confusion matrix
 [[662  30]
 [ 72 386]]

Report:
               precision    recall  f1-score   support

           0       0.90      0.96      0.93       692
           1       0.93      0.84      0.88       458
```

```
    accuracy                           0.91      1150
   macro avg       0.91      0.90      0.91      1150
weighted avg       0.91      0.91      0.91      1150

TRAINING:

Accuracy of the model:  0.9127536231884058
Error:  0.0872463768115942
```

(a) Experiment with different values of k for kNN and report 2 metrics on the training and testing sets: accuracy and error. Choose the value of k that gives the highest accuracy in testing.

[49]:
```
error = []

# Calculating error for K values between 1 and 50
for i in range(1, 50):
    classifier = KNeighborsClassifier(n_neighbors=i)
    classifier.fit(xTrain, yTrain)
    pred_knn = classifier.predict(xTest)
    error.append(np.mean(pred_knn != yTest))

    sc = metrics.accuracy_score(yTest, pred_knn)
    print("K value: ", i)
    print("Accuracy: ",sc)
    print("Error: ",1-sc)
```

```
K value:  1
Accuracy:  0.908695652173913
Error:  0.09130434782608698
K value:  2
Accuracy:  0.8930434782608696
Error:  0.1069565217391304
K value:  3
Accuracy:  0.9113043478260869
Error:  0.08869565217391306
K value:  4
Accuracy:  0.9008695652173913
Error:  0.09913043478260875
K value:  5
Accuracy:  0.9130434782608695
Error:  0.08695652173913049
K value:  6
Accuracy:  0.9130434782608695
Error:  0.08695652173913049
K value:  7
Accuracy:  0.9121739130434783
Error:  0.08782608695652172
K value:  8
```

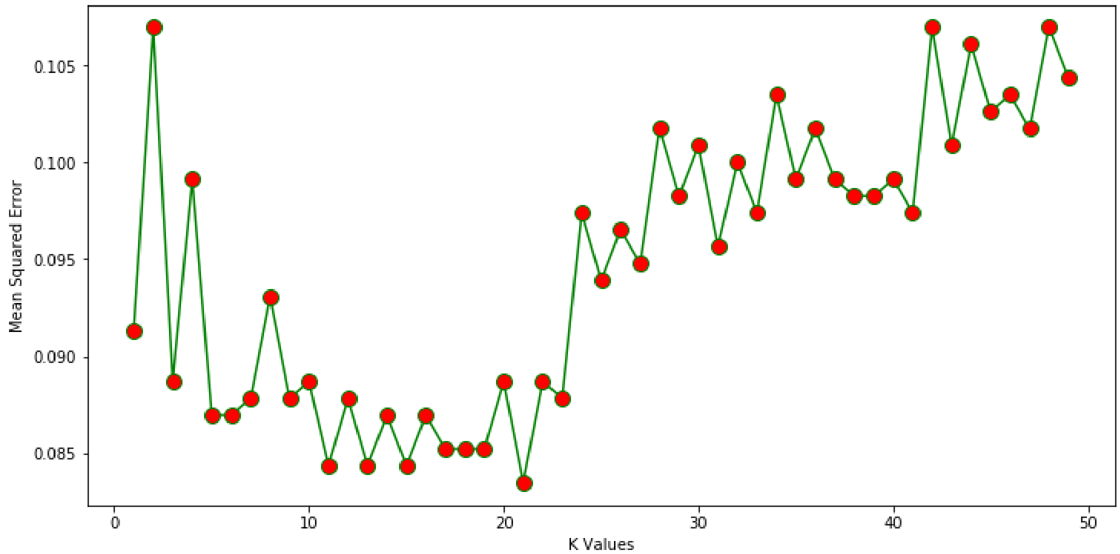
```
Accuracy:  0.9069565217391304
Error:  0.09304347826086956
K value:  9
Accuracy:  0.9121739130434783
Error:  0.08782608695652172
K value:  10
Accuracy:  0.9113043478260869
Error:  0.08869565217391306
K value:  11
Accuracy:  0.9156521739130434
Error:  0.08434782608695657
K value:  12
Accuracy:  0.9121739130434783
Error:  0.08782608695652172
K value:  13
Accuracy:  0.9156521739130434
Error:  0.08434782608695657
K value:  14
Accuracy:  0.9130434782608695
Error:  0.08695652173913049
K value:  15
Accuracy:  0.9156521739130434
Error:  0.08434782608695657
K value:  16
Accuracy:  0.9130434782608695
Error:  0.08695652173913049
K value:  17
Accuracy:  0.9147826086956522
Error:  0.0852173913043478
K value:  18
Accuracy:  0.9147826086956522
Error:  0.0852173913043478
K value:  19
Accuracy:  0.9147826086956522
Error:  0.0852173913043478
K value:  20
Accuracy:  0.9113043478260869
Error:  0.08869565217391306
K value:  21
Accuracy:  0.9165217391304348
Error:  0.08347826086956522
K value:  22
Accuracy:  0.9113043478260869
Error:  0.08869565217391306
K value:  23
Accuracy:  0.9121739130434783
Error:  0.08782608695652172
K value:  24
```

```
Accuracy:  0.9026086956521739
Error:  0.09739130434782606
K value:  25
Accuracy:  0.9060869565217391
Error:  0.0939130434782609
K value:  26
Accuracy:  0.9034782608695652
Error:  0.09652173913043482
K value:  27
Accuracy:  0.9052173913043479
Error:  0.09478260869565214
K value:  28
Accuracy:  0.8982608695652174
Error:  0.10173913043478255
K value:  29
Accuracy:  0.9017391304347826
Error:  0.0982608695652174
K value:  30
Accuracy:  0.8991304347826087
Error:  0.10086956521739132
K value:  31
Accuracy:  0.9043478260869565
Error:  0.09565217391304348
K value:  32
Accuracy:  0.9
Error:  0.09999999999999998
K value:  33
Accuracy:  0.9026086956521739
Error:  0.09739130434782606
K value:  34
Accuracy:  0.8965217391304348
Error:  0.10347826086956524
K value:  35
Accuracy:  0.9008695652173913
Error:  0.09913043478260875
K value:  36
Accuracy:  0.8982608695652174
Error:  0.10173913043478255
K value:  37
Accuracy:  0.9008695652173913
Error:  0.09913043478260875
K value:  38
Accuracy:  0.9017391304347826
Error:  0.0982608695652174
K value:  39
Accuracy:  0.9017391304347826
Error:  0.0982608695652174
K value:  40
```

```
Accuracy:  0.9008695652173913
Error:  0.09913043478260875
K value:  41
Accuracy:  0.9026086956521739
Error:  0.09739130434782606
K value:  42
Accuracy:  0.8930434782608696
Error:  0.1069565217391304
K value:  43
Accuracy:  0.8991304347826087
Error:  0.10086956521739132
K value:  44
Accuracy:  0.8939130434782608
Error:  0.10608695652173916
K value:  45
Accuracy:  0.8973913043478261
Error:  0.1026086956521739
K value:  46
Accuracy:  0.8965217391304348
Error:  0.10347826086956524
K value:  47
Accuracy:  0.8982608695652174
Error:  0.10173913043478255
K value:  48
Accuracy:  0.8930434782608696
Error:  0.1069565217391304
K value:  49
Accuracy:  0.8956521739130435
Error:  0.10434782608695647
```

[50]:
```
plt.figure(figsize=(12, 6))
plt.plot(range(1, 50), error, color='green', marker='o',
         markerfacecolor='red', markersize=10)
plt.xlabel('K Values')
plt.ylabel('Mean Squared Error')
```

[50]:
```
Text(0, 0.5, 'Mean Squared Error')
```



From the output we can see that the mean error is minimum when the value of the K is between 10 and 22.

For k =11

[51]:
```
classifier = KNeighborsClassifier(n_neighbors=11)
classifier.fit(xTrain, yTrain)
```

[51]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=11, p=2,
                     weights='uniform')
```

[52]:
```
y_pred_knn_10 = classifier.predict(xTest)
y_pred_knn_10
```

[52]:
```
array([1, 0, 0, ..., 0, 0, 0], dtype=int64)
```

[53]:
```
print("TESTING: \n")
score = classifier.score(xTest, yTest)
print("Accuracy of the model: ",score)
sc = metrics.accuracy_score(yTest, y_pred_knn)

print("\nAccuracy: ",sc)
print("Error: ",1-sc)
print("confusion matrix \n", metrics.confusion_matrix(yTest, y_pred_knn_10))
print("report: \n",metrics.classification_report(yTest, y_pred_knn_10))

print("TRAINING: \n")
score = classifier.score(xTrain, yTrain)
print("Accuracy of the model: ",score)
print("Error: ",1-score)
```

```
TESTING:
```

```
Accuracy of the model:  0.9156521739130434

Accuracy:  0.9113043478260869
Error:  0.08869565217391306
confusion matrix
 [[656  36]
 [ 61 397]]
report:
               precision    recall  f1-score   support

           0       0.91      0.95      0.93       692
           1       0.92      0.87      0.89       458

    accuracy                           0.92      1150
   macro avg       0.92      0.91      0.91      1150
weighted avg       0.92      0.92      0.92      1150

TRAINING:

Accuracy of the model:  0.9159420289855073
Error:  0.08405797101449275
```

For k = 21

[54]:
```
classifier = KNeighborsClassifier(n_neighbors=21)
classifier.fit(xTrain, yTrain)
```

[54]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=21, p=2,
                     weights='uniform')
```

[55]:
```
y_pred_knn_21 = classifier.predict(xTest)
y_pred_knn_21
```

[55]:
```
array([1, 0, 0, ..., 0, 0, 0], dtype=int64)
```

[56]:
```
print("TESTING: \n")
score = classifier.score(xTest, yTest)
print("Accuracy of the model: ",score)
sc = metrics.accuracy_score(yTest, y_pred_knn)
print("\nAccuracy: ",sc)
print("Error: ",1-sc)

print("\nconfusion matrix \n", metrics.confusion_matrix(yTest, y_pred_knn_21))
print("\nReport: \n",metrics.classification_report(yTest, y_pred_knn_21))

print("TRAINING: \n")
score = classifier.score(xTrain, yTrain)

print("Accuracy of the model: ",score)
```

```
print("Error: ",1-score)
```

```
TESTING:

Accuracy of the model:  0.9165217391304348

Accuracy:  0.9113043478260869
Error:  0.08869565217391306

confusion matrix
 [[665  27]
 [ 69 389]]

Report:
               precision    recall  f1-score   support

           0       0.91      0.96      0.93       692
           1       0.94      0.85      0.89       458

    accuracy                           0.92      1150
   macro avg       0.92      0.91      0.91      1150
weighted avg       0.92      0.92      0.92      1150

TRAINING:

Accuracy of the model:  0.9060869565217391
Error:  0.0939130434782609
```

(c) Generate a graph that includes 5 ROC curves (one for each of the 5 classifiers) on the testing set. Compute the Area Under the Curve (AUC) metric for all 5 classifiers.

[57]:
```
def plot_roc_curve(fpr, tpr):
    plt.plot(fpr, tpr, color='orange', label='ROC')
    plt.plot([0, 1], [0, 1], color='darkblue', linestyle='--')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend()
    plt.show()
```

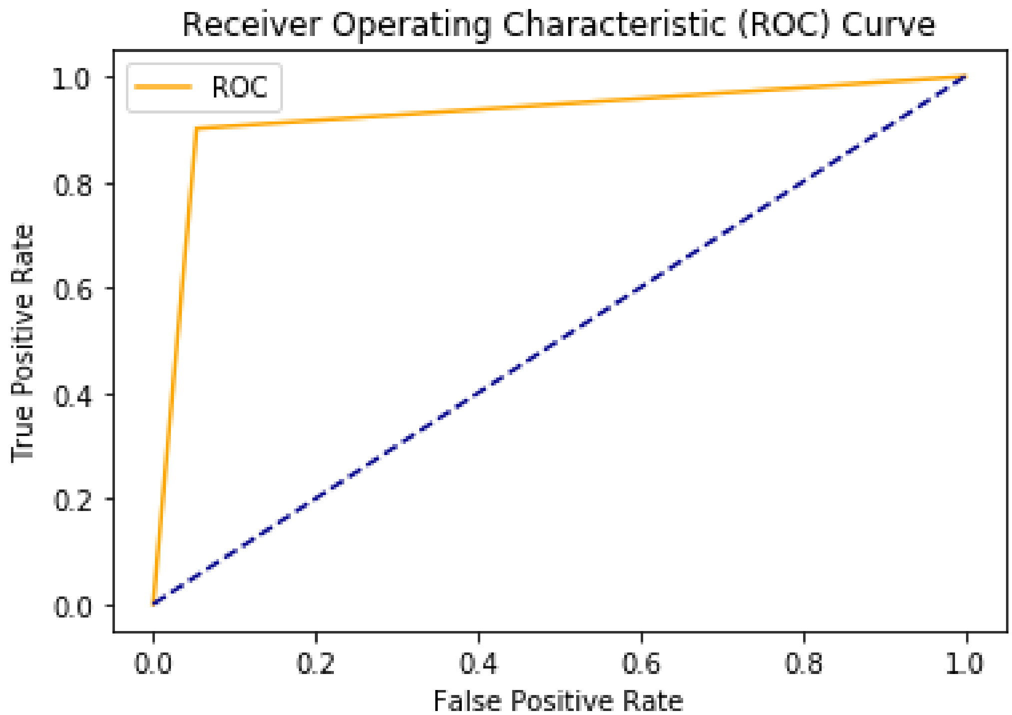
Logistic Regression

[58]:
```
auc = roc_auc_score(yTest, y_pred_reg)
print('AUC: %.2f' % auc)

fpr, tpr, thresholds = roc_curve(yTest, y_pred_reg)

plot_roc_curve(fpr, tpr)
```

```
AUC: 0.92
```



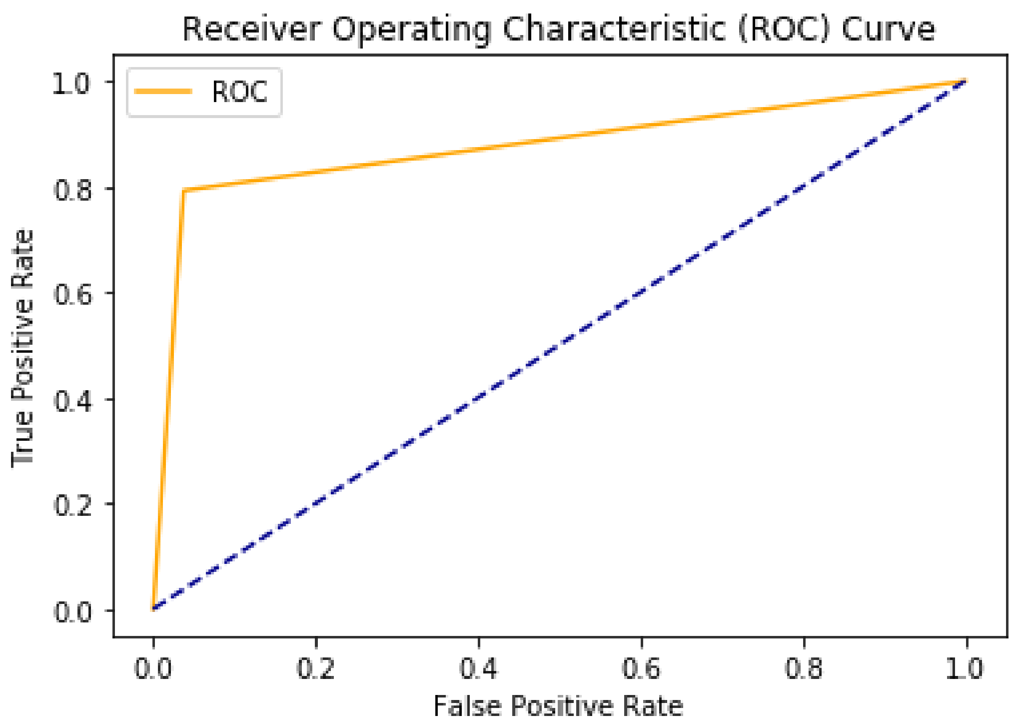
LDA

[59]:
```
auc = roc_auc_score(yTest, y_pred_lda)
print('AUC: %.2f' % auc)

fpr, tpr, thresholds = roc_curve(yTest, y_pred_lda)

plot_roc_curve(fpr, tpr)
```

```
AUC: 0.88
```



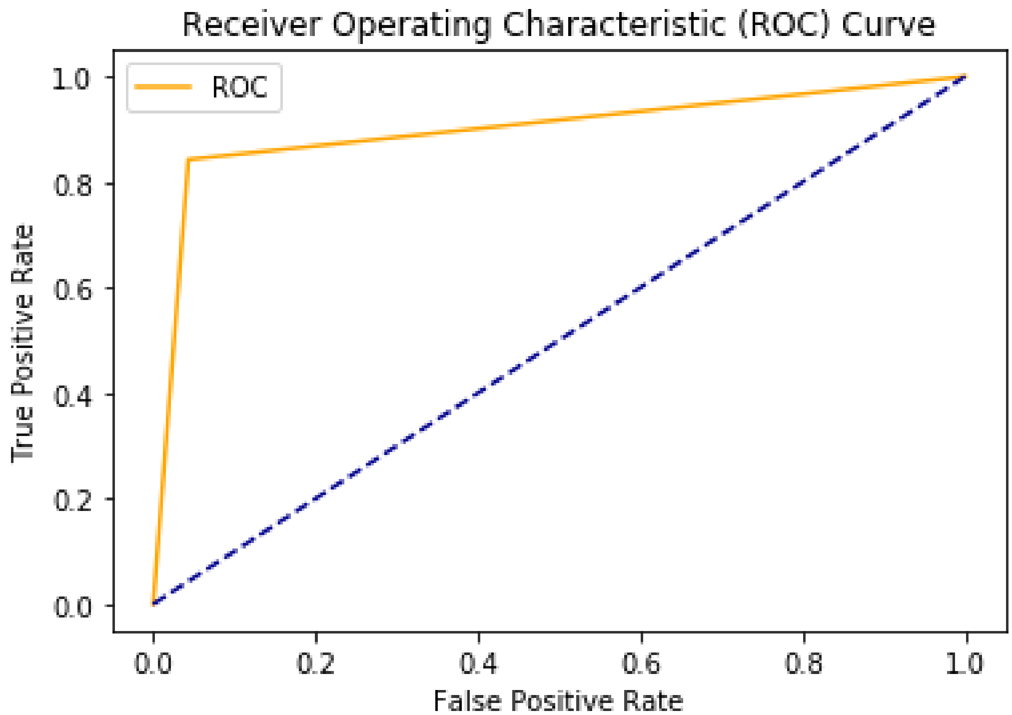
K Nearest Neighbours

[60]:
```
auc = roc_auc_score(yTest, y_pred_knn)
print('AUC: %.2f' % auc)

fpr, tpr, thresholds = roc_curve(yTest, y_pred_knn)

plot_roc_curve(fpr, tpr)
```

```
AUC: 0.90
```



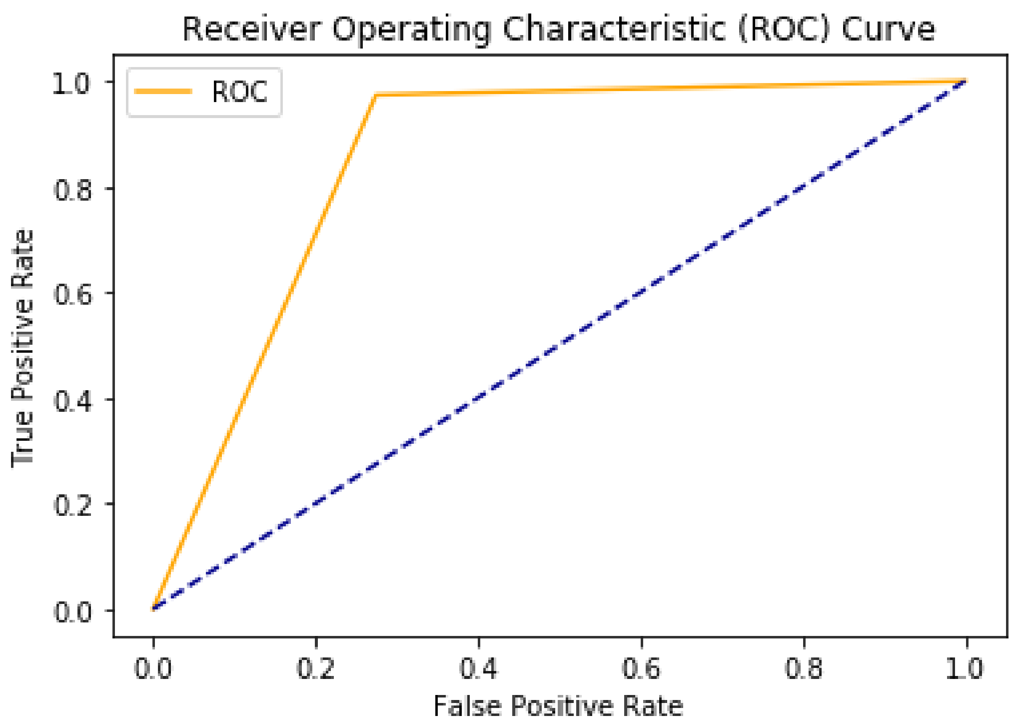
Naive Bayes

[61]:
```
auc = roc_auc_score(yTest, y_pred_nb)
print('AUC: %.2f' % auc)

fpr, tpr, thresholds = roc_curve(yTest, y_pred_nb)

plot_roc_curve(fpr, tpr)
```

```
AUC: 0.85
```



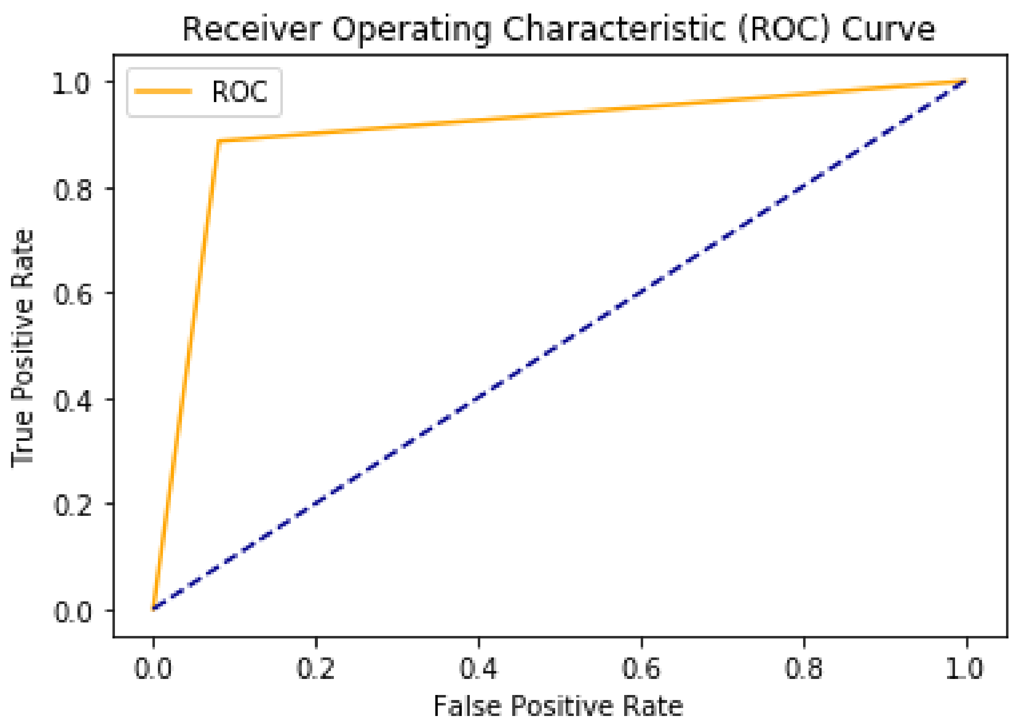
Decision Tree

[62]:
```
auc = roc_auc_score(yTest, y_pred_dtree)
print('AUC: %.2f' % auc)

fpr, tpr, thresholds = roc_curve(yTest, y_pred_dtree)

plot_roc_curve(fpr, tpr)
```

```
AUC: 0.90
```



0.0.3 Problem 3 [KNN IMPLEMENTATION]

Select 100 records from the dataset for training and 100 records for testing. Make sure that both the training and testing dataset have the same fraction of SPAM emails as the original data.

[63]:
```
xTrain1, xTest1, yTrain1, yTest1 = train_test_split(df, data_y, train_size =
↪100, test_size = 100, random_state = 0)
```

(a) Implement a function that computes the Euclidian Distance between 2 points with d features.

[151]:
```
def euclidean_distance(TestData,TestTrain):
    dist = (((TestData-TestTrain)**2).sum())**0.5
    return dist
```

(b) Write the implementation for the kNN classifier. Given the value of k and the training set, you should implement a test function that produces a predicted label for a new point x in the testing set.

[152]:
```
def label_identification(label,k):
    count_0 = 0
    count_1 = 0

    for i in range(k):
        if (label[i] == 1):
            count_1 += 1
        else:
```

```
            count_0 += 1

    if(count_0 > count_1):
        return 0
    else:
        return 1
```

[153]:
```
def predictKNN(x_train,y_train,x_test,y_test,k):
    y_pred = []
    for ki in range (x_test.shape[0]):

        distances = []
        TestData = np.array(x_test.iloc[ki,:])

        for i in range(x_train.shape[0]):

            TrainData = np.array(x_train.iloc[i,:])
            dist  = euclidean_distance(TestData,TrainData)
            distances.append((dist, i))

        distances.sort()
        label = []


        for m in range(k):
            ind = distances[m][1]
#             print(ind)
            label.append(y_train.iloc[ind])
        y_pred.append(label_identification(label,k))
    y_pred = pd.DataFrame(y_pred)

    return y_pred
```

(c) Pick several values of k (the same ones you picked in Problem 2) and print the accuracy and error metrics on the test set using your implementation of the kNN classifier.

[154]:
```
for i in range(1,40,1):
    y_pred_knn = predictKNN(xTrain1,yTrain1,xTest1,yTest1,i)
#     print(y_pred_knn)
    acc = metrics.accuracy_score(yTest1,y_pred_knn)
    print("Accuracy for K =", i, "is: ",acc)
    print("Error: ", 1 - acc)
```

```
Accuracy for K = 1 is:  0.74
Error:  0.26
Accuracy for K = 2 is:  0.64
Error:  0.36
Accuracy for K = 3 is:  0.82
Error:  0.18000000000000005
```

```
Accuracy for K = 4 is:  0.79
Error:  0.20999999999999996
Accuracy for K = 5 is:  0.8
Error:  0.19999999999999996
Accuracy for K = 6 is:  0.8
Error:  0.19999999999999996
Accuracy for K = 7 is:  0.83
Error:  0.17000000000000004
Accuracy for K = 8 is:  0.81
Error:  0.18999999999999995
Accuracy for K = 9 is:  0.83
Error:  0.17000000000000004
Accuracy for K = 10 is:  0.79
Error:  0.20999999999999996
Accuracy for K = 11 is:  0.78
Error:  0.21999999999999997
Accuracy for K = 12 is:  0.77
Error:  0.22999999999999998
Accuracy for K = 13 is:  0.77
Error:  0.22999999999999998
Accuracy for K = 14 is:  0.76
Error:  0.24
Accuracy for K = 15 is:  0.77
Error:  0.22999999999999998
Accuracy for K = 16 is:  0.78
Error:  0.21999999999999997
Accuracy for K = 17 is:  0.78
Error:  0.21999999999999997
Accuracy for K = 18 is:  0.77
Error:  0.22999999999999998
Accuracy for K = 19 is:  0.76
Error:  0.24
Accuracy for K = 20 is:  0.78
Error:  0.21999999999999997
Accuracy for K = 21 is:  0.79
Error:  0.20999999999999996
Accuracy for K = 22 is:  0.78
Error:  0.21999999999999997
Accuracy for K = 23 is:  0.77
Error:  0.22999999999999998
Accuracy for K = 24 is:  0.77
Error:  0.22999999999999998
Accuracy for K = 25 is:  0.75
Error:  0.25
Accuracy for K = 26 is:  0.75
Error:  0.25
Accuracy for K = 27 is:  0.75
Error:  0.25
```

```
Accuracy for K = 28 is:  0.76
Error:  0.24
Accuracy for K = 29 is:  0.75
Error:  0.25
Accuracy for K = 30 is:  0.78
Error:  0.21999999999999997
Accuracy for K = 31 is:  0.76
Error:  0.24
Accuracy for K = 32 is:  0.78
Error:  0.21999999999999997
Accuracy for K = 33 is:  0.77
Error:  0.22999999999999998
Accuracy for K = 34 is:  0.81
Error:  0.18999999999999995
Accuracy for K = 35 is:  0.78
Error:  0.21999999999999997
Accuracy for K = 36 is:  0.81
Error:  0.18999999999999995
Accuracy for K = 37 is:  0.79
Error:  0.20999999999999996
Accuracy for K = 38 is:  0.83
Error:  0.17000000000000004
Accuracy for K = 39 is:  0.82
Error:  0.18000000000000005
```

(d) Compare the results obtained by your implementation with those obtained with the package (on the same dataset). Are the results similar or different? If there are differences, explain why.

- The accuracy is high (83) for K values 3,7,9 and 38 in the implemented model. Where as the package model has an accuracy of 91 for K values 11 and 21.

(e) Report the running time of kNN testing averaged over all the points in the testing set.

- Running time of implemented KNN over all points for various K values from 1 to 40 is less than 40 seconds.

0.0.4 Problem 4 [CROSS VALIDATION]

(a) Implement k-fold cross-validation (CV) for training a model. The CV algorithm consists of the following steps: - Divide the entire data into k partitions of equal size. - Run k experiments. In each experiment i {1, . . . , k}, train on k 1 partitions and test on the validation set (partition i). - Record the validation error for each experiment. - Compute and print the average validation error across all k experiments

(b) Run the CV experiment for logistic regression and LDA for k {5, 10}. You can use a package for training the logistic regression and LDA models. Print for each model the average validation error for each value of k.

[82]:
```python
def cross_validation_split(x_data, y_data, k):
    kf = KFold(n_splits=k)
```

```
    kf.get_n_splits(x_data)
    err_lr = 0
    err_lda = 0
    for i_train, i_test in kf.split(x_data):
        xTrain2, xTest2 = x_data.iloc[i_train], x_data.iloc[i_test]
        yTrain2, yTest2 = y_data.iloc[i_train], y_data.iloc[i_test]

        xTrain_cv = pd.DataFrame(data = xTrain2)
        xTest_cv = pd.DataFrame(data = xTest2)
        yTrain_cv = pd.DataFrame(data = yTrain2)
        yTest_cv = pd.DataFrame(data = yTest2)

        logisticRegr = LogisticRegression()
        fit_lr = logisticRegr.fit(xTrain_cv, yTrain_cv)
        fit_lr
        y_pred_lr_cv = fit_lr.predict(xTest_cv)
        acc_lr = metrics.accuracy_score(yTest_cv,y_pred_lr_cv)
        er_lr = 1 - acc_lr
        print("Validation Error for Logistic regression: ", er_lr)
        err_lr += er_lr
        lda = LDA(n_components=1)
        fit_lda = lda.fit(xTrain_cv, yTrain_cv)
        y_pred_lda_cv = fit_lda.predict(xTest_cv)
        acc_lda = metrics.accuracy_score(yTest_cv,y_pred_lda_cv)
        er_lda = 1 - acc_lda
        print("Validation Error for LDA: ", er_lda)
        err_lda += er_lda
    print("Average validation error across all k experiments for logistic␣
↪regression: ", err_lr/k )
    print("Average validation error across all k experiments for LDA: ",␣
↪err_lda/k )
```

[89]:
```
for i in range(5,20,5):
    print("Kfolds: ", i)
    cross_validation_split(df, data_y, i)
```

```
Kfolds:  5
Validation Error for Logistic regression:  0.1934782608695652
Validation Error for LDA:  0.37173913043478257
Validation Error for Logistic regression:  0.17500000000000004
Validation Error for LDA:  0.2934782608695652
Validation Error for Logistic regression:  0.05543478260869561
Validation Error for LDA:  0.05869565217391304
Validation Error for Logistic regression:  0.10434782608695647
Validation Error for LDA:  0.060869565217391286
Validation Error for Logistic regression:  0.1815217391304348
Validation Error for LDA:  0.133695652173913
Average validation error across all k experiments for logistic regression:
```

```
0.14195652173913043
Average validation error across all k experiments for LDA:  0.183695652173913
Kfolds:  10
Validation Error for Logistic regression:  0.16956521739130437
Validation Error for LDA:  0.3543478260869565
Validation Error for Logistic regression:  0.11304347826086958
Validation Error for LDA:  0.2543478260869565
Validation Error for Logistic regression:  0.11739130434782608
Validation Error for LDA:  0.19999999999999996
Validation Error for Logistic regression:  0.18913043478260871
Validation Error for LDA:  0.2804347826086957
Validation Error for Logistic regression:  0.03260869565217395
Validation Error for LDA:  0.036956521739130443
Validation Error for Logistic regression:  0.05869565217391304
Validation Error for LDA:  0.06304347826086953
Validation Error for Logistic regression:  0.12826086956521743
Validation Error for LDA:  0.06304347826086953
Validation Error for Logistic regression:  0.05652173913043479
Validation Error for LDA:  0.05652173913043479
Validation Error for Logistic regression:  0.050000000000000044
Validation Error for LDA:  0.05217391304347829
Validation Error for Logistic regression:  0.1804347826086956
Validation Error for LDA:  0.11521739130434783
Average validation error across all k experiments for logistic regression:
0.10956521739130436
Average validation error across all k experiments for LDA:  0.1476086956521739
Kfolds:  15
Validation Error for Logistic regression:  0.1628664495114006
Validation Error for LDA:  0.2964169381107492
Validation Error for Logistic regression:  0.13355048859934848
Validation Error for LDA:  0.2996742671009772
Validation Error for Logistic regression:  0.0912052117263844
Validation Error for LDA:  0.22149837133550487
Validation Error for Logistic regression:  0.08794788273615639
Validation Error for LDA:  0.18892508143322473
Validation Error for Logistic regression:  0.14332247557003253
Validation Error for LDA:  0.254071661237785
Validation Error for Logistic regression:  0.19869706840390877
Validation Error for LDA:  0.25732899022801303
Validation Error for Logistic regression:  0.03583061889250816
Validation Error for LDA:  0.032573289902280145
Validation Error for Logistic regression:  0.032573289902280145
Validation Error for LDA:  0.03583061889250816
Validation Error for Logistic regression:  0.06514657980456029
Validation Error for LDA:  0.07166123778501632
Validation Error for Logistic regression:  0.05211726384364823
Validation Error for LDA:  0.055374592833876246
Validation Error for Logistic regression:  0.065359477124183
```

```
Validation Error for LDA:  0.07189542483660127
Validation Error for Logistic regression:  0.03594771241830064
Validation Error for LDA:  0.02941176470588236
Validation Error for Logistic regression:  0.05228758169934644
Validation Error for LDA:  0.0490196078431373
Validation Error for Logistic regression:  0.08496732026143794
Validation Error for LDA:  0.07843137254901966
Validation Error for Logistic regression:  0.09477124183006536
Validation Error for LDA:  0.0816993464052288
Average validation error across all k experiments for logistic regression:
0.0891060441549041
Average validation error across all k experiments for LDA:  0.13492083767998697
```

(c) Which model performs better? Compare the results.

- Logistic regression performs slightly better than the LDA model as the error is less for logistic regression than the LDA model. However, the difference between errors are very minimum.

Problem 5 Naive Bayes.

(a) $$P(Y=1 \mid X_1=1, X_2=0, X_3=1) = \frac{P(X_1=1, X_2=0, X_3=1 \mid Y=1)\,P(Y=1)}{P(X_1=1, X_2=0, X_3=1 \mid Y=0)\,P(Y=0) + P(X_1=1, X_2=0, X_3=1 \mid Y=1)\,P(Y=1)}$$

Using Naive Bayes Algorithm,

$$\Rightarrow \frac{P(X_1=1/Y=1)\,P(X_2=0/Y=1)\,P(X_3=1/Y=1)\,P(Y=1)}{P(X_1=1 \mid Y=0)\,P(X_2=0 \mid Y=0)\,P(X_3=1 \mid Y=0)\,P(Y=0) + P(X_1=1 \mid Y=1)\,P(X_2=0 \mid Y=1)\,P(X_3=1 \mid Y=1)\,P(Y=1)}$$

$$\Rightarrow \frac{\left(\frac{3}{4}\right)\left(\frac{1}{4}\right)\left(\frac{1}{\cancel{4}}\right)\left(\frac{\cancel{4}}{7}\right)}{\left(\frac{1}{3}\right)\left(\frac{2}{\cancel{3}}\right)(1)\left(\frac{\cancel{3}}{7}\right) + \left(\frac{3}{4}\right)\left(\frac{1}{4}\right)\left(\frac{1}{\cancel{4}}\right)\left(\frac{\cancel{4}}{7}\right)} = \frac{\frac{3}{112}}{\frac{6}{21} + \frac{3}{112}} = \frac{0.027}{\frac{735}{2352}} = \frac{0.0267}{0.3125} = \underline{0.0857}$$

$$P(Y=1 \mid X_1=1, X_2=1, X_3=1) = \frac{P(X_1=1, X_2=1, X_3=1 \mid Y=1)\,P(Y=1)}{P(X_1=1, X_2=1, X_3=1 \mid Y=0)\,P(Y=0) + P(X_1=1, X_2=1, X_3=1 \mid Y=1)\,P(Y=1)}$$

Using Naive Bayes theorem,

$$\Rightarrow \frac{P(X_1=1 \mid Y=1)\,P(X_2=1 \mid Y=1)\,P(X_3=1 \mid Y=1)\,P(Y=1)}{P(X_1=1 \mid Y=0)\,P(X_2=1 \mid Y=0)\,P(X_3=1 \mid Y=0)\,P(Y=0) + P(X_1=1 \mid Y=1)\,P(X_2=1 \mid Y=1)\,P(X_3=1 \mid Y=1)\,P(Y=1)}$$

$$\Rightarrow \frac{\left(\frac{3}{4}\right)\left(\frac{3}{4}\right)\left(\frac{1}{\cancel{4}}\right)\left(\frac{\cancel{4}}{7}\right)}{\left(\frac{1}{3}\right)\left(\frac{1}{\cancel{3}}\right)\left(\frac{\cancel{3}}{7}\right)\cdot 1 + \left(\frac{3}{4}\right)\left(\frac{3}{4}\right)\left(\frac{1}{\cancel{4}}\right)\left(\frac{\cancel{4}}{4}\right)} \Rightarrow \frac{\frac{3}{112}}{\frac{1}{21} + \frac{3}{112}} \Rightarrow \frac{0.0267}{0.075} = \underline{0.357}$$

total parameters = $\underline{7}$.

(b) $$P(Y=1 \mid X_1=1, X_2=0, X_3=1) = \frac{P(X_1=1, X_2=0, X_3=1 \mid Y=1)\,P(Y=1)}{P(X_1=1, X_2=0, X_3=1 \mid Y=0)\,P(Y=0) + P(X_1=1, X_2=0, X_3=1 \mid Y=1)\,P(Y=1)}$$

$$\Rightarrow \frac{0 \times \frac{4}{7}}{\frac{1}{3} \times \frac{3}{7} + 0} = 0.$$

$$P(Y=1 \mid X_1=1, X_2=1, X_3=1) = \frac{P(X_1=1, X_2=1, X_3=1 \mid Y=1)\,P(Y=1)}{P(X_1=1, X_2=1, X_3=1 \mid Y=0)\,P(Y=0) + P(X_1=1, X_2=1, X_3=1 \mid Y=0)\,P(Y=0)}$$

$$\Rightarrow \frac{0 \times 4/7}{0 \times \frac{3}{7} + 0 \times \frac{4}{7}} = \infty$$

total parameters = $\underline{15}$.

ⓐ parameter estimation:

no. of classes $(k) = 2$.

no. of categorical variables $(p) = 3$.

compute $P(Y=0)$, since $[P(Y=1) = 1 - P(Y=0)]$

compute $P(X_n = 0 | Y = y)$ $n = \{1,2,3\}$ and $y = \{0,1\}$

$[P(X_n = 1 | Y = y) = 1 - P(X_n = 0 | Y = y)]$

$\therefore$ total parameters $= P(X_n = 0 | Y = y) + P(Y=0)$

$= (k)(p) + (k-1)$

$= (2)(3) + 1$

$= \underline{\underline{7}}$

ⓑ Parameter Estimation:

no. of classes $= 2$ (k)

no. of categorical variables $(p) = 3$

$\Rightarrow k(2^p - 1) + (k-1)$

take all possible combinations of X_1, X_2 and X_3.

$\Rightarrow 2(2^3 - 1) + (1)$

$\Rightarrow 2(8-1) + 1$

$\Rightarrow \underline{\underline{15}}$.

Problem 6 Logistic Regression.

(a) Predicted probability $\Rightarrow \hat{P}(x) = \dfrac{e^{+\beta_0+\beta_1 x_1 + x_2}}{\left(1+e^{\beta_0+\beta_1 x_1 + x_2}\right)}$

$h(x) = \dfrac{e^{-z}}{1+e^{-z}}$

$z = (\theta^T x)$

$$= \frac{e^{-6+0.05x_1+x_2}}{1+e^{-6+0.05x_1+x_2}}$$

$P(x) = \dfrac{e^{z}}{1+e^{z}}$

$$1-P(x) = 1 - \frac{e^{-6+0.05x_1+x_2}}{1+e^{-6+0.05x_1+x_2}}$$

$e^{z}(1-P(x)) = P(x)$.

$$= \frac{1}{1+e^{-6+0.05x_1+x_2}}$$

$\dfrac{P(x)}{1-P(x)} = e^{z}$

$\ln\left(\dfrac{P(x)}{1-P(x)}\right) = z$

Let $x_1 = 40$ hrs $x_2 = 3.5$ GPA.

$$\ln\left(\frac{P(x)}{1-P(x)}\right) = -6+0.05(40)+3.5$$

$$\ln\left(\frac{P(x)}{1-P(x)}\right) = -0.5$$

$$\frac{P(x)}{1-P(x)} = e^{-0.5}$$

The probability that a student gets an A in class $\Big\}$ $= \underline{0.38}$

(b)

$$\ln\left(\frac{P(x)}{1-P(x)}\right) = \beta_0 + \beta_1 X_1 + \beta_2 X_2$$

$$0.5 \Rightarrow -6 + 0.05 * X_1 + 1 * 3.5$$

$$0.05 X_1 = 3.5 - 0.5 - 6$$

$$X_1 = \frac{-3}{0.05}$$

$$\boxed{X_1 = 60 \text{ hours}}$$