Data Mining Assignment 2

Zhengyuan Zhu
CSE5334 - Data Mining
University of Texas at Arlington

April 4, 2020

Problem 1: Non-parameteric density estimation

Solution for 1.1

First, we set up all the parameters for generating the Gaussian random data where the $\mu_1 = 5$ and $\sigma_1 = 1$

```
data_5_1 = np.random.normal(5, 1, 1000)
```

Then, we write a function p = mykde(X, h, x) that performs kernel density estimation on data X with bandwidth h for 1-Dimensional data. The function take X as the samples that calculate the density of gaussian distribution on x. It traverses all the samples and add each sample's density one by one.

```
def mykde(X, h, x):
    density = np.zeros(len(X))
    for i in range(len(X)):
        density += np.exp(-(x-X[i])**2/(2*h**2))/(h*np.sqrt(2*np.pi))/len(
                                                            X)
    return density
```

The figure below is the visualization of the data distribution:

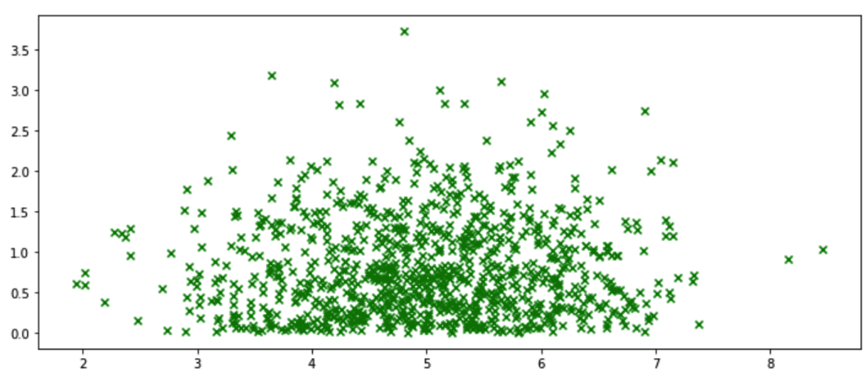


Figure 1: Fake data with μ_1, σ_1

Then we do some visualizations of the probability density function. The figure below shows that the different densities with different bandwidth. The red line represents the ground truth distribution and the histogram represents the samples distribution. It shows that the distribution curve becomes more flat when bandwidths get larger and larger.

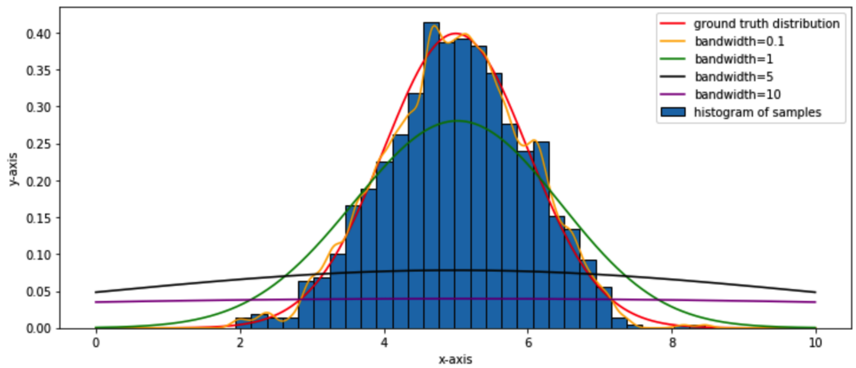


Figure 2: Histogram of X along with the different estimated probability density function

Solution for 1.2

For the problem 1.2, the methodology essentially is the same as problem 1.1. The only different is that we add a new random normal distribution data with $\mu_2 = 0$ and $\sigma = 0, 2$. We use the codes below for the visualization of results. And the conclusion is pretty much the same as problem 1.1. The larger bandwidth we got, the more flat probability density function in the figure.

```
fig = plt.figure(figsize=(12, 5))
ax = fig.add_subplot(111)
plt.hist(data_5_1, bins=30, edgecolor='black', density=True, label='
                                          histogram of samples')
# Plot kernel function
x = np.arange(0, 10, 0.01)
y = normal(x, 5, 1)
d1 = mykde(data_5_1, 0.1, x)
d2 = mykde(data_5_1, 1, x)
d3 = mykde(data_5_1, 5, x)
d4 = mykde(data_5_1, 10, x)
# Plot the ground truth distribution and estimations
plt.plot(x, y, color='red', label='ground truth distribution')
ax.plot(x, d1, label='bandwidth=0.1', color='orange')
ax.plot(x, d2, label='bandwidth=1', color='green')
ax.plot(x, d3, label='bandwidth=5', color='black')
ax.plot(x, d4, label='bandwidth=10', color='purple')

plt.legend()
plt.xlabel('x-axis')
plt.ylabel('y-axis')
```

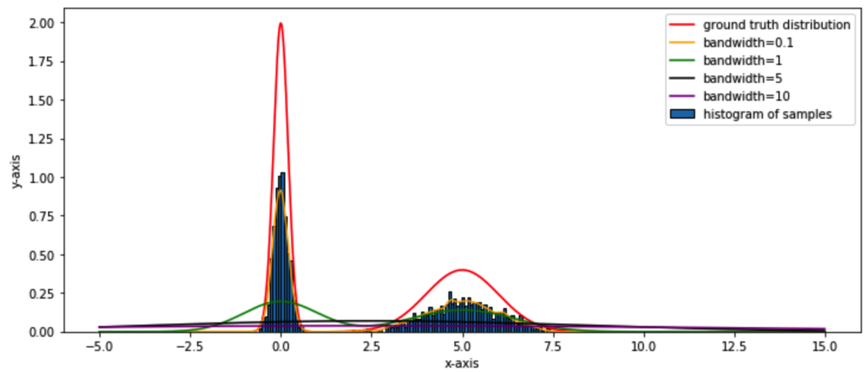


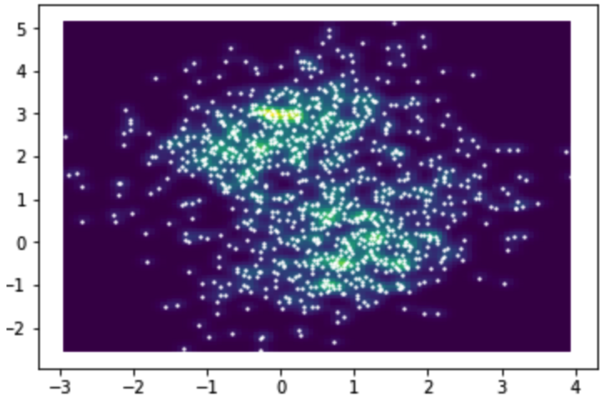
Figure 3: Histogram of X along with the different estimated probability density function

Solution for 1.3

When we have 2 sets of 2-D Gaussian random data, we need to modify mykde function that can fit the 2-D gaussian distribution. I have tried a lot but failed, so that I decide to modify the sklearn vanilla Kernel Density and stack the data we generated to a grid form. The source codes of mykde2D is simple and easy to understand.

```
from sklearn.neighbors import KernelDensity

def mykde2D(x, y, bandwidth, xbins=100j, ybins=100j, **kwargs):
    # create grid of sample locations (default: 100x100)
    xx, yy = np.mgrid[x.min():x.max():xbins,
                      y.min():y.max():ybins]

    xy_sample = np.vstack([yy.ravel(), xx.ravel()]).T
    xy_train  = np.vstack([y, x]).T

    kde_skl = KernelDensity(bandwidth=bandwidth, **kwargs)
    kde_skl.fit(xy_train)

    # score_samples() returns the log-likelihood of the samples
    z = np.exp(kde_skl.score_samples(xy_sample))
    return xx, yy, np.reshape(z, xx.shape)
```

The visualization of 2D gaussian distribution is a little bit tricky. The white dots represent the 2D gaussian distribution's samples. And I use a colored mesh to create a pseudocolor plot with a non-regular rectangular grid. The shade that cover the figure shows the μ and σ as results of mykde2D. The results shows that when bandwidth is small(such as 0.1), the estimated distribution seems like to be "overfitting". And vice versa the estimated distribution become "underfitting" when bandwidth is large.



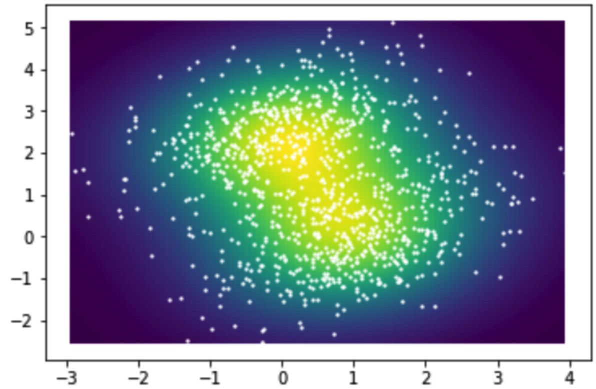
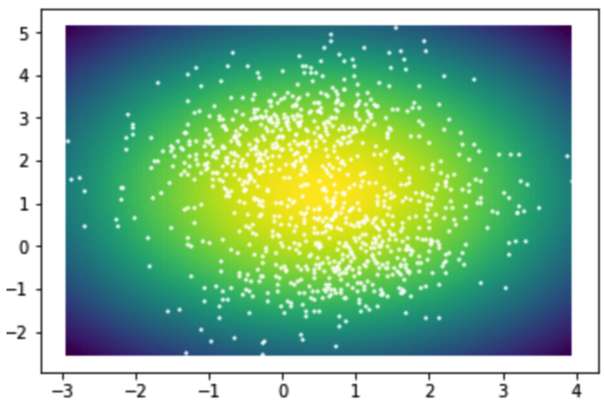


Figure 4: Test your function mykde on this data with h = 0.1

Figure 5: Test your function mykde on this data with h = 1



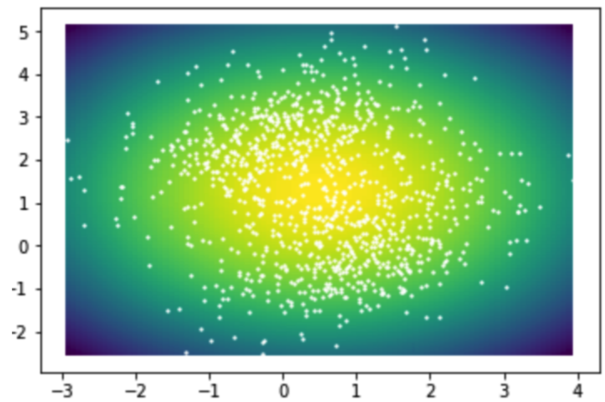


Figure 6: Test your function mykde on this data with h = 5

Figure 7: Test your function mykde on this data with h = 10

Problem 2 Solution

Solution 2.1

To solve the problem, the first thing is to generate 1000 training instances in two different classes (500 in each) from multi-variate normal distribution using the following parameters for each class. The source codes are below with samples.

```
mul1, sigma1 = [1, 0], [[1, 0.75], [0.75, 1]]
mul2, sigma2 = [0, 1], [[1, 0.75], [0.75, 1]]
size = 500

train_data2D_1 = np.random.multivariate_normal(mean=mul1, cov=sigma1, size=size)
train_data2D_1_label = np.zeros((size, 1))
print(train_data2D_1_label.shape)

```

```
train_data2D_2 = np.random.multivariate_normal(mean=mul2, cov=sigma2, size=size)
train_data2D_2_label = np.ones((size, 1))
X_train = np.vstack([train_data2D_1, train_data2D_2])
y_train = np.vstack([train_data2D_1_label, train_data2D_2_label])
print(X_train[:5], X_train[-5:])
print(y_train[:5], y_train[-5:])

# build test set
mul1, sigma1 = [1, 0], [[1, 0.75], [0.75, 1]]
mul2, sigma2 = [0, 1], [[1, 0.75], [0.75, 1]]
size = 500

test_data2D_1 = np.random.multivariate_normal(mean=mul1, cov=sigma1, size=size)
test_data2D_1_label = np.zeros((size, 1))
print(test_data2D_1_label.shape)

test_data2D_2 = np.random.multivariate_normal(mean=mul2, cov=sigma2, size=size)
test_data2D_2_label = np.ones((size, 1))
X_test = np.vstack([test_data2D_1, test_data2D_2])
y_test = np.vstack([test_data2D_1_label, test_data2D_2_label])
print(X_test[:5], X_test[-5:])
print(y_test[:5], y_test[-5:])
---------------------------------------------------------------------
(500, 1)
[[-0.19002878 -0.14795651]
 [-0.27783423 -1.46171299]
 [ 0.78327396  0.39920896]
 [ 0.57195261 -0.04425206]
 [ 1.0097563  -1.48342926]] [[ 1.0917887   2.14877639]
 [ 0.20435192  1.65422036]
 [-1.49104023  0.3028843 ]
 [-0.48226042  0.62071859]
 [-0.26311759 -0.31374102]]
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]] [[1.]
 [1.]
 [1.]
 [1.]
 [1.]]

 (500, 1)
[[ 1.54609753 -0.01456578]
 [ 1.49064799  0.83087897]
 [-0.59881474 -1.20925996]
 [ 0.34422204 -0.58984347]
 [ 0.42700309 -1.39169054]] [[-0.21408786  0.96185428]
 [ 0.56305549  1.13239575]
 [ 2.30235643  3.27293605]
```

```
 [ 1.5609519   1.07252668]
 [-1.14839655  0.92135342]]
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]] [[1.]
 [1.]
 [1.]
 [1.]
 [1.]]
```

Then I implement your Naive Bayes Classifier **[pred, posterior, err] = myNB(X,Y,X test,Y test)** whose inputs are the training data **X**, labels **Y** for **X**, testing data **X_test** and labels **Y_test** for **X_test** and returns predicted labels **pred**, posterior probability **posterior** with which the prediction was made and error rate **err**.

```python
import pandas as pd
import numpy as np
print(pd.__version__)

def myNB(X, y, X_test, y_test):
    def norm(x, mean, std):
        return 1/(np.sqrt(2*np.pi)*std)*np.exp(-(x-mean)**2/2*(std**2))

    X_train, y_train = pd.DataFrame(X, columns=['Feature1', 'Feature2']), pd.DataFrame(y, columns=['Class'])
    X_test, y_test = pd.DataFrame(X_test, columns=['Feature1', 'Feature2']), pd.DataFrame(y_test, columns=['Class'])
    y_unique = y_train['Class'].unique()
    train_set = pd.concat([X_train, y_train], axis=1)
    prior = np.zeros(len(y_unique))
    conditional = np.reshape(np.zeros(len(y_unique)*len(X_train.columns)*2), (len(y_unique), len(X_train.columns), 2))


    for i in range(0,len(y_unique)):
        prior[i]=(sum(y_train['Class']==y_unique[i])+1)/(len(y_train['Class'])+len(y_unique))
#     print("The prior probability of each class is: ", prior)

    for i, h in enumerate(X_train.columns.values.tolist()):
        for j in range(0,len(y_unique)):
            class_feature = train_set[h].loc[(train_set['Class']==y_unique[j])]
            mean = np.mean(class_feature)
            var = np.std(class_feature)
            conditional[i][j] = [mean, var]
#             print("mean and standard variance of current feature is: ", h, j, mean, var)
#     print(conditional)
```

```python
#     print("Prior distribution is: ", prior)
    pred_probs = []
    for idx, row in X_test.iterrows():
        probs = []
        for cIdx, pri in enumerate(prior): # class 0 1
            for fIdx, feat in enumerate(row): # feature 0 1
                pri *= norm(feat, conditional[fIdx][cIdx][0], conditional[
                                                         fIdx][cIdx][1])
            probs.append(pri)
        pred_probs.append(probs)
    pred = [np.argmax(p) for p in pred_probs]
    # calculate error rate
#     print(np.array(pred).shape)
#     print(y_test.to_numpy().shape)
    err = np.mean(np.array(pred)!=y_test.to_numpy().squeeze())
#     print(err)

    return pred, conditional, err

pred, posterior, err = myNB(X_train, y_train, X_test, y_test)
```

For the experiments part, I perform the experiments 10 times and take an average of error rate. The source codes and results are shown below.

```python
# perform the experiments 10 times
avg_err, run_time = 0, 10
for i in range(run_time):
    train_data2D_1 = np.random.multivariate_normal(mean=mul1, cov=sigma1,
                                          size=size)
    train_data2D_1_label = np.zeros((size, 1))

    train_data2D_2 = np.random.multivariate_normal(mean=mul2, cov=sigma2,
                                          size=size)
    train_data2D_2_label = np.ones((size, 1))
    X_train = np.vstack([train_data2D_1, train_data2D_2])
    y_train = np.vstack([train_data2D_1_label, train_data2D_2_label])

    test_data2D_1 = np.random.multivariate_normal(mean=mul1, cov=sigma1,
                                          size=size)
    test_data2D_1_label = np.zeros((size, 1))

    test_data2D_2 = np.random.multivariate_normal(mean=mul2, cov=sigma2,
                                          size=size)
    test_data2D_2_label = np.ones((size, 1))
    X_test = np.vstack([test_data2D_1, test_data2D_2])
    y_test = np.vstack([test_data2D_1_label, test_data2D_2_label])

    pred, posterior, err = myNB(X_train, y_train, X_test, y_test)
    print("current time error: ", err)
    avg_err += err/run_time
print("The average error: ", avg_err)
-----------------------------------------------------------------------
current time error:  0.117
current time error:  0.057
```

```
current time error:  0.101
current time error:  0.061
current time error:  0.086
current time error:  0.08
current time error:  0.079
current time error:  0.08
current time error:  0.071
current time error:  0.09
The average error:  0.08219999999999998
```

I have performed prediction on the testing data with source code. The accuracy, precision, recall, confusion matrix as well as a scatter plot of data points whose labels are color coded are shown below.

```
from sklearn import metrics
import matplotlib.pyplot as plt

train_data2D_1 = np.random.multivariate_normal(mean=mul1, cov=sigma1, size=size)
train_data2D_1_label = np.zeros((size, 1))

train_data2D_2 = np.random.multivariate_normal(mean=mul2, cov=sigma2, size=size)
train_data2D_2_label = np.ones((size, 1))
X_train = np.vstack([train_data2D_1, train_data2D_2])
y_train = np.vstack([train_data2D_1_label, train_data2D_2_label])

test_data2D_1 = np.random.multivariate_normal(mean=mul1, cov=sigma1, size=size)
test_data2D_1_label = np.zeros((size, 1))

test_data2D_2 = np.random.multivariate_normal(mean=mul2, cov=sigma2, size=size)
test_data2D_2_label = np.ones((size, 1))
X_test = np.vstack([test_data2D_1, test_data2D_2])
y_test = np.vstack([test_data2D_1_label, test_data2D_2_label])

pred, posterior, err = myNB(X_train, y_train, X_test, y_test)

# report the metrics
acc = metrics.accuracy_score(np.array(pred), y_test.squeeze())
print("Accuracy = %.3f" % (sum([p==a for p, a in zip(np.array(pred), y_test.squeeze())])/len(np.array(pred))))
precision = metrics.precision_score(np.array(pred), y_test.squeeze())
print("Precision = %.3f" % precision)
recall = metrics.recall_score(np.array(pred), y_test.squeeze())
print("Metrics = %.3f" % recall)
cm = metrics.confusion_matrix(np.array(pred), y_test.squeeze())
print("Confusion Matrix = {}".format(cm))

# draw data points
plt.scatter(train_data2D_1[:, 0], train_data2D_1[:, 1], label='$\mu_1, \sigma_1$')
```

```
plt.scatter(train_data2D_2[:, 0], train_data2D_2[:, 1], label='$\mu_2, \
                                            sigma_2$')
plt.legend()
plt.show()
------------------------------------------------------------------------
Accuracy = 0.912
Precision = 0.896
Metrics = 0.926
Confusion Matrix = [[464  52]
                    [ 36 448]]
```

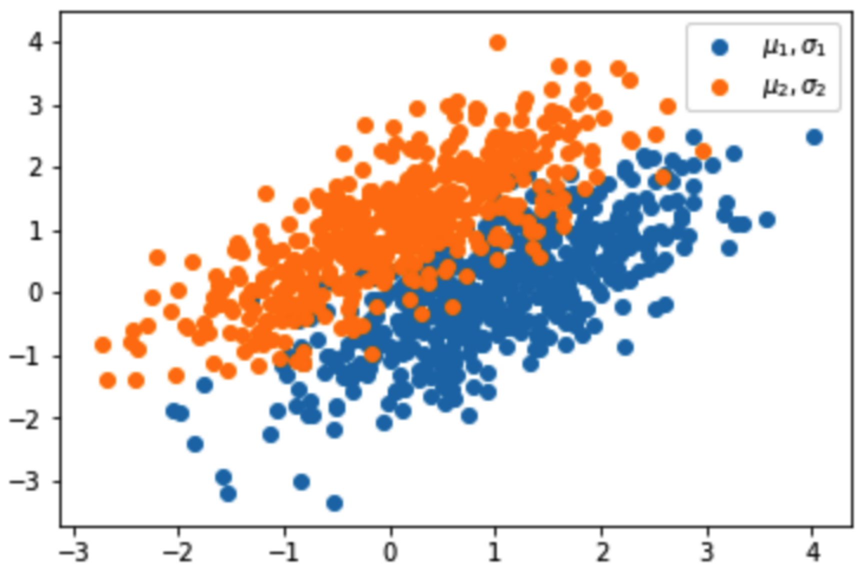


Figure 8: Dataset distribution with two differnet colors

And in my training data, I have changed the number of examples in each class to 10,20,50,100,300,500 and perform prediction on the testing data with my code. The plot of changes of accuracies are shown below. The result shows that the more exmaples we have, more higher accuracy we obtain.

```
# change the number of examples
accs = []
x_axis = [0, 1, 2, 3, 4, 5]
for n in [10,20,50,100,300,500]:
    train_set_1 = train_data2D_1[:n]
    train_label1 = np.zeros((n, 1))
    train_set_2 = train_data2D_2[:n]
    train_label2 = np.ones((n, 1))
    X_train = np.vstack([train_set_1, train_set_2])
    y_train = np.vstack([train_label1, train_label2])
    pred, posterior, err = myNB(X_train, y_train, X_test, y_test)
    acc = metrics.accuracy_score(np.array(pred), y_test.squeeze())
    accs.append(acc)
```

```


plt.plot(x_axis, accs, 'ro-', label='Changes of accuracies')
plt.xlabel("number of examples")
plt.ylabel("accuracy")
plt.legend()
plt.show()
```

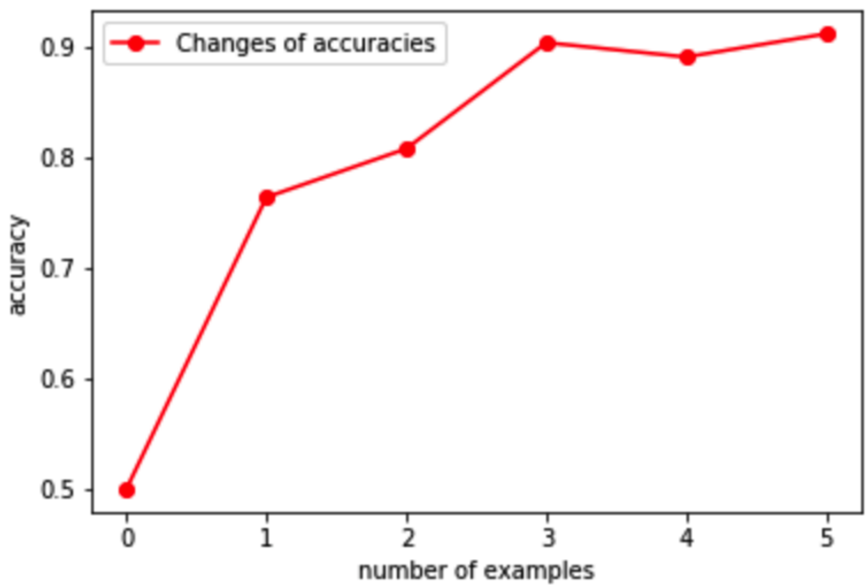


Figure 9: The changes of accuracies along with differnet number of examples

And when in the training data, change the number of examples in class 0 as 700 and the other as 300. Perform prediction on the testing dataset, I found that the accuracy becomes way much lower because of the unbalanced data.

```
# change the number of examples
train_data2D_1 = np.random.multivariate_normal(mean=mul1, cov=sigma1, size=700)
train_data2D_1_label = np.zeros((700, 1))

train_data2D_2 = np.random.multivariate_normal(mean=mul2, cov=sigma2, size=300)
train_data2D_2_label = np.ones((300, 1))

train_set_1 = train_data2D_1[:700]
train_label1 = np.zeros((700, 1))
train_set_2 = train_data2D_2[:300]
train_label2 = np.ones((300, 1))
X_train = np.vstack([train_set_1, train_set_2])
y_train = np.vstack([train_label1, train_label2])
pred, posterior, err = myNB(X_train, y_train, X_test, y_test)
acc = metrics.accuracy_score(np.array(pred), y_test.squeeze())
```

```
print("The accuracy when  change the number of examples in class 0 as 700 and the other as 300:", acc)
----------------------------------------------------------------------
The accuracy when  change the number of examples in class 0 as 700 and the other as 300: 0.834
```

Then the ROC and AUC curve are shown below:

```
No Skill: ROC AUC=0.500
Logistic: ROC AUC=0.863
```

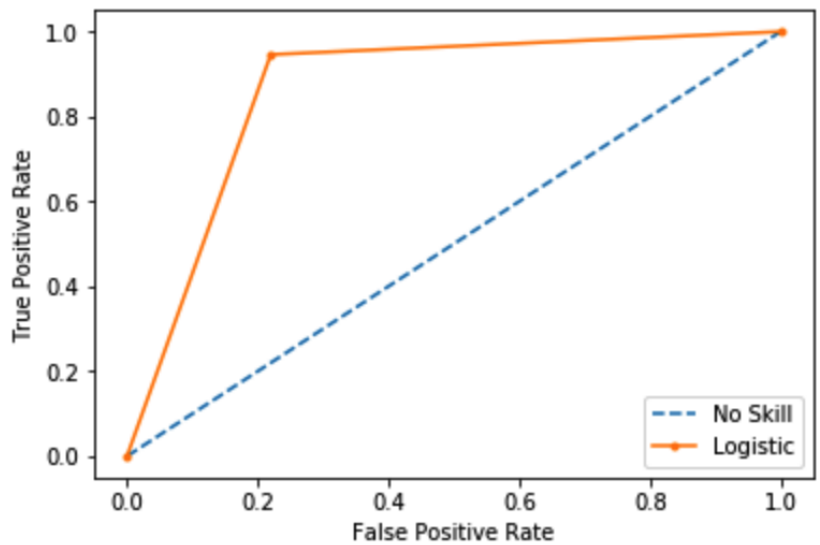


Figure 10: ROC curves from the dataset above

In the last part, I reproduce the tfidf matrix in the homework1 and print some of the reviews.

```
# Generate tfidf matrix
!pip install nltk
import nltk
nltk.download('wordnet')
nltk.download('stopwords')
import pandas as pd

review=pd.read_csv('./data/Amazon_Reviews.csv')
review['Label']=review['Label'].map({'__label__2 ':1,'__label__1 ':0})
docs_num = review.shape[0]

# Preprocessing for tokenize, stopwords removal, lemmatization
from nltk.tokenize import RegexpTokenizer
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords

tokenizer=RegexpTokenizer(r'\w+')
lemmatizer=WordNetLemmatizer()

```

```
for idx, row in review.iterrows():
    tmp = []
    tokens = tokenizer.tokenize(row['Review'])
    for w in tokens:
        if w.lower() not in stopwords.words('english'):
            tmp.append(lemmatizer.lemmatize(w.lower()))
    review.iloc[idx, 0] = ' '.join(tmp)

DF = {}
for idx, row in review.iterrows():
    for w in row['Review'].split():
        try:
            DF[w].add(idx)
        except:
            DF[w] = {idx}
DF = {k: len(v) for k, v in DF.items()}
vocab = [k for k, v in DF.items()]
vocab_len = len(vocab)

from collections import Counter
import numpy as np

TFIDF = {}
for idx, row in review.iterrows():
    rw = row['Review'].split()
    cnt = Counter(rw)
    for w in np.unique(rw):
        tf = cnt[w]/len(rw)
        df = DF[w]
        idf = np.log((docs_num+1)/(df+1))
        TFIDF[(idx, w)] = tf*idf

n = 0
for k, v in TFIDF.items():
    if n<5:
        print(k, v)
        n += 1
    else:
        break

"""
P: like(54) great(38) good(35) love(33) best(20)
N: however(15) disappointed(11) waste(10) poor(10) hard(8)
"""
def gen_vector(idx, review):
    Q = np.zeros(10)
    P = np.zeros(10)
    ten_words = ['like', 'great', 'good', 'love', 'best',
                 'however', 'disappointed', 'waste', 'poor', 'hard']
    for w in ten_words:
        for k, v in TFIDF.items():
            if k[0]==idx and k[1]==w:
                Q[ten_words.index(w)] = v
        for word in review:
```

```
            if word==w:
                P[ten_words.index(w)] += 1

    return P, Q

tfidf_vectors = []
count_vectors = []
for idx, row in review.iterrows():
    rw = row['Review'].split()
    P, Q = gen_vector(idx, rw)
    tfidf_vectors.append([Q, row['Label']])
    count_vectors.append(P)

for i in range(10):
    print("tf-idf vector for document {} is {}".format(i, tfidf_vectors[i]))
--------------------------------------------------------------------------------
tf-idf vector for document 0 is [array([0.        , 0.        , 0.        , 0.        , 0.05008433,
       0.        , 0.        , 0.        , 0.        , 0.        ]), 1]
tf-idf vector for document 1 is [array([0.        , 0.        , 0.        , 0.        , 0.09799108,
       0.        , 0.        , 0.        , 0.        , 0.        ]), 1]
tf-idf vector for document 2 is [array([0.02129965, 0.        , 0.        , 0.        , 0.03266369,
       0.        , 0.        , 0.        , 0.        , 0.        ]), 1]
tf-idf vector for document 3 is [array([0.05878704, 0.        , 0.        , 0.        , 0.0300506 ,
       0.        , 0.        , 0.        , 0.        , 0.        ]), 1]
tf-idf vector for document 4 is [array([0.        , 0.        , 0.03727823, 0.        , 0.04899554,
       0.        , 0.        , 0.        , 0.        , 0.        ]), 1]
tf-idf vector for document 5 is [array([0.        , 0.        , 0.        , 0.        , 0.06091338,
       0.        , 0.        , 0.        , 0.        , 0.        ]), 1]
tf-idf vector for document 6 is [array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]), 0]
tf-idf vector for document 7 is [array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]), 1]
tf-idf vector for document 8 is [array([0.        , 0.        , 0.        , 0.03937682, 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ]), 1]
tf-idf vector for document 9 is [array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]), 1]
```

The I use tf-idf weight matrix as features and perform 5-fold cross-validation with the Naive Bayes classifier. The result (average accuracy, precision and reacall across all folds) are shown below.

```
from sklearn.model_selection import KFold
from sklearn.naive_bayes import GaussianNB
from sklearn import metrics

```

```
dataset = [[v[0], v[1]] for v in tfidf_vectors]
kf = KFold(n_splits=5)
avg_acc, avg_precision, avg_recall = [], [], []
for train_idx, test_idx in kf.split(dataset):
    gnb = GaussianNB()
    X_train, y_train = [dataset[i][0] for i in train_idx], [dataset[i][1]
                                          for i in train_idx]
    X_test, y_test = [dataset[i][0] for i in test_idx], [dataset[i][1] for
                                           i in test_idx]
    pred = gnb.fit(X_train, y_train).predict(X_test)

    acc = metrics.accuracy_score(np.array(pred), y_test)
    print("Accuracy = %.3f" % (sum([p==a for p, a in zip(np.array(pred),
                                          y_test)])/len(np.array(pred))))
    precision = metrics.precision_score(np.array(pred), y_test)
    print("Precision = %.3f" % precision)
    recall = metrics.recall_score(np.array(pred), y_test)
    print("Recall = %.3f" % recall)
    print('-----'*4)
    avg_acc.append(acc)
    avg_precision.append(precision)
    avg_recall.append(recall)

print("Average Accuracy is: ", sum(avg_acc)/5)
print("Precision Accuracy is: ", sum(avg_precision)/5)
print("Recall Accuracy is: ", sum(avg_recall)/5)
--------------------------------------------------------------------------
                                              ------


Accuracy = 0.675
Precision = 0.955
Recall = 0.636
--------------------
Accuracy = 0.625
Precision = 1.000
Recall = 0.583
--------------------
Accuracy = 0.700
Precision = 0.913
Recall = 0.677
--------------------
Accuracy = 0.725
Precision = 1.000
Recall = 0.667
--------------------
Accuracy = 0.641
Precision = 1.000
Recall = 0.622
--------------------
Average Accuracy is:  0.6732051282051282
Precision Accuracy is:  0.9735177865612649
Recall Accuracy is:  0.6370809225647934
```

The result shows that cross validation can reduce bias and every data points get to be tested exactly once and is used in training k-1 times. The variance of the resulting estimate is reduced as k increases.

Reference

- https://jakevdp.github.io/PythonDataScienceHandbook/05.13-kernel-density-estimation.html
- https://stats.stackexchange.com/questions/206963/adaptive-variable-bandwidth-in-2d-gaussian-kernel-density-estimator-to-account