Homework 1

Chandrima Bhattacharya, Zeyu Wang

In [30]:

```
print ("Question 1")
```

```
Question 1
```

In [1]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from scipy.spatial.distance import pdist
from sklearn.metrics.pairwise import euclidean_distances
```

In [2]:

```
# a: Import the train/test files from Digit Recognizer
train = pd.read_csv('F:/Annie/CornellMS/Semester 4/Machine Learning/HW1/Digit/train.cs
v', delimiter=',')
test = pd.read_csv('F:/Annie/CornellMS/Semester 4/Machine Learning/HW1/Digit/test.csv',
delimiter=',')
train.shape
```

Out[2]:

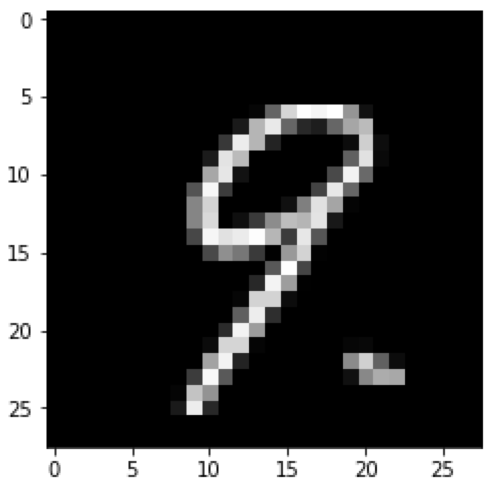
```
(42000, 785)
```

In [3]:

```
# b: Display one digit
display_image = test.iloc[2]
display_image = np.array(display_image, dtype='uint8')
pixels = display_image.reshape((28, 28))
plt.imshow(pixels, cmap='gray')
```

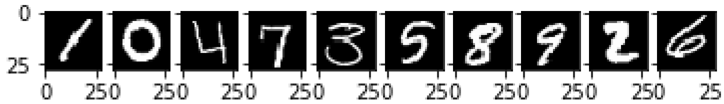
Out[3]:

```
<matplotlib.image.AxesImage at 0x250f4f3e860>
```



In [4]:

```
# b: Display one of each digit
def display_digit(file):
    unique = file['label'].unique()
    digits = {}
    for i in unique:
        digits[i] = np.where(train.label==i)[0][:1]
    fig, ax = plt.subplots(1, 10, sharex='col', sharey='row')
    for i in unique:
        display_image = train.iloc[list(digits.values())[i],1:785]
        display_image = np.array(display_image, dtype='uint8')
        pixels = display_image.reshape((28, 28))
        ax[i].imshow(pixels, cmap='gray')
display_digit(train)
```



In [5]:

```
# c: Prior probabilities of each digit
prior_prob = train['label'].value_counts(normalize=True)
print (prior_prob)
# Yes, it is almost uniform across digit because probability of each digit is .1 approx
imately.
```

```
1    0.111524
7    0.104786
3    0.103595
9    0.099714
2    0.099452
6    0.098500
0    0.098381
4    0.096952
8    0.096738
5    0.090357
Name: label, dtype: float64
```

In [25]:

```
print ("Answer", "\n", "The prior probabilities of each class is approximately 0.1, whi
ch is our expectation for nearly evenly distributed class of 10 categories of number.")
```

```
Answer
 The prior probabilities of each class is approximately 0.1, which is our
expectation for nearly evenly distributed class of 10 categories of numbe
r.
```

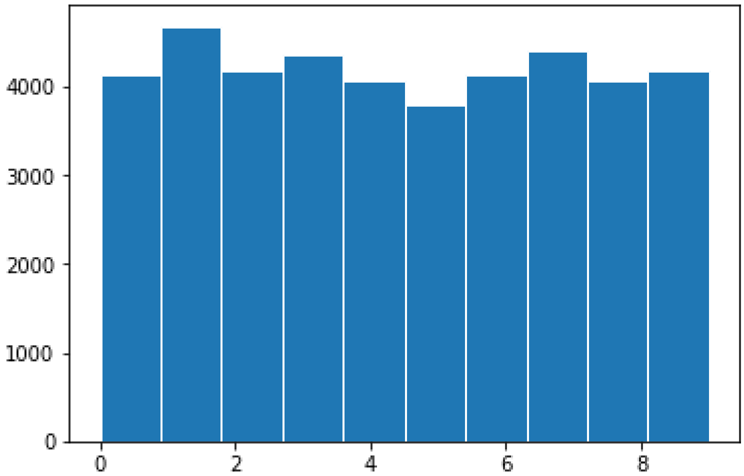
In [6]:

```
# c: Normalized histogram for each digit
plt.hist(train['label'], edgecolor='white',linewidth=1)

# Yes, the following plot reflects that it is nearly even.
```

Out[6]:

```
(array([4132., 4684., 4177., 4351., 4072., 3795., 4137., 4401., 4063.,
        4188.]),
 array([0. , 0.9, 1.8, 2.7, 3.6, 4.5, 5.4, 6.3, 7.2, 8.1, 9. ]),
 <a list of 10 Patch objects>)
```



In [27]:

```
print ("Answer", "\n", "The following plot can be mostly called even. There are 42000 d
igits in the training set, hence we would expect each digit to have a representation of
4200 which we can see approximately from the histogram. We notice that the number of 1s
are the maximum and the number of 5s is the least.")
```

```
Answer
 The following plot can be mostly called even. There are 42000 digits in t
he training set, hence we would expect each digit to have a representation
of 4200 which we can see approximately from the histogram. We notice that
the number of 1s are the maximum and the number of 5s is the least.
```

In [7]:

```
# d: Select examples of each digit
labels = np.asarray(train['label'])
def example_sample(labels):
    result = []
    for i in range(10):
        indices = np.where(labels == i)
        index = indices[0][0]
        result.append(index)
    return result
examples = example_sample(labels)
print(examples)
```

```
[1, 0, 16, 7, 3, 8, 21, 6, 10, 11]
```

In [8]:

```
# Calculate L2 distance
pixels = np.asarray(train[train.columns[1:]])
def nearest_points(example, data):
    nearest = []
    for e in example:
        target = data[e]
        dis = float('inf')
        for d in range(len(data)):
            if d == e:
                continue
            else:
                distance = np.linalg.norm(target - data[d])
                if distance < dis:
                    dis = distance
                    point = d
        nearest.append([dis, point])
    return nearest
nearest = nearest_points(examples, pixels)
print (nearest)
```

```
[[1046.5954328201515, 12950], [489.67948701165744, 29704], [1380.877257398
354, 9536], [1832.6649993929605, 8981], [1356.8809822530493, 14787], [106
6.3676664265472, 30073], [1446.5113203843239, 16240], [863.5010133172977,
15275], [1593.7775879965184, 32586], [910.5767403135224, 35742]]
```

In [9]:

```
# print error examples
error_examples = []
for i in range(len(nearest)):
    if labels[nearest[i][1]] == labels[examples[i]]:
        error_examples += [[examples[i], labels[examples[i]]]]
    else:
        error_examples += [[examples[i], labels[nearest[i][1]], '*']]
print(error_examples)
```

```
[[1, 0], [0, 1], [16, 2], [7, 5, '*'], [3, 4], [8, 5], [21, 6], [6, 7], [1
0, 8], [11, 9]]
```

In [10]:

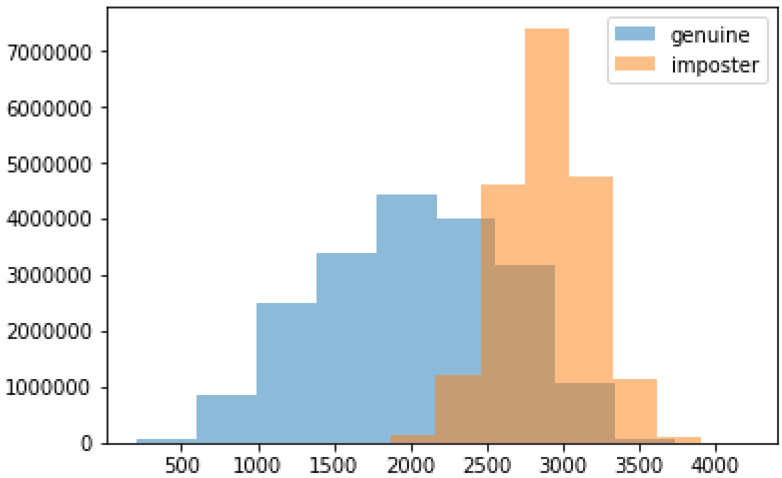
```
# e: Genuine and imposters
digit_0 = np.array(train[train.columns[1:]][train.label == 0])
digit_1 = np.array(train[train.columns[1:]][train.label == 1])
genuine_0 = np.append([], pdist(digit_0))
genuine_1 = np.append([], pdist(digit_1))
imposter = euclidean_distances(digit_0, digit_1)
```

In [11]:

```
# Plot genuine and imposters
plt.hist(np.append(genuine_0, genuine_1), alpha = 0.5, label = 'genuine')
plt.hist(imposter.flatten(), alpha = 0.5, label = 'imposter')
plt.legend(loc='upper right')
```

Out[11]:

```
<matplotlib.legend.Legend at 0x2509afc27f0>
```



In [12]:

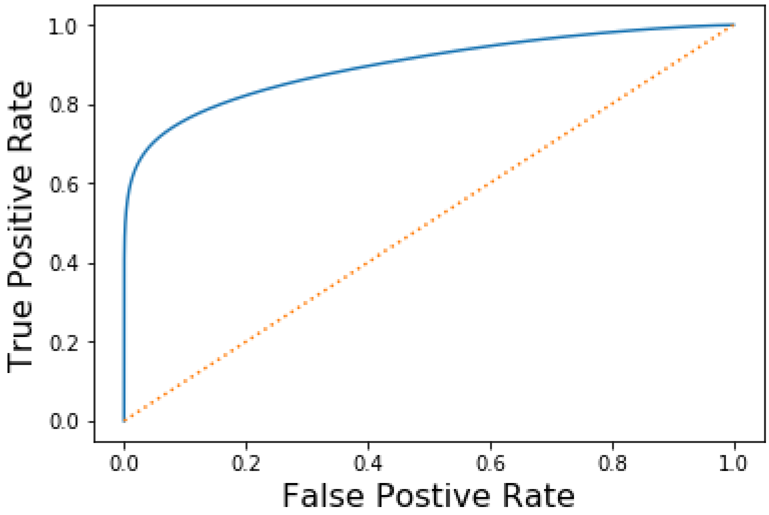
```
# f: ROC curve generation
def roc(genuine, imposter):
    theta = np.linspace(min(genuine), max(imposter) + 1, 1000)
    tpr = np.array([])
    fpr = np.array([])
    for t in theta:
        tp = np.sum(genuine < t)
        fp = np.sum(imposter < t)
        fn = len(genuine) - np.sum(genuine < t)
        tn = len(imposter) - np.sum(imposter < t)
        tpr = np.append(tpr, tp/(tp + fn))
        fpr = np.append(fpr, fp/(fp + tn))
    return tpr, fpr
```

In [13]:

```
tpr, fpr = roc(np.append(genuine_0, genuine_1), imposter.flatten())
# Plot ROC curve
fig = plt.figure()
plt.plot(fpr, tpr)
plt.plot([0, 1], [0, 1], ':')
plt.xlabel('False Postive Rate', fontsize=16)
plt.ylabel('True Positive Rate', fontsize=16)
```

Out[13]:

```
Text(0,0.5,'True Positive Rate')
```

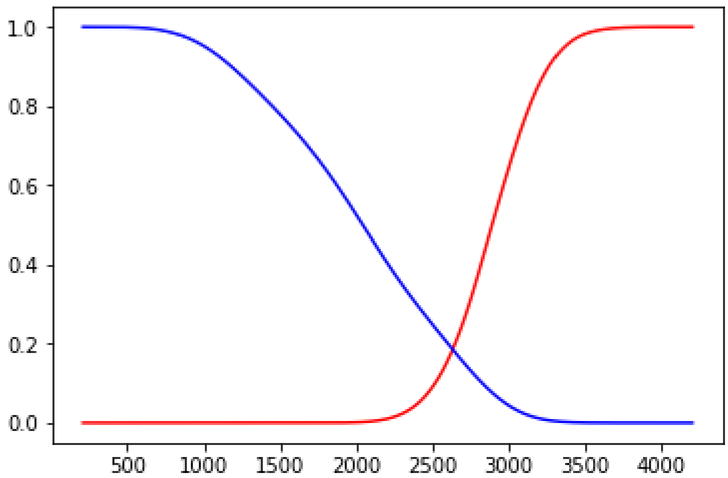


In [14]:

```
fnr = 1 - tpr
theta = np.linspace(min(np.append(genuine_0, genuine_1)), max(imposter.flatten()) + 1,
1000)
fig = plt.figure()
plt.plot(theta, fpr, 'r')
plt.plot(theta, fnr, 'b')
```

Out[14]:

```
[<matplotlib.lines.Line2D at 0x2509b081748>]
```



In [15]:

```
# Equal error generation
def equalError(fpr, fnr):
     diff = fpr - fnr
    for i in range(len(diff)):
        if diff[i] > 0:
            break
    return (fpr[i - 1] + fpr[i])/2
equalError(fpr, fnr)
```

Out[15]:

```
0.1865334958330681
```

In [29]:

```
# When a classifier guesses randomly, error rate will be high if it is a multiclass pro
blem but if it is a 2-class problem
# Also, probability of occurence of each class affects the error rate
print ("Answer", "\n", "When we have a 2-class problem, and a majority occuring class i
n it, random guess might work decently well, especially when we output the major class.
But when data-classes increases, error rate of random guess will start increasing. Rand
om guess should not be used because we are not trying to find the features and weighing
them to output class, hence usually there is possibility of incurring a high error rat
e.")
```

```
Answer
 When we have a 2-class problem, and a majority occuring class in it, rand
om guess might work decently well, especially when we output the major cla
ss. But when data-classes increases, error rate of random guess will start
increasing. Random guess should not be used because we are not trying to f
ind the features and weighing them to output class, hence usually there is
possibility of incurring a high error rate.
```

In [16]:

```
# g: knn implementation
#def split_data(data):
#    data = np.array(data)
#    np.random.shuffle(data)
#    return [data[i::n] for i in range(n)]

def knn(tr, te, k):
    tr_label = tr.T[0].T
    te_label = te.T[0].T
    tr_data = tr.T[1:].T
    te_data = te.T[1:].T
    y_hat = np.zeros(len(te_label))
    for i in range(len(te_data)):
        distance = euclidean_distances([te_data[i]], tr_data)
        ind = np.argpartition(distance[0], k)[:k]
        nearest = tr_label[ind]
        counts = np.bincount(nearest)
        y_hat[i] = np.argmax(counts)
        if i % 1000 == 0: print(i)
    accuracy = sum(y_hat==te_label)/len(te_label)
    print("Accuracy: ", accuracy)
    return y_hat, accuracy
```

In [17]:

```
# h: 3-fold classification
#data = split_data(train, 3)
three = [i for i in np.linspace(0, len(train),4)]
print (train.shape[0], three)
data = np.array(train)
np.random.shuffle(data)
sol = []
for i in range(0, 3):
    t1, t2, t3, t4, t5, t6 = three[i], three[i+1], three[(i+1)%3], three[(i+1)%3 +1], t
hree[(i+2)%3], three[(i+2)%3 +1]
    print (t1, t2, t3, t4, t5, t6 )
    test_val = data[int(t1):int(t2),:]
    train_val = np.concatenate((data[int(t3):int(t4),:], data[int(t5):int(t6),:]), axis
=0)
    print (train_val.shape, test_val.shape)
    y, acc = knn(train_val, test_val, 5)
    sol += [y, acc]
```

```
42000 [0.0, 14000.0, 28000.0, 42000.0]
0.0 14000.0 14000.0 28000.0 28000.0 42000.0
(28000, 785) (14000, 785)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
12000
13000
Accuracy:  0.9637857142857142
14000.0 28000.0 28000.0 42000.0 0.0 14000.0
(28000, 785) (14000, 785)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
12000
13000
Accuracy:  0.9656428571428571
28000.0 42000.0 0.0 14000.0 14000.0 28000.0
(28000, 785) (14000, 785)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
12000
13000
Accuracy:  0.9641428571428572
```

In [18]:

```
# Average accuracy
print (np.mean(acc))
```

```
0.9641428571428572
```

In [19]:

```
# i: Confusion matrix
test_label = test_val.T[0].T
def confusion(y_hat, y):
    table = np.zeros(shape=(10,10), dtype = np.uint16)
    np.set_printoptions(precision=0)
    for i in range(0, len(y_hat)):
        table[int(y_hat[i])][y[i]] += 1
    return table
confusion(sol[4:6][0], test_label)
```

Out[19]:

```
array([[1427,    0,    8,    1,    1,    5,    8,    0,    6,    6],
       [   0, 1572,   20,    4,   11,    3,    5,   20,   18,    6],
       [   2,    5, 1349,    6,    0,    1,    0,    3,    3,    4],
       [   0,    1,    5, 1387,    0,   25,    0,    1,   25,    9],
       [   0,    0,    1,    0, 1316,    2,    2,    6,    8,   14],
       [   1,    0,    1,   13,    0, 1179,    6,    0,   19,    1],
       [   6,    1,    3,    2,    8,   15, 1341,    0,    4,    0],
       [   1,    5,   30,    7,    2,    0,    0, 1373,    5,   28],
       [   0,    0,    2,    6,    1,    3,    1,    0, 1266,    2],
       [   1,    1,    1,    7,   30,    8,    0,   18,   18, 1288]],
      dtype=uint16)
```

In [20]:

```
#j : Train-test classification using the entire data
def knn_classifier(tr, te, k):
    tr_label = tr.T[0].T
    tr_data =  tr.T[1:].T
    y_hat = np.zeros(len(te))
    for i in range(len(te)):
        distance = euclidean_distances([te[i]], tr_data)
        ind = np.argpartition(distance[0], k)[:k]
        nearest = tr_label[ind]
        counts = np.bincount(nearest)
        y_hat[i] = np.argmax(counts)
        if i % 1000 == 0: print(i)
    return y_hat
```

```
train_data = np.array(train)
test_data = np.array(test)
test_labels = knn_classifier(train_data, test_data, 5)
```

```
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
12000
13000
14000
15000
16000
17000
18000
19000
20000
21000
22000
23000
24000
25000
26000
27000
```

In [24]:

```
final_out = pd.DataFrame(test_labels, columns=['Label'])
final_out['ImageId'] = range(1, len(final_out) + 1)
final_out = final_out.set_index('ImageId')
final_out.to_csv('F:/Annie/CornellMS/Semester 4/Machine Learning/HW1/Digit/Digit_Submis
sion.csv')
```

In [15]:


```
print ("Question 2")
```

Question 2

In [1]:

```
import pandas as pd
import numpy as np
from sklearn.cross_validation import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
```

```
C:\Users\Chandrima\Anaconda3\lib\site-packages\sklearn\cross_validation.p
y:41: DeprecationWarning: This module was deprecated in version 0.18 in fa
vor of the model_selection module into which all the refactored classes an
d functions are moved. Also note that the interface of the new CV iterator
s are different from that of this module. This module will be removed in
0.20.
  "This module will be removed in 0.20.", DeprecationWarning)
```

In [2]:

```
# a: Import the train/test files from Titanic
train = pd.read_csv('F:/Annie/CornellMS/Semester 4/Machine Learning/HW1/Titanic/train.c
sv', delimiter=',')
test = pd.read_csv('F:/Annie/CornellMS/Semester 4/Machine Learning/HW1/Titanic/test.cs
v', delimiter=',')
```

In [3]:

```
# Check training features and values present
train.notna().sum()
```

Out[3]:

```
PassengerId    891
Survived       891
Pclass         891
Name           891
Sex            891
Age            714
SibSp          891
Parch          891
Ticket         891
Fare           891
Cabin          204
Embarked       889
dtype: int64
```

In [4]:

```
def clean_data(data):
    data = data.drop(columns=['Name'], axis=0) #Name has unique value for each passenge
r and will not affect the model
    data['Sex'] = data['Sex'].replace(['male', 'female'], [0,1])
    data['Age'] = data['Age'].fillna(round(data.Age.mean()))
    data['Embarked'] = data['Embarked'].replace(['S', 'C', 'Q', np.nan], [0, 1, 2, 3])
    tickets = data.Ticket.unique()
    tickets_dic = dict(zip(tickets, range(len(tickets))))
    cabin = data.Cabin.unique()
    cabin_dic = dict(zip(cabin, range(len(cabin))))
    data = data.replace({'Ticket': tickets_dic, 'Cabin': cabin_dic})
    return data
```

In [5]:

```
train_data = clean_data(train)
train_data
```

Out[5]:

	PassengerId	Survived	Pclass	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin
0	1	0	3	0	22.0	1	0	0	7.2500	0
1	2	1	1	1	38.0	1	0	1	71.2833	1
2	3	1	3	1	26.0	0	0	2	7.9250	0
3	4	1	1	1	35.0	1	0	3	53.1000	2
4	5	0	3	0	35.0	0	0	4	8.0500	0
5	6	0	3	0	30.0	0	0	5	8.4583	0
6	7	0	1	0	54.0	0	0	6	51.8625	3
7	8	0	3	0	2.0	3	1	7	21.0750	0
8	9	1	3	1	27.0	0	2	8	11.1333	0
9	10	1	2	1	14.0	1	0	9	30.0708	0
10	11	1	3	1	4.0	1	1	10	16.7000	4
11	12	1	1	1	58.0	0	0	11	26.5500	5
12	13	0	3	0	20.0	0	0	12	8.0500	0
13	14	0	3	0	39.0	1	5	13	31.2750	0
14	15	0	3	1	14.0	0	0	14	7.8542	0
15	16	1	2	1	55.0	0	0	15	16.0000	0
16	17	0	3	0	2.0	4	1	16	29.1250	0
17	18	1	2	0	30.0	0	0	17	13.0000	0
18	19	0	3	1	31.0	1	0	18	18.0000	0
19	20	1	3	1	30.0	0	0	19	7.2250	0
20	21	0	2	0	35.0	0	0	20	26.0000	0
21	22	1	2	0	34.0	0	0	21	13.0000	6
22	23	1	3	1	15.0	0	0	22	8.0292	0
23	24	1	1	0	28.0	0	0	23	35.5000	7
24	25	0	3	1	8.0	3	1	7	21.0750	0
25	26	1	3	1	38.0	1	5	24	31.3875	0
26	27	0	3	0	30.0	0	0	25	7.2250	0
27	28	0	1	0	19.0	3	2	26	263.0000	8
28	29	1	3	1	30.0	0	0	27	7.8792	0
29	30	0	3	0	30.0	0	0	28	7.8958	0
...	...	...	...	...	...	...	...	...	...	...
861	862	0	2	0	21.0	1	0	660	11.5000	0
862	863	1	1	1	48.0	0	0	661	25.9292	136

	PassengerId	Survived	Pclass	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin
863	864	0	3	1	30.0	8	2	148	69.5500	0
864	865	0	2	0	24.0	0	0	662	13.0000	0
865	866	1	2	1	42.0	0	0	663	13.0000	0
866	867	1	2	1	27.0	1	0	664	13.8583	0
867	868	0	1	0	31.0	0	0	665	50.4958	144
868	869	0	3	0	30.0	0	0	666	9.5000	0
869	870	1	3	0	4.0	1	1	8	11.1333	0
870	871	0	3	0	26.0	0	0	667	7.8958	0
871	872	1	1	1	47.0	1	1	222	52.5542	42
872	873	0	1	0	33.0	0	0	668	5.0000	115
873	874	0	3	0	47.0	0	0	669	9.0000	0
874	875	1	2	1	28.0	1	0	274	24.0000	0
875	876	1	3	1	15.0	0	0	670	7.2250	0
876	877	0	3	0	20.0	0	0	128	9.8458	0
877	878	0	3	0	19.0	0	0	671	7.8958	0
878	879	0	3	0	30.0	0	0	672	7.8958	0
879	880	1	1	1	56.0	0	1	276	83.1583	145
880	881	1	2	1	25.0	0	1	232	26.0000	0
881	882	0	3	0	33.0	0	0	673	7.8958	0
882	883	0	3	1	22.0	0	0	674	10.5167	0
883	884	0	2	0	28.0	0	0	675	10.5000	0
884	885	0	3	0	25.0	0	0	676	7.0500	0
885	886	0	3	1	39.0	0	5	16	29.1250	0
886	887	0	2	0	27.0	0	0	677	13.0000	0
887	888	1	1	1	19.0	0	0	678	30.0000	146
888	889	0	3	1	30.0	1	2	614	23.4500	0
889	890	1	1	0	26.0	0	0	679	30.0000	147
890	891	0	3	0	32.0	0	0	680	7.7500	0

891 rows × 11 columns

In [6]:

```
X = train_data.drop(columns=['Survived'], axis=0)
y = train_data['Survived']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state
= 0)
```

In [7]:

```
sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
```

In [8]:

```
lr = LogisticRegression()
lr.fit(X_train_std, y_train)
lr.coef_
```

Out[8]:

```
array([[-1.62829646e-04, -7.46489676e-01,  1.26253889e+00,
        -5.79799973e-01, -4.41016207e-01, -5.20773726e-02,
        -8.49053018e-02,  4.30454845e-02,  2.49439696e-01,
         1.64205111e-01]])
```

In [9]:

```
predictions = lr.predict(X_test)
print(confusion_matrix(y_test, predictions))
print(classification_report(y_test, predictions))
```

```
[[110   0]
 [ 67   2]]
             precision    recall  f1-score   support

          0       0.62      1.00      0.77       110
          1       1.00      0.03      0.06        69

avg / total       0.77      0.63      0.49       179
```

In [10]:

```
print ("Answer", "\n" ,"I used the coef_ function defined in scipy which outputs the co
efficient for features in the decision matrix. Then I choose those features which has a
feature weight of more than 0.3. I had initially dropped names column as all values are
unique and wouldn't have made any difference.")
```

```
Answer
 I used the coef_ function defined in scipy which outputs the coefficient
for features in the decision matrix. Then I choose those features which ha
s a feature weight of more than 0.3. I had initially dropped names column
as all values are unique and wouldn't have made any difference.
```

In [6]:

```
# The values influencing the positive and the negative class seems to be Pclass, Sex, S
ibSp and Parch.
# I chose is based on |lr.coef_[feature]| > .3
X = train_data.drop(columns=['SibSp', 'Parch', 'Fare', 'Embarked', 'PassengerId'], axis
=0)
y = train_data['Survived']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state
= 0)
X_train_std = sc.fit_transform(X_train)
lr.fit(X_train_std, y_train)
predictions = lr.predict(X_test)
print(classification_report(y_test, predictions))
print(confusion_matrix(y_test, predictions))
# We see the Precision for the following have increased after dropping the following co
lumns
```

```
             precision    recall  f1-score   support

          0       0.72      0.98      0.83       110
          1       0.93      0.41      0.57        69

avg / total       0.81      0.76      0.73       179

[[108   2]
 [ 41  28]]
```

In [12]:

```
# c: Train on entire training set and predict test set
X_train = train_data.drop(columns=['Survived', 'SibSp', 'Parch', 'Fare', 'Embarked', 'P
assengerId'], axis=0)
X_test = clean_data(test).drop(columns=['SibSp', 'Parch', 'Fare', 'Embarked', 'Passenge
rId'], axis=0)
y_train = train_data['Survived']
```

In [13]:

```
sc.fit(X_train)
X_train_std = sc.fit_transform(X_train)
lr.fit(X_train_std, y_train)
predictions = lr.predict(X_test)
```

In [14]:

```
final_out = pd.DataFrame(predictions, columns=['Survived'])
final_out['PassengerId'] =test['PassengerId']
final_out = final_out.set_index('PassengerId')
final_out.to_csv('F:/Annie/CornellMS/Semester 4/Machine Learning/HW1/Titanic/Titanic_Su
bmission.csv')
```

1.

Let $E(X) = \mu$ and $E(Y) = v$

$Var(X-Y) = E[(X-Y)^2] - E[X-Y]^2$

$= E[X^2-2XY+Y^2] - (\mu-v)^2$

$= E[X^2]-2E[XY]+E[Y^2] - (\mu^2-2\mu v+v^2)$

$= (E[X^2]-\mu^2) + (E[Y^2]-v^2) -2(E[XY]-\mu v)$

$= Var(X) + Var(Y) - 2Cov(X,Y)$

2.

Let the probability of defective widgets be P(A) and that of normal ones be P(B).

Samely let actual testing positive probability be P(C) and negative probability be P(D).

According to given conditions:

$P(C|A) = 0.95 = P(D|B)$

So $P(D|A) = 1-P(C|A) = 0.05 = P(C|B)$

While $P(A) = 10^{-5} = 0.00001$

Obviously $P(B) = 1 - P(A)$

So $P(C) = P(C|A)P(A) + P(C|B)P(B)$

(a)

As the result,

$P(A|C)=P(C|A)P(A)/P(C)$

$= P(C|A)P(A)/[P(C|A)P(A) + P(C|B)P(B)]$

$=0.95*10^{-5}/[0.95*10^{-5} + 0.05*(1-10^{-5})]$

$=1.8997 * 10^{-4}$

That is the chances of actually defective widgets when the test shows defective.

(b)

Let the annually number of producing defective widgets be N_A and that of normal ones be N_B

Samely thrown defective widgets be M_A and that of normal ones be M_B

Since the factory make $N = 10^7$ widgets a year

$N_A = NP(A) = 100$ and $N_B = N-N_A = 10^7-100$

$M_A = N_AP(C|A) = 95$

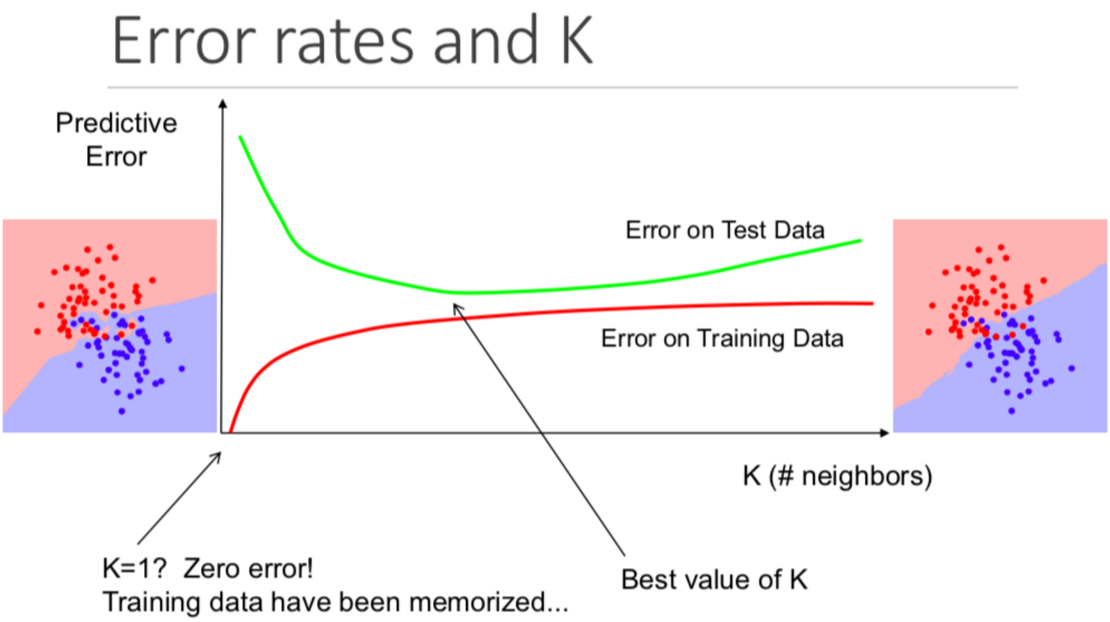
$M_B = N_BP(C|B) = 499995$

So, 499995 good widgets are thrown away and 5 bad widgets are still shipped to customers each year.

3.

(a) In training data, when k = 1, prediction error will always be 0, while as k increases to n, the 0-1 prediction error will also increase.

(b) As the value of k increases, 0-1 prediction error will first decrease because of decrease of variance, and then increase, since more neighbor lead to larger variance again.



(c) I suggest to use a 10 folds cross validation, since as the number of folds increase, the mesuremant of accuracy will increase, while more folds will lead to huge computational works.

(d) Since kNN is a mainly based on neighboring data, I suggest adding higher weight to nearer neighbor and lower weight to farther neighbor in order to avoid the caveat.

(e) First, when input dimension raises, distance between data becomes extremely far, which means density of data and weight of data decreases fast;

Second, the requirement of data will increase for high dimension input, which will take much longer time for kNN which need to traverse all of the data for every single data.

Reference:

[1]: https://glowingpython.blogspot.com/2012/04/k-nearest-neighbour-classifier.htmlhttps://glowingpython.blogspot.com/2012/04/k-nearest-neighbour-classifier.html

[2]: https://stackoverflow.com/questions/52366421/how-to-do-n-d-distance-and-nearest-neighbor-calculations-on-numpy-arrays

[3]: https://stats.stackexchange.com/questions/49692/why-do-researchers-use-10-fold-cross-validation-instead-of-testing-on-a-validati

[4]: https://medium.com/30-days-of-machine-learning/day-3-k-nearest-neighbors-and-bias-variance-tradeoff-75f84d515bdb