CS5785 Applied Machine Learning

Homework #1 (due 09/13/17)

Name: Weisi Zhang, Dexing Xu
Email: wz337@cornell.edu, dx49@cornell.edu

1 Digits Recognizer

1.(a) Join the Digit Recognizer competition on Kaggle. Download the training and test data. The competition page describes how these files are formatted.

We imported the libraries needed and loaded the data first.

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from collections import Counter

data = np.loadtxt('train.csv', dtype=np.int32, delimiter=',', skiprows=1)
dimension = int(data[1,1:].shape[0] ** 0.5)
```

1.(b) Write a function to display an MNIST digit. Display one of each digit (demonstrated in Fig.1).

```
def displayDigitTwo(data, dimension):
    #cut down sample to improve runtime
    sample_data = data[np.random.choice(data.shape[0], 1000, replace=False), :]
    plt.figure(figsize = (15, 7))
    plt.suptitle('MNIST digit', fontsize=18)
    row_index = 0
    label = 0
    while True:
        current_label = sample_data[row_index, 0]
        if current_label == label:
            plt.subplot(2, 5, label + 1)
            plt.title('Digit {label}'.format(label=label))
            plt.imshow(sample_data[row_index, 1:].reshape((dimension, dimension)), cmap='binary')
            label += 1
        if label > 9:
            break
        row_index += 1
displayDigitTwo(data, dimension)
```

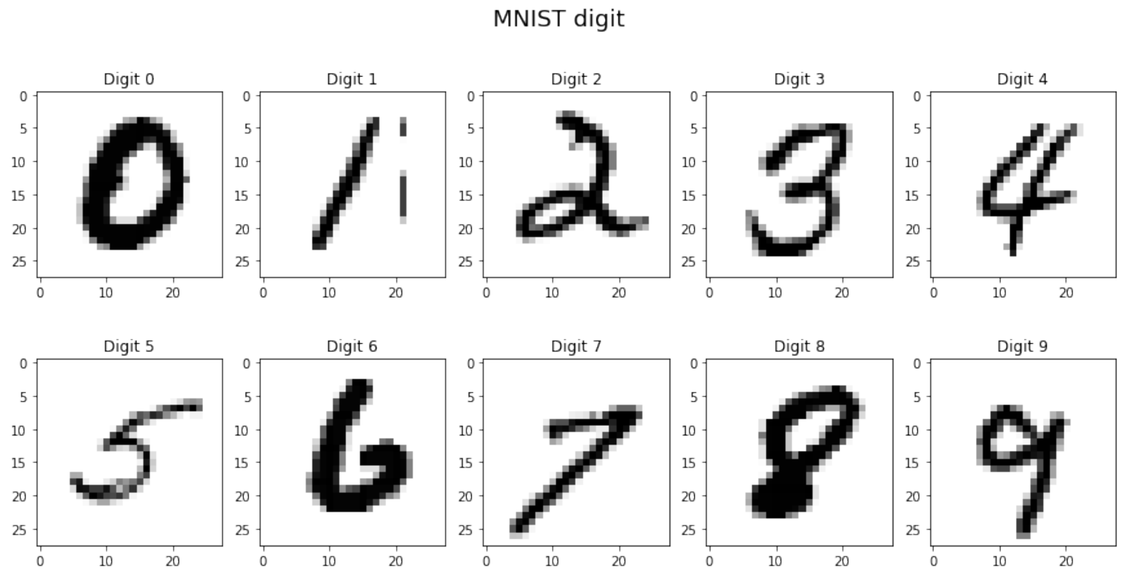


Figure 1: Display 9 random MNIST digit

1.(c) Examine the prior probability of the classes in the training data. Is it uniform across the digits? Display a normalized histogram of digit counts. Is it even?

As indicated by the histogram below, the distribution of the prior probability of each digit are mostly uniform, with one being the highest occurred number (11.2%) and five being the lowest occurred number (9.0%).

```
def displayHistogram(data):
    prior = []
    sample_size = float(data.shape[0])
    for i in range(10):
        prior.append((data[data[:, 0] == i, :]).shape[0] / sample_size)
    plt.title("Prior Probability vs. Digit")
    plt.xlabel("Digit")
    plt.ylabel("Prior Probability")
    plt.xticks(range(10))
    plt.bar(range(10), prior, 0.75, color = "blue")
    print "prior distribution max: {0}%".format(max(prior)*100)"
    print "prior distribution min: {0}%".format(min(prior)*100)"
displayHistogram(data)
```

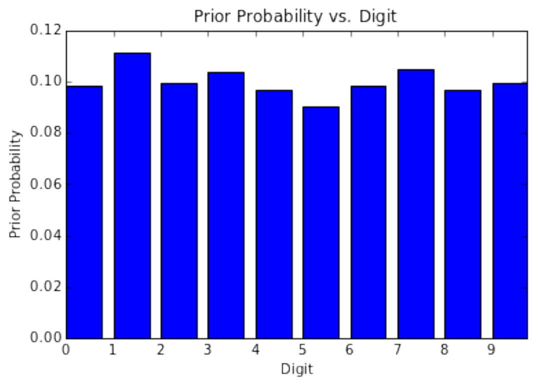


Figure 2: Normalized Histograms of the Prior Probability of Labels in the Training Set

1.(d) Pick one example of each digit from your training data. Then, for each sample digit, compute and show the best match (nearest neighbor) between your chosen sample and the rest of the training data. Use L2 distance between the two images' pixel values as the metric. This probably won't be perfect, so add an asterisk next to the erroneous examples (if any).

Nearest neighbor is a special case of kNN when k equals to 1. Fig.3 displays a set of comparison for each of the sample digit and its nearest neighbor. As shown, we have mistakenly classified digit 9 in this sample. However, through observations from multiple trials, the overall performance of 1NN is actually quite good. This is reasonable since overlaps between the handwritten digits are pretty small.

```
def kNN(training_data, test_data, k):
    neighbors = np.sqrt(np.sum((training_data - test_data)**2, axis = 1))
    best_match = neighbors.argsort()[:k + 1][1:] # best K matches
    label = data[best_match, 0]
    cnt = Counter()
    top_vote = Counter(label).most_common(1)[0][0]
    return top_vote, best_match

sample_data = data[np.random.choice(data.shape[0], 1000, replace=False), :]
plt.figure(figsize = (15, 15))
plt.suptitle('One Nearest Neighbor MNIST digit', fontsize=18)
row_index = 0
label = 0
while True:
    current_label = sample_data[row_index, 0]
    if current_label == label:
        top_vote, best_match = kNN(data[:,1:], sample_data[row_index, 1:], 1)
        plt.subplot(4, 5, current_label+1+5*(current_label//5))
        plt.title('Digit {label}'.format(label=label))
        plt.imshow(sample_data[row_index, 1:].reshape((dimension, dimension)), cmap='binary')
```

```
            plt.subplot(4, 5, current_label+1+5*(current_label//5+1))
            if current_label == top_vote:
                plt.title('1NN of Digit {label}'.format(label=label))
            else:
                plt.title('1NN of Digit {label}, True label {top_vote}*'.format(
                label=label, top_vote=top_vote))
            plt.imshow(data[best_match[0], 1:].reshape((dimension, dimension)), cmap='binary')

            label += 1
        if label > 9:
            break
        row_index += 1
```

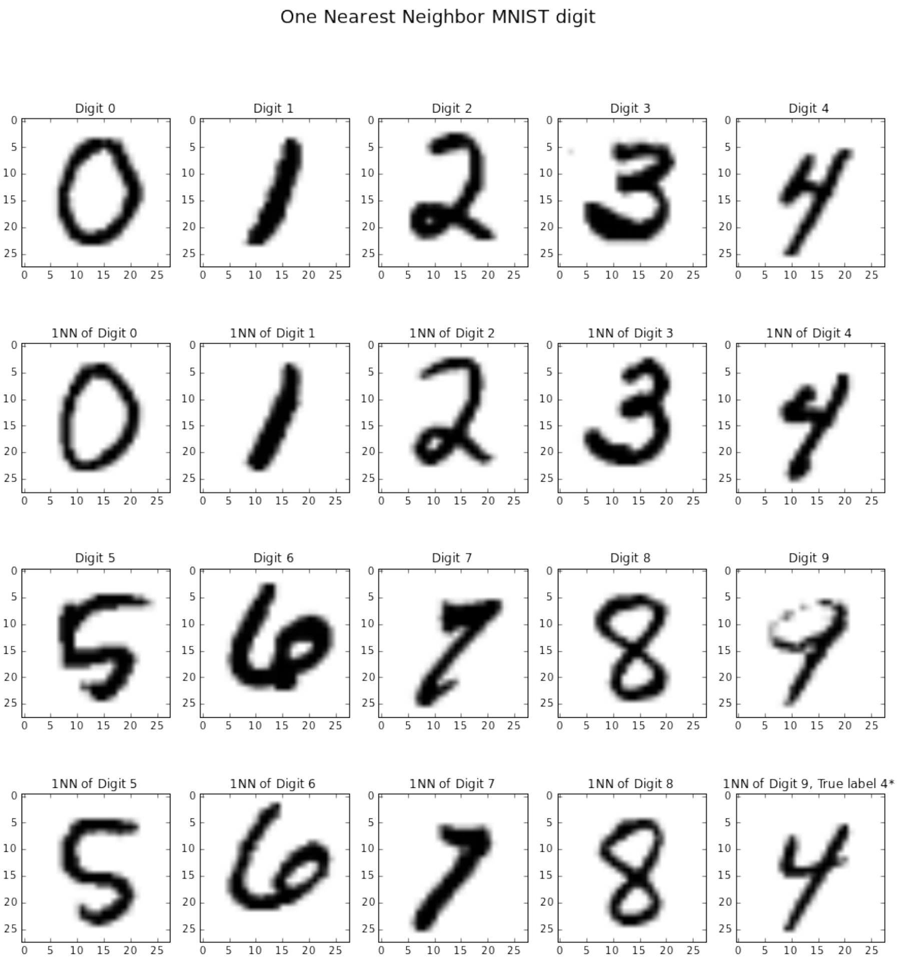


Figure 3: One Nearest Neighbor for MNIST digit

1.(e) Consider the case of binary comparison between the digits 0 and 1. Ignoring all the other digits, compute the pairwise distances for all genuine matches and all impostor matches, again using the L2 norm. Plot histograms of the genuine and impostor distances on the same set of axes.

As shown in Figure 4, when L2 distance reaches 2000, we start to see some imposter classifications. However, the distributions of genuine and imposter matches overlap little overall. That is why we can classify digits through kNN.

```
# data only contains 1
ones = data[data[:,0] == 1][:,1:]
# data only contains 0
zeros = data[data[:,0] == 0][:,1:]
# genuine between 1 and 1
def l2_matrix(x, y):
    m = x.shape[0]
    n = y.shape[0]
    x2 = np.sum(x**2, axis=1).reshape((m, 1))
    y2 = np.sum(y**2, axis=1).reshape((1, n))
    xy = x.dot(y.T)
    dists = np.sqrt(x2 + y2 - 2*xy)
    return dists

ones_genuine = l2_matrix(ones, ones)
zeros_genuine = l2_matrix(zeros, zeros)
impostor_matches = l2_matrix(zeros, ones)

ones_upper = ones_genuine[np.triu_indices(ones_genuine.shape[0],1)]
zeros_upper = zeros_genuine[np.triu_indices(zeros_genuine.shape[0],1)]
genuine = np.concatenate((ones_upper, zeros_upper), axis=0)
impostor = impostor_matches.flatten()

plt.figure(figsize = (12, 6))
_n, _bins, _ = plt.hist(genuine, label='genuine',alpha=0.3, range=(0, 4500), bins=30)
_n2, _bins2, _ = plt.hist(impostor, label='impostor',alpha=0.3, range=(0, 4500), bins=30)
plt.xlabel('l2 distance')
plt.ylabel('counts')
plt.title('histograms of the genuine and impostor distances')
plt.legend()
```

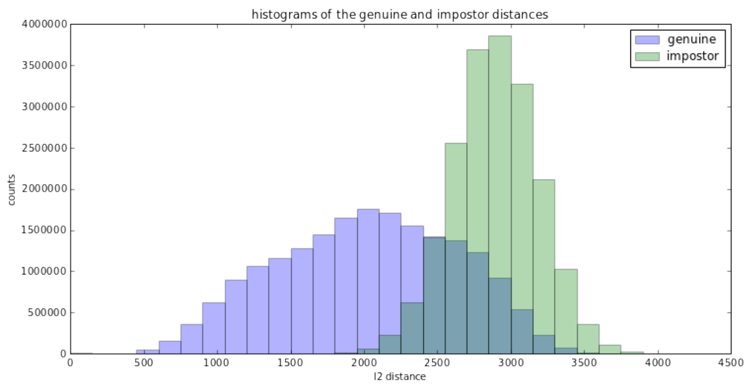


Figure 4: Distribution of genuine distances and impostor distances

1.(f) Generate an ROC curve from the above sets of distances. What is the equal error rate? What is the error rate of a classifier that simply guesses randomly?

As shown in Fig.5, the equal error rate is the intersection between the anti-diagonal and ROC curve, it is approximately 0.18. The random guess accuracy is 0.5.

```python
len_gen = float(len(genuine))
len_imp = float(len(impostor))
tpr = []
fpr = []
for thresh in xrange(200, 4200, 5):
    tpr.append(np.count_nonzero(genuine < thresh) / len_gen)
    fpr.append(np.count_nonzero(impostor < thresh) / len_imp)

fig = plt.figure(figsize = (12, 8))
ax = fig.add_subplot(111)
ax.plot(fpr, tpr, label='roc curve')
ax.set_ylabel('TPR')
ax.set_xlabel('FPR')
ax.set_title('ROC Curve')
ax.plot([0, 1], [0, 1], color='green', linestyle='--', label='anti diagonal')
ax.plot([0, 1], [1, 0], color='red', linestyle='--', label='roc random')

major_ticks = np.arange(0, 1, 0.1)
minor_ticks = np.arange(0, 1, 0.05)

ax.set_xticks(major_ticks)
ax.set_xticks(minor_ticks, minor=True)
ax.set_yticks(major_ticks)
ax.set_yticks(minor_ticks, minor=True)
ax.grid(which='both')
ax.grid(which='minor', alpha=0.2)
ax.grid(which='major', alpha=0.5)
ax.legend()
```

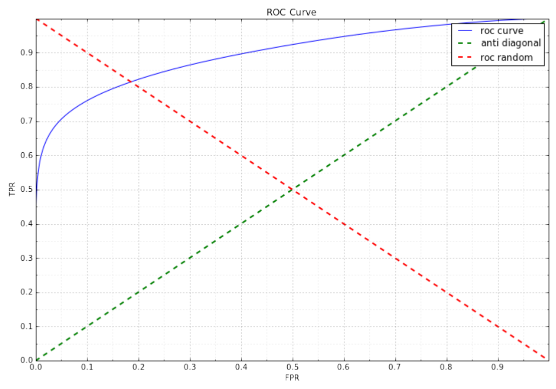


Figure 5: ROC Curve

1.(g) Implement a K-NN classifier. (You cannot use external libraries for this question; it should be your own implementation.)

Implementation: In order to make the most use of RAM space and save computation time, we expanded the terms and computed multiple test data at once using matrix multiplication. We first created a matrix of l2 distance, picked the nearest k item, and then returned the top voted label.

This works because our ”test data” (pseudo test data) and training data are not large, and we can fit the matrix multiplication under my RAM. The bottleneck of this particular implementation is that when the training data and test data become large, the matrix multiplication will not fit under the RAM. Then, we will have to modify the implementation to do batch processing.

```
def l2_matrix(x, y):
    m, n= x.shape[0], y.shape[0]
    x2 = np.sum(x**2, axis=1).reshape((m, 1))
    y2 = np.sum(y**2, axis=1).reshape((1, n))
    dist_matrix = np.sqrt(x2 + y2 - 2*(x.dot(y.T)))
    return dist_matrix

def get_majority_vote(a):
    majority_vote = Counter(a).most_common(1)[0][0]
    return majority_vote

def kNN(training_data, test_data, k, label_data):
    # step one: do matrix multiplication get a matrix of l2 distance
    dist_matrix = l2_matrix(test_data, training_data)

    # step two: for each comparison keep the top k and its index
    least_k_indices = dist_matrix.argsort(axis=1)[:,:k]
    labels = label_data[least_k_indices.flatten()].reshape((test_data.shape[0], k))

    #step three: get majority votes
    classification_result = np.apply_along_axis(get_majority_vote, 1, labels)

    return classification_result
```

1.(h) Using the training data for all digits, perform 3 fold cross-validation on your K-NN classifier and report your average accuracy.

See below for the average accuracy for different k.

```
def l2_matrix(x, y):
    m, n= x.shape[0], y.shape[0]
    x2 = np.sum(x**2, axis=1).reshape((m, 1))
    y2 = np.sum(y**2, axis=1).reshape((1, n))
    dist_matrix = np.sqrt(x2 + y2 - 2*(x.dot(y.T)))
    return dist_matrix

def get_majority_vote(a):
    majority_vote = Counter(a).most_common(1)[0][0]
    return majority_vote

def kNN(training_data, test_data, k, label_data):
    # step one: do matrix multiplication get a matrix of l2 distance
    dist_matrix = l2_matrix(test_data, training_data)

    # step two: for each comparison keep the top k and its index
    least_k_indices = dist_matrix.argsort(axis=1)[:,:k]
    labels = label_data[least_k_indices.flatten()].reshape((test_data.shape[0], k))

    #step three: get majority votes
    classification_result = np.apply_along_axis(get_majority_vote, 1, labels)

    return classification_result
```

3 Cross Validation Average Accuracy for K = 1, 3, 5...19

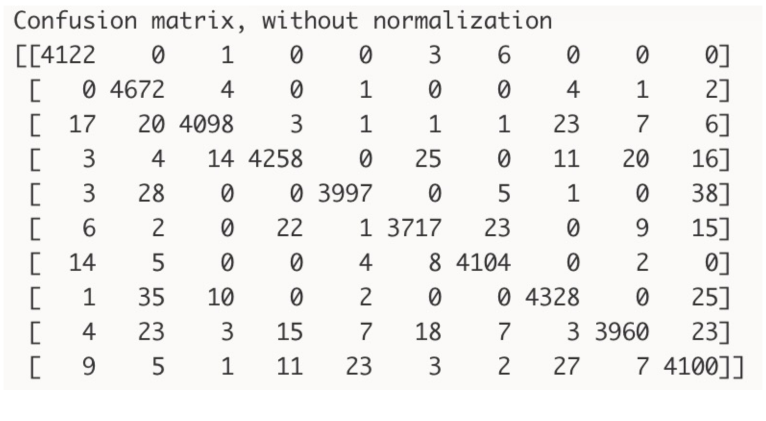
K	Average Accuracy	K	Average Accuracy
1	0.964380952381	11	0.960285714286
3	0.966166666667	13	0.958047619048
5	0.965619047619	15	0.956261904762
7	0.96380952381	17	0.954666666667
9	0.962285714286	19	0.952785714286

We need to pick the optimal k for our training data. As shown in the table above, the average accuracy reaches the highest when k equals to 3. This makes sense since the prediction is purely based on its neighborhood, K-NN performs better when K is smaller.

1.(i) Generate a confusion matrix (of size 10x10) from your results. Which digits are particularly tricky to classify?

As shown in the normalized confusion matrix heat map in Fig.8, we can see that 8 is particularly tricky to identify.

```
confusion_matrix = []
for digit in range(0, 10):
    test_data = data[[data[:,0] == digit]]
    i_th_results = kNN(data[:,1:], test_data[:,1:], 3, data[:,0])
    confusion_matrix.append(dict(Counter(i_th_results)))

matrix = []
for classification_result in confusion_matrix:
    row = []
    for digit in range(0, 10):
        if digit in classification_result:
            row.append(classification_result[digit])
        else:
            row.append(0)
    matrix.append(row)
print matrix
print np.array(matrix)

# Plot non-normalized confusion matrix
import itertools
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting ‘normalize=True‘.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

plt.figure()
plot_confusion_matrix(np.array(matrix), classes=range(0,10),
                      title='Confusion matrix, without normalization')

# # Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(np.array(matrix), classes=range(0,10), normalize=True,
                       title='Normalized confusion matrix')

plt.show()
```



```
Confusion matrix, without normalization
[[4122    0    1    0    0    3    6    0    0    0]
 [   0 4672    4    0    1    0    0    4    1    2]
 [  17   20 4098    3    1    1    1   23    7    6]
 [   3    4   14 4258    0   25    0   11   20   16]
 [   3   28    0    0 3997    0    5    1    0   38]
 [   6    2    0   22    1 3717   23    0    9   15]
 [  14    5    0    0    4    8 4104    0    2    0]
 [   1   35   10    0    2    0    0 4328    0   25]
 [   4   23    3   15    7   18    7    3 3960   23]
 [   9    5    1   11   23    3    2   27    7 4100]]
```

Figure 6: Confusion Matrix (without normalization)

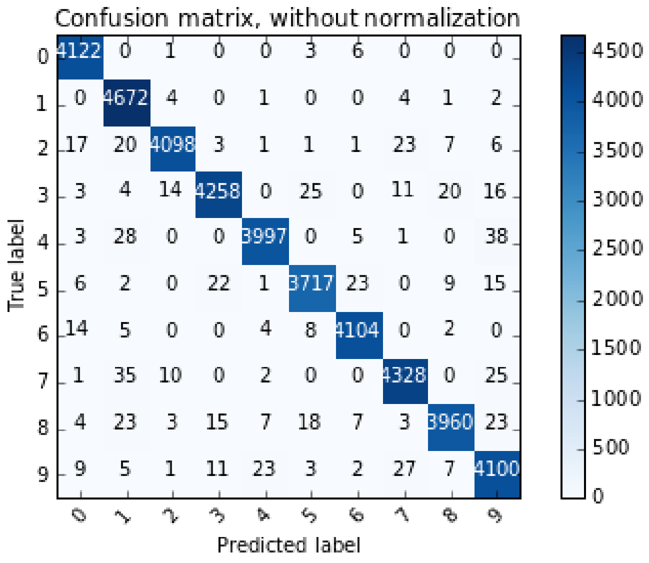


Figure 7: Confusion Matrix (without normalization)

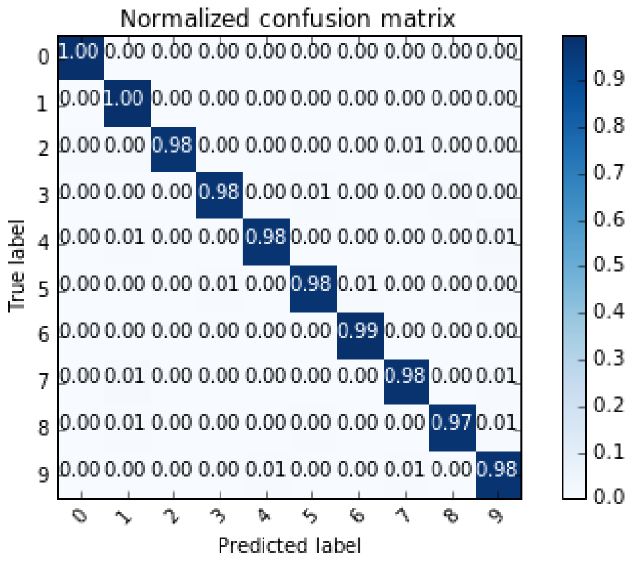


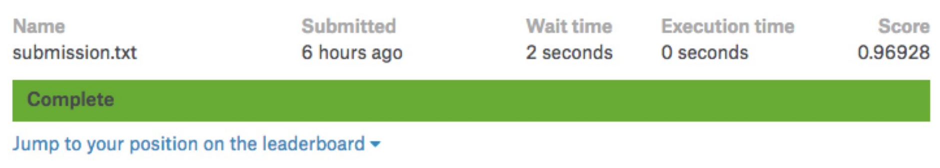
Figure 8: Confusion Matrix (with normalization)

1.(j) Train your classifier with all of the training data, and test your classifier with the test data. Submit your results to Kaggle.

After determining our k (3 in the case), we ran the model on test set and submitted the result to kaggle.

As shown in Fig.9, the performance of KNN is not that accurate and the runtime is rather slow. This makes sense since KNN is a rather naive classifier.

```
classification_result = kNN(data[:,1:], test_data, 3, data[:,0])

with open("submission.txt", "wb") as f:
    f.write("ImageId,Label\n")
    label_id = 0
    for item in classification_result:
        label_id += 1
        f.write("{0},{1}".format(str(label_id),str(item)))
        f.write("\n")
```



Name	Submitted	Wait time	Execution time	Score
submission.txt	6 hours ago	2 seconds	0 seconds	0.96928

Complete

Jump to your position on the leaderboard

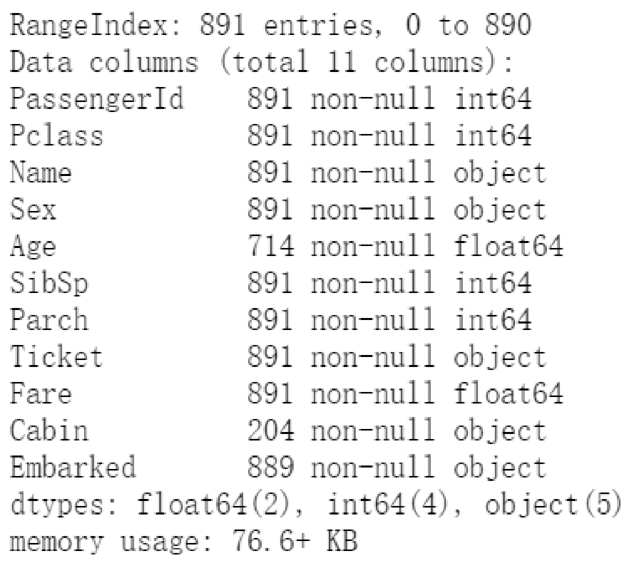
Figure 9: Final Submission to Kaggle

2 The Titanic Disaster

First, we load the data uses following function

```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import sys
import seaborn as sb
def get_data():
    train_data = pd.read_csv("train.csv")
    test_data = pd.read_csv("test.csv")
    return train_data, test_data
train_data,test_data = get_data()
train_features = train_data.drop(['Survived'],axis=1)
train_features.info()
```

Then we can see the train features as following



```
RangeIndex: 891 entries, 0 to 890
Data columns (total 11 columns):
PassengerId    891 non-null int64
Pclass         891 non-null int64
Name           891 non-null object
Sex            891 non-null object
Age            714 non-null float64
SibSp          891 non-null int64
Parch          891 non-null int64
Ticket         891 non-null object
Fare           891 non-null float64
Cabin          204 non-null object
Embarked       889 non-null object
dtypes: float64(2), int64(4), object(5)
memory usage: 76.6+ KB
```

Figure 10: train feature status 1

Since the “Name”,“PassengerId”is irrelevant to the survive situation of those passengers, therefore we drop these features out. Also, the “Cabin” and “Embarked” features are somewhat repeat the information which “Pclass”indicates, therefore we drop them out.

```
sb.heatmap(train_features.corr())
```

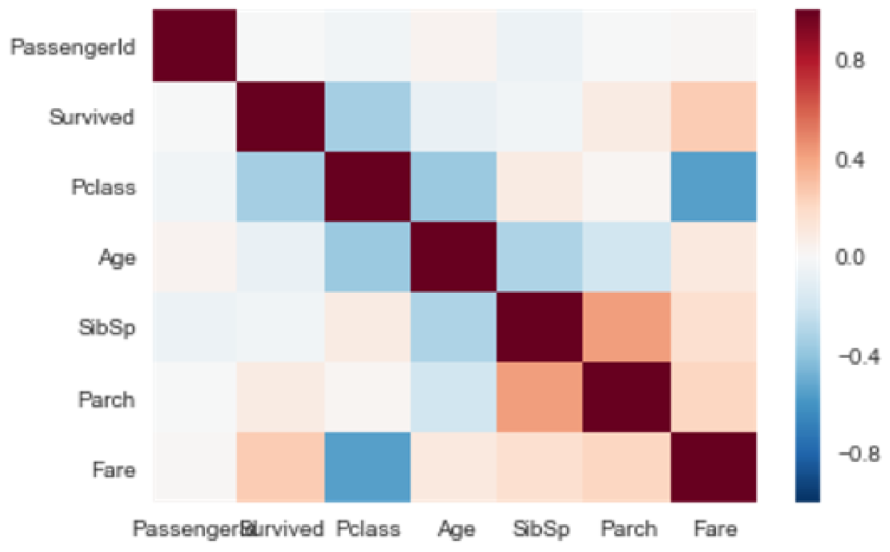
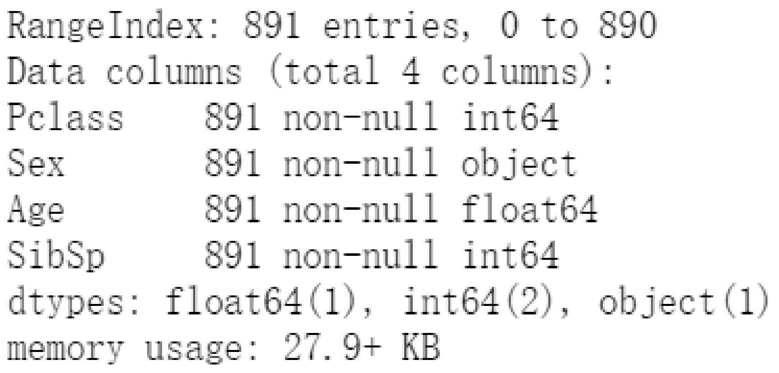


Figure 11: Independence Checking

We can see that "Parch" and "Fare" are self relevant therefore we also need to drop these features. Also, there are "Age" is null, we need to fill in the data to do the prediction. We choose the medium of the "Age" to fill in those null data showing as following:

```
#data preprocessing
train_mid = 28
new_age = train_features['Age'].fillna(train_mid)
train_features['Age'] = new_age
test_mid = 27
test_features['Age'] = test_mid
train_features.info()
test_features.describe()
test_features.info()
```

Finally, we obtain the train features as following



```
RangeIndex: 891 entries, 0 to 890
Data columns (total 4 columns):
Pclass    891 non-null int64
Sex       891 non-null object
Age       891 non-null float64
SibSp     891 non-null int64
dtypes: float64(1), int64(2), object(1)
memory usage: 27.9+ KB
```

Figure 12: train features status 2

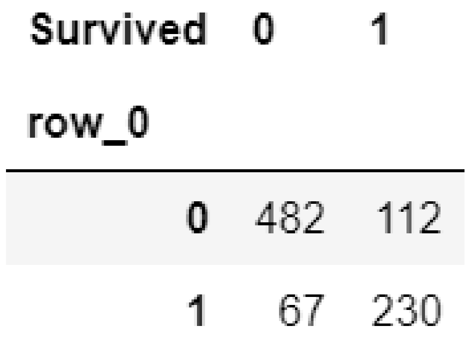
We can train the model now:

```
from sklearn import linear_model
train_features = train_features.replace({'Sex':{'female':0,'male':1}})
log_model = linear_model.LogisticRegression()
X = train_features
Y = train_data.loc[:,"Survived"]
log_model.fit(X,Y)
preds = log_model.predict(X = train_features)
pd.crosstab(preds,train_data["Survived"])
```

And we get the result as follow:

Then we have to preprocess the test features:

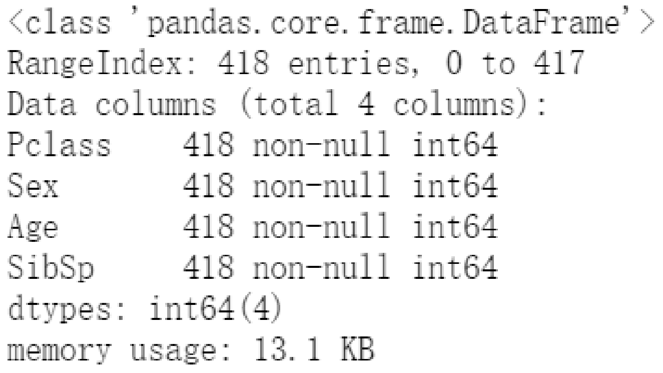
```
test_features.drop(['Fare','Parch'],axis=1)
test_features.info()
test_features = test_features.replace({'Sex':{'female':0,'male':1}})
test_features.info()
```



Survived	0	1
row_0		
0	482	112
1	67	230

Figure 13: training result

We get the test features as following:



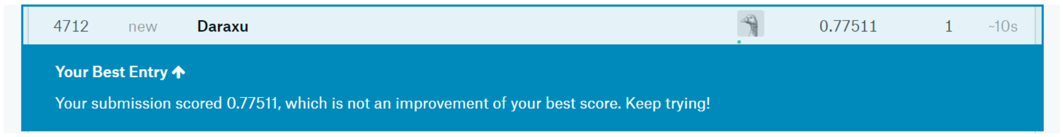
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 418 entries, 0 to 417
Data columns (total 4 columns):
Pclass    418 non-null int64
Sex       418 non-null int64
Age       418 non-null int64
SibSp     418 non-null int64
dtypes: int64(4)
memory usage: 13.1 KB
```

Figure 14: test features

Make test set predictions and create a submission for Kaggle, save the submission to CSV

```
test_preds = log_model.predict(X=test_features)
submission = pd.DataFrame({"PassengerId":test_data["PassengerId"], "Survived":test_preds})
submission.to_csv("gender_submission.csv", index=False)
```

Then we submit the result to Kaggle, and get the score 0.77511.


4712 new Daraxu 0.77511 1 ~10s

Your Best Entry

Your submission scored 0.77511, which is not an improvement of your best score. Keep trying!

Figure 15: titanic submission result

3 Written Exercise

1.
Proof:

$E(X-Y) = E(X) - E(Y)$

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

$Cov(X,Y) = E[(X-E(X))(Y-E(Y))]$

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

2.

a.

Let $A = \{widget\,is\,defective\}$, $B = \{widget\,is\,tested\,positive\}$, then $A^C = \{widget\,is\,not\,defective\}$, $B^C = \{widget\ is\ tested\ negative\}$

We know that $P(B|A) = 0.95$, $P(B^C|A^C) = 0.95$, and $P(A) = \frac{1}{100000}$

$$P(A|B) = \frac{P(A,B)}{P(B)} = \frac{P(B|A)P(A)}{P(B|A)P(A) + P(B|A^C)P(A^C)} = \frac{0.95/100000}{0.95/100000 + 0.05 * 99/100000} = 0.00019$$

Thus, the chances that it's actually defective given the positive result is 0.00019.

b.

$P(throw\,away\,good\,widget) = P(B, A^C) = P(B|A^C)P(A^C) = (1-P(B^C|A^C))P(A^C) = 0.05 * 99999/100000 = 0.0499995$

The number of good widgets thrown away per year is 10,000,000*0.499995=499995.

$P(bad\ widgets\ shipped\ to\ customers) = P(B^C, A) = P(A) - P(B, A) = P(A) - P(B|A)P(A) = 1/100000 - 0.95/100000 = 0.0000005$

The number of bad widgets shipped to customers per year is 10,000,000*0.0000005=5.

3.

a. When K = N or it is large, samples which do not locates in the overlapping area will be classified with correct label, and samples just lying on the border of the overlapping area they would correctly classified. However, those samples which still in the overlapping area would be wrongly labeled. On the contrary, if K = 1, samples locate outside the overlapping area would be wrongly classified.

b. When using the held-out to train the model, the error rate may decreases first and then increases as the K increases. When the number of neighbors is small, result of prediction may have big difference; when the number of neighbors becomes too large, it will count to the majority of the training set.

c. More K-fold means longer processing time and more storage space, and each test set would be small so that the variance would be large. While less k-fold means less processing time but more biased in cross-validation. The choice of K should depend on the size of the data and we usually make K = 5.

d. We set weight based on the Euclidean distance, closer and higher weight, also they will be valued more when doing the votes.

e.Large amount of calculation; Need a lot of memory;