

CS 5785 Applied Machine learning

Homework 1

Team Member: Queenie Liu, Xueqi Wei

Programming Exercise

1. Digit Recognizer

```
[ ]: #packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import seaborn as sns
from sklearn import neighbors
from statistics import mode
from sklearn import model_selection
from sklearn.metrics.pairwise import euclidean_distances
from sklearn import metrics
from sklearn.metrics import confusion_matrix
import matplotlib.cm as cm
import collections
from collections import Counter
```

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

```
[ ]: #load dataset
dtrain = pd.read_csv('digit-train.csv', header = 0, dtype = np.int)
X_dtest = pd.read_csv('digit-test.csv', header = 0, dtype = np.int)
#dtrain.head()
X_dtrain = dtrain.drop('label', axis = 1)
Y_dtrain = dtrain['label']

#get the height and width
height = width = int((dtrain.shape[1] - 1) ** 0.5)
print(height, width)
```

- (b) Write a function to display an MNIST digit. Display one of each digit.

```
[ ]: #display the index[2].
dis_mnist_digit = []
for i in range(0,10):
    dis_mnist_digit.append(dtrain[dtrain.label == i].index[2])
```

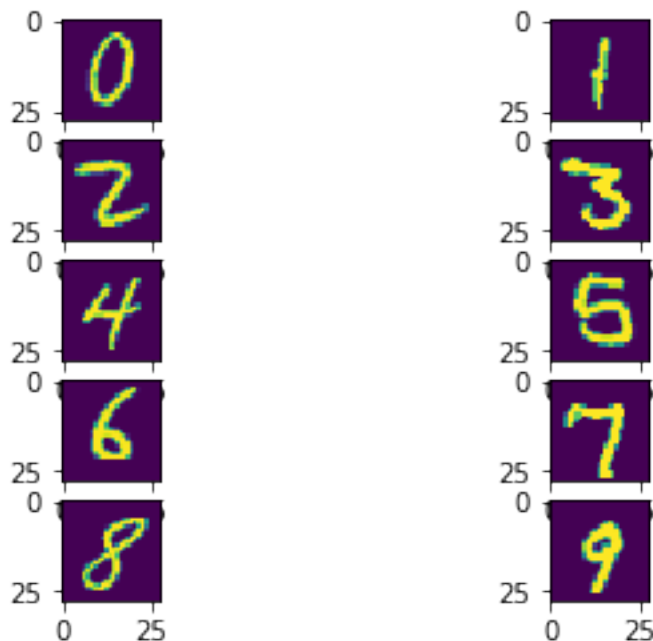
```
dis_mnist_digit = {i:dis_mnist_digit[i] for i in range(0,10)}
print(dis_mnist_digit)
```

```
[4]: #make a subplot.
fig, ax = plt.subplots(nrows = 5, ncols = 2)           #fit the 10 images into 5*2
        ↳subplots to make it easier to observe.

def generate_image(index):
    image_matrix = X_dtrain.loc[index].values.reshape((height,width))
        ↳#load the height&width in previous part.
    return image_matrix

#plot one of each digit.
ax[0, 0].imshow(generate_image(dis_mnist_digit[0]))
ax[0, 1].imshow(generate_image(dis_mnist_digit[1]))
ax[1, 0].imshow(generate_image(dis_mnist_digit[2]))
ax[1, 1].imshow(generate_image(dis_mnist_digit[3]))
ax[2, 0].imshow(generate_image(dis_mnist_digit[4]))
ax[2, 1].imshow(generate_image(dis_mnist_digit[5]))
ax[3, 0].imshow(generate_image(dis_mnist_digit[6]))
ax[3, 1].imshow(generate_image(dis_mnist_digit[7]))
ax[4, 0].imshow(generate_image(dis_mnist_digit[8]))
ax[4, 1].imshow(generate_image(dis_mnist_digit[9]))

plt.show()
```



- (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?

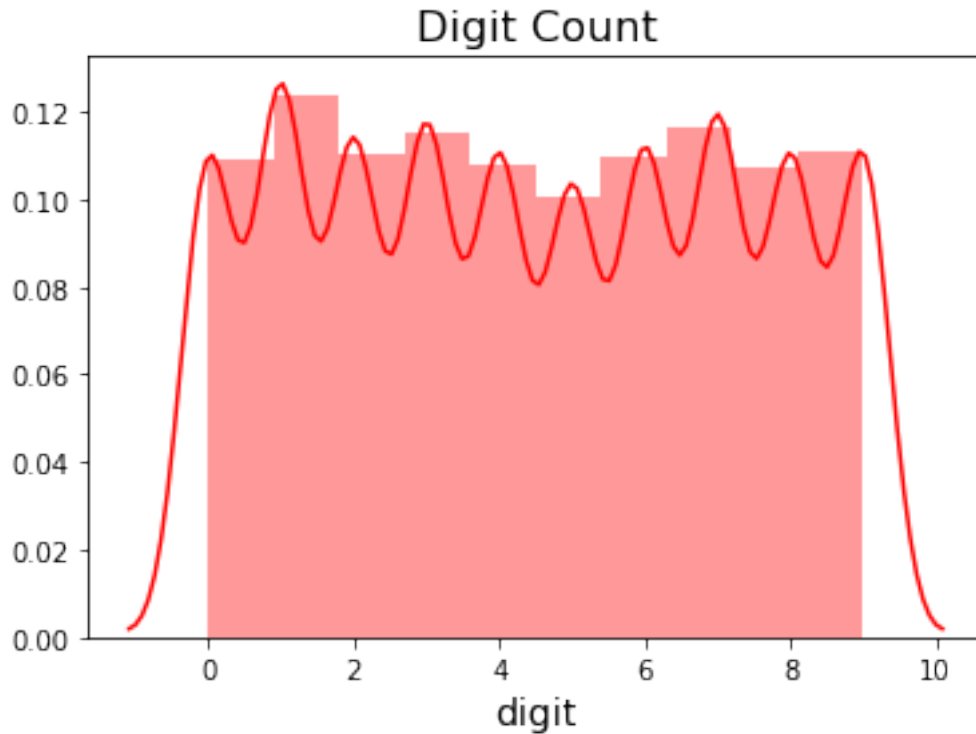
```
[5]: #examine the prior probability.
dtrain_freq = pd.DataFrame(Y_dtrain.value_counts()).reset_index()
→                                     #count the classes.
dtrain_freq.sort_values(by = ['label'])
→                                     #sort the digits by no. of counts

#Index: ["Digit"]
#Label: ["Count"]
```

```
[5]:  digit  count
0      1   4684
1      7   4401
2      3   4351
3      9   4188
4      2   4177
5      6   4137
6      0   4132
7      4   4072
8      8   4063
9      5   3795
```

```
[6]: #display a normalized histogram of digit counts.
sns.distplot(Y_dtrain, bins = 10, color = 'red')
plt.title('Digit Count', fontsize = 16)
plt.xlabel('digit', fontsize = 14)
```

```
[6]: Text(0.5, 0, 'digit')
```



It seems like a uniform distribution across the digits and almost even.

- (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).

```
[7]: #from above, we picked the [2] digit
def generate_nearest_neighbor(index):
    #compute the nearest neighbor
    #of our example
    neighbors = np.argsort(((np.array(X_dtrain) - np.array(X_dtrain.
    iloc[index]))**2).sum(-1))
    nearest_neighbor = neighbors[1]
    if Y_dtrain[index] == Y_dtrain[best_match]:
        right = True
    return nearest_neighbor

nearest_neighbor = []
#show the best match
#for each of the [2] digit
for i in range(0,10):
    nearest_neighbor.append(generate_nearest_neighbor(dis_mnist_digit[i]))
```

```
nearest_neighbor
→ #display the [n] of
→ the best match
```

[7]: [25196, 29923, 4704, 9240, 7204, 10674, 34338, 10446, 14340, 8935]

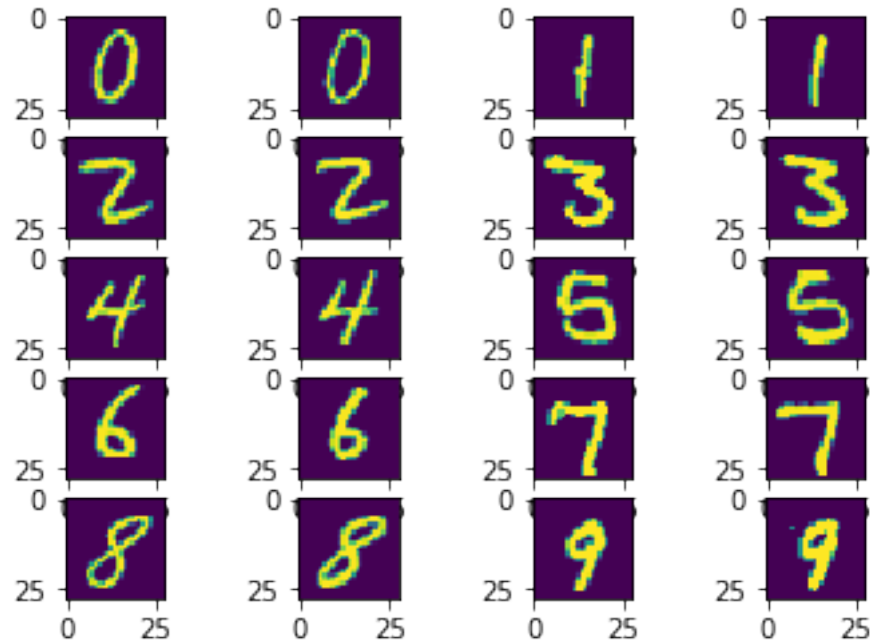
```
[8]: #import pandas as pd
#import matplotlib.pyplot as plt
#plot the orginal digits and their best match into a 5*4 subplots

fig, ax = plt.subplots(5,4)

def ex_img(index):
→ #plot each pair of digits side
→ by side
    image_matrix = X_dtrain.loc[index].values.reshape((height,width))
    return image_matrix

ax[0, 0].imshow(ex_img(dis_mnist_digit[0]))
ax[0, 1].imshow(ex_img(25196))
ax[0, 2].imshow(ex_img(dis_mnist_digit[1]))
ax[0, 3].imshow(ex_img(29923))
ax[1, 0].imshow(ex_img(dis_mnist_digit[2]))
ax[1, 1].imshow(ex_img(4704))
ax[1, 2].imshow(ex_img(dis_mnist_digit[3]))
ax[1, 3].imshow(ex_img(9240))
ax[2, 0].imshow(ex_img(dis_mnist_digit[4]))
ax[2, 1].imshow(ex_img(7204))
ax[2, 2].imshow(ex_img(dis_mnist_digit[5]))
ax[2, 3].imshow(ex_img(10674))
ax[3, 0].imshow(ex_img(dis_mnist_digit[6]))
ax[3, 1].imshow(ex_img(34338))
ax[3, 2].imshow(ex_img(dis_mnist_digit[7]))
ax[3, 3].imshow(ex_img(10446))
ax[4, 0].imshow(ex_img(dis_mnist_digit[8]))
ax[4, 1].imshow(ex_img(14340))
ax[4, 2].imshow(ex_img(dis_mnist_digit[9]))
ax[4, 3].imshow(ex_img(8935))

plt.show()
```



- (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.

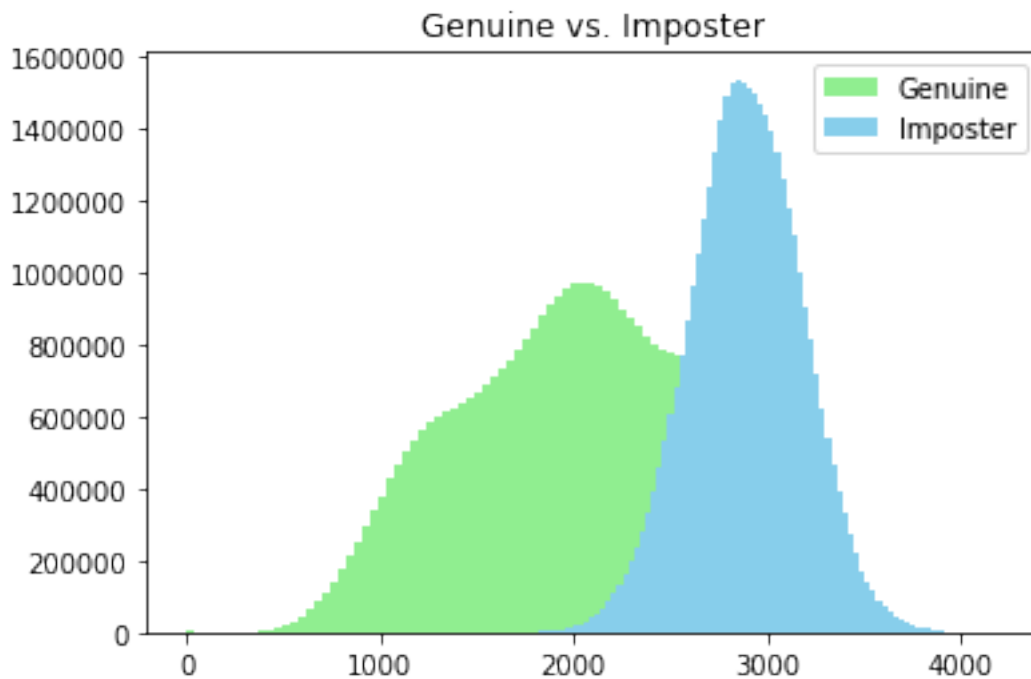
```
[9]: #import numpy as np
      #import matplotlib.pyplot as plt
      #from sklearn.metrics.pairwise import euclidean_distances

      binary_y = Y_dtrain[Y_dtrain.isin([0,1])].sort_values()
      binary_x = X_dtrain.loc[list(binary_y.index)]

      X = np.array(binary_x)
      Y = np.array(binary_y).reshape((4684+4132),1)
      → #from above questions we can observe that [0]4132, [1]4684.
      y_score = euclidean_distances(X,X)
      → #find the pairwise distances for all genuine and
      → impostor matches.
      y = euclidean_distances(Y,Y)

      #compute genuine and imposter by stack arrays in sequence horizontally.
      genuine = np.hstack([y_score[:4132,:4132].reshape(1,4132*4132), y_score[4132:
      →,4132:].reshape(1,4684*4684)])
      imposter = np.hstack([y_score[:4132,4132:].reshape(1,4132*4684), y_score[4132:,
      →4132].reshape(1,4132*4684)])
```

```
#plot histograms of the genuine and imposter distances on the same set of axes.
plt.hist(genuine[0], alpha = 1, bins = 100, color = 'lightgreen', label = 'Genuine',)
plt.hist(imposter[0], alpha = 1, bins = 100, color = 'skyblue', label = 'Imposter',)
plt.title('Genuine vs. Imposter')
plt.legend()
plt.show()
```



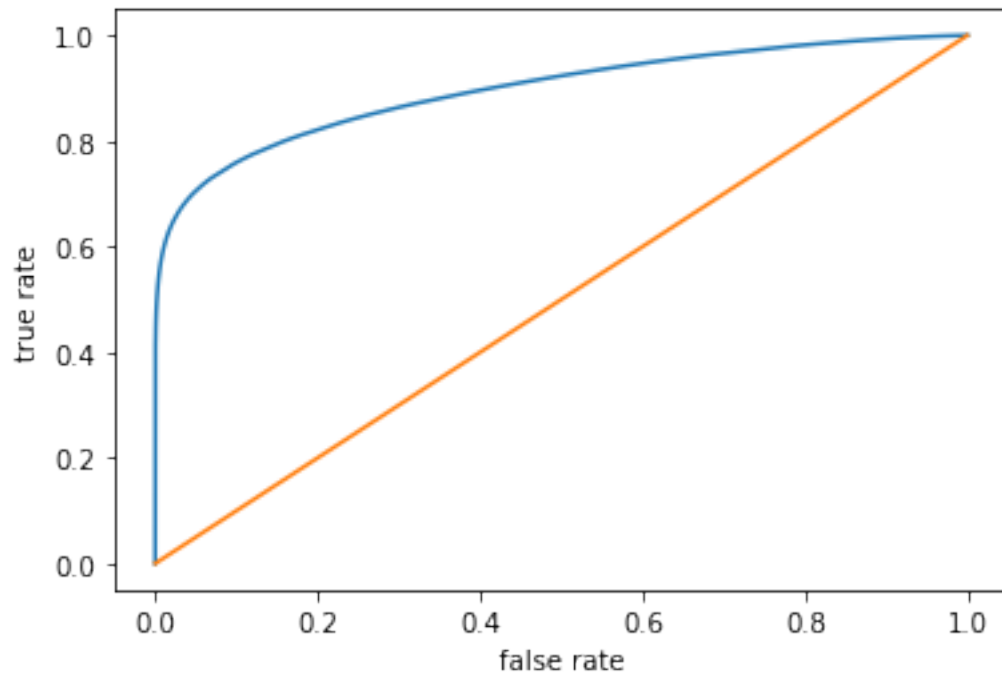
- (f) Generate an ROC curve from the above sets of distances. What is the equal error rate? What is the error rate of a classifier that simply guesses randomly?

```
[12]: #from sklearn.metrics.pairwise import euclidean_distances
#from sklearn import metrics

#Generate an ROC curve.
y_score = - (euclidean_distances(X,X)).reshape(1, (4684+4132)*(4684+4132))
#from above questions we can observe that [0]4132, [1]4684.
y = (euclidean_distances(Y,Y)).reshape(1, (4684+4132)*(4684+4132))
fp,tp, thresholds = metrics.roc_curve(y[0],y_score[0])
plt.plot(tp, fp)
#plot the ROC curve
plt.plot([0,1],[0,1])
```



```
plt.plot(color = 'colors')
plt.xlabel('false positive rate')
plt.ylabel('true positive rate')
plt.show()
```



```
[27]: #Compute the equal error rate.
def generate_equal_error_rate(tp,fp,thresholds):
    →                                     #False Positive Rate (FPR)
    n = np.argmin(np.abs((1-tp)-fp))
    →                                     #True Positive Rate (TPR)
    eer = 1-np.mean(((1-tp)[n],fp[n]))
    return eer
eer = generate_equal_error_rate(tp,fp,thresholds)
print(eer)
```

0.1855478851724539

According to the above, the EER is about 0.186. Guess randomly, the EER of a classifier may be 0.5.

(g) Implement a K-NN classifier.

```
[27]: #from collections import Counter
#implement a K-NN classifier
```

```

def generate_k_nn_classifier(X_dtrain,Y_dtrain,X_dtest ,n):
    Y_test=[]
    for i in range(len(X_dtest)):
        neighbors = np.argsort(((np.array(X_dtest.iloc[i])-np.
→array(X_dtrain))**2).sum(-1))
        prediction = Counter(Y_dtrain.iloc[neighbors[:n]]).most_common(1)[0][0]
        Y_test.append(prediction)
    return Y_test

```

```
[11]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import neighbors
from statistics import mode
from sklearn import model_selection
from sklearn.metrics import confusion_matrix

[2]: dtrain = pd.read_csv('digit-train.csv', header=0, dtype=np.int)
X_dtest = pd.read_csv('digit-test.csv', header=0, dtype=np.int)
dtrain.head()
X_dtrain = dtrain.drop('label', axis = 1)
Y_dtrain = dtrain['label']

[3]: #from collections import Counter
#implement a K-NN classifier

def generate_k_nn_classifier(X_dtrain,Y_dtrain,X_dtest ,n):
    Y_test=[]
    for i in range(len(X_dtest)):
        neighbors = np.argsort(((np.array(X_dtest.iloc[i])-np.
→array(X_dtrain))**2).sum(-1))
        prediction = Counter(Y_dtrain.iloc[neighbors[:n]]).most_common(1)[0][0]
        Y_test.append(prediction)
    return Y_test
```

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

```
[5]: #from sklearn import model_selection
#use our K-NN classifier to perform 3 fold cross-validation

X_train, X_test, y_train, y_test = model_selection.train_test_split(X_dtrain,
→Y_dtrain, random_state=2, test_size=0.5)
Y_prediction = generate_k_nn_classifier(X_train,y_train,X_test,10)
```

After generating a confusion matrix below, we achieved the average accuracy which is 0.958.

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

```
[12]: #import pandas as pd
#from sklearn.metrics import confusion_matrix

c_matrix = confusion_matrix(np.array(y_test).reshape(1,21000)[0],np.
    →array(Y_prediction))
k_nn_c_matrix = pd.DataFrame(c_matrix,(x for x in range(10)),(x for x in_
    →range(10)))
k_nn_c_matrix
```

```
[12]:
```

	pred 0	pred 1	pred 2	pred 3	pred 4	pred 5	pred 6	pred 7	\
true 0	2045	3	1	0	0	2	7	1	
true 1	0	2344	5	1	2	0	4	3	
true 2	24	36	1934	5	1	3	6	51	
true 3	4	15	13	2071	0	19	4	19	
true 4	2	27	0	0	1986	0	10	3	
true 5	5	10	1	41	1	1775	26	0	
true 6	13	6	0	0	2	9	2010	0	
true 7	1	35	8	0	5	0	0	2155	
true 8	5	38	7	38	9	37	6	8	
true 9	11	8	4	17	15	2	1	34	

	pred 8	pred 9
true 0	1	1
true 1	1	1
true 2	5	5
true 3	15	15
true 4	0	72
true 5	4	18
true 6	3	0
true 7	0	28
true 8	1821	37
true 9	2	1977

According to the matrix above, we observe that 8 is the trickies to classify.

```
[13]: #calculate average accuracy

accu = sum(k_nn_c_matrix.apply(lambda x: max (x),axis = 1))/sum(k_nn_c_matrix.
    →apply(lambda x: sum(x), axis =1))
print(accu)
```

```
[13]: 0.958
```

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

```

[:]: #load new dataset for test

X_dtest = pd.read_csv('digit-test.csv', header=0, dtype=np.int)
x_dtest = X_dtest.drop('label', axis = 1)
Y_dtest = X_dtest['label']

[:]: #test our K-NN classifier with the test data
#from sklearn import model_selection

X_train, X_test, y_train, y_test = model_selection.train_test_split(x_dtest,
→Y_dtest, random_state=2, test_size=0.5)
Y_prediction = generate_k_nn_classifier(X_train,y_train,X_test,10)

```

2. Titanic

```
In [1]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
from sklearn.linear_model import LogisticRegression
```

```
In [2]: # read in data
train = pd.read_csv("train.csv")
test = pd.read_csv("test.csv")
test.head()
```

Out[2]:

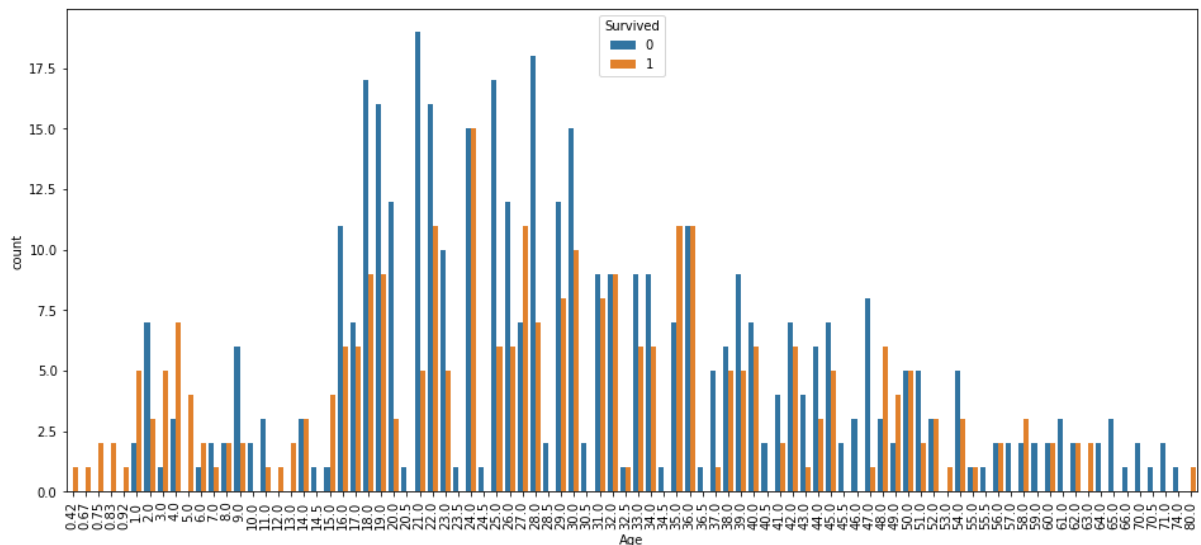
	PassengerId	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embark
0	892	3	Kelly, Mr. James	male	34.5	0	0	330911	7.8292	NaN	
1	893	3	Wilkes, Mrs. James (Ellen Needs)	female	47.0	1	0	363272	7.0000	NaN	
2	894	2	Myles, Mr. Thomas Francis	male	62.0	0	0	240276	9.6875	NaN	
3	895	3	Wirz, Mr. Albert	male	27.0	0	0	315154	8.6625	NaN	
4	896	3	Hirvonen, Mrs. Alexander (Helga E Lindqvist)	female	22.0	1	1	3101298	12.2875	NaN	

```
In [3]: train.head()
```

```
Out[3]:
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Ca
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	N
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...)	female	38.0	1	0	PC 17599	71.2833	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	N
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	N

```
In [4]: # find pattern for Age vs Survived
plt.figure(figsize = (16,7))
sns.countplot(x="Age", hue = "Survived", data = train)
plt.xticks(rotation="vertical")
plt.show()
```



We noticed that there are people who are between 15 and 30 have a lower survival rate. We can then separate people into several age groups: (0-15), (15-30), (30-55), (55+) so that we can have a better prediction.

After reviewing the dataset, we want to drop "PassengerId", "Name", "Ticket", "Embarked" which are obviously unhelpful for prediction. We also want to drop "Cabin" since it contains too many NaN.

```
In [5]: # cleaning data
# replace null value in "age" with mean
def mean_to_null_train(cols):
    Age = cols[0]
    Pclass = cols[1]
    if pd.isnull(Age):
        return int(train[train["Pclass"] == Pclass]["Age"].mean())
    else:
        return Age
# replace null value in "Fare" with mean
def replace_Fare_null_train(cols):
    Fare = cols[0]
    Pclass = cols[1]
    if pd.isnull(Fare):
        return int(train[train["Pclass"] == Pclass]["Fare"].mean())
    else:
        return Fare
# start replacing
train["Age"] = train[["Age", "Pclass"]].apply(mean_to_null_train,axis=1)
train["Fare"] = train[["Fare", "Pclass"]].apply(replace_Fare_null_train,
axis=1)
train.drop(["PassengerId", "Ticket", "Cabin", "Embarked", "Name"],axis=1,
inplace = True)
# add different age group to make better prediction
train["AgeGroup"] = ""
train.loc[train["Age"] < 15, "AgeGroup"] = '<15'
train.loc[(15 <= train["Age"]) & (train["Age"] < 30), "AgeGroup"] = "15-30"
train.loc[(30 <= train["Age"]) & (train["Age"] < 50), "AgeGroup"] = "30-50"
train.loc[train["Age"] >= 55, "AgeGroup"] = ">50"
```

```
In [6]: # turn categorical variables into dummy variables
sex = pd.get_dummies(train["Sex"], drop_first = True)
pclass = pd.get_dummies(train["Pclass"],drop_first=True)
age = pd.get_dummies(train["AgeGroup"], drop_first = True)
# put new variables with original dataset
train = pd.concat([train,pclass,sex,age],axis=1)
# drop catagorical columns
train.drop(["Sex", "Age", "Pclass", "AgeGroup"],axis=1,inplace = True)
```



```
In [7]: train.head()
```

```
Out[7]:
```

	Survived	SibSp	Parch	Fare	2	3	male	15-30	30-50	<15	>50
0	0	1	0	7.2500	0	1	1	1	0	0	0
1	1	1	0	71.2833	0	0	0	0	1	0	0
2	1	0	0	7.9250	0	1	0	1	0	0	0
3	1	1	0	53.1000	0	0	0	0	1	0	0
4	0	0	0	8.0500	0	1	1	0	1	0	0

```
In [8]: # same process for test data
def mean_to_null_test(cols):
    Age = cols[0]
    Pclass = cols[1]
    if pd.isnull(Age):
        return int(test[test["Pclass"] == Pclass]["Age"].mean())
    else:
        return Age
def replace_Fare_null_test(cols):
    Fare = cols[0]
    Pclass = cols[1]
    if pd.isnull(Fare):
        return int(test[test["Pclass"] == Pclass]["Fare"].mean())
    else:
        return Fare


test["Fare"] = test[["Fare", "Pclass"]].apply(replace_Fare_null_test,axis=1)
test["Age"] = test[["Age", "Pclass"]].apply(mean_to_null_test,axis=1)
test.drop(["Name", "Ticket", "Cabin", "Embarked"],axis=1,inplace = True)

test["AgeGroup"] = ""
test.loc[test["Age"] < 15, "AgeGroup"] = "<15"
test.loc[(15 <= test["Age"]) & (test["Age"] < 30), "AgeGroup"] = "15-30"
test.loc[(30 <= test["Age"]) & (test["Age"] < 50), "AgeGroup"] = "30-50"
test.loc[test["Age"] >= 55, "AgeGroup"] = ">55"

test.dropna(inplace = True)
sex = pd.get_dummies(test["Sex"], drop_first = True)
pclass = pd.get_dummies(test["Pclass"],drop_first=True)
age = pd.get_dummies(test["AgeGroup"], drop_first=True)
test = pd.concat([test,pclass,sex,age],axis=1)
test.drop(["Sex", "Age", "Pclass", "AgeGroup"],axis=1,inplace = True)
test_X = test.drop(["PassengerId"],axis=1,inplace = False)
```


```
In [9]: # prepare training dataset
X_train = train.drop("Survived",axis=1)
y_train = train["Survived"]
# train our model
logistic_model = LogisticRegression()
logistic_model.fit(X_train,y_train)
# make prediction
predictions = logistic_model.predict(test_X)
# output
df = pd.DataFrame(data = {"PassengerId" : test["PassengerId"], "Survived" : predictions})
df.to_csv(path_or_buf="result.csv", index=False)
```

```
//anaconda3/lib/python3.7/site-packages/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
FutureWarning)
```

 Getting Started Prediction Competition

Titanic: Machine Learning from Disaster

Start here! Predict survival on the Titanic and get familiar with ML basics

 Kaggle · 10,877 teams · Ongoing









[Overview](#)
[Data](#)
[Notebooks](#)
[Discussion](#)
[Leaderboard](#)
[Rules](#)
[Team](#)
[My Submissions](#)
[Submit Predictions](#)

Your most recent submission

Name	Submitted	Wait time	Execution time	Score
result.csv	11 hours ago	0 seconds	0 seconds	0.77033

Complete

[Jump to your position on the leaderboard](#)

6053	chenzhaoming		0.77033	1	13h
6054	Yifeng Wu		0.77033	2	13h
6055	khaleel		0.77033	5	11h
6056	Xueqi Wei	 	0.77033	4	11h
<div> Your Best Entry ↑ </div> <div> Your submission scored 0.77033, which is an improvement of your previous score of 0.66028. Great job! </div> <div>  Tweet this! </div>					
6057	gina574574		0.77033	1	10h
6058	lzoom777		0.77033	1	10h

In []:

Written-up

1.

Prove $\text{var}[X - Y] = \text{var}[X] + \text{var}[Y] - 2\text{cov}[X, Y]$

Sol:

$$\begin{aligned}\text{var}[X - Y] &= E[(X - Y)^2] - E[X - Y]^2 \\&= E[X^2 - 2XY + Y^2] - E[X - Y]^2 \\&= E[X^2] - 2E[XY] + E[Y^2] - E[X - Y]^2 \\&= E[X^2] - 2E[XY] + E[Y^2] - E[X]^2 + 2E[X]E[Y] - E[Y]^2 \\&= (E[X^2] - E[X]^2) + (E[Y^2] - E[Y]^2) - 2(E[XY] - E[X]E[Y])\end{aligned}$$

Since $\text{var}[X] = E[X^2] - E[X]^2$, $\text{var}[Y] = E[Y^2] - E[Y]^2$,

$$\begin{aligned}\text{and } \text{cov}[X, Y] &= E[(X - E[X])(Y - E[Y])] = E[XY - XE[Y] - YE[X] + E[X]E[Y]] \\&= E[XY] - E[X]E[Y] - E[X]E[Y] + E[X]E[Y] \\&= E[XY] - E[X]E[Y]\end{aligned}$$

Thus, $\text{var}[X - Y] = \text{var}[X] + \text{var}[Y] - 2\text{cov}[X, Y]$

2.

(a)

Let X denotes the result of the detector, X = 1 means the result is positive, X = 0 means the result is negative.

Let D denotes the status of the widgets, D = 1 means the widget is defective, D = 0 means the widget is not defective.

$$P(X = 1|D = 1) = 95\% = 0.95$$

$$P(X = 0|D = 0) = 95\% = 0.95$$

$$P(D = 1) = \frac{1}{100000}$$

$$P(X = 1) = P(X = 1|D = 1)P(D = 1) + P(X = 1|D = 0)P(D = 0)$$

$$= 0.95 \times \frac{1}{100000} + (1 - 0.95) \times (1 - \frac{1}{100000})$$

$$= 0.050009$$

$$P(D = 1|X = 1) = \frac{P(X = 1|D = 1) \times P(D = 1)}{P(X = 1)}$$

$$= \frac{0.95 \times \frac{1}{100000}}{0.050009}$$

$$= 0.0001899658$$

(b)

$$\text{good widgets that are thrown away} = 10,000,000 \times P(X = 1) \times P(D = 0|X = 1)$$

$$= 10,000,000 \times 0.050009 \times (1 - 0.00018999658)$$

$$= 499,995$$

$$P(D = 1|X = 0) = \frac{P(X = 0|D = 1)P(D = 1)}{P(X = 0)}$$

$$= \frac{(1-0.95)\frac{1}{100000}}{(1-0.050009)}$$

$$= 5.26320775670507 \times 10^{-7}$$

$$\text{bad widgets shipped} = 10,000,000 \times P(X = 0) \times P(D = 1|X = 0)$$

$$= 10,000,000 \times (1 - 0.050009) \times 5.26320775670507 \times 10^{-7}$$

$$= 5$$

3.

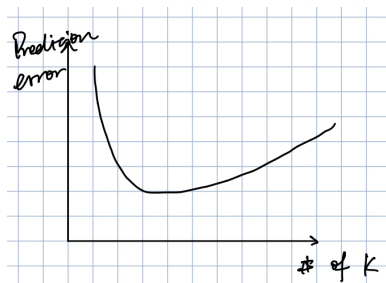
a)

When $K = 1$, there will be no error. Since the prediction for training data point x_i includes (x_i, y_i) as part of the example training data used by kNN, the point we use for prediction is the same point that we are trying to test when $k = 1$. Thus, there will be no prediction error.

When K becomes higher and get closer to n , there will be more neighbors that we are considering. The model becomes more complex, so the prediction error goes up.

When $K = n$, the prediction error will be $\frac{1}{2}$. In the training dataset, there are two classes that each contains exactly half of the training data. When we consider $K = n$ nearest neighbors, the kNN classifier will found that the probability that x_i belongs to any of the two classes equals to $\frac{1}{2}$, which means the prediction error is $\frac{1}{2}$.

b)



When K goes down, as stated in part a), the particular training set we choose will be more and more heavily influencing our prediction and the error in validation set becomes higher and higher.

When K goes up, the prediction error will go down at first since we have more information and can make better predictions. However, when K keeps going up and becomes very large, the model will be over-fitted and result in the error going up as the above picture.

c)

For the computational requirements in a n -fold validation, when n gets larger, the model becomes complex, the

run time will increase. For validation accuracy. Large n means less bias but higher variance and high accuracy but long run time. Small n means more bias but less variance, lower accuracy but shorter running time. Thus, n should not be too large or too small. I recommend using $n = 5$ or $n = 10$.

d)

When K is very large, the points with a long distance to our data point can only give us a few information about the classification of our data. In this case, we can put more weight on the closer points and less weight on the farther points to have a better prediction. The weight function should go up when distance becomes smaller and go down when distance becomes larger.

e)

In high dimension, the distance metric becomes meaningless since we take all dimensions into account in kNN but in reality some dimensions are useless and may have a bad impact on our prediction. Also, high dimension will make the computation process very complex, which will lead to a long time in calculation.