CS 5785: Applied Machine Learning **(Due: 09/18/19)**

Submission Assignment #1

Instructor: Nathan Kallus *Name:* Konstantinos Ntalis, Max Wulff, *Netid:* kn442, mcw232

Course Policy: This homework is due on September 5th, 2019 at 12:01AM. Upload your homework to CMS. Please upload the submission as a single .zip file. A complete submission should include:.

- A write-up as a single .pdf file.
- Source code and data files for all of your experiments (AND figures) in .py files if you use Python or .ipynb files if you use the IPython Notebook. If you use some other language, include all build scripts necessary to build and run your project along with instructions on how to compile and run your code.

Problem 1: Digit Recognizer

(a) We download and import the training and test data from Kaggle.

```
train = pd.read_csv("/content/drive/My Drive/AML - Fall 2019/train.csv")
test = pd.read_csv("/content/drive/My Drive/AML - Fall 2019/test.csv")
```

(b) We first create a function that displays an MNIST digit given its rowindex in the set. We then use this function to display the first occurence of each digit from the training set.

```
def MNISTdigit(rowindex):
    img = np.array(train.iloc[rowindex,1:]).reshape(-1,28)
    plt.imshow(img, cmap = 'Greys')
    plt.show()

for i in range(0,10):
    row = 0
    while train.iloc[row,0] != i:
        row += 1
MNISTdigit(row)
```

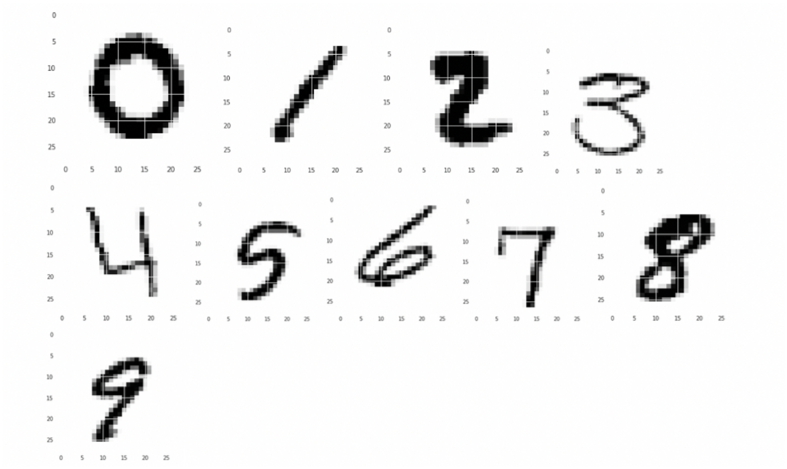


Figure 1: MNIST digits

(c) We use the following simple code to count the frequency of each occurrence and create a normalized histogram.

```
    train['label'].value_counts(sort = False , normalize = True).plot('bar')
```

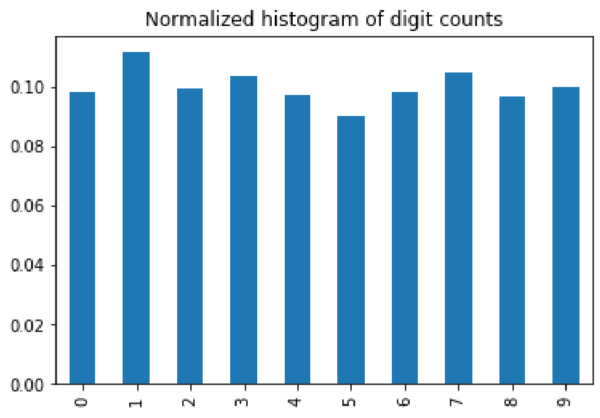


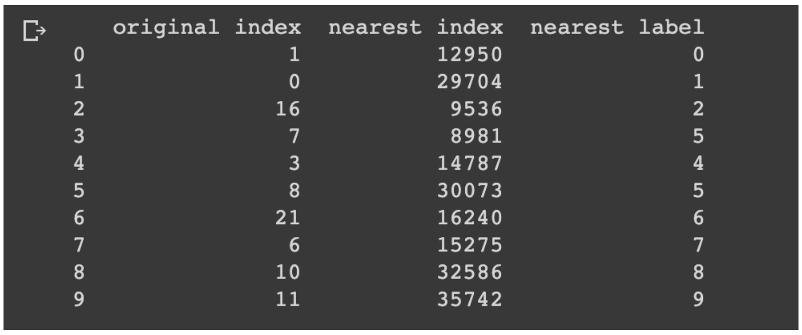
Figure 2: Plot of the frequency of digit counts

The class distribution is relatively even, with the digit 5 being the most infrequent and 1 the most frequent.

(d) To approach this question, we start by getting the index of the first instance of each digit. We can then calculate the distance from all other instances using a simple loop.

```
train_array = train.to_numpy()
nearest = []

for i in index:
  l2_dist = np.argsort(((train_array[i,1:] - train_array[:,1:])**2).sum(-1))
  nearest.append(l2_dist[1])
```

Here it's worth noting, that we skip the first element, so that we exclude the distance from itself. Having done that, it's straightforward to compute the best match (see jupyter notebook for details). The results can be seen in the following table Our Number 3's closest neighbor is a Number 5. Other than that every other's nearest



```
   original index  nearest index  nearest label
0               1          12950              0
1               0          29704              1
2              16           9536              2
3               7           8981              5
4               3          14787              4
5               8          30073              5
6              21          16240              6
7               6          15275              7
8              10          32586              8
9              11          35742              9
```

Figure 3: Best match between sample and the rest of the training data

neighbor was itself.

(e) For this part, we won't include all the code, which can be found in the notebook. The idea is that we create a large matrix where 1's are stacked below zeros. This number of rows of this matrix will be the amount of zeros plus the amount of ones we have in the set. The number of columns will of course be the fixed number of pixels per digit. From that, we can use sklearn's euclidean distances function to construct a distance matrix, which will essence calculate the pairwise distances of the above matrix.

```
dis = sklearn.metrics.pairwise.euclidean_distances(zo)
```

Note here that the **zo** is the stacked matrix with zeros on top of ones. As a result, we can think of the distance matrix as a matrix consisting of four major blocks. Genuine 0's on the upper left, genuine 1's on the lower right and imposters on the off-diagonal blocks (0-1 and 1-0 respectively). Having this structure in mind, it's easy to create two stacked vectors of genuine matches and imposter matches and then plot the required histograms.

```
#plot the genuine and imposter distances on the same plot
bins = np.linspace(0, 4000, 80)

plt.hist(gen, bins, alpha = 0.5, label = 'Genuine')
plt.hist(imp, bins, alpha = 0.5, label = 'Imposter')
plt.legend(loc = 'upper left')
plt.title('Histogram of Genuine and Imposter Distances')
plt.xlabel('Distance')
plt.ylabel('Frequency')
plt.show()
```

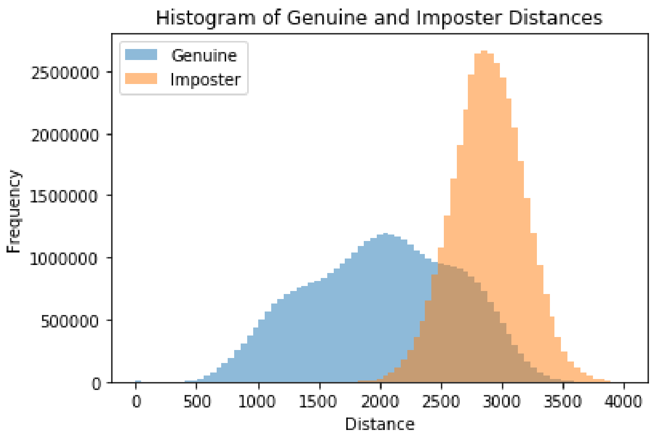


Figure 4: Histogram of Genuine/Imposter distances

(f) To calculate FPR, TPR and threshold for the ROC curve, we stack genuine and imposter distances in a long vector and also and a new column of labels (0 for imposter, 1 for genuine) in order to use sklearn's ROC curve function.

```
#use zero to mark genuine matches
markg = np.zeros(gen.shape)

#use one to mark imposter matches
marki = np.ones(imp.shape)

#create two master vectors of distances and corresponding labels
dist = np.hstack([gen, imp])
mark = np.hstack([markg, marki])

fpr, tpr, threshold = sklearn.metrics.roc_curve(mark, dist)
```

Furthermore, using the above set of distances, we can generate an ROC curve. The code to generate the plot can be found in the jupyter notebook included in the submission.

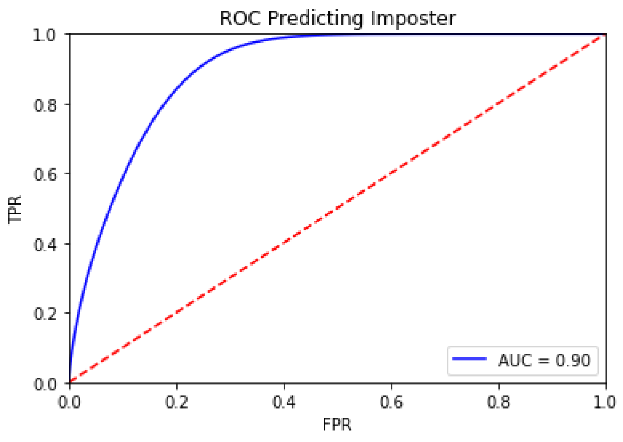


Figure 5: ROC Curve

As for the equal error rate, we use the following code to calculate it. Keeping in mind that it is defined as the threshold point where FPR = FNR, we get:

```
    EER_fpr = fpr[np.nanargmin(np.absolute(((1-tpr) - fpr)))]
    EER_tpr = tpr[np.nanargmin(np.absolute(((1-tpr) - fpr)))]
```

which is equal to

```
    (0.18554784422124979 0.814452073876342)
```

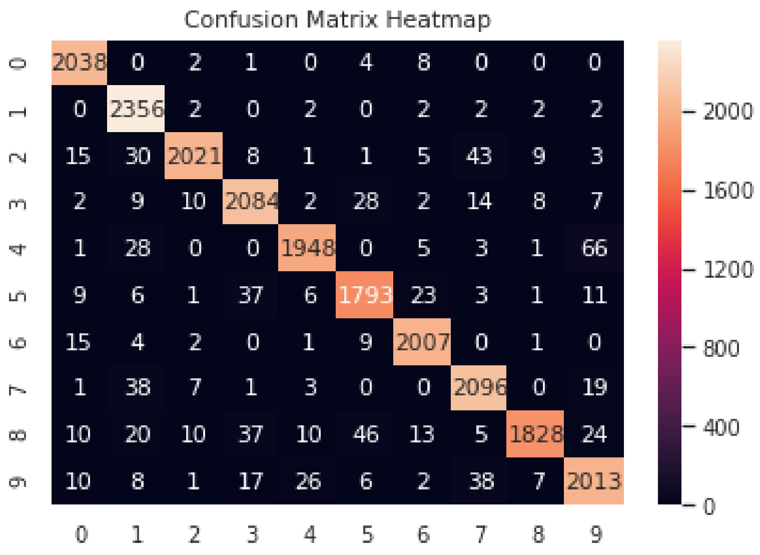
respectively. Finally, the classifier that guesses randomly has a 50% error rate.

(g) The implementation of our KNN is based on two functions, KNNtrain and KNNPred. The first one is responsible for calculating the distances, sorting them and picking the k nearest ones, according to the given input. KNNPred carries out the majority vote, and as the name suggests, returns a prediction.

```
#create function to run KNN. take in a training set, test set, and k.
#output 3 arrays: (1) the actual labels (2) the nearest distances
#(3) the labels of the nearest neighbors

#trainSet includes the label in the first column
#testSet does not include the first column
def KNNtrain(trainSet, testSet, k):
  nearest = np.zeros((testSet.shape[0],k))
  nearestlabels = np.zeros((testSet.shape[0],k))


  for i in range(testSet.shape[0]):
    l2_dist = np.argsort(((testSet[i,:] - trainSet[:,1:])**2).sum(-1))
    nearest[i,:] = l2_dist[0:k]

  for i in range(nearest.shape[0]):
    for j in range(nearest.shape[1]):
      nearestlabels[i,j] = trainSet[int(nearest[i,j]),0]


  return(nearest, nearestlabels)

#make a prediction based on the majority vote k nearest neighbors
def KNNPred(labels):

  prediction = np.zeros(labels.shape[0])
  for i in range(labels.shape[0]):
    vote = stats.mode(labels[i,:])[0][0]
    prediction[i] = vote
  return prediction
```

(h) We can now randomly split the training data into two halves, train the classifier on the first half of the data, and test it on the second half. The accuracy is 96%.

(i) We can plot the confusion matrix for our model using seaborn.

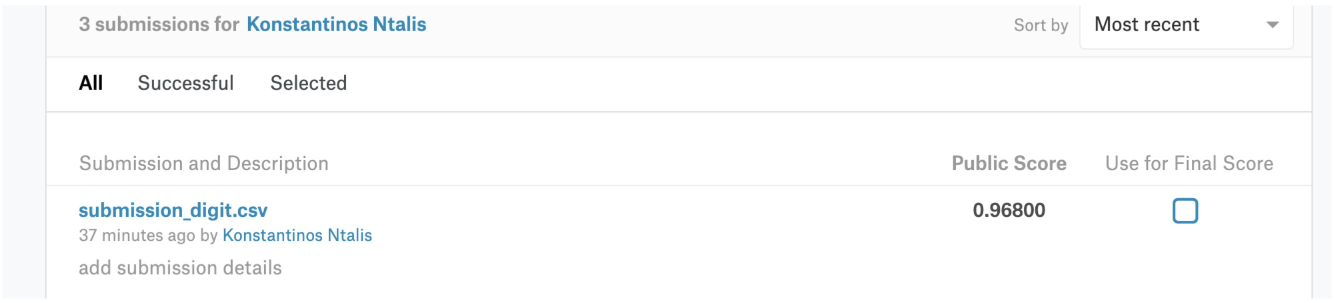


	0	1	2	3	4	5	6	7	8	9
0	2038	0	2	1	0	4	8	0	0	0
1	0	2356	2	0	2	0	2	2	2	2
2	15	30	2021	8	1	1	5	43	9	3
3	2	9	10	2084	2	28	2	14	8	7
4	1	28	0	0	1948	0	5	3	1	66
5	9	6	1	37	6	1793	23	3	1	11
6	15	4	2	0	1	9	2007	0	1	0
7	1	38	7	1	3	0	0	2096	0	19
8	10	20	10	37	10	46	13	5	1828	24
9	10	8	1	17	26	6	2	38	7	2013

Figure 6: 10x10 confusion matrix

(j) We finally train the classifier with all of the training data, and test it with the test data.

```
neighbor_final, labels_final = KNNtrain(train, test, k = 5)
```



3 submissions for Konstantinos Ntalis   Sort by Most recent

All   Successful   Selected

Submission and Description	Public Score	Use for Final Score
submission_digit.csv 37 minutes ago by Konstantinos Ntalis add submission details	0.96800	☐

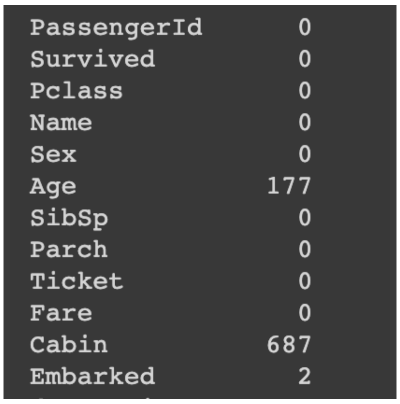
Figure 7: Kaggle Digit Recognizer submission

Problem 2: The Titanic Disaster

(a) We start by loading the titanic dataset.

```
train = pd.read_csv("/content/drive/My Drive/train_titanic.csv")
test = pd.read_csv("/content/drive/My Drive/test_titanic.csv")
```

(b) Before fitting logistic regression, we firstly explore the structure of the dataset, in order to make any necessary imputation if needed. Indeed, there are missing values in three categories. We choose to impute Age

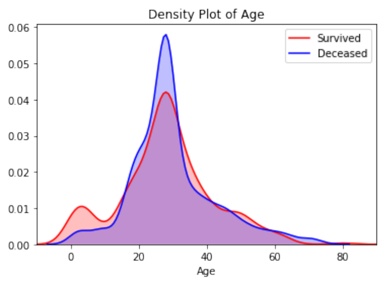


```
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
```

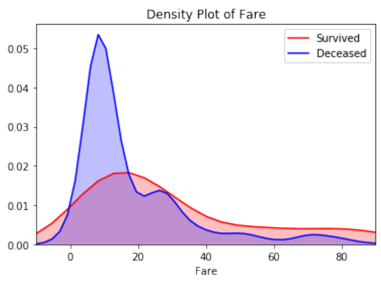
Figure 8: Missing values in the training set

using the median of the Age feature and Embarked using the most frequent value. As for the Cabin, since the missing values comprise 77% of the data, we choose to drop this feature completely. It should also be mentioned that exactly the same procedure has to be applied to the test set, so that the two sets have the same structure. Furthermore, we choose to drop features that seem irrelevant for the prediction (tickets, passengerID and name) and also create a new binary feature ('Alone') by combining SibSp and Parch, that indicated whether a passenger is travelling alone (see jupyter notebook for more details). Finally, we use dummy variables to transform our categorical features, in order to fit the logistic regression model.

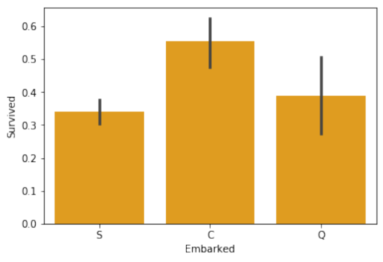
At this point, we can also produce some exploratory plots in order to study the relationship between our features and the response before fitting the regression. From the below plots, we can note that Sex and Fare



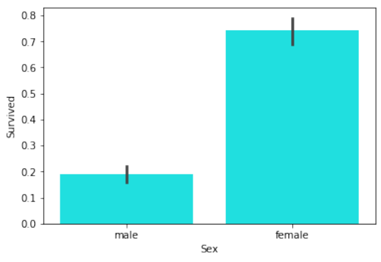
(a)



(b)



(c)



(d)

Figure 9: Plots for exploratory data analysis

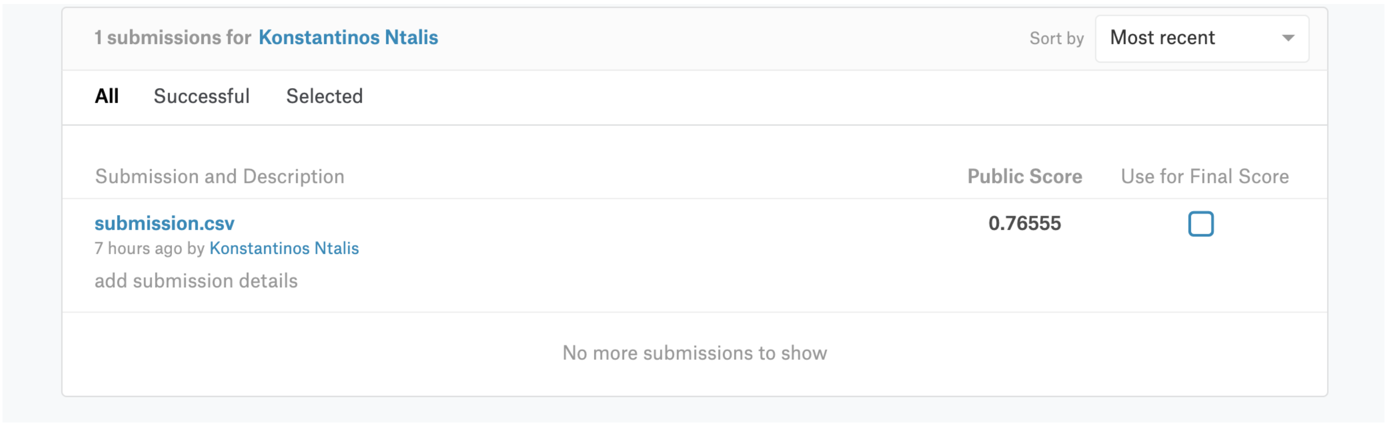
seem to be playing a major role for the survival of a passenger. Women and passengers with expensive fares seem to have greater chances of survival. On the other hand, Age does not seem to play a significant role. Finally, there is some difference in passengers that have embarked at different points. Those who boarded in France seem to have the highest survival rate.

(c) We can now fit the logistic regression model

```
    from sklearn import linear_model

    x = train_imputed[['Age', 'Fare', 'Pclass_2', 'Pclass_3', 'Sex_male', 'Embarked_Q', '
                                                            Embarked_S', 'Alone']]
    y = train_imputed['Survived']

    lr = linear_model.LogisticRegression()
    lr.fit(x, y)
```

and predict on the test data

```
    final_pred = lr.predict(test_imputed)
```



1 submissions for Konstantinos Ntalis

Sort by Most recent

All Successful Selected

Submission and Description	Public Score	Use for Final Score
submission.csv 7 hours ago by Konstantinos Ntalis add submission details	0.76555	☐

No more submissions to show

Figure 10: Kaggle Titanic submission

Problem 3: Written Exercises

(1)

$$
\begin{aligned}
Var[X-Y] =& E[(X-Y)^2] - E[X-Y]^2 && (0.1)\\
=& E[X^2 - 2XY + Y^2] - E[X-Y]^2 && (0.2)\\
=& E[X^2] - 2E[XY] + E[Y^2] - E[X-Y]^2 \quad \text{by linearity of expectation} && (0.3)\\
=& E[X^2] - 2E[XY] + E[Y^2] - E[X-Y]E[X-Y] && (0.4)\\
=& E[X^2] - 2E[XY] + E[Y^2] - E[X]^2 - E[Y]^2 + 2E[X]E[Y] && (0.5)\\
=& (E[X^2] - E[X]^2) + (E[Y^2] - E[Y]^2) - 2E[XY] + 2E[X]E[Y] && (0.6)\\
=& Var[X] + Var[Y] - 2 * Cov[X,Y] && (0.7)
\end{aligned}
$$

(2)

$$
\begin{aligned}
P(Pos|Def) =& 0.95\\
P(Neg|Good) =& 0.95\\
P(Def) =& 0.00001
\end{aligned}
$$

(a)

$$\begin{aligned} P(Def|Pos) =& \frac{P(Pos|Def)P(Def)}{P(Pos|Def)P(Def) + P(Pos|Good)P(Good)} && (0.8)\\ =& \frac{.95(.00001)}{.95(.00001) + .05(.99999)} && (0.9)\\ =& .00019 && (0.10) \end{aligned}$$

(b) Total Good: 9999900
Total Bad: 100
Good Thrown Away = $P(Pos|Good)*$Total Good = 499995
Bad Shipped = $P(Neg|Bad)*$Total Bad = 5

(3a) Let's first consider the boundary cases. When $k = 1$, since (x_i, y_i) is included, then the the prediction error would be zero. When $k = n$, everything is included and since the classes are balanced, we would have 50% chance getting it right. As we vary k from n to 1, the $0 - 1$ prediction error on the training data will, although not smoothly, almost monotonically decrease. As we gradually increase the complexity of our model, the bias decreases while the variance increases. We therefore expect our classifier to over-fit on the training set, reaching a perfect prediction rate when $k = 1$.

(3b) As k varies from 1 to n the validation error will first decrease and then increase again. This is because total error is a combination of the bias and variance. As k increases the bias of the fit also increases, however the variance goes down. The lowest total error will occur at the minimum of the combination of the two.

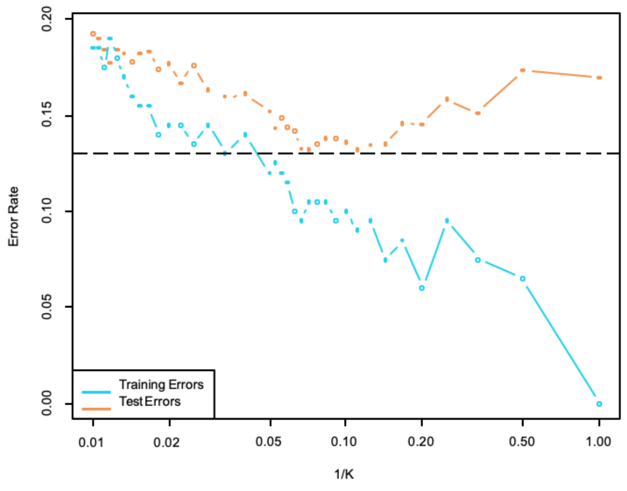


Figure 11: Source for Image: Chen, Yudong. ORIE 4740 Fall 2018 at Cornell University, Lecture 5-6.

(3c) To avoid this you could weight each neighbor's vote based on how far they are from the point. The closer the neighbor is the more their vote could count.

(3d) Memory constraints can occur here. If we are working in a high dimension, then the storage of large vectors and the calculation of the distance matrix can be very computationally expensive. Secondly, the curse of dimensionality can play a big role. Suppose that we have so many features, that inevitably some of them are irrelevant for the classification problem. Suppose further that we're trying to predict whether an individual will develop heart disease. If, apart from the BMI and the blood pressure, we include much more irrelevant features, such as their shoe size and their income, then two observations that would be close in two dimensions (using BMI and blood pressure) can suddenly appear to be very far apart in a higher dimensional space.

(3ei)

```
Log Train
                true red  true blue
predicted red          6          3
predicted blue         3          8

Log Test
                true red  true blue
predicted red          3          1
predicted blue         2          4
```

(3eii)

```
KNN Train
                true red  true blue
predicted red          5          4
predicted blue         4          7

KNN Test
                true red  true blue
predicted red          1          2
predicted blue         4          3
```

(3eiii) The training accuracy for the logistic regression algorithm was slightly higher than for KNN with $k = 1$. However the test accuracy was much higher. This could be because with $k = 1$, variance in a KNN algorithm is quite high, meaning a jagged boundary that can catch many test points on the wrong side. This can drive down the test error quite a lot.

```
Log Algo Train Accuracy:  0.7
Log Algo Test Accuracy:  0.7
KNN Algo Train Accuracy:  0.6
KNN Algo Test Accuracy:  0.4
```