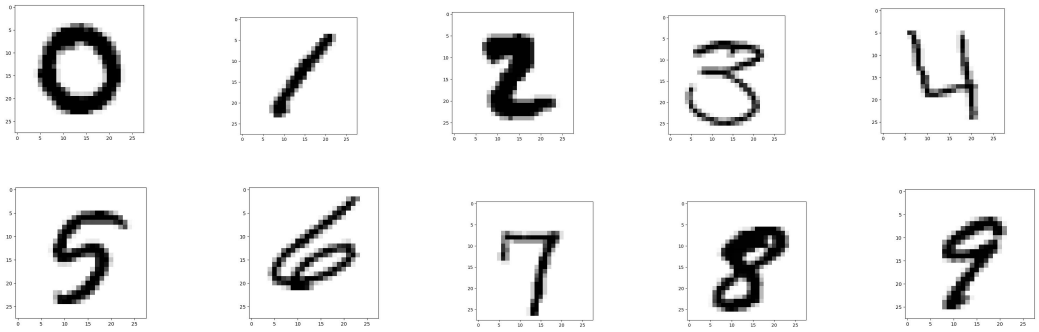
Programming 1

a) Completed. No results to discuss.

b) One of each digit displayed in figure1.

Figure 1:



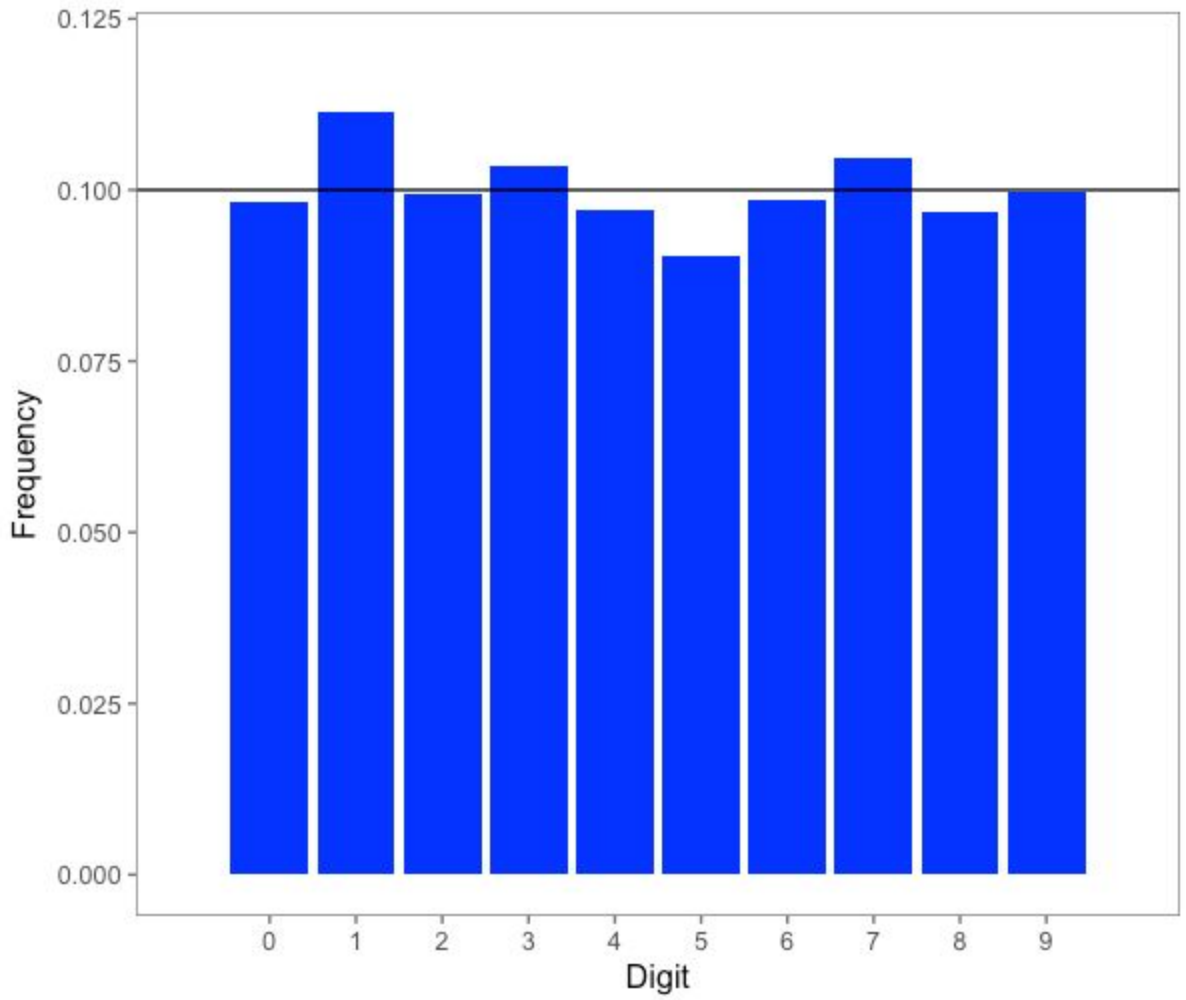
The information for each element includes a representation of the original written digit in 28x28 grayscale pixels. Each pixel is valued between 0 and 255, with 0 the lightest and 255 the darkest. To display the original digits we created a function that takes in the values corresponding to a single element as a row, creates a square array with the integer values cast to floating point, and plots the result. Digits are selected by scanning through the data for each label in sequence. The code for this section is in hw1_modules.py and display_dig.py

c) The prior probability of each class in the data is not uniform. Figure 2 displays the counts and relative frequency of each digit and Figure 3 shows a histogram of relative frequencies, along with a line at the expected frequency. Both the histogram and the counts are uneven. The digits 1, 3, and 7 are overrepresented, with 1 being the most common. The digits 0,2,4,5,6 and 8 are under represented, with 5 being the least common.

Figure 2:

	Digit									
	0	1	2	3	4	5	6	7	8	9
Count	4132	4684	4177	4351	4072	3795	4137	4401	4063	4188
Frequency	0.098	0.112	0.099	0.104	0.097	0.090	0.099	0.105	0.097	0.100

Figure 3 :



The counts were generated using basic statistic methods in r, and the histogram was generated with ggplot. The code for this section is in digit_prior_freq.Rhistory

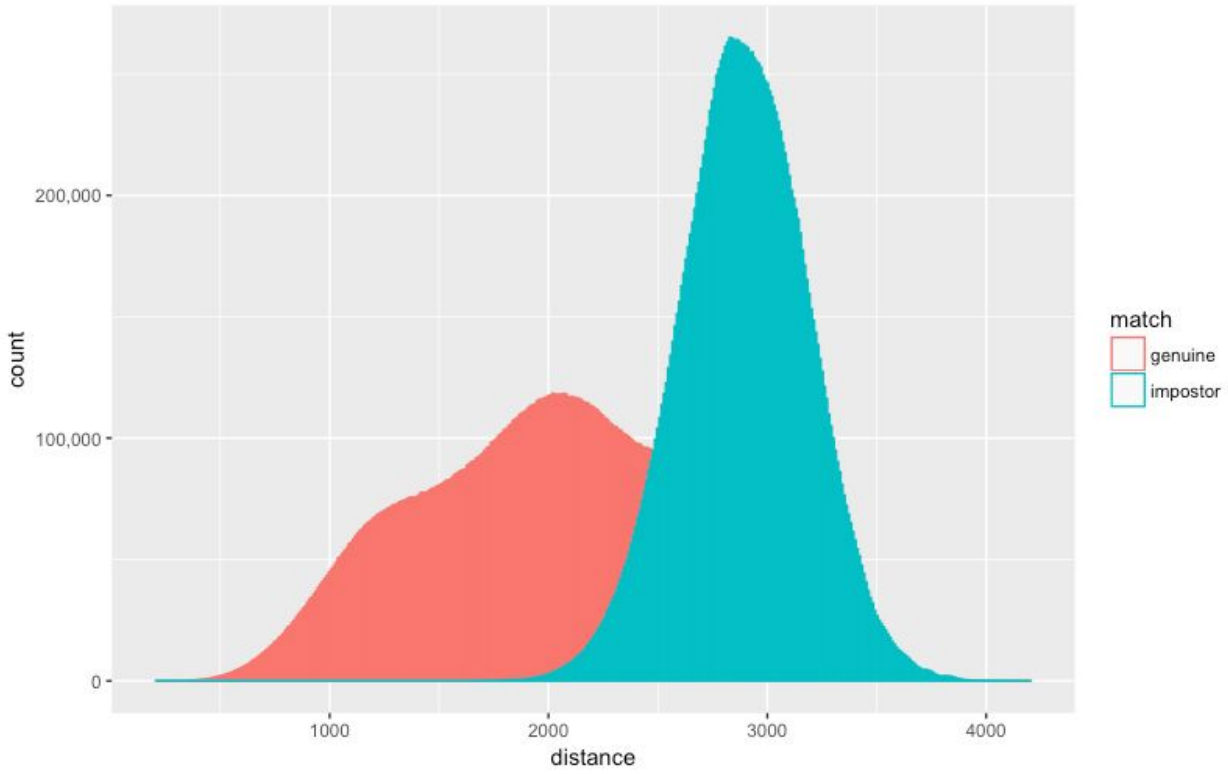
d) For this section we manually selected 10 entries in the data, representing one of each digit type. The distance between two elements is defined as the euclidean (L2) distance between the pixel values of each element. The distance between each digit and every element in the training data is then computed, and the closest match to each digit is found. The indexes, distance to closest match, and the labels of each match are stored and displayed in a table. The only digit that was misidentified by this method was 3, which was confused with a 5.
Figure 4:

Digit	SourceIdx	MatchIdx	MatchDist	MatchLabel
0	1	12950	1046.595433	0
1	0	29704	489.679487	1

2	16	9536	1380.877257	2
*3	7	8981	1832.664999	5*
4	3	14787	1356.880982	4
5	8	30073	1066.367666	5
6	21	16240	1446.51132	6
7	6	15275	863.501013	7
8	10	32586	1593.777588	8
9	11	35742	910.57674	9

We coded the original L2 distance calculation, which worked properly, but was too slow to handle the full set of training data. We also originally used two nested for loops to iterate over each digit and each element of the data to calculate the distance, which had similar performance problems. These were replaced by a distance function in the scipy library and working on all elements at once with matrix operations, rather than iterating over elements. The code for this section is in hw1_modules.py and distances.py

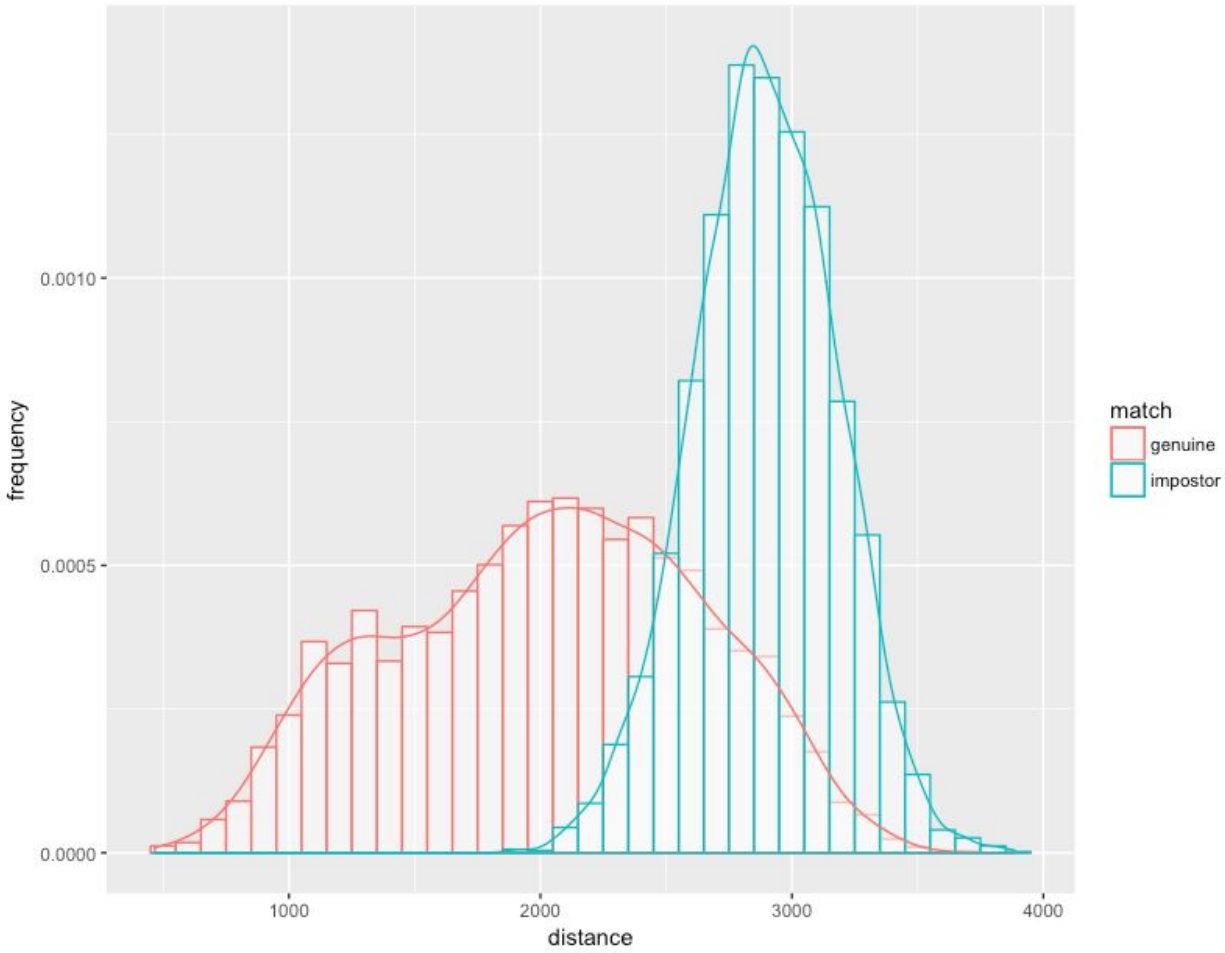
e) For this section, a subset of the data is taken where each element has a label of 0 or 1. The pairwise distance (again L2 distance on pixel values) is calculated between each element of this set, and every other element. A histogram (figure 5) shows the distribution of the “impostor” matches (01 and 10) and “genuine” matches (11 and 00) relative to distance. In the histogram you can see a clear distinction between the distribution of the two kinds of matches. Genuine matches are more widely distributed, with a mean distance of around 2000. The impostor matches are more narrowly distributed, with a mean distance just under 3000.
Figure 5:



The scipy library used to generate the pairwise distances returns these distances in a condensed array (equivalent to the lower triangular portion of the array flattened), so there are additional steps in the code to map the condensed array to the rows and columns of the normal square distance matrix. This ended up being unnecessary as we had to expand the condensed array for the next section. We then write out to CSV a line for each pair of elements, where the first column is whether the labels on the two elements match (True for genuine match, False for impostor match), and the second column is the distance between the two elements. This CSV was then loaded in R to create figure 5, a histogram showing the distribution of impostor and genuine matches over distance.

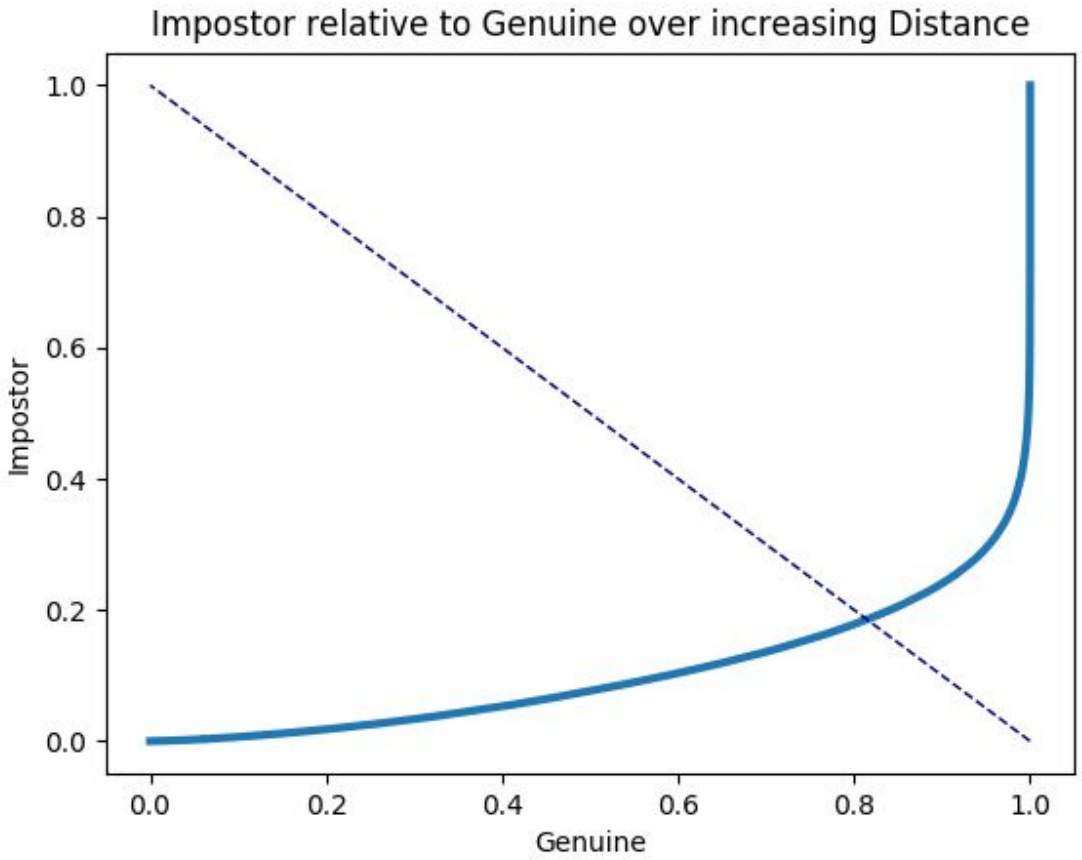
After the previous histogram was generated and written up, our deliverables were changed. Figure 6 shows a normalized histogram of the distances, overlaid with a density function line. The bin sizes on this histogram are larger, and due to time constraints was done with a random sample of 10,000 rows, versus the ~3 million rows used to generate the previous plot.

Figure 6



The code for this section is in hw1_modules.py, binary.py, binary_histogram.Rhistory, and binary_histogram_norm.Rhistory.

f) Using the same distance calculations from the previous section, we generated an ROC curve (figure 7) and calculated the EER to be 0.18555. This means that a classifier using distance as a threshold, could produce an optimal result where both genuine and impostor matches are classified with an ~%18 error rate. A classifier guessing randomly would have a 50% EER. Since most formulations of ROC look at a true positive as being above a threshold, and this particular model has true positives ('genuine') as being below a threshold (distance), this ROC curve appears flipped about the diagonal axis compared to a standard ROC. The dashed line represents the EER, and it intersects our curve at the correct point (~18%).
Figure 7



The ROC curve was calculated with a scipy library, and uses the distance matrix from the last section (in square form) and a binary label matrix created by taking outer product of the array of labels, and using the equality operator. In this label matrix element Xij is True if the label of element i in our data matches the label of element j, and False otherwise. This is more computationally efficient than the method we used in the previous section. The EER was calculated from the same results used for the ROC curve. This section uses the code in hw1_modules.py and binary.py.

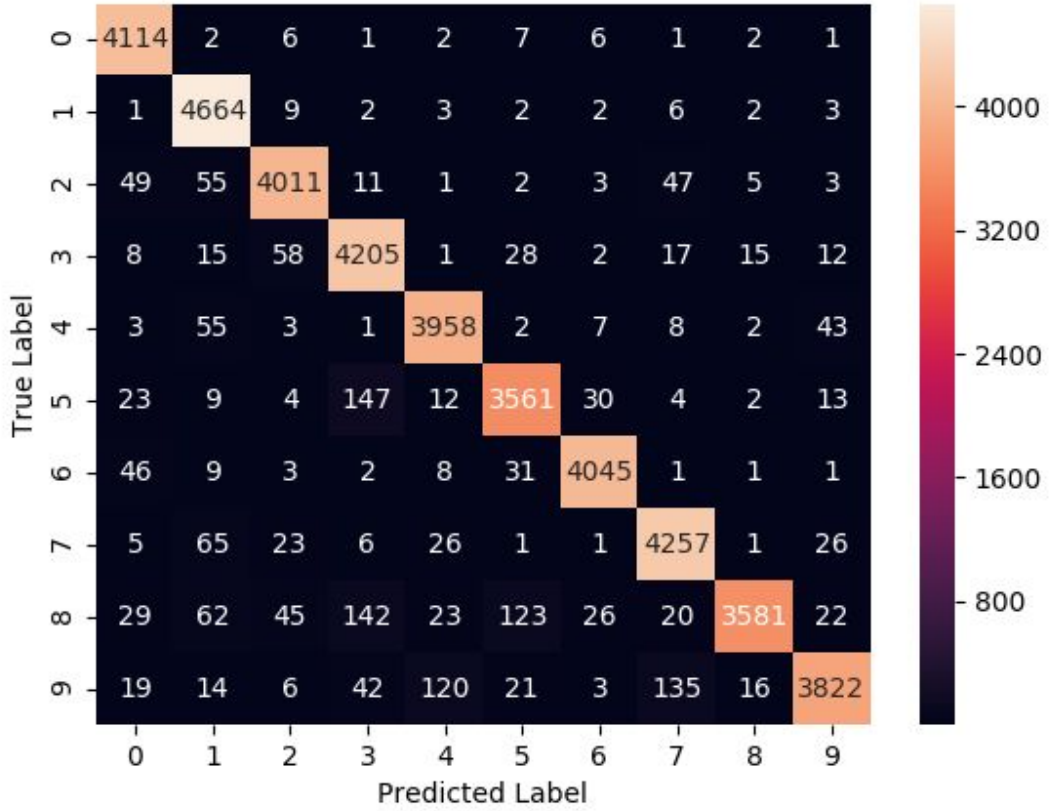
g) We implemented a knn classifier for this section. There are no results to discuss. The classifier itself takes an array of training observations, a matching array of training labels, an array of testing observations, and an integer value for k. It returns an array of predicted labels the same length as the number of elements tested. The function first computes a distance matrix between all the elements in the training set and all the elements in the testing set. Then, for each test element, it creates a temporary array of the training labels, and the distance to each training element. It then uses a partition sort library, where for a given integer n, the array will be sorted such that the n smallest values will occupy the 0 to (n-1) index positions, and all the larger values will occupy the n and higher index positions. Within each partition there is no further sorting. Partition sorting our 2 column label and distance array on the passed parameter k ensures that the k smallest distances are in the first k rows, along with their matching label.

The function then determines the most frequently seen label within the first k rows, and that becomes the prediction for that test element. It then iterates to the next test element and repeats the process. Once all elements have been given a prediction, the prediction array is returned. Although this method worked well for small training sets, larger training sets utilize an excessive amount of memory, and all the subsetting, indexing, and sorting slows down while the system tries to allocate and swap all the memory required. Code used for this section hw1_modules.py.

h) Using all training data and 3 fold cross validation, our average accuracy was .957333. We used an sklearn library to generate three pairs of index arrays, which we used to subset our training data into “training” and “testing” for the purposes of cross validation. For each set of “training” and “testing”, we passed the “training” values, “training” labels, and “testing” values to our k nearest neighbor classifier, with k=3. We then compared our known “testing” labels with the predictions from the knn classifier, and determined our accuracy as the percentage correctly predicted. This was averaged over the three cross validation runs. Code used for this section hw1_modules.py, and knn.py.

i) Our confusion matrix is shown in figure 7. The digits 8 and 9 were the most difficult to classify, along with 5 to a lesser degree. 8 was often misclassified as a 3 or a 5, 9 was often misclassified as a 4 or a 7, and 5 was often misclassified as a 3. These are all plausible results given the similarities of the shapes of those digits. The most reliable predictions were for 0 and 1, with 4 and 6 being the next most reliable predictions. The frequent misclassification of 9 as a 4, along with the general reliability of predicting 4, may indicate that our model overpredicts 4 as

a label in general.



The confusion matrix was calculated at the same time as the cross validation. After each cross validation run, the predicted labels and the true labels were passed to an sklearn library, which generated a confusion matrix. These matrices were summed together to form a cumulative measure, which was then plotted as a heatmap to create the figure above. Code used for this section hw1_modules.py, and knn.py.

j) Submitted to kaggle with accuracy of .96614, for rank 1145. The size of the training and testing data exceeded the available memory we had for computation. Dividing the real test data into 5k row chunks allowed us to processes each chunk in memory, without needing to swap to disk, which greatly improved performance. The output from each chunk was concatenated together before writing out to csv for submission. Code used for this section, hw1_modules.py, hw1_main.py.

Part 2. Titanic

Introduction

The data set is consist of information of all passengers that were on Titanic, the notorious sunken ship. The data documented detailed passenger information such as age, gender, travel companion, ticket price, and cabin class. (figure 1) It is interesting to us to see if we can predict passengers' survival by looking at all the attributes. Particularly, we will use logistic regression as our prediction method.

Discussion

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	891.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.361582	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	13.019697	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	22.000000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	35.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

Figure 1. A summary of the dataset with all the attributes mean, range, standard deviation. There are 891 registered passengers in the dataset.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
PassengerId    891 non-null int64
Survived       891 non-null int64
Pclass         891 non-null int64
Name           891 non-null object
Sex            891 non-null object
Age            891 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(5), object(5)
memory usage: 83.6+ KB
```

Figure 2. The number of data entries for each attributes

Based on figure 2, cabin data has significant number of missing values. This cloud lead to very biased prediction. Therefore, we will not take into account cabin data in our final model. Moreover, ‘Ticket’, ‘Name’, and ‘PassengerId’ attributes are not descriptive in survival rate. We will remove them as well.

We first approach the problem by evaluating the correlation of survival rate with each attributes. This way, we can get rid of some attributes that does not show a correlation with survival rate. Therefore, prediction accuracy can be improved. And this will make avoid unnecessary computation.

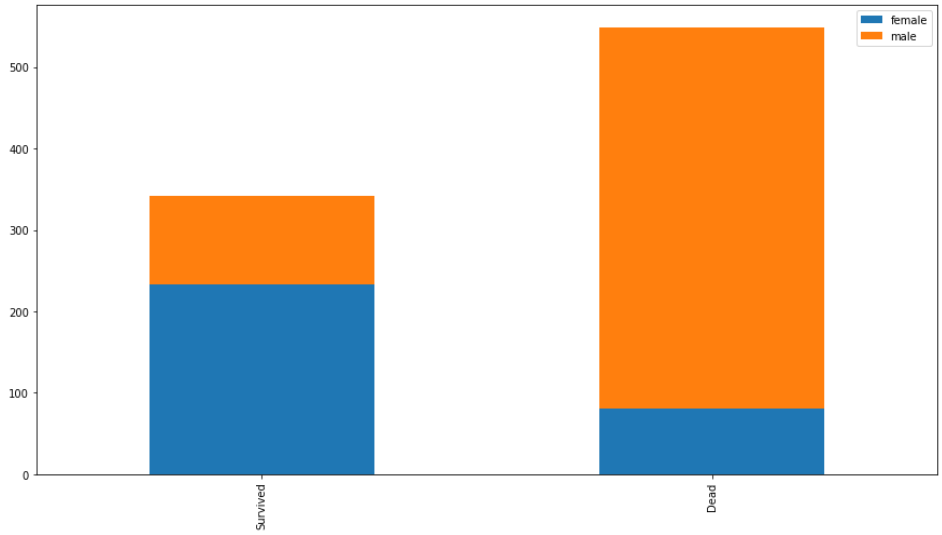


Figure 3. Number of survival and dead in correlation to gender.

The first attribute we analyzed is gender. It can be seen clearly that female has a dominant number in survival, while male dominates the Dead number. We will incorporate gender attribute in our final model because it shows a strong correlation that suggests females are much more likely to survive.

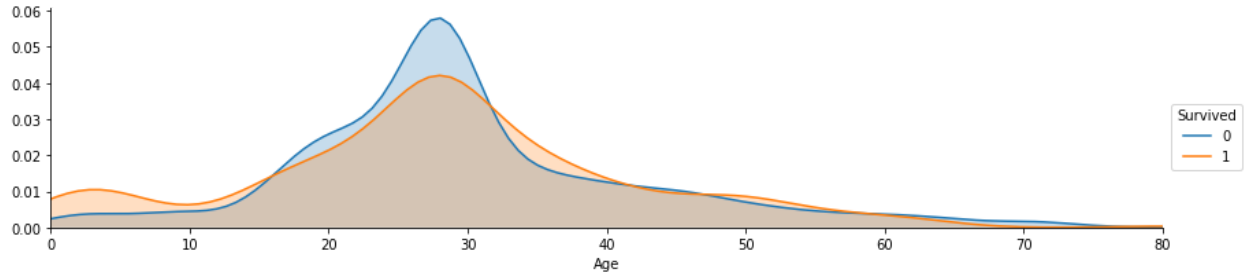


Figure 4. Survival rate in correlation to age.

The second attribute we check is age. Based on figure 3, There is a correlation between age and survival. This is indicated by the crosses of the areas of survived distribution and dead distribution. It can be seen that passengers are more likely to not survive around age 30, and youngers (especially age less than 12) are likely to survive more than elderlies.

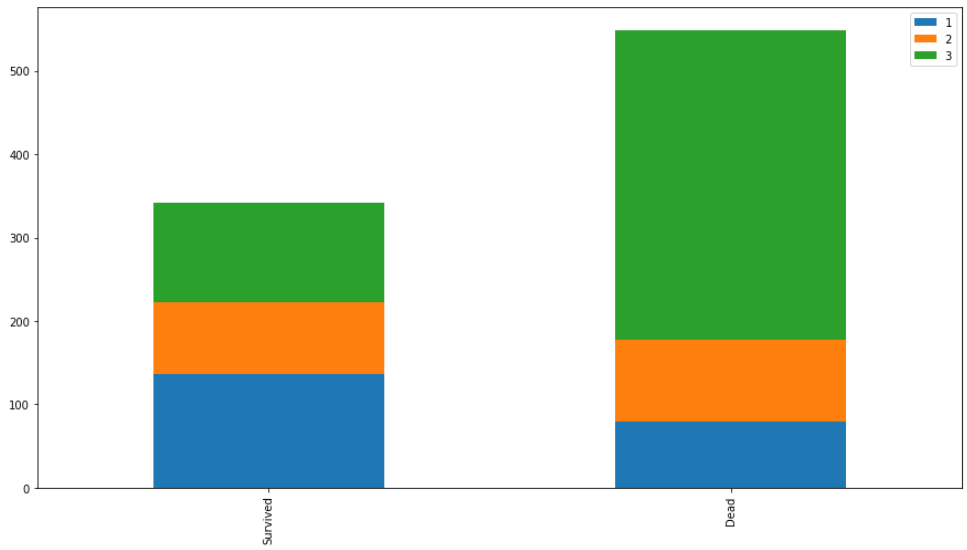


Figure 5. The survival rate by cabin class

Passenger class has a correlation to survival, as we can see that There are proportionally more people survived in class 1 than in in class 3.

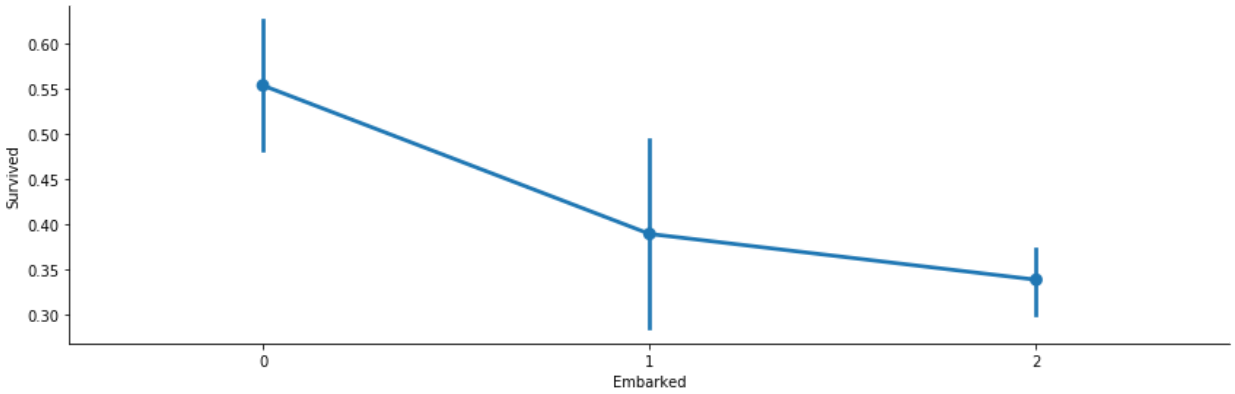


Figure 6. Embarked data in correlation to survival rate in proportion

Clearly, there is a huge distinction of survival proportion in correlation to embarked data. We will keep embarked attribute

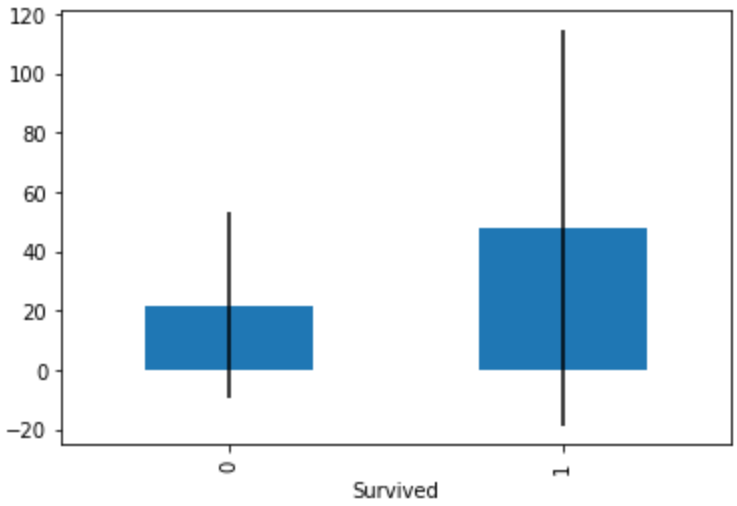


Figure 7. Fare vs. survival

The survivor range and mean ticket fare are higher than those that did not survive. Higher ticket fare means higher chance of survival.

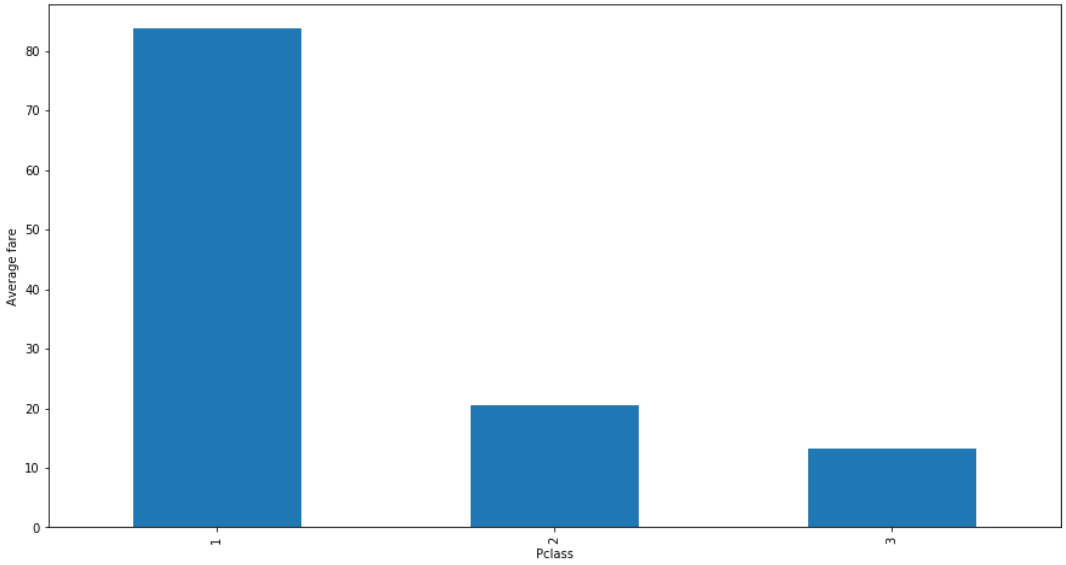


Figure 8, Passenger class vs. average fare, Higher class means higher price

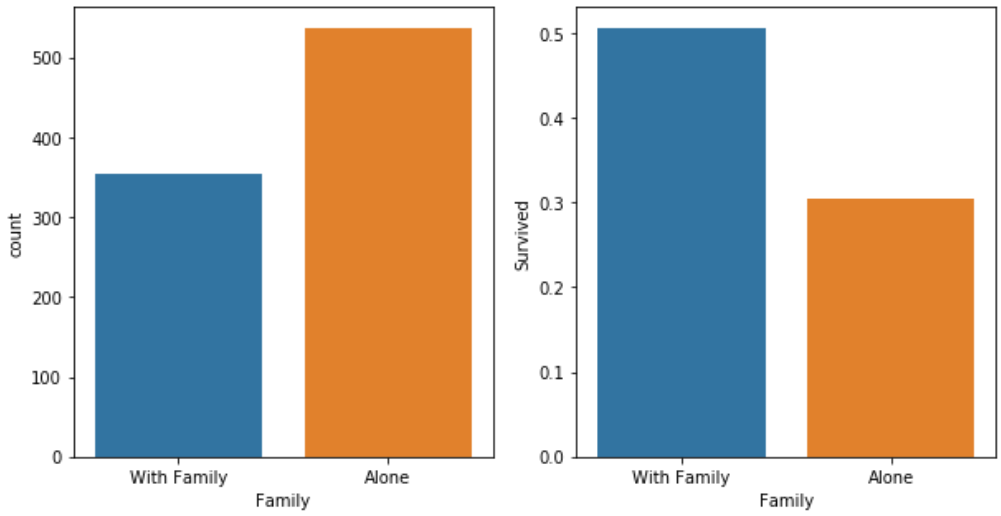


Figure 9. Travel with family vs. travel alone.

We grouped together 2 attributes, travel with siblings and travel with kids as traveling with family. It seems that travel with family have a higher chance of survival.

Conclusion

We used the best model for cross validation logistic regression, incorporating all the attributes we found to have a correlation with survival and made the prediction on the testing data. Our prediction accuracy is 0.80359, and we are ranked as #7018.

	Features	Coefficient Estimate
0	Pclass	-0.820086
1	Sex	-2.489233
2	Age	-0.027817
3	SibSp	-0.294562
4	Parch	-0.081023
5	Fare	0.004568
6	Embarked	-0.139252

Figure 10. The Correlation coefficients of each variable in the logistic regression model.

Written Part:

1. Proof. var [X − Y] = var [X]+var [Y]−2cov[X,Y].

   var [X − Y] = E[X - Y]^2 - [E[X - Y]]^2
   = E[X^2 + Y^2 - 2XY] - [[EX]^2 - 2[EX][EY] + [EY]^2
   = (E[X^2] - [EX]^2) + (E[Y^2] - [EY]^2) - (2E[XY] - [EX][EY])
   = var [X]+var [Y]−2cov[X,Y]

2. (a) according to Bayes Rule:
   P(D|+) = (P(+|D) * P(D))/ P(+)
   = (P(+|D) * P(D))/ (P(+|D) * P(D) + P(+|N) * P(N))
   = (95% * 0.001%) / ((95% * 0.001%) + (5% * 99.999%)
   = 0.019%
   Suppose the test shows that a widget is defective, the chances that it's actually defective given the test result is 0.019%

   (b) P(N|+) * P(+) = (1 - P(D|+)) * P(+)
   = 99.981% * ((95% * 0.001%) + (5% * 99.999%)
   =4.99%

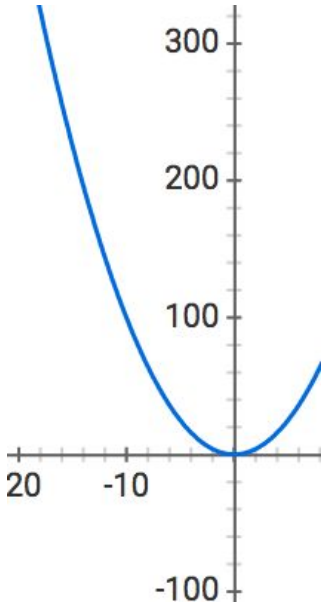
   4.99% * 10000000 = 499000
   Every year, there will be 499000 good widgets thrown out.

   P(D|-) = (P(-|D) * P(D))/ P(-)
   = (P(-|D) * P(D)) / (P(+|D) * P(D) + P(+|N) * P(N))
   = (5% * 0.001%) / ((5% * 0.001%) + (95% * 99.999%)
   = 0.0000526%

   0.0000526% * 10000000 = 6.26
   Every year, there will be on average 6.26 bad widgets sent to customer.

3. (a) When k is equal to n, there would be an equal number of each class as a neighbor, and the prediction error would be 50%. The prediction error would then go down exponentially as k approaches 1. And since the testing point is also in the training data, k=1 means it's pointing to itself, and the prediction error would be 0.

   (b) It will have a parabolic like shape. Where the prediction error goes down exponentially as k goes from n to a small number (depending on the data, usually 3-5). At this point the prediction error will reach an optimum. As k continues to get closer to 1, the error goes up a little again as the model now becomes very sensitive to noise.



(c) Depends on the data set. Usually K ranges from 5 to 10 are yield pretty good results. If K is too large, we increase the computational requirements linearly in proportion to k. If k is too small, we may not get a representative sample of data for each fold, which could possibly lead to biased and inaccurate results.

(d) A modification could be that to set the weights of the k nearest neighbors to be inversely proportional to the distance to the testing point. So the ones that are nearer have a higher weight in votes.

(e) Knn is computationally expensive, it is very slow for huge data set. Also, for high dimensional space the distance to all neighbors increases, which makes the closest neighbors less likely to be in the same class. In higher dimensions the data points move outward towards the edges of the feature space, and are closer to the outer boundary than to their neighbors. This makes classification harder as neighbors will all lie in the same general direction, opposite the boundary.

Citations:

http://www.numpy.org/
https://www.scipy.org/
http://scikit-learn.org/stable/index.html
http://matplotlib.org/
http://seaborn.pydata.org/
http://ggplot2.org/
https://pypi.python.org/pypi/feather-format/

https://stackoverflow.com/questions/5323818/condensed-matrix-function-to-find-pairs/14839010#14839010
https://stackoverflow.com/questions/13079563/how-does-condensed-distance-matrix-work-pdist
https://medium.com/dataholiks-distillery/l2-distance-matrix-vectorization-trick-26aa3247ac6c

https://stackoverflow.com/questions/27433798/how-to-change-y-axis-range-to-percent-from-number-in-barplot-with-r
https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf
https://stackoverflow.com/questions/34226400/find-the-k-smallest-values-of-a-numpy-array
https://stackoverflow.com/questions/28339746/equal-error-rate-in-python#42761279
http://ahmedbesbes.com/how-to-score-08134-in-titanic-kaggle-challenge.html