# Uber vs Lyft Preference Predictor

**Sarah Wessel, Viraj Patil**

## Abstract

The aim of our project is to create a classification algorithm which can predict whether Uber or Lyft will be more affordable for a given source, destination, car type, hour of day and day of the week. In the following pages, we describe our dataset, data cleaning process, and proposed classification techniques for accomplishing our goal.

## Introduction

As ridesharing is becoming more common, people find themselves struggling to choose between the popular options. The question that will come to most people's mind is whether to use Uber or Lyft. There are so many factors that can affect the price including time of the ride request, usage by others in the area, weather, etc. While the companies' price algorithms are hidden from the consumer, we can use past data that includes relevant factors to create a predictive model. The model described below will help by determining which rideshare company is more affordable ahead of time. It will be trained on data of past Uber and Lyft rides in Boston and will yield a prediction of whether Uber or Lyft will be cheaper. Consumers can use this tool to predict whether Uber or Lyft will be more affordable when planning future rides, and rideshare drivers can use this information to decide which service to work for on a given day, at a given location.

## Experimental Results

### Data

We started with the following dataset from Kaggle to train our model. It includes 650,000 rows of combined Uber/Lyft rides in Boston.

Data Source: https://www.kaggle.com/ravi72munde/uber-lyft-cab-prices

One issue with this dataset was that it only describes individual rides with either Uber/Lyft. There is no comparison between Lyft and Uber ride prices, unless we assume that each rider has considered both companies and chosen to ride with the less expensive one. Since this is an unsafe assumption, we reformatted the data to compare Uber and Lyft prices for common inputs.

### Data Processing:

Code: https://colab.research.google.com/drive/17ZfOnSHJZw54113XGQbg0XLDU5_cHL9P

As a part of this cleaning process, we needed to decide on a set of common attributes to use to group data points. After some deliberation, we settled on using source, destination, day of week, hour of day, and type of car as the common attributes because they each contain information that could affect the price of the ride. We then used Python to convert the data to the new format (see above link). We calculated the day of week and hour of day using the existing timestamp in our data, and calculated type of car by mapping the existing name attribute onto a set of five common car types (shared, standard, lux, xl, and black_xl). Once these values were calculated, we binned the data by these “common attributes”, removed bins which did not have both Uber and Lyft data, and then determined whether Uber or Lyft would be preferred for each remaining bin (by comparing average price). Each “bin” was then stored as a row in our new dataset. After this work was done, the dataset includes the following attributes:

| Name | #Variants | Examples | Conversion |
|---|---|---|---|
| day_of_week | 7 | 0, 3, 6 | N/A |
| hour | 24 | 1, 9, 17 | N/A |
| source | 12 | ‘Fenway’, ‘Back Bay’ | ‘Fenway’ -> 1 |
| destination | 12 | ‘Fenway’, ‘Back Bay’ | ‘Fenway’ -> 1 |
| type | 5 | ‘shared’, ‘standard’ | ‘Shared’ -> 1 |
| avg_distance | c (Continuous ) (0.3 - 5.7 miles) | 1.25, 3.16 | N/A |
| preference | 2 | ‘Uber’, ‘Lyft’ | Uber -> 0, Lyft -> 1 |

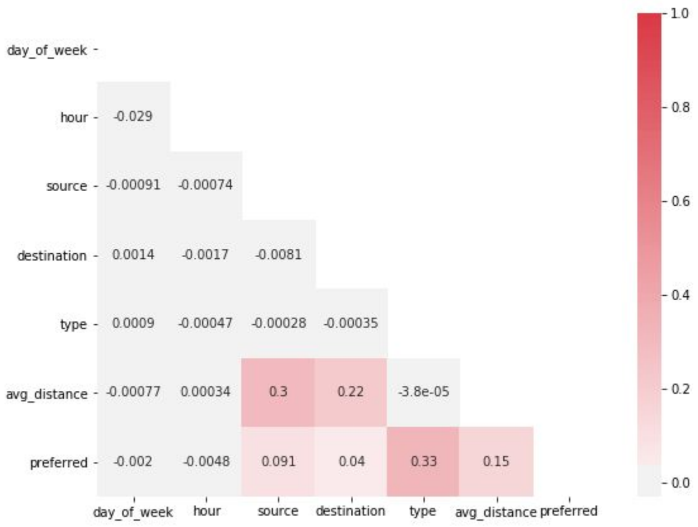
A feature vector for our models can contain the above parameters. There is one final step remaining which is to change discrete strings to numeric formats. Values such as day_of_week, hour, and distance are numerical and do not require any further manipulation. Other values are discrete, so we mapped them to a unique numerical value before inputting into the model. Altogether, considering all the possible values for each discrete feature, there can be many unique feature vectors:

$$7 * 24 * 12 * 12 * 5 * c * 2 = 241{,}920c$$

The data source has a limited scope of rideshare trips by providing data in between 12 major locations in Boston. As a result, it is safe to say that the number of variants for avg_distance will be around 12 * 11 = 132 distances. The original dataset is large with 650,000 entries and even after binning the data, we still have 56,000 rows in our data set (*bit.ly/uber-lyft-preference-data*). Given all these possibilities, we decided to cut down on training and testing data during development, utilizing smaller random bins of data for reasonable training times.

## Data Analysis:

To better understand our data, we analyzed correlations between variables, the results of which are shown in the heatmap below. Small correlations can be seen between the type of car and preferred ride share service as well as the trip distance and preferred ride share service. There also seems to be slight correlations between source, destination, and distance of trip, which intuitively makes sense. It should also be noted that the limited correlations between our features and target variable may limit the predictability of our target.



## Models

Since we care about grouping user inputs into either Uber or Lyft classes, our models will utilize classification methods. We chose to experiment with the following models:

1. Logistic Regression
2. Naive Bayes - our data sanitization step described above will remove dependent columns such as distance (which is related to source and destination). This will maintain the I.I.D assumption that the model is built off of.
3. Decision Trees
4. K-Nearest Neighbors
5. Support Vector Machines
6. Neural Networks

We chose to implement methods 1 - 5 using the sklearn library. Each model had the same steps: training hyperparameters, training the model, making predictions, and testing accuracy. Given these similarities, we decided to abstract the shared logic into a higher level class called SklearnModel. Specific model classes will extend and configure SklearnModel with the correct parameters. Here is a more in depth explanation of the model's design and implementation. Methods are listed in the order they are invoked.

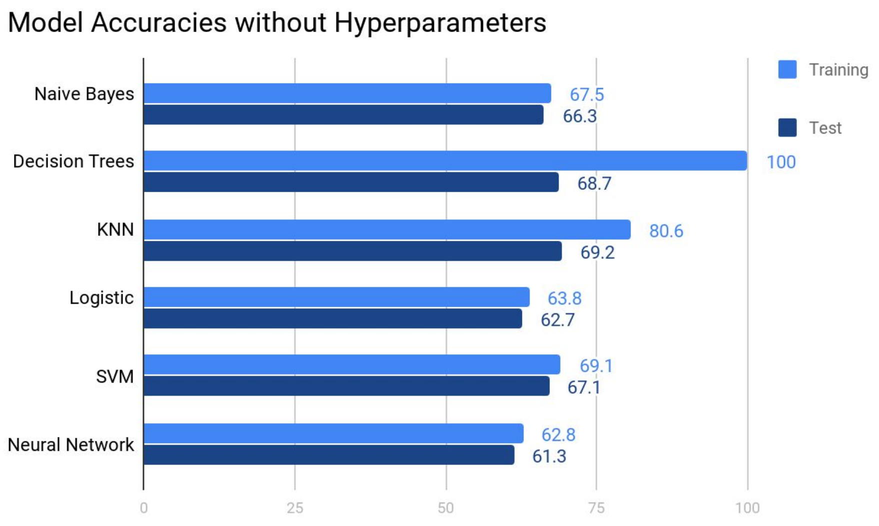
**SklearnModel**:

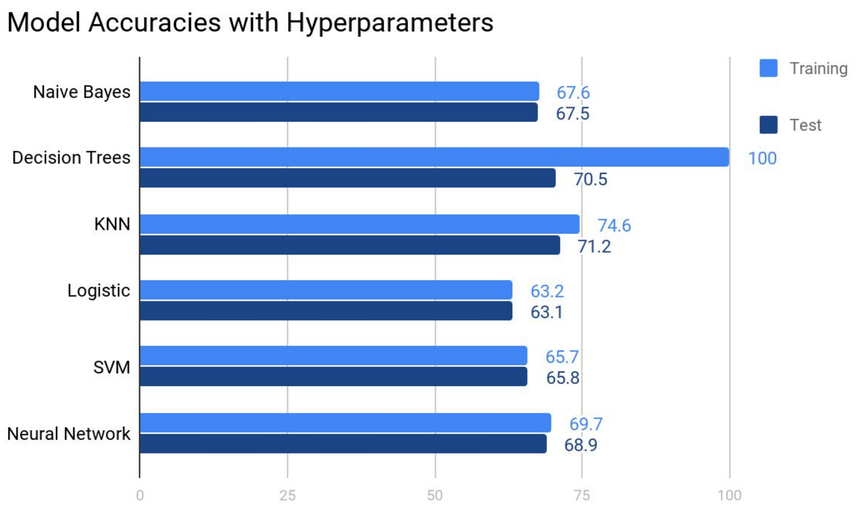
- constructor(classifier, parameters)
    - Creates a SklearnModel
    - Takes in two inputs. The first one is a class such as KNeighborsClassifier, DecisionTreeClassifier, etc. The second specifies relevant hyperparameters in a format that is understood by sklearn.
    - Example parameter input for KNN: {'n_neighbors': [2, 3, 5, 7, 10, 20, 50, 75, 100]}
- tune_hyperparameters(x_trn, y_trn)
    - Utilizes the GridSearchCV module to train hyperparameters based on the training data.
- train(x_trn, y_trn, hyperparameters)
    - Trains the model based on the provided training data and hyperparameters
- predict(x_vals)
    - Predicts the classes (Uber or Lyft) for the provided data.
- accuracy(x_vals, y_vals)
    - Provides the classification error of the trained model

Given this setup, we were able to efficiently create models for Logistic Regression, Naive Bayes, Decision Trees, KNN, and SVM. We tuned the values for each of the models and got the following results.

## Results

Overall, the results showed lower classification error for the training data for all the models that we trained. This is expected as there will always be a little overfitting to training data. In the case of the Decision Tree, the model had a 100% accuracy for the training data which makes sense as the leaf size is set 1 by default in the sklearn library. Other models see around 1-4% increase in accuracy for training data which implies a relatively low level of overfitting. The best accuracy for testing data is achieved by K-nearest neighbours (KNN) with a 71.2% accuracy.





As expected, there are some small but clear improvements for each of the models when hyperparameters are included. The downside here is that the training time significantly increased. For example, our initial implementation of SVM had a training runtime of O(N^4) and introducing hyperparameters increased the complexity to the point where the program would not finish execution. To address this issue, we swapped in the LinearSVC library which ran successfully.

Finally, for the Neural Network implementation, we decided to use our own programs from the last assignment and catered it towards this application. One additional component required was hyperparameter training. In order to do this, we ran the NN with varying learning rates, number of hidden layers, iteration counts, nodes per layer, and activation functions per layer. Doing this over the entire input would take far too long, so we used a condensed subset of 5000 rows. We set the hyperparameters to whichever combination of the above variables let to the highest accuracy in the condensed training data. This training is very computationally intensive and took over 3 hours for just 2  hidden layer combinations! However, even with just 2 hidden layers we see a ~5% improvement from our initial values.

**Conclusion**

Given the limited correlations between the features and target, we were pleased with the performance of our models. In the future, it would be interesting to see whether the use of other features like season, month, or weather could improve performance. It would also be interesting to explore more configurations neural network to see if we can further improve performance.

## Participant Contributions

Sarah: Worked to synthesize data from kaggle by writing code to parse and bin the dataset. She also researched relevant sklearn libraries and created several model implementations.

Viraj: Worked on standardizing the data and converting it into a format that is usable by the ML models. He also built the architecture for the prediction program and contributed more models.

Both discussed approaches for each step of the project before implementing anything and worked on the writeups together. Both worked on Neural Network implementation.