New York City Taxi Tip Predictions

Zach Price & Aditi Kharkwal
Department of Data Science
University of Washington
Seattle, Washington 98105
`zprice12@uw.edu & akhark@uw.edu`

Abstract

Every month, millions of taxi rides are responsible for transporting people throughout New York City. Tips account for a significant portion of each taxi driver's revenue. Our goal was to create an effective model that can act as a tool for NYC yellow taxi drivers looking to maximize their tip amounts. If the model can isolate certain significant factors for increasing tip amounts, NYC yellow taxi drivers will be able to incorporate information on these factors in their routes to boost tip revenue. The project also allowed for an evaluation of a variety of machine learning algorithms to determine which possess the highest predictive power in this situation. Using yellow taxi data from July 2021 in New York City, we predicted the tip amounts of yellow taxi rides from July 2022 in New York City through various techniques, including multiple linear regression, ridge regression, gradient boosted tree, K-nearest neighbors, and random forest. These models were compared with their mean-squared error of predictions of July 2022 tips. We found that the gradient boosted tree produced the lowest mean-squared error of 3.263.

1 Data

1.1 Data Source

The data was pulled directly from New York City's Taxi & Limousine Commission government website, which can be found at Taxi Data. Two parquet files were used from this website: "Yellow Taxi Trip Records" from July 2021 and "Yellow Taxi Trip Records" from July 2022. Each entry represents a single taxi ride with a variety of information on the trip. Prior to cleaning, each data set contained information on approximately 2.8 million rides.

1.2 Data Cleaning

Overall, the raw data was generally clean, requiring only some minor data cleaning before it could be run through our models. First, we dropped columns from the two data sets that we didn't wish to use in our models, such as 'vendorID', 'storeandfwdflag', and ' totalamount'. 'vendorID' was dropped because it's simply an ID for each driver. 'storeandfwdflag' was dropped because we could not find information on what this binary variable represented. 'totalamount' was dropped because it's not possible to calculate 'totalamount' without also knowing the tip amount.

After some brief exploration of the data, we also realized that a variety of variables contained values that were impossible for what they represented. Thus, we filtered the data to drop rows that had 'passengercount', 'tripdistance', or 'fareamount' less than zero. Inherently, the measurements of passengers, trip distances, and fare amounts can't be negative values. We also removed rows with NA's and any duplicate rows.

In addition, certain rows were removed based on the lack of prevalence in the data sets. Rides with 'passengercount' greater than 7 were removed because there were only 157 of these in the data. Furthermore, rows with 'RatecodeID' equal to 99 were removed because there were 63 of these in the data. After all the cleaning was completed, the data for July 2021 and July 2022 each contained roughly 2.2 million rows of taxi ride information.

1.3 Feature Engineering

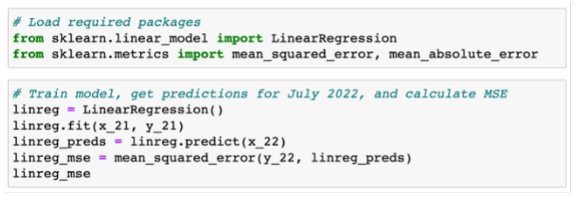
To allow our models to be more accurate, we had to utilize features in the data set to create other helpful features and adjust existing features. First, we used 'tpeppickupdatetime' and 'tpepdropoffdatetime' to create a 'duration' variable. These were datetime type variables; thus, we simply needed to subtract the pick-up time from the drop-off time and convert this into seconds. Additionally, we used 'tpeppickupdatetime' and 'tpepdropoffdatetime' to create 'starthour' and 'endhour' for each ride. We simply determined the start hour of each ride by its pick-up time and the end hour by its drop-off time.

Lastly, we had to encode the categorical variables because some of the algorithms do not have built-in handling for categorical variables. We decided to target encode 'paymenttype' and 'RatecodeID' with the mean of each level's tip amount. For example, payment type had three possible values, 1, 2, and 3; we found the mean tip amount for each of these groups and mapped each value to its mean tip amount.

In the end, our features included 'passengercount', 'tripdistance', 'RatecodeID', 'PULocationID', 'DOLocationID', 'paymenttype', 'fareamount', 'extra', 'mtatax', 'tollsamount', 'improvementsurcharge', 'congestionsurcharge', 'airportfee', 'duration', 'starthour', and 'endhour'. The features were saved in the data frames 'x21' and 'x22' for July 2021 and July 2022, respectively. The response was 'tipamount', saved in 'y21' and 'y22'.

2 Results

2.1 Multiple Linear Regression



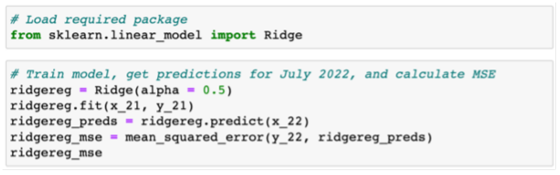
```
# Load required packages
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error
```

```
# Train model, get predictions for July 2022, and calculate MSE
linreg = LinearRegression()
linreg.fit(x_21, y_21)
linreg_preds = linreg.predict(x_22)
linreg_mse = mean_squared_error(y_22, linreg_preds)
linreg_mse
```

Figure 1: Multiple linear regression code

We started with one of the simpler modeling techniques, multiple linear regression. The code for this model is shown above in Figure 1. No model selection was done, we simply ran the multiple linear regression on all the features with the tip amount as the response. We did this intentionally for better comparison with the ridge regression model. The mean-squared error produced by the multiple linear regression model was 4.396.

2.2 Ridge Regression



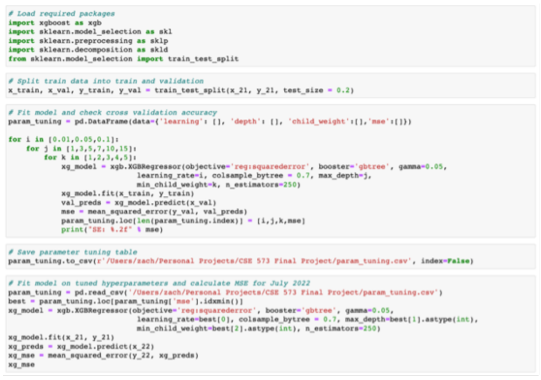
```
# Load required package
from sklearn.linear_model import Ridge
```

```
# Train model, get predictions for July 2022, and calculate MSE
ridgereg = Ridge(alpha = 0.5)
ridgereg.fit(x_21, y_21)
ridgereg_preds = ridgereg.predict(x_22)
ridgereg_mse = mean_squared_error(y_22, ridgereg_preds)
ridgereg_mse
```

Figure 2: Ridge regression code

The second model we utilized was ridge regression, shown above in Figure 2. Ridge regression adds a penalty to the weight of parameter coefficients to the cost function; thus, it forces the coefficients of unimportant features to go toward zero. We tried a variety of alpha values, ultimately settling on 0.5 because they all produced similar results. The mean-squared error for the ridge regression model was the same as the multiple linear regression, 4.396 (slightly different past 4 decimal places). This implies that all features were generally important in predicting the tip amount because the ridge regression model would have produced a different mean-squared error than the multiple linear regression by reducing the effect of unimportant variables.

2.3 Gradient Boosted Tree



```python
# Load required packages
import xgboost as xgb
import sklearn.model_selection as skl
import sklearn.preprocessing as sklp
import sklearn.decomposition as skld
from sklearn.model_selection import train_test_split
```

```python
# Split train data into train and validation
x_train, x_val, y_train, y_val = train_test_split(x_21, y_21, test_size = 0.2)
```

```python
# Fit model and check cross validation accuracy
param_tuning = pd.DataFrame(data={'learning': [], 'depth': [], 'child_weight':[],'mse':[]})

for i in [0.01,0.05,0.1]:
    for j in [1,3,5,7,10,15]:
        for k in [1,2,3,4,5]:
            xg_model = xgb.XGBRegressor(objective='reg:squarederror', booster='gbtree', gamma=0.05,
                                         learning_rate=i, colsample_bytree = 0.7, max_depth=j,
                                         min_child_weight=k, n_estimators=250)
            xg_model.fit(x_train, y_train)
            val_preds = xg_model.predict(x_val)
            mse = mean_squared_error(y_val, val_preds)
            param_tuning.loc[len(param_tuning.index)] = [i,j,k,mse]
            print("SE: %.2f" % mse)
```

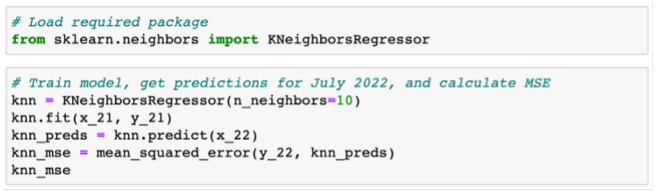
```python
# Save parameter tuning table
param_tuning.to_csv(r'/Users/zach/Personal Projects/CSE 573 Final Project/param_tuning.csv', index=False)
```

```python
# Fit model on tuned hyperparameters and calculate MSE for July 2022
param_tuning = pd.read_csv('/Users/zach/Personal Projects/CSE 573 Final Project/param_tuning.csv')
best = param_tuning.loc[param_tuning['mse'].idxmin()]
xg_model = xgb.XGBRegressor(objective='reg:squarederror', booster='gbtree', gamma=0.05,
                             learning_rate=best[0], colsample_bytree = 0.7, max_depth=best[1].astype(int),
                             min_child_weight=best[2].astype(int), n_estimators=250)
xg_model.fit(x_21, y_21)
xg_preds = xg_model.predict(x_22)
xg_mse = mean_squared_error(y_22, xg_preds)
xg_mse
```

Figure 3: Gradient boosted tree code

We spent the most time creating our gradient boosted tree model, depicted above in Figure 3. This involved creating validation sets from the training data to calculate MSE of various iterations of the XGBoost model as we hyperparameter tuned for learning rate, max depth, and minimum child weight. Then, we found the minimum MSE, which was 2.603, of the validation sets and used the hyperparameters, learning rate of 0.05, max depth of 10, and minimum child weight of 4, from this to create the final XGBoost model. This final model produced a testing MSE of 3.263.

2.4 K-Nearest Neighbors



```python
# Load required package
from sklearn.neighbors import KNeighborsRegressor
```

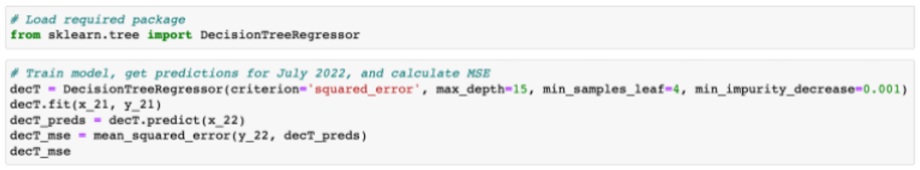
```python
# Train model, get predictions for July 2022, and calculate MSE
knn = KNeighborsRegressor(n_neighbors=10)
knn.fit(x_21, y_21)
knn_preds = knn.predict(x_22)
knn_mse = mean_squared_error(y_22, knn_preds)
knn_mse
```

Figure 4: K-nearest neighbors code

The K-nearest neighbors model, shown in Figure 4, took a long time to run but produced the worst mean-squared error out of all our models. Since we utilized many features and had 2.2 million rows of data, the prediction step for the KNN model took a long time. For each row, the model compared that point's location in an extremely high dimensional space to millions of other points, placing much computational stress on our machine. Thus, we couldn't tune for the number of neighbors and somewhat arbitrarily chose 10. The MSE for the KNN model was 6.217.

2.5 Decision Tree

In addition to the gradient boosted tree, we also ran a normal regression decision tree, displayed in Figure 5 below, with similar parameters to those used in the XGBoost final model. We wanted to see if the gradient boosted tree would perform significantly better than a simple decision tree. This

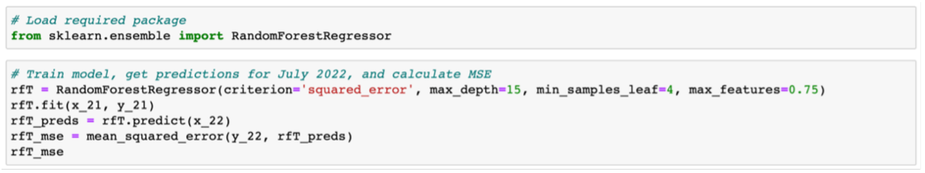

```
# Load required package
from sklearn.tree import DecisionTreeRegressor

# Train model, get predictions for July 2022, and calculate MSE
decT = DecisionTreeRegressor(criterion='squared_error', max_depth=15, min_samples_leaf=4, min_impurity_decrease=0.001)
decT.fit(x_21, y_21)
decT_preds = decT.predict(x_22)
decT_mse = mean_squared_error(y_22, decT_preds)
decT_mse
```

Figure 5: Decision tree code

turned out to not be the case as the decision tree produced a mean-squared error of 3.302, which was a very similar value to the error of the XGBoost model.

2.6 Random Forest


```
# Load required package
from sklearn.ensemble import RandomForestRegressor

# Train model, get predictions for July 2022, and calculate MSE
rfT = RandomForestRegressor(criterion='squared_error', max_depth=15, min_samples_leaf=4, max_features=0.75)
rfT.fit(x_21, y_21)
rfT_preds = rfT.predict(x_22)
rfT_mse = mean_squared_error(y_22, rfT_preds)
rfT_mse
```

Figure 6: Random forest code

We wanted to compare the gradient boosted tree model to another type of ensemble regression decision tree; thus, we ran a random forest model with the code depicted in Figure 6. The random forest model was created with similar parameters to those used in the XGBoost final model. This model produced a mean-squared error of 3.567, which was clearly worse than the error of the gradient boosted tree model and surprisingly worse than that of the regression decision tree.

3 Conclusion

3.1 Findings

Figure 7 in the appendix displays the mean-squared error of each of the six models when using July 2021 New York City yellow taxi data to predict the tip amounts for July 2022 New York City yellow taxi rides. We can see that the K-nearest neighbors model generated by far the worst MSE at 6.217. The multiple linear regression and ridge regression models resulted in the same MSE of 4.396. Finally, the tree algorithms produced the three lowest errors, with the gradient boosted tree model performing the best out of all of them with a MSE of 3.263. This means that on average, the predicted tip amounts using the XGBoost model were $1.81 away from the actual tip amounts for yellow taxi drivers in New York City in July 2022. It's unsurprising that the XGBoost model produced the best results as it's considered to be a powerful algorithm, and it was the only one that we truly hyperparameter tuned for. It was surprising how well the normal regression decision tree performed in comparison to the XGBoost model. Considering the average tip amount was $2.36, we were hoping for better accuracy in our models than this. However, a few significantly high tip amounts might have inflated the errors as our models struggled to generalize to these cases.

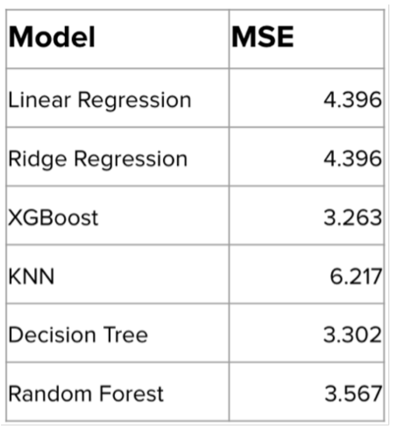
3.2 Further Research

There are many areas of potential further research stemming from this project. First, we would like to scale the features and response to potentially reduce computational time and complexity. This might also help control for the overall rise in tip amounts in July 2022 compared to July 2021. In addition, we could try utilizing principal component analysis to reduce dimensionality and potentially lower model errors. Furthermore, we would like to hyperparameter tune for all models, rather than just XGBoost, and add some model-selection. We might also remove rows with tip amounts greater than a certain threshold because these are rare and difficult for our model to generalize to, which inflates the mean-squared error. Lastly, we could consider binning the tip amount to turn it into a categorical response that we could run classification algorithms on.

4 Broader Impact

The broader impact of this project as of now is minimal due to the sizable errors in the predictions of our models. However, with future improvement, this model could be a very useful tool for helping New York City taxi drivers adjust their routes to maximize tip revenue. Hopefully, this will allow some taxi drivers to make more money with their time. If this tool were to become heavily utilized, there could be issues with taxi drivers targeting certain regions or times to provide their services because that is when the highest tips are available. This might make it difficult for residents of New York City to find rides during times or areas outside of these optimal factor levels.

5 Appendix



Model	MSE
Linear Regression	4.396
Ridge Regression	4.396
XGBoost	3.263
KNN	6.217
Decision Tree	3.302
Random Forest	3.567

Figure 7: Mean-squared error of models

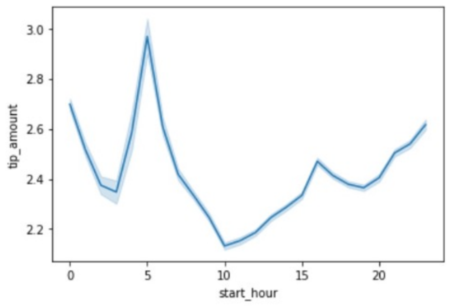


Figure 8: Time series of start hour vs tip amount for July 2021

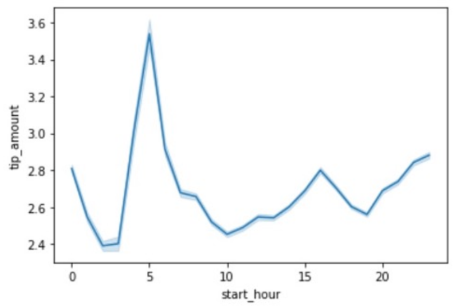


Figure 9: Time series of start hour vs tip amount for July 2022

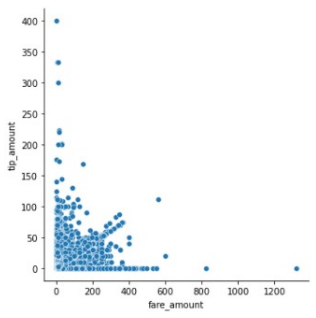


Figure 10: Scatter plot of fare amount vs tip amount

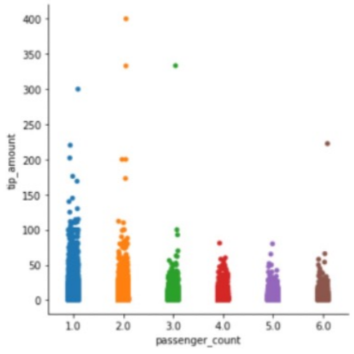


Figure 11: Scatter plot of passenger count vs tip amount

6 Code

The code used to clean the data and create the models can be found at Taxi Tip Prediction Repository. Unfortunately, the parquet files that hold the yellow taxi data for July 2021 and July 2022 exceeded GitHub's limit of 25mb; thus, the best way to access the raw data is through Taxi Data.

7 References

Arif, A. (2023, February 14). *What is ridge regression in machine learning*. Dataaspirant. Retrieved March 16, 2023, from https://dataaspirant.com/ridge-regression/

Badole, M. (2023, March 2). *Mastering multiple linear regression: A comprehensive guide*. Analytics Vidhya. Retrieved March 16, 2023, from https://www.analyticsvidhya.com/blog/2021/05/multiple-linear-regression-using-python-and-scikit-learn/

Harode, R., & Malik, S. (2020, April 23). *XGBoost: A deep dive into boosting*. Medium. Retrieved March 16, 2023, from https://medium.com/sfu-cspmp/xgboost-a-deep-dive-into-boosting-f06c9c41349

Sham, K. (2020, June 15). *Linear regression in python with scikit-learn*. Medium. Retrieved March 16, 2023, from https://medium.com/analytics-vidhya/linear-regression-in-python-with-scikit-learn-e1bb8a059cd2

Singh, A. (2023, March 13). *KNN algorithm: Guide to using K-nearest neighbor for regression*. Analytics Vidhya. Retrieved March 16, 2023, from https://www.analyticsvidhya.com/blog/2018/08/k-nearest-neighbor-introduction-regression-python/

Slatery, K. (2020, March 14). *Decision trees: Understanding the basis of Ensemble Methods*. Medium. Retrieved March 16, 2023, from https://towardsdatascience.com/decision-trees-understanding-the-basis-of-ensemble-methods-e075d5bfa704