MSML 651 Final Project Report

Prashant Swarnapuri (pswarnap@umd.edu)

New York City Taxi Fare Prediction

**Abstract**

**This project will attempt to predict the fare of a New York City taxi ride, when the ride is booked. Taxi fares are typically based on mileage, travel time, and in certain cases the location of the pickup or drop-off location. Prediction of travel time is often the trickiest aspect of fare estimation in taxi rides. While this project will not attempt to predict travel time directly, it is reasonable to assume that any predicted fare is at least in part a function of travel time. This project attempts to provide a much better estimation of fares by include additional features from external datasets. While distance is the most obvious aspect of fares, weather is another critical external factor which can affect the travel time and thus the fares. A weather data set was sourced from the NOAA which contains details about precipitation including rain, snow, and ice which can affect the travel time between locations. Additional context to can be added to trips, by identifying pickup and drop-off locations which are likely to induce special fare rates such as airports.**

**Motivation**

Taxi cabs are an important means of transportation between the various boroughs of New York. High-volume ride sharing providers like Uber and Lyft do provide trip fare predictions ahead of time. However, their pricing is highly volatile based on supply and demand of drivers. The New York Metro subway system is often a cheap means of travelling within the city, however it is likely to take longer, involving one or more line switches to get to the destination. Having a means to predict the fare of regular cab trip ahead of time, will allow for easy decision making for the consumer between taking a cab, using a ride sharing service or taking the subway.

**Datasets**

The data being used for this project is hosted on Kaggle made available by Google [1]. The data set consists of taxi trips in New York City for a period of between 2009 to 2015. This data set contains 55 million rows of training data, which is expected to be challenging in the absence of a big data cluster for computation.

The columns included in this data set are:

* **pickup\_datetime** - value indicating when the taxi ride started.
* **pickup\_longitude** - longitude coordinate of where the taxi ride started.
* **pickup\_latitude** - latitude coordinate of where the taxi ride started.
* **dropoff\_longitude** - longitude coordinate of where the taxi ride ended.
* **dropoff\_latitude** - latitude coordinate of where the taxi ride ended.
* **passenger\_count** - the number of passengers in the taxi ride.

The target column in the **fare\_amount** column which denotes the fare paid at the end of the trip in US dollars.

One of the major contributors of delays on the road is the weather conditions. Precipitation such as rain or snow will often affect the flow of traffic contributing to increase in travel time and thus fare price. Weather data from stations is available via Mesonet for several different locations in New York City [2]. The weather data from the following stations were be considered:

1. Station ID NYC at Central Park
2. Station ID LGA at LaGuardia International Airport

Station ID JRB at Wall Street, only went operational in 2016 and hence does not overlap with the duration of the source dataset. Precipitation Accumulation in 1 hour intervals, Snow Depth and Ice Accretion are expected to be the features of interest for this project. The file downloaded from ASOS had weather conditions for both LGA and NYC stations from a period of 2009 to 2016. There are various weather elements included in these files. However, our fields of interest are:

* **p01i** – precipitation accumulation in the last 1 hour in inches
* **ice\_accretion\_1hr** – ice accretion in the last 1 hour in inches
* **snowdepth** – current snow depth in inches

**Coding Environment**

The coding environment for this project was the Databricks cloud environment Runtime 9.1 [3]. The stack installed by default with this environment is Spark 3.1.2 and Scala 2.12. The PySpark interface was used to interact with the Spark platform using the Python programming language [4]. The Spark MLLib library was used for machine learning modules. The MLFlow module was used to track the various runs of the machine learning model. Where possible the random seed has been set to 42, to ensure reproducibility of results.

Considering the size of the data, all initial testing and cross validation of the models was done using a randomly sampled data set comprising of 1% of the total dataset i.e. 550k data records. The datasets are read from the Databricks File System, which were uploaded ahead of time from the local computer. It consists of the training data file (train.csv.zip) from the original Kaggle dataset and weather data from ASOS for the city of New York (asos.csv).

Visualization were plotted using Pandas and Seaborn libraries by converting the Spark data frames to Pandas data frames [5][6].

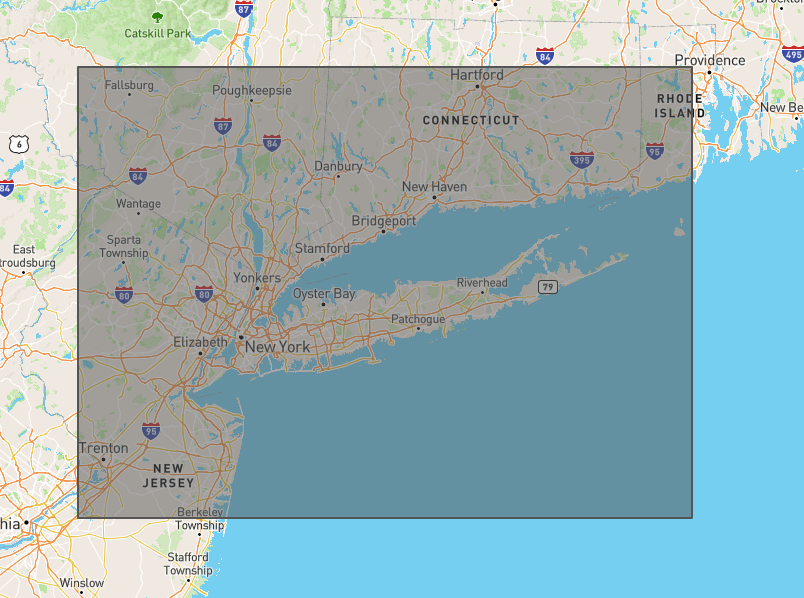
**Data Exploration and Cleansing**

The range of coordinates for each of the trips were checked to see for bad values. For the state of the New York, the longitude values should typically be negative and latitude values should be positive.

As we can see from Table I, there are at least a few bad values which do not follow this convention. The large positive and negative values are likely because of GPS error or difference in coordinate projection system. The outlier values will be removed from the dataset as it is not feasible to fix these data points without more information. In addition, a rough boundary denoting the New York City area will be drawn and all records with pickup or drop-off coordinates outside this region will be removed, as they are either likely to be bad coordinates or outliers. This bounding box is visualized in Fig. 1.

TABLE I. STATISTICS OF COORDINATE VALUES IN THE DATASET

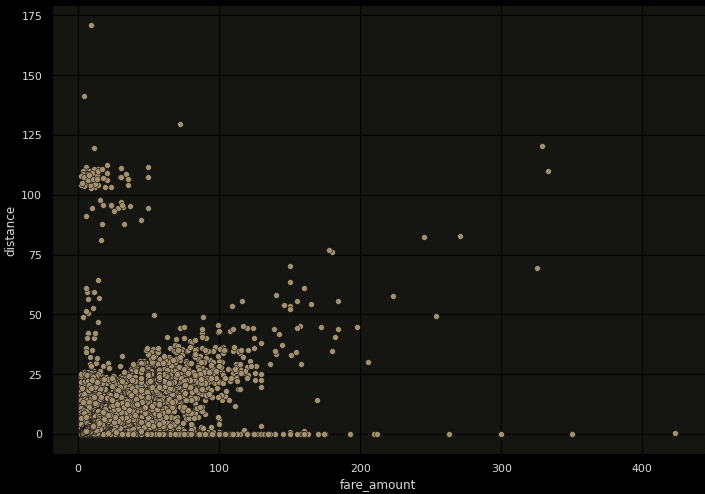
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **Pick Lat** | **Pick Lon** | **Drop Lat** | **Drop Lon** |
| **mean** | 39.92 | -72.51 | 39.92 | -72.51 |
| **stddev** | 9.64 | 12.85 | 9.63 | 12.78 |
| **min** | -3492.26 | -3442.06 | -3547.89 | -3442.02 |
| **max** | 3408.79 | 3457.63 | 3537.13 | 3457.62 |



**Fig 1. Map showing the extent of coordinates being considered for this analysis (created using geojson.io)**

The minimum fare amount for taxi ride is $2.50 [7]. All records with fares below this amount or above $500 will be filtered out from the dataset. In addition, records with passenger counts above 6 will also not be considered.

Looking at the distribution of fares and comparing it to the distance of the trip we can see a clear linear trend between the two variables as seen in Fig. 2 below. It is clear to see the distance does factor quite a bit into the fare amount, as is expected.



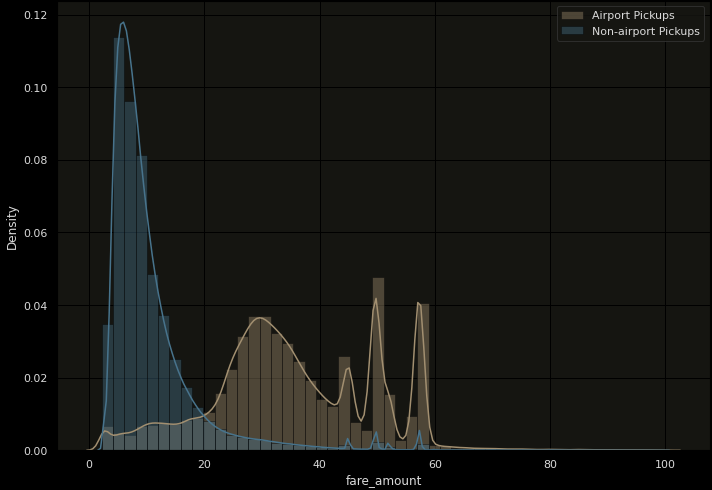
**Fig 2. Scatterplot showing the relationship between distance and fare\_amount**

When the fare distribution is plotted against the year of the trip, another clear trend emerges. The averages fares go up with time from $10 in the year 2009 to $13 in 2015 as seen in Fig. 3 below. The higher fares in the year 2008 can actually be attribute to a small number of trips in the UTC year 2009, bleeding into the year 2008 in Eastern time. This data point can be ignored.



**Fig. 3. Line graph showing the trend in average fares over the years**

Some special trips such as trips to and from airports are likely to have special fares which cannot be accounted for with a simple distance heuristic. Using the coordinates of major airports, the trips which have pickup or drop-off coordinates in the vicinity of the airport can be identified. Plotting out the distribution by separating the airport pickups vs non-airport pickups, it can be observed that airport pickups tend to have higher fares that non-airport pickups as seen in Fig. 4 below. In addition, there seems to be some special fixed fares rules at around the $50 and $60 mark.



**Fig. 4. Comparative fare distributions of airport pickups vs non-airport pickups**

**Feature Engineering**

The distance of trip is likely to be the best indicator of the fare. The easiest way to calculate the distance is the consider the pickup and drop-off coordinates and calculate the greatest circle distance between the two points. Ideally, if there was more detailed trips information, in terms of GPS pings along the routes, it would have been trivial to calculate the exact distance covered by the trip. In lieu of that, the greatest circle distance is decent proxy within the extents of the city of New York.

The greatest circle (or haversine) distance formula is given below [8]:



where, δ1 = end longitude, φ1 =start longitude, δ2 = end latitude, φ2 = start longitude, r = radius of sphere

The radius of the Earth is assumed to be 6371 kilometers. The output of this function is the circle distance of the trip in kilometers. (column: “distance”)

The time of the actual trip is crucial to understand some of the trends in the fare pricing. The pickup timestamp from the source data was shifted from UTC to New York local time for accuracy (column: “localtime”). From the local timestamp, the following values were generated which were considered critical to fare estimate –

* Pickup Year – The year of the trip in YYYY format. Fares tend to increase in the later years due to cost of living increases. (column: “pickup\_year”)
* Pickup Month – The month of the trip in MM format. Fares can be seasonal in nature. Certain months are not as busy as other (column: “pickup\_month”)
* Pickup Day of Week – The day of week of the trip where 0 indicates Sunday and 6 indicates Saturday. This feature was included as Weekends and weekday fares may vary. (column: “pickup\_dow”)
* Pickup Hour of Day – The hour of day when the trip started ranging from 0 to 23. Day and night fares might vary, and peak hours might have higher fares. (column: “pickup\_hour”)

Two further binary features were generated from these timestamp specific variables:

* Night Time – Indicating if trip started after 10PM in the night or before 6AM in the morning local time i.e. “pickup\_hour” greater than 22 or less than 6. (columns: “is\_nighttime”)
* Peak Time – Indicating if this was weekday trip between 6AM to 10AM or 4PM to 8PM i.e. “pickup\_dow” not in (0,6) and “pickup\_hour” between 6 to 10 or between 16 to 20. (columns: “is\_peaktime”)

For the weather dataset, the data was filtered down to just the precipitation, snow, and ice columns as they are most likely to affect conditions on the road. The actual values of measurement differ in measurement methodologies, sensor models as well as calibration errors [9]. In order to avoid biases due to these issues, the precipitation columns were transformed into simple one hot encoded flags indicating if precipitation of any kind was present. The columns “is\_precip”, “is\_snow”, and “is\_ice”, we generated from “p10i”, “snowdepth”, and “ice\_accretion\_1h” respectively. For the purposes of this project each weather station has its own set of features. The initial dataset was then broken into two sets using the station ID key (LGA vs NYC). Each of feature in the dataset was then prefixed with the station ID to uniquely identify the feature. The final list of relevant features was - “nyc\_is\_precip”, “nyc\_is\_snow”, “nyc\_is\_ice”, “lga\_is\_precip”, “lga\_is\_snow”, and “lga\_is\_ice”.

Airport trips were considered as trips of special significance as they are likely to affect the fares as seen in our exploratory data analysis. There are three airports in the vicinity of the New York City – LaGuardia (LGA), John F. Kennedy (JFK), and Newark (EWR). Using the coordinates of all 3 airports, all trips in the dataset were flagged if the pickup or drop-off was within kilometer of any of these airports. The two flags thus generated were “is\_pickup\_near\_airport” and “is\_dropoff\_near\_airport”.

All the features at this point, are either numeric or Boolean in nature. This enables us to easily package the features in a feature matrix for the processing in Spark MLLib. All the features are combined using a vector assembler, and the column is labelled as “features”.

**Model Training**

As per the dataset documentation a naïve distance based approach yields a RMSE close to ~6-8. Our target is to improve on this baseline score, using advanced feature selection, complimentary datasets, and better regression algorithms.

Initial baselines were developed using a simple Linear Regression model with the following base features – “distance”, "pickup\_latitude", "pickup\_longitude", "dropoff\_latitude" and "dropoff\_longitude". The baseline model was single run with all of the records, with default parameters for the Linear Regression model. The baseline linear regression model returned a root mean square value (RMSE) of 5.45.

In addition to the Linear Regression model, the Random Forest and the Gradient Boosted Trees model were developed to improve on the performance of the baseline model. The base RMSE for the Random Forest model is 4.94 and for the Gradient Boosted Model is 4.52. The additional engineered features were added to the model at this point as described in the feature engineering section of this document.

In order to tune the hyper-parameters for various models accurately, for the final run, each model was set up a parameter grid map with a range of plausible values. This parameter grid was then passed into a cross validation module with 4 folds. The parameters and the ranges of values tested are listed below. This cross validation was done using only 1% of the records due to time and performance constraints using the entire dataset. In-depth description of each of these parameters can be found in the PySpark documentation [10] [11] [12]

. TABLE II. ARRAY OF HYPER-PARAMETERS TESTED BY MODEL

|  |  |  |
| --- | --- | --- |
| **Model** | **Parameter** | **Values** |
| Linear Regression | regParam | [0, 0.15,0.3, 0.5] |
| elasticNetParam | [0.0,0.3,0.5,0.8] |
| Random Forest | maxDepth | [3, 5, 10] |
| numTrees | [5, 10 , 20] |
| Gradient Boosted | maxDepth | [3, 5, 10] |
| maxBins | [8, 16, 32] |

For recording the results for each run of the experiment the MLFlow module was used to capture the metrics [13]. The hyper-parameters for the model with the best metric was logged using the MLFlow API. Databricks provides an interface to view the MLFlow results, after each model fit call is executed. The results of the cross validation runs are listed in Table II below.

TABLE III. CROSS VALIDATION AND HYPER-PARAMETER OPTIMIZATION RESULTS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Model** | **Parameter** | **Best Value** | **CV Avg RMSE** | **Test RMSE** |
| Linear Regression | regParam | 0.3 | 5.30 | 5.23 |
| elasticNetParam | 0 |
| Random Forest | maxDepth | 10 | 4.33 | 4.26 |
| numTrees | 20 |
| Gradient Boosted | maxDepth | 10 | 4.35 | 4.27 |
| maxBins | 32 |

A few observations that were noteworthy during parameter optimization -

1. The RMSE of the linear regression model was not very sensitive to the changes in the hyper- parameters. The average RMSE for was consistently in 5.1-5.3 range.
2. For the random forest model, higher depth and number of trees resulted in better performance. The RMSE improved from 5.8 to 4.3 with increase in both parameters.
3. The GBT model improved with increase in maxBins. The maxDepth parameter did not cause huge changes in the performance of the model. However, coincidentally the best RMSE was the highest tested value (10).

Using the best hyper-parameter values from the cross validation runs, the final model was trained using the whole data set, for each of these three methods. The final RMSE values using the held out test set for each of the three models are shown below.

TABLE IV. PERFORMANCE OF FINAL MODELS WITH FULL DATASET WITH BEST HYPER-PARAMETERS

|  |  |
| --- | --- |
| **Model** | **Final RMSE** |
| Linear Regression (regParam=0.3, elasticNetParam=0) | 5.31 |
| Random Forest (maxDepth=10, numTrees=20) | 4.31 |
| Gradient Boosted Trees (maxDepth=5, maxBins=32) | 4.06 |

**Conclusions and Future Work**

In this project, the addition of weather data and some domain specific feature engineering allowed us to improve the performance of the baseline models. Using the Gradient Boosted model, we were able to achieve a RMSE of $4.08, about 1.92 better than the baseline defined by the dataset.

In terms of feature engineering, this work can further be improved upon by using a routing API such as Google Maps to get a better sense of actual asphalt distance between two points.

Regarding model training, due to performance and time constraints, the ability to perform hyper parameter optimization was limited. It is likely that the Random Forest and Gradient Boosted models, could show better results with higher hyper-parameter values. The lack of a core XGBoost and LightGBM implementation in the PySpark environment made it not tenable to test out these much advanced algorithms within this project.

**References**

[1] New York City Taxi Fare Prediction Dataset -<https://www.kaggle.com/c/new-york-city-taxi-fare-prediction/data>

[2] Mesonet ASOS API - <https://mesonet.agron.iastate.edu/request/download.phtml?network=NY_ASOS>

[3] Databricks Documentation – https://docs.databricks.com/applications/machine-learning/index.html

[4] PySpark API Reference - https://spark.apache.org/docs/latest/api/python/

[5] Pandas API Reference - <https://pandas.pydata.org/docs/reference/index.html#api>

[6] Seaborn API Reference - https://seaborn.pydata.org/api.html

[7] NYC Standard Taxi Fares - <https://www1.nyc.gov/site/tlc/passengers/taxi-fare.page>

[8] Greatest Circle Distance - <http://www.movable-type.co.uk/scripts/latlong.html>

[9] Automated Surface Observing System Guide - <https://www.weather.gov/media/asos/aum-toc.pdf>

[10] RandomForestRegressor PySpark Documentation - <https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.regression.RandomForestRegressor.html>

[11] GBTRegressor PySpark Documentation - <https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.regression.GBTRegressor.html#pyspark.ml.regression.GBTRegressor>

[12] LinearRegression PySpark Documentation - <https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.regression.LinearRegression.html#pyspark.ml.regression.LinearRegression>

[13] MLFlow Documentation - <https://www.mlflow.org/docs/latest/index.html>