Analyzing 1 Billion+ NYC Yellow Taxi Rides for Ride Demands Prediction

Ruilin Zhong, Jialei Zheng and Shengjia Zhang
sz2547@columbia.edu, rz2331@columbia.edu, jz2672@columbia.edu

Abstract

Taking taxi has always been a fast, convenient way to travel around the city. Our project analyze over 1 Billion NYC yellow taxi trip data and apply random forest algorithms through MapReduce on PySpark to predict ride demands, fare, and tip hourly in each neighborhood of New York City. Our algorithm also gives relative importance of features on predicted ride demands. Finally, this project develops a web application for taxi drivers to make better pickup decisions.

***Keywords**: Yellow Taxi; Data Analysis; AWS; Random Forest; Big Data; Spark; MLlib; PySpark; MapReduce*

I. Introduction

Taxi has always been an important way to travel in the city. Each year there are over 100 million yellow taxi trip in Manhattan. In recent years however, Uber has taken over lots of business from taxies in NYC. One of the reason is that Uber can better allocate the drivers to serve the area with more customers through Apps on cell phone to shorten waiting time and provide better service. So it would be very useful if taxi driver can know when and which area have the most business and in which neighborhood can he find the customers that are willing to give a large amount of tips. Also, such studies can help the NYC Taxi & Limousine Commission better allocate the taxi resources. A big data solution on solving this problem is necessary for many agents: passengers (who wants to know how, where, and when to catch a cheap cab), drivers (who want to know where to pick up most passengers), CEOs of Uber and Lyft (who want to make their business strategy most profitable), and NYC governments (who want to monitor NYC taxi and car-sharing businesses).

II. Related Works

The New York City Taxi & Limousine Commission has released a staggeringly detailed historical dataset covering over 1.1 billion individual taxi trips in the city from January 2009 through June 2015. In the blog of Todd W. Schneider, New York taxi riding pattern is analyzed. It shows some very interesting pattern for different Taxi(Yellow taxi, Uber and Lyft) and for different area as well as pick up time.[1]

What's more, another blog[2] use regression and random forest machine model to predict the Taxi demands in different during different time and get pretty valuable results.

Those researches shows great value on New York Taxi data research and specifically, taxi riding prediction.

III. System Overview

After training the model, we use AWS cloud service to build a simple web application. This app is aimed to predict the riding number and riding fare when given the taxi pick up time and location.

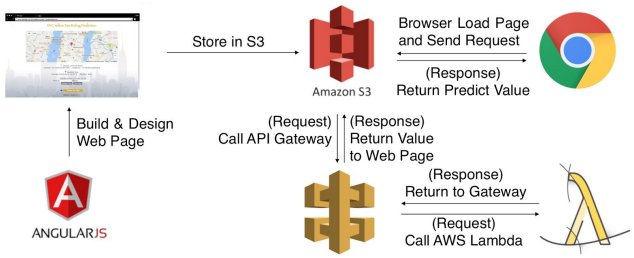


Figure 1

The flowchart shows how this website design and build up. First of all, AngularJS is used for front end design and controlling. Google Map API is used for displaying map on the web page. Open Weather API is used for collecting current weather data like temperature and weather condition. After user select the pick up location in the map, pick up time and click predict button, page request will send to AWS Lambda function through API Gateway. In order to make our website accessible to all users, we put the front end file into S3 buckets, thus anyone can visit this website through visiting the URL of that S3 bucket file.

API Gateway is defined in AWS console or in a swagger file. It's used for getting request from a particular web page, then transmit the request to a certain AWS Lambda function. The http request headers and body templates are

defined in the API Gateway. Thus a well-defined request message will be sent to AWS Lambda function.

AWS Lambda function can be triggered by the request sent from API Gateway and it is used for calculation and taxi ride prediction. After AWS Lambda function gets the pick up information (location, weather and time), well-defined machine learning module (decision tree) is used for calculating the prediction result. Finally, that result will be sent back to the front end, and then users can get the result.

IV. Dataset and its Processing

IV - 1 Data source and description

In this project three datasets are used: TLC yellow taxi trip record dataset, Manhattan weather dataset crawled from http://weathersource.com, and a dataset that contains geographical coordinates to define Manhattan neighborhoods obtained from NYC OpenData.

The historical weather data are fetched from http://weathersource.com/, which provides free historical weather data lookup through HTTP request. User provides location and time information in the HTTP request and will receive the corresponding weather data.

The Manhattan neighborhood geographical coordinates dataset is downloaded with a tab function on NYC OpenData website.

The official TLC trip record dataset contains data for over 1 billion taxi trips from January 2009 through June 2016. Each individual trip record contains precise location coordinates for where the trip started and ended, timestamps for when the trip started and ended, plus a few other variables including fare amount, payment method, and distance traveled. In our processing, we will first throw out unnecessary information like drop-off location & time, travel distance. Then we run the Ray-casting algorithm to turn coordinates of latitude and longitude into our pre-defined neighborhoods polygons of NYC. Ray-casting algorithm can tell whether a coordinate lies in the polygon defined by a series of coordinates. Then for each set of unique feature combination of pickup_location, hour, weather, is_bussiness_day, temperature_category, we calculate the average rides in hour, average trip fare and tips. Finally we combine the data of each month and turn it into the correct format to feed into the model for training.

We use 12 months of raw data of individual yellow taxi rides. Each month of individual rides are contained in a csv file with size between **1.5 - 3G**. We train our predictive model on 12 months (a year) of data from July 2015 to June 2016. The total number of individual rides are more than **100 million**, and the total size of our one-year dataset is about **25G**.

IV-2 Defining variables

We aim to predict three outcome variables that measure the ride demands:

- ***Count*** (numerical): hourly total number of pickup rides in a neighborhood, data fetched and computed from the TLC taxi datasets.
- ***Fare*** (numerical): Expected (average) trip fare (after-tax but before-tip) for each ride, data fetched and computed from the TLC taxi datasets.
- ***Tip*** (numerical): Expected (average) tip fare for each ride, fetched and computed from the TLC taxi datasets.

All the three outcome variables are numerical and continuous.

We predict *Count*, *Fare*, and *Tip* based on 2 continuous (numerical) and 3 categorical features:

- ***Temperature*** (numerical, but could also be categorical if desired): average *daily* temperature in Fahrenheit in Manhattan when the pickup happens. For the convenience of modeling, temperature data is categorized into 10 levels: 0-10F, 11-20F, 21-30F, …, 81-90F. Data is fetched from http://weathersource.com/.
- ***Hour*** (numerical, but could also be categorical if desired): Time when the pickup happens, only keeping the hour part (e.g. 0:23 would be 0; 18:59 would be 18), data fetched from the TLC datasets.
- ***Weather*** (categorical): *daily* weather condition in 3 categories: rain, snow, no_rain_no_snow, with data fetched from http://weathersource.com/
- ***Businessday*** (numerical; but could also be categorical if desired): binary variable that denotes either Federal holidays and weekends, or business days, determined by Python package "holidays"
- ***Location*** (categorical): the neighborhood where the pickup location lies within. In total there are 18 neighborhoods in Manhattan below West 110th Street and East 95th Street): Battery Park City & Lower Manhattan, Chinatown, Clinton, East Village, Gramercy, Hudson Yards & Chelsea & & Flatiron & Union Square, Leno Hill & Roosevelt Island, Lincoln Square, Lower East Side, Midtown

Midtown South, Murray Hill Kips Bay, SoHo & TriBeCa & Civic Center & Little Italy, Stuyvesant Town Cooper Village, Turtle Bay & East Midtown, Upper East Side & Carnegie Hill, Upper West Side, West Village, Yorkville. Location data for each pickup is recorded in the TLC datasets in the format of geographical coordinates (pickup longitude and latitude). Each coordinate is mapped to its corresponding neighborhood with a dataset that contains the definition of each neighborhood as defined by a 2-D polygon with geographical coordinates of all the vertices of the polygon. The 2-D polygon dataset that defines each Manhattan neighborhood is downloaded from NYC OpenData.

From the raw TLC dataset with a year of more than 100 million individual ride information from 2015/07 to 2016/06, we obtain our post-processed data by calculating *Count, Fare,* and *Tip* for each unique combination of {*Temperature*, *Hour*, *Weather*, *Business*, *Location*}. We will discuss how to map geographical coordinates of each pickup to one of the 18 neighborhoods in Section V-2.

V. ALGORITHM

V-1 The ray-casting algorithm

We collect a dataset from NYC OpenData that defines each of the 18 Manhattan neighborhoods by a 2-D polygon on the map, with geographical coordinates (longitude and latitude) for all vertices of each polygon. Then we determine whether a given coordinate (longitude and latitude) of a ride's pickup location lies within or outside each of the 2-D polygon.

To determine the relative position of a given 2-D point with respect to a given 2-D polygon, we use the ray-casting algorithm, with its pseudocode given as follows.

Ray-casting(P, polygon)

Input:

(a) P, a 2-D point defined by coordinates
(b) polygon, a 2-D polygon defined by coordinates of all its vertices

Algorithm:

```
count ← 0
for each side in polygon:
  if ray_intersects_segment(P,side) then
      count ← count + 1
if is_odd(count) then
  return inside
else
  return outside
```

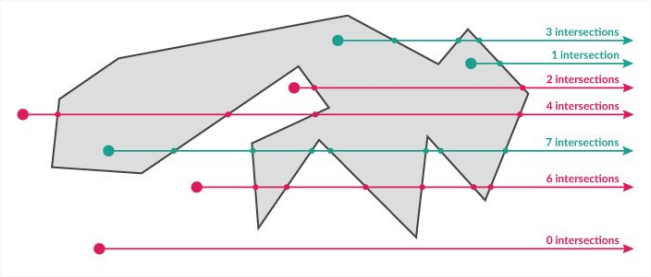


Figure 2

We modify the above algorithm to take into account of the boundary conditions. If a point lies on the edge or vertex of a polygon, we count the point as *within* the polygon.

V-2 Random forest regression algorithm

Given that our data displays non-linear relationship between many feature variables and outcome variables through our exploratory data analysis, and our regression problem involves both numerical and categorical features with relatively high dimensionality, we decide to approach this ride demands prediction problem with Random Forest regression, a "panacea" considered for many data science problems.

Random forest is a type of ensemble learning method, where a group of weak decision tree models combine to form a powerful model. It can handle both classification and regression problems. The advantages of using random forest include:

- It can well handle both categorical and numerical (continuous) features in a single model, which is exactly what we have for our prediction problem.
- It is very powerful to deal with large data set with very high dimensionality, without concern on

overfitting. By limiting the maximum number of tree depth and pruning, one could easily avoid the overfitting problem. The learning procedure would automatically identify important features and discard less important ones, to achieve an effect of dimensionality reduction. As each month of our dataset has a size of 1G +, and our categorical features have relatively large dimensionality, random forest would be a great choice.

- Random forest model can rank the importance of features. This would be a great plus to interpret our models and findings.
- It accurately handles unbalanced data and non-linearity, as displayed by our dataset.
- It is a non-parametric algorithm such that it does not require any assumption about how the data is distributed.
- It retains high level of accuracy even when large chunk of data is missing
- It is a decision tree algorithm, so that it would be easy to interpret the result.

Figure 3

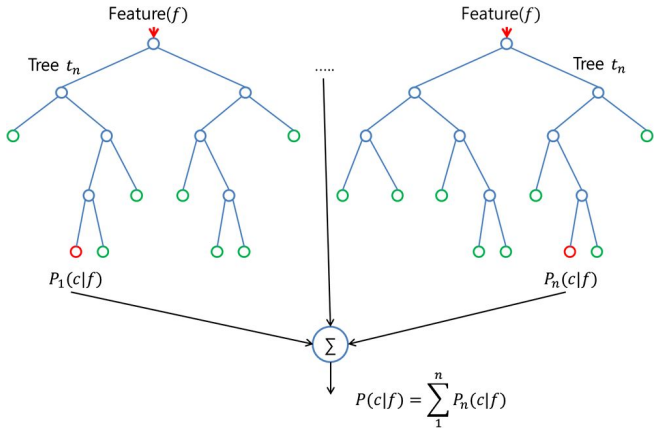


Figure 4

As illustrated in the following figure, the random forest algorithm works as follows.

1. Assume total number of trees is N. Then it samples N cases at random but with replacement. This sample will be the training set for growing the tree.
2. If there are M input variables, a number m<M is specified such that at each tree node, m variables are selected at random out of the M. The best split on these m is used to split the node. The value of m is held constant while we grow the forest.
3. Each tree is grown to the largest extent possible (until reaching the max depth of trees as we set in advance) and there is no pruning. Pruning takes place after we finish growing all the trees.
4. Predict new data by aggregating the predictions of the N trees (i.e., majority votes for classification, and average for regression).

We randomly divide our post-processed dataset as described in Section V-1 into 20-80%. We train the random forest model on the 80% training dataset through k-fold cross validation. Then we report the test error on the 20% test dataset. We fit three different random forest models to predict *Count*, *Fare*, and *Tip*. The metric that measures model performance is root-mean-square error (RMSE) on *Count*, *Fare*, *Tip* respectively, as this is a regression problem and all the three outcome variables are numerical and continuous.

After the random forest regression model is constructed, we tune the model by experimenting with different parameters. Key parameters that may significantly influence the model performance are:

- **Number of decision trees in the random forest *(n_tree):*** as N grows, model has a declining

variance on the test prediction. Usually more trees give better test performance.

- **Maximum depth of each decision tree *(max_deep):*** Deeper trees are almost always better subject to requiring more trees for similar performance, but may raise overfitting problem if there is not sufficiently large number of trees to support a large depth. This is a direct result of the bias-variance tradeoff. Deeper trees reduces the bias; more trees reduces the variance.
- **How many features to test for each split *(mode_feature_strat):*** The more useless features there are, the more features you should try. This needs tuned through cross-validation.
- **Maximum number of bins on bottom level of each tree (*max_bin*)**: Putting a limit on the maximum number of bins (nodes) on bottom level of decision trees would help further avoid overfitting problems.

A detailed guide and formal definition of random forest regression algorithm could be found in Murphy (2012).

VI. Software Package Description

VI - 1. Processing the raw data

There are multiple python scripts used for data fetching and processing. First we need to fetch history weather data through weather source API. Apply for a key at http://weathersource.com/ and put the key into the url in ***getWeather.py***, then sepcify the start and end date of the days you wish to collect. Run `$python getWeather.py,` this will produce ***his_weather.csv*** that contains the data.

Then download NYC yellow taxi trip data from http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml And put these data in the same directory with the .py files. Then edit the ***rmcol_taxi.py***, change the variable "input_file" to name of the taxi data in csv format.

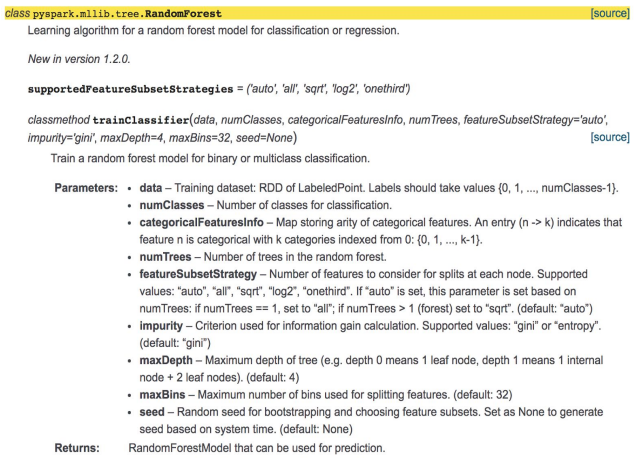
Run `$python rmcol_taxi.py` This will produce a csv file with name start with "rmcoled". This step will remove some unnecessary columns in the original dataset and reduce its size. Then edit the process_taxi.py and change the input file name into the file produced in the last step. Run `$python process_taxi.py,` this will produce a csv file with name started with "processed". This step may take a long time to process since we need to run the ray-casting algorithm to determine in which neighbor the pickup location is. Each ride's coordinate (longitude and latitude) will be tested with respect to each of the 18 2-D polygons that represent the 18 Manhattan neighborhoods. The testing will stop once the algorithm finds a neighborhood (polygon) within which the given pickup point (2D-coordinate) belongs to.

Next, we edit and run ***countTaxi.py***, which will produce a csv file with name start with "count_". This step will create a set of unique feature combination of *Location*, *Hour, Weather*, *Businessday*, *Temperature*, and count the average rides in hour (*Count*), average fare (*Fare*) and tips (*Tip*). If multiple months of taxi data are used, now we have many csv files named "count_...". Put all these file name into the in_file_list in combine.py and use `$python combine.py` to combine these data into one csv files named combined_taxi_tripdata.csv. Then run `$python final_process.py` to produce 3 separate txt file that can be used to train the model.

VI - 2. Implementing and tuning random forest regression with PySpark MLlib

We implement the random forest regression algorithm introduced in Section V-3 in a MapReduce fashion, with Apache PySpark MLlib.

PySpark MLlib has a built-in function that implements random forest regression and train the model through k-fold cross validation.



class `pyspark.mllib.tree.RandomForest` [source]

Learning algorithm for a random forest model for classification or regression.

New in version 1.2.0.

`supportedFeatureSubsetStrategies` = *('auto', 'all', 'sqrt', 'log2', 'onethird')*

classmethod `trainClassifier`(*data*, *numClasses*, *categoricalFeaturesInfo*, *numTrees*, *featureSubsetStrategy='auto'*, *impurity='gini'*, *maxDepth=4*, *maxBins=32*, *seed=None*) [source]

Train a random forest model for binary or multiclass classification.

Parameters:
- **data** – Training dataset: RDD of LabeledPoint. Labels should take values {0, 1, ..., numClasses-1}.
- **numClasses** – Number of classes for classification.
- **categoricalFeaturesInfo** – Map storing arity of categorical features. An entry (n -> k) indicates that feature n is categorical with k categories indexed from 0: {0, 1, ..., k-1}.
- **numTrees** – Number of trees in the random forest.
- **featureSubsetStrategy** – Number of features to consider for splits at each node. Supported values: "auto", "all", "sqrt", "log2", "onethird". If "auto" is set, this parameter is set based on numTrees: if numTrees == 1, set to "all"; if numTrees > 1 (forest) set to "sqrt". (default: "auto")
- **impurity** – Criterion used for information gain calculation. Supported values: "gini" or "entropy". (default: "gini")
- **maxDepth** – Maximum depth of tree (e.g. depth 0 means 1 leaf node, depth 1 means 1 internal node + 2 leaf nodes). (default: 4)
- **maxBins** – Maximum number of bins used for splitting features. (default: 32)
- **seed** – Random seed for bootstrapping and choosing feature subsets. Set as None to generate seed based on system time. (default: None)

Returns: RandomForestModel that can be used for prediction.

Figure5 : PySpark built-in function for random forest regression

To fit the above built-in function "RandomForest.trainRegressor", we first need to convert our dataset into the desired format: RDD of LabeledPoint.

RDD, abbreviated for Resilient Distributed Dataset, is the primary user-facing API in Spark since its inception. At the core, an RDD is an immutable distributed collection of elements of the dataset, partitioned across nodes in the cluster that can be operated in parallel with a low-level API that offers transformations and actions.

Once creating a spark session, we load our dataset and convert it into desired RDD format, and then randomly split the dataset into 80% - 20% as training data versus test data separately. We then call the built-in function "RandomForest.trainRegressor" to train the random forest regression model on the training data, and test it on the test data.

```
model = RandomForest.trainRegressor(trainingData, categoricalFeaturesInfo=cat_var,
                                    numTrees=n_tree, featureSubsetStrategy=mode_feature_strat,
                                    impurity='variance', maxDepth=max_deep, maxBins=max_bin)
```

Figure 6: Calling PySpark MLlib built-in function "RandomForest.trainRegressor" to implement random forest regression

As discussed in Section V-3, we measure the model performance with root-mean-squared-error (RMSE) on each outcome variable. Our function would output the RMSE score on test dataset for *Count, Fare*, and *Tip* respectively.

To tune our model performance, we extensively experiment our model with different combinations of parameters and aim to obtain the best-fit model with the minimum root-mean-squared-error (RMSE) on test dataset. To make our testing process more intuitive and interactive, we create a Jupyter Notebook and activate the Apache Spark MapReduce function on the Jupyter Notebook server. For each outcome variable (*Count, Fare, Tip*), we tune a random forest model separately on the 5 feature variables.

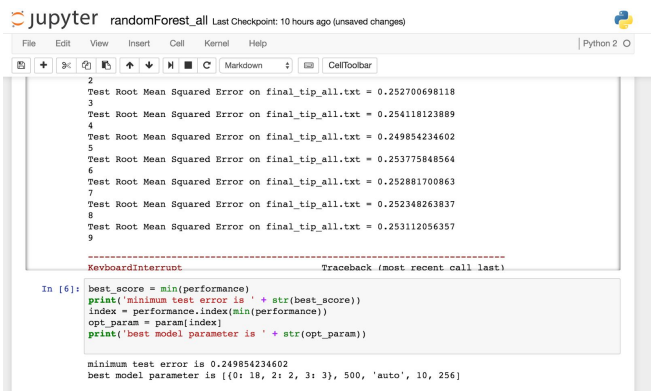


Figure 7: Model tuning interface on Jupyter Notebook

Specifically, we try each unique combination of the following parameters as shown in Figure 8.

```
for mode_feature in ['auto', 'all', 'log2', 'onethird']:
    for max_bin_size in [32, 64, 128, 256]:
        for deep in xrange(4, 15):
            for n in [10, 20, 50, 70, 100, 150, 200, 250, 500, 800, 1000, 1200, 1500, 2000, 2500, 3000]:
                testRMSE, model_param = train_model(data=dataset, cat_var=cat_var_dic, n_tree=n,
                    mode_feature_strat=mode_feature, max_deep=deep, max_bin=max_bin_size, test_portion=0.2)
```

Figure 8: tune random forest model parameters with each combination.

Note that in Figure 8, ***mode_feature*** denotes the features to test for each split, where 'auto' takes squared root of total number of features, 'all' takes all features, 'log2' takes the logarithm of total number of features to the base 2, and 'onethird' takes ⅓ of total number of features. Parameter ***max_bin_size*** limits the maximum number of nodes on the bottom level of each decision tree. Parameter ***n*** denotes number of decision trees in this random forest. Parameter ***deep*** denotes the maximum depth of each tree allowed.

Finally, with the best-performed model for each outcome variable, we let PySpark save the model, so that with any given new datapoint with features, we could use the saved model to predict *Count*, *Fare*, or *Tip*. We then incorporate such predictive model into the backend of our web app.

As for the relative importance of features, PySpark built-in function could not give such an estimates. We hereby implement the same model with Python scikit-learn package that has a built-in function to implement random forest regression with an output of relative feature importance. Though the scikit-learn built-in function does not implement random forest in a MapReduce fashion and hereby requires much longer computation time, it gives us a great overview of which features play the most influential role to determine ride *Count*, *Fare*, and *Tip*.

class sklearn.ensemble. **RandomForestRegressor** (*n_estimators=10, criterion='mse', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_split=1e-07, bootstrap=True, oob_score=False, n_jobs=1, random_state=None, verbose=0, warm_start=False*) [source]

Figure: Python scikit-learn built-in function for implementing random forest regression

VI - 3. Implementing Web App

The front end of the web mainly used AngularJS for developing. Google Map API and Open Weather API are also used for map display and weather prediction.

Index.html file is the front-end display file, and AngularJS controller file is maps.js. They are all stored in AWS S3 Bucket so that it can be easily reached from browser.

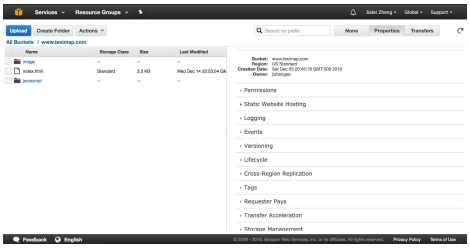


Figure 9: Put frontend files in S3 bucket

3. Swagger File and API Gateway
URL mapping from frontend to back end is in Swagger file Swagger.json. Which defines the rule for sending request from front page to AWS Lambda Function.

Figure 10: Use API Gateway for mapping

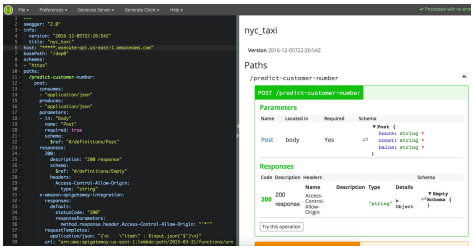


Figure 11: Use swagger file to define API Gateway

4. AWS Lambda Function
AWS Lambda function is used for developing backend in this part. After getting request with other information (like weather, time and location), back end file would predict the riding number around this place in this hour. The result will send back to frontend page.

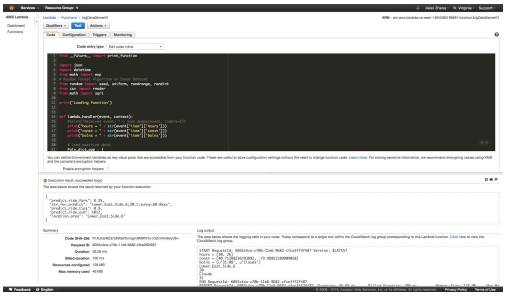


Figure 12: Use Lambda function to develop backend

VII. Experiment Results

VII - 1 Exploratory data analysis

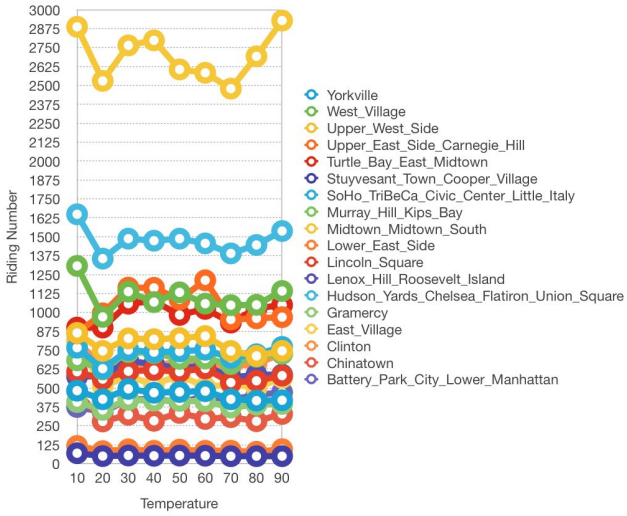


Figure 13

Figure above shows ***Count*** (riding numbers) change when ***Temperature*** change in different ***Location***. We can find the maximum riding occurs in the hottest/coldest weather.

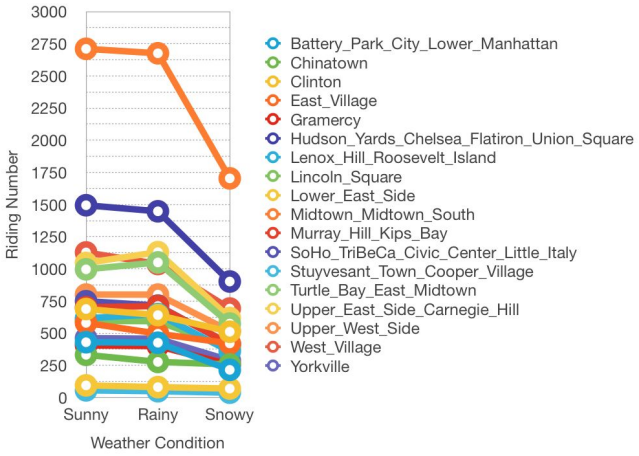


Figure 14

This figure shows riding ***Count*** changes when ***Weather*** condition change. Surprisingly, more riding event happen in sunny day, not in rainy or snowy day as we expected.

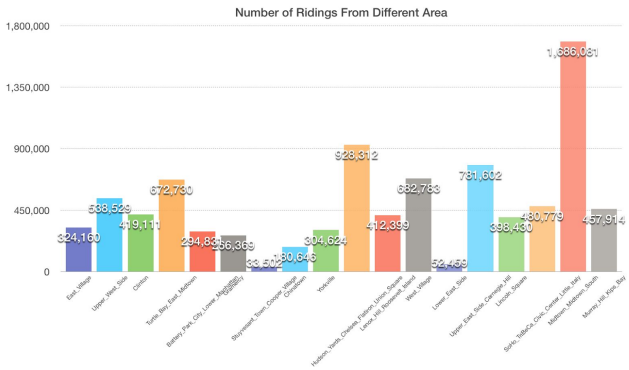


Figure 15

The figure above indicates total riding ***Count*** in different area. It is obvious that Midtown has the maximum taxi riding.

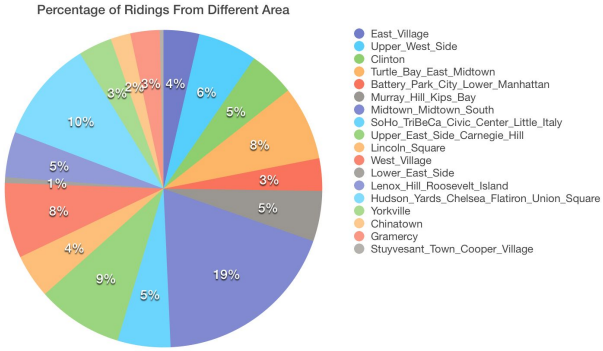


Figure 16

This figure shows the percentage of total riding ***Count*** in different area.

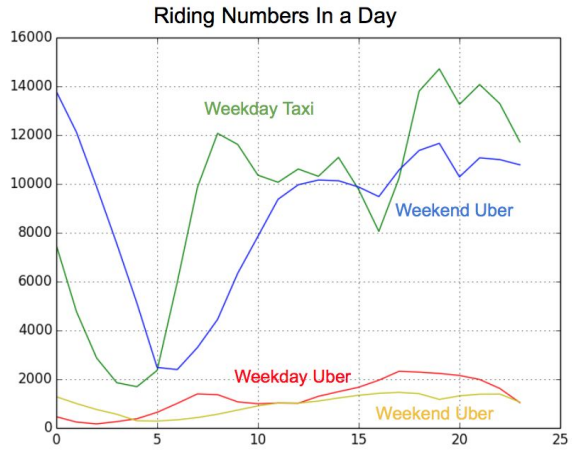


Figure 17

This figure indicates the total riding numbers change in ***Business*** day and non ***Business*** day. The riding event move 2 ~ 3 hours backwards from weekday line to weekend line.

VII - 2 Prediction performance with random forest

We measure the prediction performance on each of the outcome variables with the metric "Root Mean Square Error (RMSE)". RMSE is defined as follows:

$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}}$ where $\hat{y}_i$ denotes the predicted value and y_i denotes the realized outcome value; n denotes the total number of data points.

The best-performed models for each of the three outcome variables are summarized in Table 1.

	Count	***Fare***	***Tip***
RMSE on test dataset	**87**	**$1.04**	**$0.22**
n_trees	1500	500	700
Max_depth _tree	18	6	10
n_features_ per_split	sqrt(N)	N	sqrt(N)

Table 1: RMSE of best models tuned for predicting *Count*, *Fare*, and *Tip,* where N denotes the total number of data points in the training data

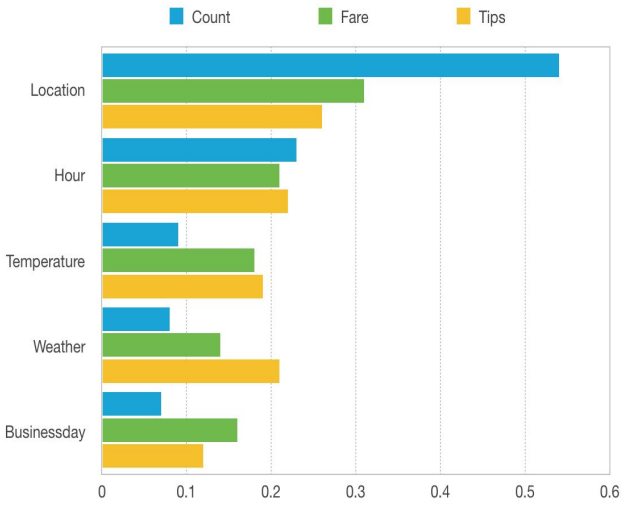


Figure 18: relative importance of features given by random forest regression models

Figure 18 shows how each feature relatively affects ridding *Count*, *Fare* and *Tip*. *Location* and *Hour* are two most important features for predicting all the three outcome variables: *Count, Fare,* and *Tip*. For *Count*, the predictive power of *Location* dominates all other features. As for *Tip*, *Weather* matters more than it is for the other outcome variables.

VII - 3 Web app demo

(http://www.teximap.com.s3-website-us-east-1.amazonaws.com/) Open the URL, you'll get this webpage (shown below). Click the pick up location in the map, and you'll see the longitude and latitude change. Current weather condition can be also seen in this page which is updated automatically. Then you can change the pick up time in the blank, and click "Update Time" button. Or you can just use the current time by clicking "Time Now" button. Finally, click the "Predict Taxi Ride" and you can see the prediction result of taxi riding number as well as riding fare (including tips) at the end of the page.

We would like to express our greatest appreciation to Professor Lin and all the Teaching Assistants for their help and support on this project.

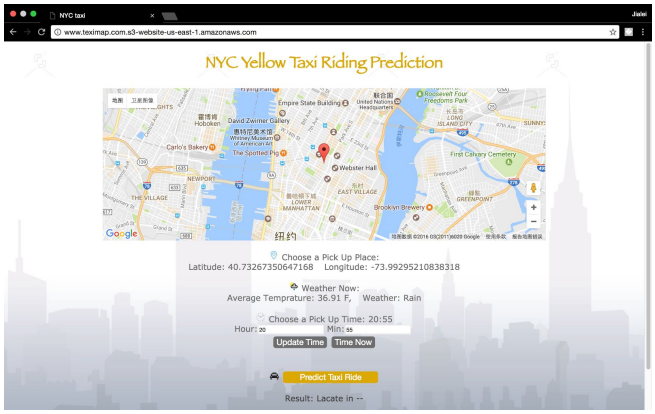


Figure 19: Web Page

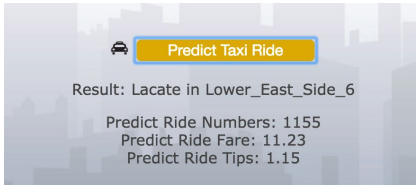


Figure 20: Prediction result

VIII. Conclusion

This project analyzes more than 1 billion raw data on NYC yellow taxi rides with additional weather, temperature, and geographical-coordinates-defined neighborhood information to find out the relationship between Manhattan yellow taxi ride demands and relevant features on location, pickup time, temperature, weather, and businessday_or_holiday. We construct a random forest regression algorithm to predict hourly pickup ride amount, average fare amount, and average tip amount by 5 features: location, pickup time, temperature, weather, and businessday_or_holiday. We implement this random forest machine learning model in a MapReduce fashion with Apache PySpark MLlib and tune the model to achieve an outstanding performance measured by RMSE. We also incorporate our predictive model into the backend of a web app. Through AWS S3 bucket, API Gateway, and AWS lambda function, we deploy a web app to predict the ride amount, fare amount, and tip amount for any selected point on Google Map at any time given the current-day weather and temperature. Our web app is easily accessible on desktop as well as mobile devices. Since our solution package (big data algorithm + web app) could be easily scaled up to solve other big data problems with Uber, Lyft, other car-sharing datasets, and even airport flight demands, transportation demands, it has huge business

value. Our solution is also easily applicable for other cities and countries outside NYC.

ACKNOWLEDGMENT

We would like to express our greatest appreciation to Prof. Ching-Yung Lin and all the Teaching Assistants for their help and support on this project.

REFERENCES

[1] Todd W. Schneider "Analyzing 1.1 Billion NYC Taxi and Uber Trips, with a Vengeance" http://toddwschneider.com/posts/analyzing-1-1-billion-nyc-taxi-and-uber-trips-with-a-vengeance/

[2] Yunrou Gong, Bin Fang, shuo zhang and Jingyu Zhang, Predict New York City Taxi Demand, http://blog.nycdatascience.com/student-works/predict-new-york-city-taxi-demand/