Uber Price Estimator

Web Application

12/23/2016

Advisor: Prof. Sahu

A web application which generates estimated Uber trip price

Chuhan Xiong, Rong Lei, Zijun Chen, Heng Ji

Link to GitHub Code: https://github.com/chuhanxiong/UberPriceEstimator/

YouTube Video Demo Link: https://www.youtube.com/watch?v=jdh0oKmiK7M

Contents

Introduction 2
Backend Work 2
Data Preparation 2
Machine-Learning Linear Regression with Multivariate 4
Build Training Model 4
Uber Price Prediction 5
Deployment 6
EC2 Instance and Libraries Preparation 6
Application Structure 6
Frontend Work 8
Framework & Structure 8
Libraries & APIs 10
Google Map API 10
Autocompletion 11
Responsive Sidebar 12
Openweathermap API 13
Weather Effects Plugin 15
Dynamic Weather Icon 17
Result 18
Without Weather Influence 18
With Weather Influence 19
Conclusion 20
Reference 21

Introduction

Nowadays, as Uber becomes increasingly popular around the world, its expansion brings a huge impact to the taxi industry. More and more people prefer to use Uber instead of a taxi not only because of waiting time, but also the price [1]. Therefore, we want to provide passengers a convenient way to get an estimated cost for an Uber trip. In our project, we built an Angularjs - Python Flask web application that can estimate Uber trip price in New York City based on the trip distance, weather conditions and several other factors. Once customer submits a trip request (origin location, destination, number of passengers) through the front-end, it will send a data package, containing the trip information and current weather index to the backend. The team used Machine Learning linear regression with multivariate to build the training model in the backend to predict the Uber price.

Backend Work

Data Preparation

We have a folder called "taxi_data" that contains all the yellow taxi trip data [2] from July 2015 to June 2016. There are 12 csv files in that folder, and each one of them has trip data within a specific month. For example, "2015-07.csv" contains all the New York City yellow taxi trip data in July 2015.

Since there is no information about the weather in the trip data, we have to obtain this information from other sources [3], and join it with the original yellow trip data. By doing this, we will have information about pickup time, passenger number, trip distance, total amount, and weather condition for each single trip.

We used pandas to process the data files. For taxi data, it contains the following attributes: pickup time, passenger number, trip distance, and total amount. For weather data, we modified the data format in order to match up with taxi data for better

processing. For example, we used integer 1 to represent ‘sunny’ in weather condition, 2 for ‘clouds’ etc. After we gathered all the weather information from the Internet, now we have to join the weather data and the trip data together. We did this in two steps:

First, we used Spark map-reduce to separate our big data set into different parts.

```
taxiFilePPRDD = taxiFileContent.map(lambda line: line.split(','))
.map(lambda cols: (getFileName(cols[0]),(parsePickUpTime(cols[0]), int(cols[1]), float(cols[2]), float(cols[3]))))
.reduceByKey(lambda a, b: appendAB(a,b))
# taxiFilePPRDD.saveAsTextFile("test")
```

Figure 1.1.1: Spark Map-Reduce

Basically, we split one day’s data into 24 pieces, since a day has 24 hours. In other words, each single day’s data will be divided into 24 different csv files, where each file records the trip data within a specific hour in a day. Theoretically, one month’s data will be mapped into 720 parts (24x30) associated with different keys. In our case, the keys are the file names. For example, 2015-07-01-15.csv will record yellow taxi trip information between 15pm and 16pm on 2015-07-01. After this whole map-reduce process, our data set looks like this:

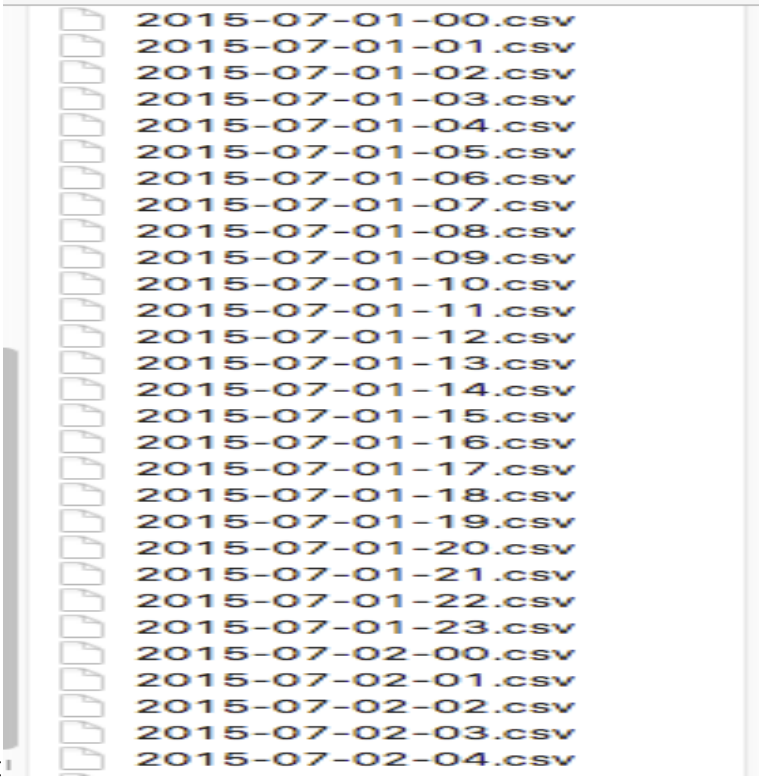


Figure 1.1.2: Raw Data Set

Next, we can join the weather data and the trip data together according to the trip data's file name. For instance, all the weather data that are associated with 14pm-15pm on 2015-07-02 will be added to the file: 2015-07-02-14.csv. All the rows in this file will share the same weather information. Finally, each csv file will look like this:

	tpep_pickup_datetime	passenger_count	trip_distance	total_amount	temp	wind	weather
2	1	1	2.3	14.15	73	5.8	3
3	1	1	10.48	38.34	73	5.8	3
4	1	1	0.69	5.3	73	5.8	3
5	1	1	2.6	14.65	73	5.8	3
6	1	1	0.69	7.25	73	5.8	3

Figure 1.1.3: Final CSV file

Machine-Learning Linear Regression with Multivariate

Build Training Model

The dimension of our training data set is six since each training point has six features: pickup time, passenger number, trip distance, temperature, wind speed, and weather condition. The corresponding training labels set has one column: total amount. We used python's library: pandas to read, write to and from csv files, as well as collect data from different files to build our training data and training labels sets.

```
train_data_df = pd.DataFrame()
train_lable_df = pd.DataFrame()
train_data_df_list = []
train_lable_df_list = []
```

Figure 1.2.1: Build Training Data Set part 1

```
try:
    df = pd.read_csv(store_dir+f,usecols=['tpep_pickup_datetime','passenger_cou
    train_data_df_list.append(df[['tpep_pickup_datetime','passenger_count','trip
    train_lable_df_list.append(df[['total_amount']])
except:
    pass
```

Figure 1.2.2: Build Training Data Set part 2

```
train_data_df = pd.concat(train_data_df_list)
train_lable_df = pd.concat(train_lable_df_list)
print 'finished concatinating'

train_data = train_data_df.as_matrix()
train_labels = train_lable_df.values.flatten()
print 'finished converting to matrix'
```

Figure 1.2.3: Build Training Labels Set

Besides, we also leveraged python's library sklearn to train our model, and saved it as a file ("model.pkl"), which we could load on our EC2 server to do Uber price prediction.

```
clf = linear_model.LinearRegression()
model = clf.fit(train_data, train_labels)
print 'finished training'

joblib.dump(model, 'model.pkl')
print 'finished dumping'
```

Figure 1.2.4: Model Traning

Uber Price Prediction

On our server side (EC2 instance), we just need to provide one API to our front-end – "/getPrice" as following:

```
@app.route('/getPrice', methods = ['POST'])
def getPrice():
    data=json.loads(request.data)
    pickupTime = int(data[1]['timeNo'])
    ppl = int(data[1]['passengerNo'])
    dist = float(data[1]['distanceinmile'])
    temp = float(data[0]['temp'])
    wind = float(data[0]['wind'])
    weather = float(data[0]['weather'])

    return str(calculatePrice(pickupTime, ppl, dist, temp, wind, weather))
```

Figure 1.2.5: API - getPrice

Once we gather all the information we need: pickup time, passenger number, trip distance, temperature, wind speed, as well as weather condition, we will use the training model, which will be imported at the beginning when our server starts, to do the price prediction based on the input (test data) features:

```
def calculatePrice(a_pickupTime, a_ppl, a_dist, a_temp, a_wind, a_weather):
    test_data = [a_pickupTime,a_ppl,a_dist,a_temp,a_wind,a_weather]
    estimated_price = math.ceil(clf.predict(test_data)[0])
    # print 'estimated total_amount: ', estimated_price
    return estimated_price
```

Figure 1.2.6: calculatePrice() Function

Note: "clf" here is the model that we will load from the file "model.pkl" when our server starts.

Deployment

EC2 Instance and Libraries Preparation

We deployed a Python Flask application on AWS EC2. After getting one EC2 instance running on the cloud, we need to ssh to the instance to install all the packages, and python libraries we need: apache2, libapahe2-mod-wsgi, flask, sklearn, numpy, scipy, and scikit-learn.

Application Structure

Now, we need to create a directory for our Flask application. In our case, we created a directory in our home dir to work in, and link to it from the site-root defined in apache's configuration (/var/www/html by default)

ubuntu@ip-172-31-26-182:~$ mkdir uberestimator

ubuntu@ip-172-31-26-182:~$ sudo ln –sT uberestimator /var/www/html/uberestimator

After that, we can upload our local folder to our ec2 server's uberestimator folder. The folder's structure looks like this:

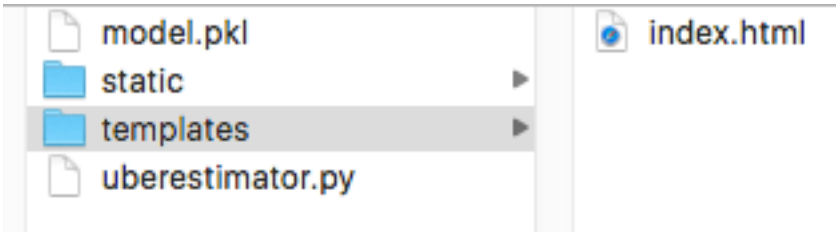


Figure 1.3.1: Application Folder Structure part 1

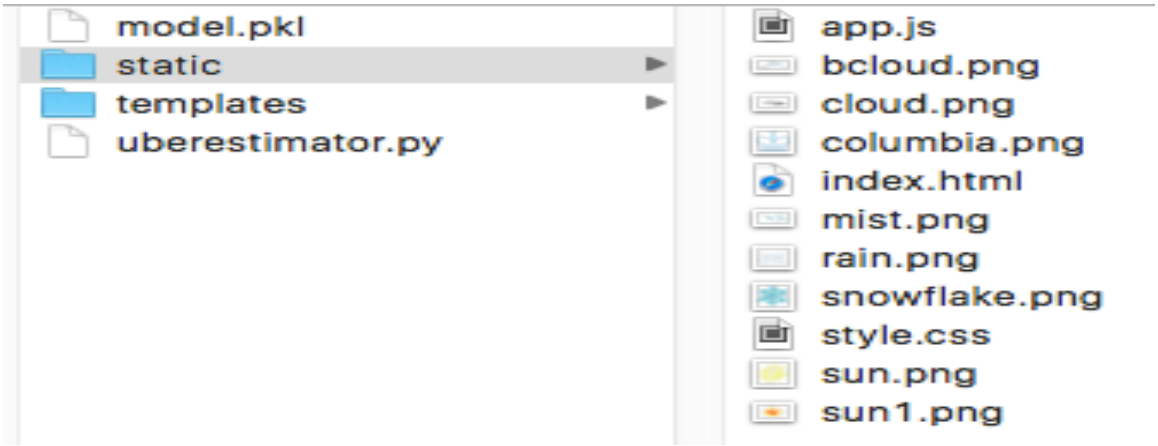


Figure 1.3.2: Application Folder Structure part 2

model.pkl is our training model obtained locally through Spark streaming and python's libraries as we discussed before. The static folder contains all the resources, including Javascript, CSS files and pictures, imported by the front-end. The templates folder contains index.html. The requirements.txt file can be ignored in our case. (It is useful if you want to deploy a Flask application through AWS beanstalk since it will use this file to download all the required libraries for the application)

Next, we need to create an uberestimator.wsgi file to load the app:

```
import sys

sys.path.insert(0, '/var/www/html/uberestimator')

from uberestimator import app as application
```

The last step is to enable mod_wsgi. Since the apache server displays html pages by default, to serve dynamic content from a Flask application, we will have to make some changes in the apache configuration file that is located at /etc/apache2/sites-enabled/000-

default.conf. Essentially, we need to add the following block just after the DocumentRoot /var/www/html line:

```
WSGIDaemonProcess uberestimator

WSGIScriptAlias / /var/www/html/uberestimator/uberestimator.wsgi

        <Directory /var/www/html/uberestimator>

                WSGIProcessGroup uberestimator

                WSGIApplicationGroup %{GLOBAL}

                Order deny,allow

                Allow from all

        </Directory>
```

By doing this, the apache server will display our Flask index page once users load our server's public DNS. Finally, we just need to restart apache server to run our Flask application:

```
sudo /etc/init.d/apache2 restart

sudo apachectl restart
```

Frontend Work

Framework & Structure

We selected Angularjs [4] as our frontend framework. An ng-app "plunker" was created to support the entire structure of html, and two ng-controllers: "WeatherCtrl" and "MainCtrl". Additionally, it provides a service function that can wrap separate useful data from these two controllers together. The service function was defined to deliver variables through different controllers, as we could not do it directly. Thus, the frontend can send a

single data pack to backend, instead of sending data separately from multiple controllers. The definition of the service function is shown as below:

```
app.service('datapack', function() {
    var weatherinfo = {};
    var routeinfo = {};


    return {
        getweather: function() {
            return weatherinfo;
        },
        setweather: function(value) {

            weatherinfo = value;
        },
        getroute: function() {
            return routeinfo;
        },
        setroute: function(value) {
            routeinfo=value;
        }
    }
});
```

Figure 2.1.1: Service function

The “WeatherCtrl” controller contains all the functions that relate to weather call, including the openweathermap API, a dynamic icon function and a weather effects plugin. The “MainCtrl” controller contains google map service functions and an interface connecting the frontend with the backend. Google map service functions will be introduced later.

```
var datas=JSON.stringify([JSON.parse(datapack.getweather()),JSON.parse(datapack.getroute())]);//string(array)
console.log(JSON.parse(datas));
$scope.showprice=true;
$scope.uberprice=0.0;
$http({
  method : 'POST',
  url : 'http://localhost:5000/getPrice',
  data : JSON.parse(datas), //parse to json before sending
  'Content-Type': 'application/json'

  }).success(function(data){
      /*called for result & error because 200 status*/
      $scope.uberprice = data
  })
  .error(function(data){
      alert(data)
  });
```

Figure 2.1.2: API Call to Backend

In order to display the routes better, our team sets google map as full screen, and puts all functional operations in the mouse responsive sidebar. Such a sidebar can automatically hide when mouse leaves its range.

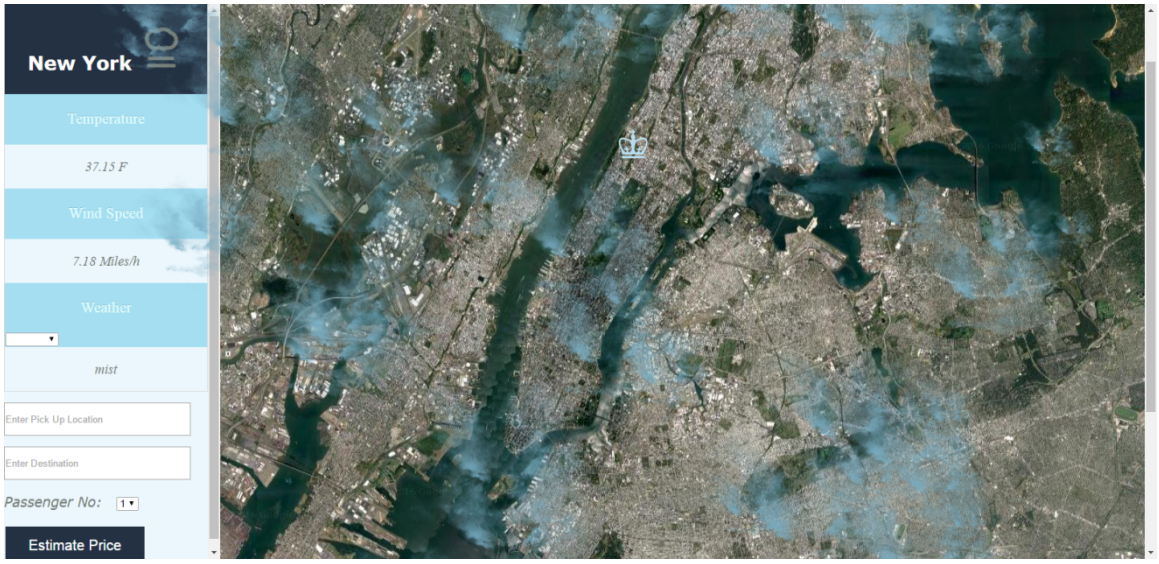
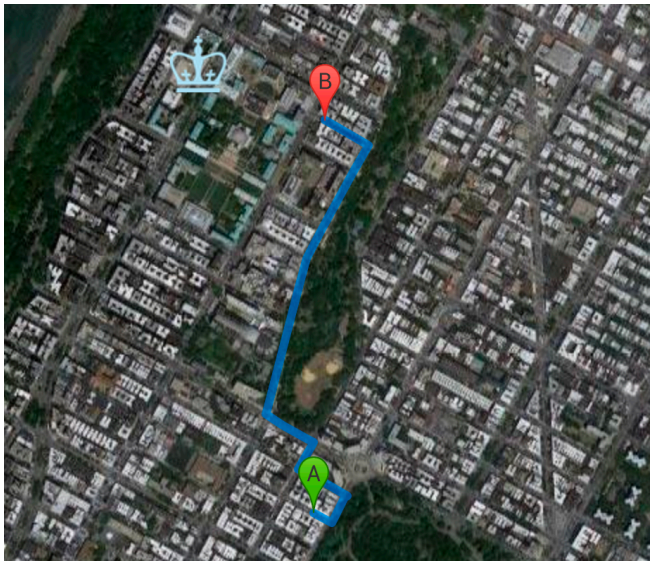


Figure 2.1.3: Screenshot of EC2 App

Libraries & APIs



Google Map API

Figure 2.2.1: Google Map Route

At front end, we applied several Google Map API [5] functions, including an offline map, markers, and getDirections to get the information about an Uber trip. By calling getDirections, we sent the request containing origin and destination locations to

the server. Then, we calculated directions (using 'DRIVING' transportation method) with the DirectionsService object. Result will be displayed as a poly-line drawing the route on a map.

To realize this, first we wrapped the request with specified origin location and destination. These two parameters are bound as ng-models so that they can be derived from the two input-boxes on the web page.

```
var request = {
  origin: $scope.directions.origin.getPlace().formatted_address,
  destination: $scope.directions.destination.getPlace().formatted_address,
  travelMode: google.maps.DirectionsTravelMode.DRIVING
};
```

Figure 2.2.2: Request Form

To use directions in the Google Maps JavaScript API, we create an object of type DirectionsService and call directionsService.route to initiate the wrapped request to the Directions service by passing it a DirectionsRequest object that literally contains the input terms and a callback.

```
directionsService.route(request, function (response, status) {...
```

Figure 2.2.3: Direction Service

After receiving the response, we can convert it into a JSON Object and then parse the information we need to compute the results (i.e. trip distance, duration).

Autocompletion

We included AngularJS library in order to work with Google Maps, where we used the gmPlacesAutocomplete directive to realize the autocompletion of places. This directive listens for users' input and provides place predictions based on the input. This is realized by the property that when the place changes, the 'gmPlacesAutocomplete::placeChanged' event is broadcasted.

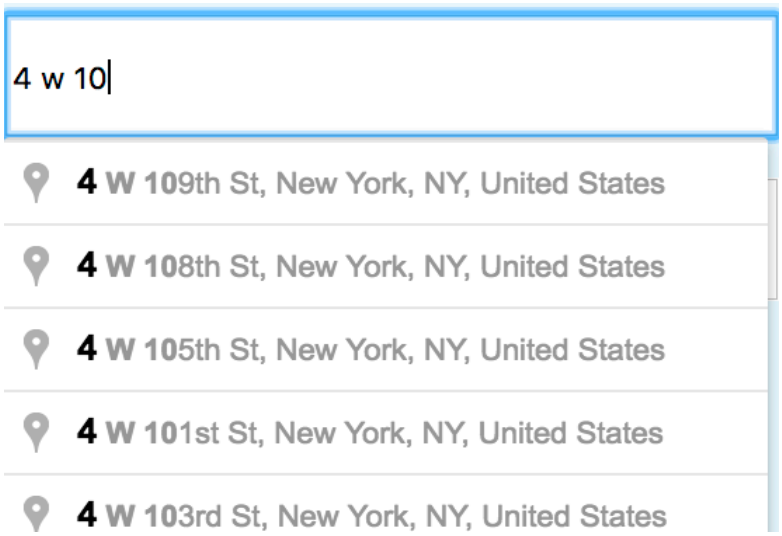


Figure 2.2.4: Auto-completion

As shown in the screen shot, when users type in the location (whether as origin or destination), the predictions will be provided in a dropdown list, so that uses can quickly choose the right place.

Responsive Sidebar

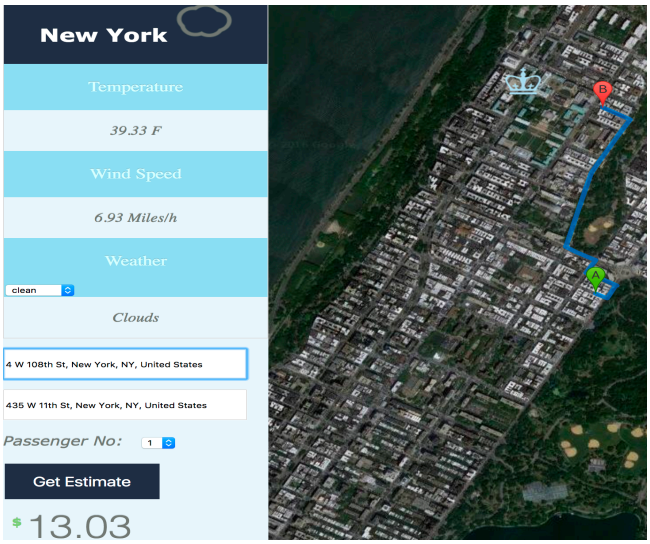


Figure 2.3.1: Responsive Sidebar

We collected three modules in the sidebar: the real-time weather parameters, places autocompletion including the request button for getting the distance, and the returned price result. At first, we need to create a top-layer-fixed sidebar, and let it include the three modules. This part can be realized in the html file like this:

```
nav.sidebar:hover + .main{
  margin-left: 200px;
}
```

Figure 2.3.2: Sidebar Class

Since we want to show the full-screen map on the web page, now we need to implement the mouse-responsive function on the sidebar. The following part of our css

```
<nav class="navbar navbar-inverse sidebar navbar-fixed-top" role="navigation">
```

file will let the nav bar to be shown on mouseover.

Figure 2.3.3: Sidebar CSS Style

Openweathermap API

The team utilized a weather API from openweathermap company to fetch real time weather information, including weather type, wind speed, temperature. The API is called through http "get" request as the form of:

$http.get("http://api.openweathermap.org/data/2.5/weather?q=New%20York&appid=apikey")

Such an API checks the current weather through its frequently updated database every time our web is refreshed. Then, the front end can parse the JSON format data, and

post the real time weather on panel

Figure 2.4.1: Weather Panel

Based on the yellow taxi trip data fetched by backend, the team analyzed the potential influences that different types of weathers may cause on the modeling of pricing, and classified the weather types into seven majority groups. Namely, they are "clear-day",

"partly-cloudy-day", "cloudy", "rain", "sleet" and "fog". Moreover, to simplify the data transmit, these weather types are converted into numbers ranging from 1 to 7.

Weather Effects Plugin

A jQuery plugin [6] for displaying the float weather effect was added by our team. Through this plugin, website can present dynamic weather effects, which always change according to New York's current weather. Screenshots are presented as below:

Figure 2.5.1: Sunny & Broken Clouds

Figure 2.5.2: Clouds & Rain

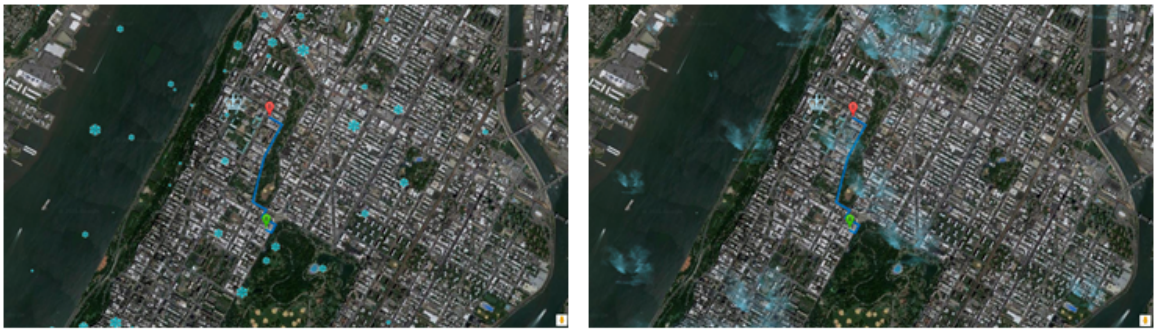


Figure 2.5.3: Snow & Mist

The team combined the weather code conversion process with the selection of this weather effect plugin in the same if-else statement, ensuring the weather effect will switch in real time according to openweathermap API's data. Screenshot of the code are presented as following:

```
//==========convert weather data==========
if ($scope.wtr=="clear sky"|| $scope.wtr=="clear")//clear
  {$scope.weather=1;
    $(document).snowfall({round:false,minSize:420,maxsize:420,maxSpeed:0,minSpeed:0,image :"i/sun1.png",flakeCount:1});
  }
else if ($scope.wtr=="scattered clouds"|| $scope.wtr=="broken clouds")//overcast
   {$scope.weather=2;
   $(document).snowfall({round:false,minSize:400,maxsize:420,maxSpeed:0,minSpeed:0,image :"i/bcloud.png",flakeCount:8});
  }
  else if ($scope.wtr=="few clouds"|| $scope.wtr=="clouds")//cloudy
    {$scope.weather=3;
     $(document).snowfall({round:false,minSize:400,maxsize:420,maxSpeed:0,minSpeed:0,image :"i/cloud.png",flakeCount:4});
    }
  else if ($scope.wtr=="shower rain"|| $scope.wtr=="rain"|| $scope.wtr=="thunderstorm")//rain
    {$scope.weather=4;
     $(document).snowfall({round:false,minSize:350,maxsize:360,maxSpeed:12,minSpeed:10,image :"i/rain.png",flakeCount:25});
    }
  else if ($scope.wtr=="snow")//snow
    {$scope.weather=5;
     $(document).snowfall({round:true,minSize:4,maxSize:35,text:'XX',image :"i/snowflake.png",flakeColor:'',collection:'',flakeCount:50});
    }
  else if ($scope.wtr=="mist"|| $scope.wtr=="haze"|| $scope.wtr=="fog")//haze
    {$scope.weather=6;
     $(document).snowfall({round:false,minSize:300,maxsize:300,maxSpeed:0,minSpeed:0,image :"i/mist.png",flakeCount:25});
    }
```

Figure 2.5.4: Weather Convert

Weather effect can also change manually with a selection button that locates at weather panel, and the weather type that will be sent to backend is replaced by this manually selected type. With the help of this function, we could check whether the weather variable may influence the estimated price when origin and destination locations are fixed, without refreshing the website and re-entering these location parameters.

Dynamic Weather Icon

Similarly, a dynamic weather icon will float beside the weather panel, right next to “New York”. This icon can vary according to the data outcome that is fetched by openweathermap API. The team realized this effect with the help of a plugin called “skycon” [7][8]. The types of icons are shown below:

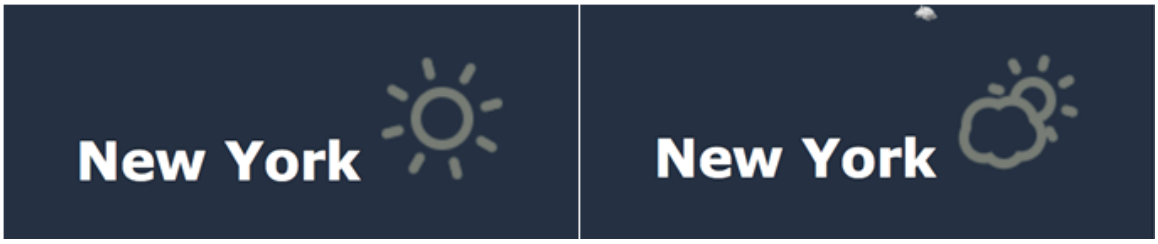


Figure 2.6.1: Clear & Broken Clouds

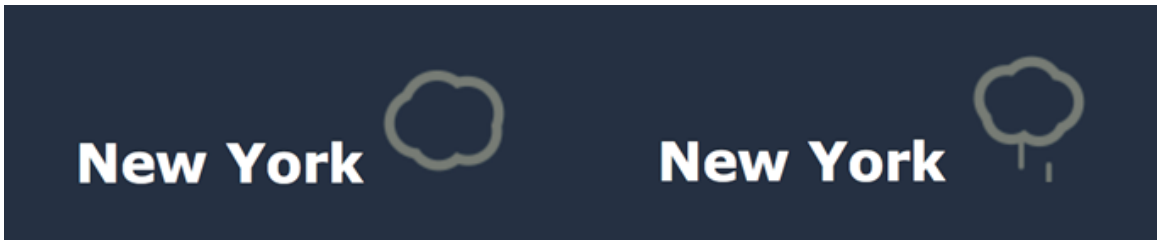


Figure 2.6.2: Clouds & Rain

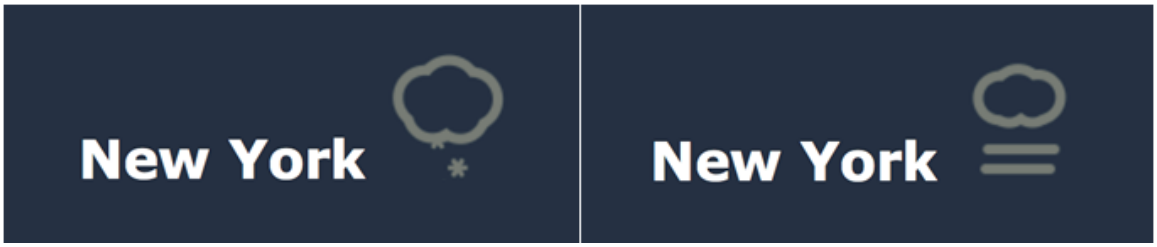


Figure 2.6.3: Snow & Mist

Figure 2.6.4: Windy

Result

Without Weather Influence

Our team tested the web application by computing the value of R^2, and created test and trained data by splitting the combined files. The Linear Regression model was trained on the training data only. Here, 90% of the data was used for training and the remaining 10% for evaluating. By importing python's library sklearn, the team used r2_score function to compute R^2. The result was 0.936, indicating that about 93.6% of the test data fitted in with our model. The team used cross_val_predict function to visualize prediction errors.

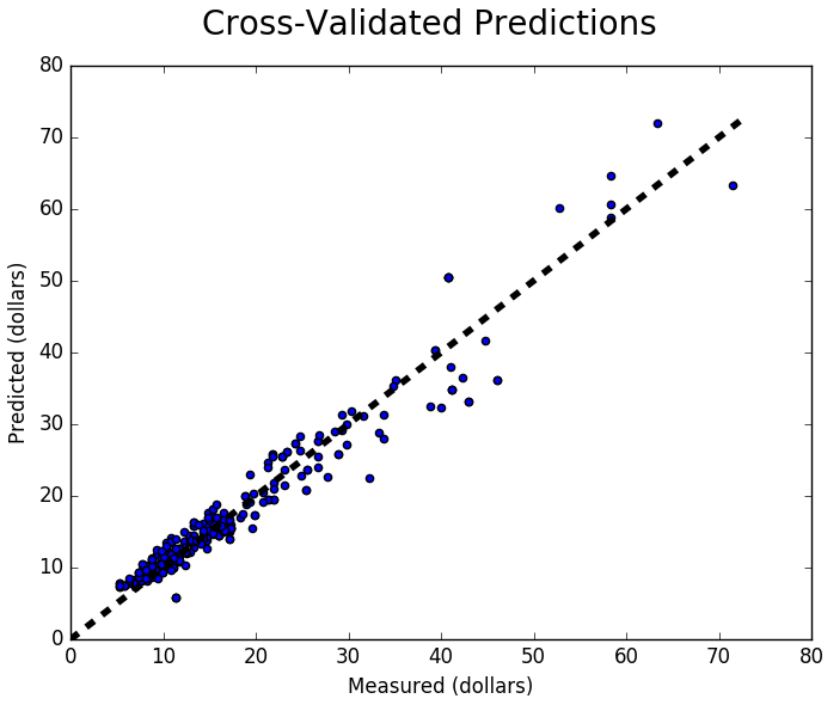


Figure 3.1.1: Cross-Validated Predictions without Weather Variables

Above graph is a comparison between predicted price and the real trip total amount. In this case, we removed weather conditions variables from our training model. Its x axis represents test data label, and its y axis represents the corresponding prediction made by our machine learning training model. The graph indicates the performance of

our predictor is good if we only consider about the trip distance and the price for our training model.

With Weather Influence

Similarly to previous error calculation, in this case, we will add weather features into our training model. It turns out that our estimated price will not match with the real price at all. In fact, the estimated price trends to stay at a constant value. This result indicates that the price and weather conditions are not linearly related to each other; therefore, we should not train our model with weather conditions.

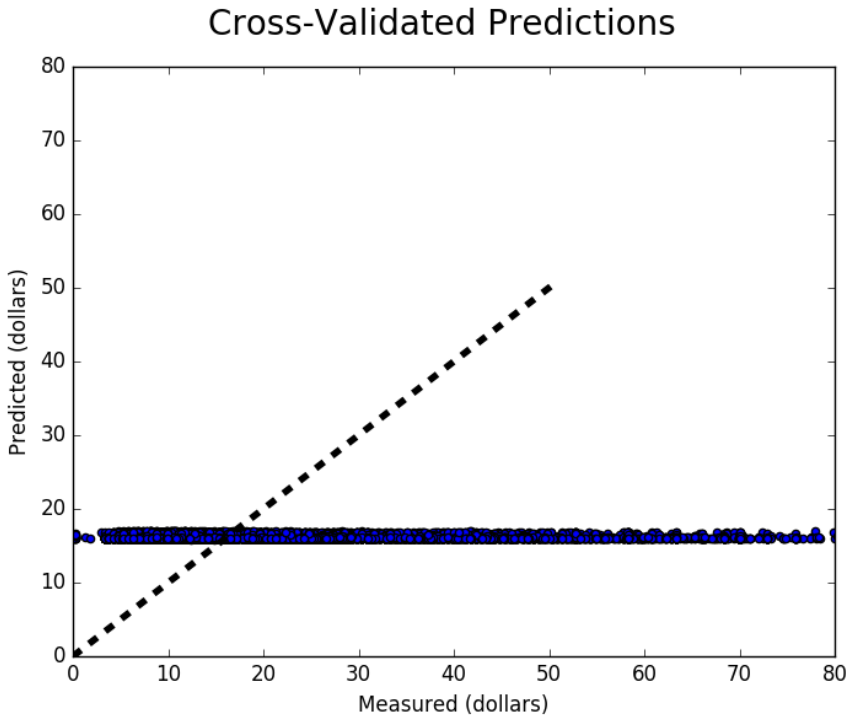


Figure 3.2.1: Cross-Validated Predictions with Weather variables

Conclusion

In our project, we built Linear Regression model based on the dataset collected from the Internet, and used the training model to predict the Uber price for customers. Initially, we took the weather factors into consideration, joined the weather data with taxi trip data. However, it turned out that the estimated price stayed at a stable value. Therefore, we built two models, with and without consideration of weather factors, to determine whether we should take weather features to build our training model. We applied Coefficient of Determination or R^2, which is commonly used to evaluate how well a linear regression model fits the data, to test the two models. Ideally, we expected the r2_score result would be close to 1, which means that the prediction error rate tends to be zero. Nevertheless, the result shows that the model with weather conditions outputs an R^2 value close to 0. This implies that weather features are not linearly related to the price. On the other side, the model without considering weather factors gives an R^2 value of 0.936, indicating that our training data fits well with the labels. Therefore, finally, we chose the second model, the one without weather variables, as our price estimator training model.

We tested our estimator by comparing its estimated price with the one from the real Uber app. It turns out within the range of New York City, if we consider the price given by Uber app to be the true value, our Uber price estimation error rate ranges from 3% to 5% if the real price is between $20 and $50. The error range is within $2 if the real price estimated by Uber app is below $10. Thus, our Uber price estimator does achieve a good performance in terms of its closeness to the real Uber app estimated price.

Reference

1. Uber, Lyft Cars Arrive Much Faster Than Taxis, Study Says, Sep 8, 2014, http://www.forbes.com/sites/ellenhuet/2014/09/08/uber-lyft-cars-arrive-faster-than-taxis/#4ce1bf9d5f73, Accessed on Nov 23, 2016

2. TLC_Trip_Record_Data, NYC Taxi& Limousine Commission, 2016, http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml, Accessed on Nov 20, 2016

3. Weather History for KNYC-January, 2016, WEATHER UNDERGROUND, 2016, https://www.wunderground.com/history/airport/KNYC/2016/1/1/DailyHistory.html?req_city=Manhattan&req_state=NY&req_statename=&reqdb.zip=10023&reqdb.magic=3&reqdb.wmo=99999&MR=1, Accessed on Nov 20, 2016

4. AngularJS, v1.4.8, Google, angularjs.org, 2015, https://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js, Accessed on Nov 20, 2016

5. google-map-js, Google, 2016, http://jvandemo.github.io/angularjs-google-maps/dist/angularjs-google-maps.js, Accessed on Dec 23, 2016

6. jQuery, v3.1.1, jQuery Foundation, jquery.org, 2016 https://ajax.googleapis.com/ajax/libs/jquery/3.1.1/jquery.min.js, Accessed on Nov 20, 2016

7. skycons.js, Github, https://rawgithub.com/darkskyapp/skycons/master/skycons.js, Accessed on Dec 23, 2016

8. Snowfall jquery plugin, v2.0, Github, 2012, https://cdn.rawgit.com/loktar00/JQuery-Snowfall/master/src/snowfall.jquery.js, Accessed on Dec 23, 2016