NYC Taxi Fare Prediction

Rider Fare Prediction in The Big Apple

Allen Kong Dec 7, 2020 · 5 min read

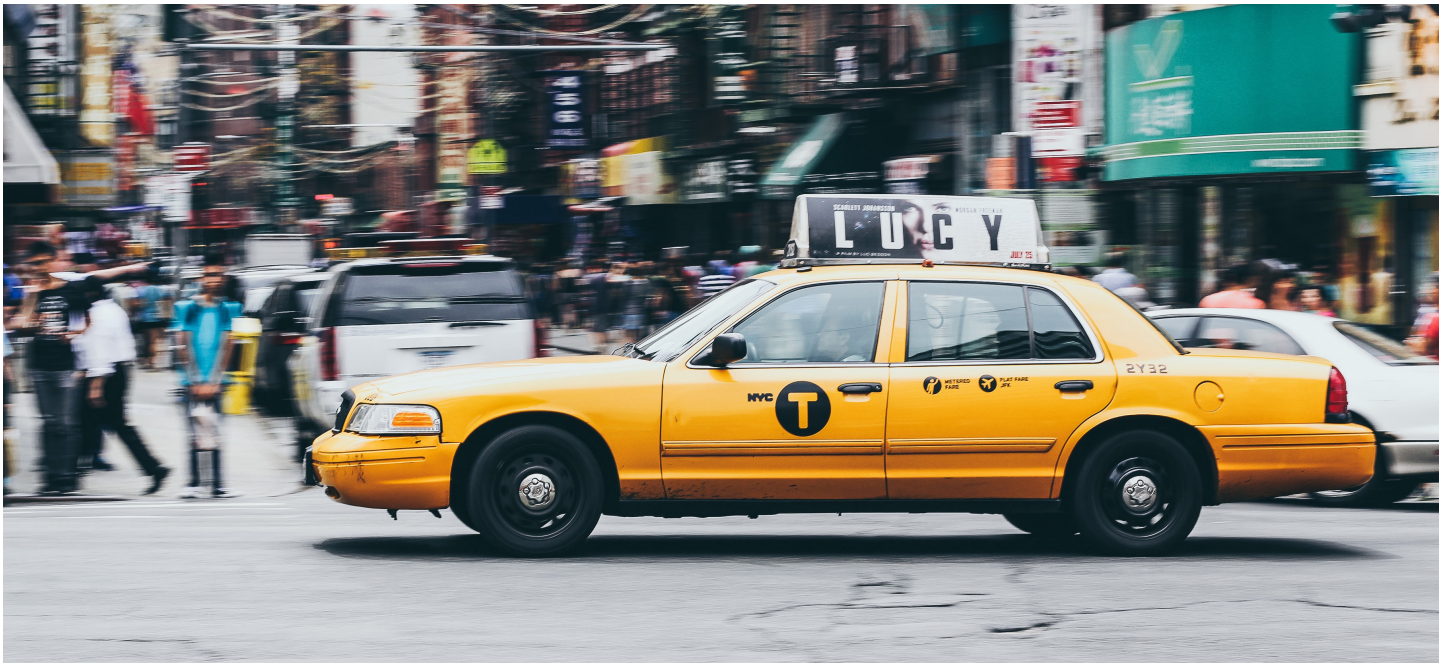

Photo by Alexander Redl on Unsplash

The Data

Loading the Data

The data for this project can be found on Kaggle in the New York City Taxi Fare Prediction competition held by Google Cloud. The entire training set consists of about 55 million rows of NYC taxi fare data. There is not enough memory in the CPU to train a model using all this data so I decided to use 4 million rows of training data which is

```
df = pd.read_csv('/kaggle/input/new-york-city-taxi-fare-prediction/train.csv
test_df = pd.read_csv('/kaggle/input/new-york-city-taxi-fare-prediction/test
df.shape, test_df.shape
```

```
((4000000, 8), (9914, 7))
```

Hosted on Jovian View File

Let's take a quick peek at how the data looks like.

```
df.head()
```

	key	fare_amount	pickup_datetime	pickup_longi
0	2009-06-15 17:26:21.0000001	4.5	2009-06-15 17:26:21 UTC	-73.84
1	2010-01-05 16:52:16.0000002	16.9	2010-01-05 16:52:16 UTC	-74.01
2	2011-08-18 00:35:00.00000049	5.7	2011-08-18 00:35:00 UTC	-73.98

Hosted on Jovian View File

We can see that there are 8 columns which consist of the unique key, the fare amount and 6 features. Before jumping into the statistics, let's check the sample for null entries.

Null Values

```
df.isnull().sum().sort_index()/len(df)
```

```
dropoff_latitude     0.000007
dropoff_longitude    0.000007
fare_amount          0.000000
key                  0.000000
passenger_count      0.000000
pickup_datetime      0.000000
```

There appears to be an insignificant amount of null entries in the sample so it should be fine to remove them from the sample. Now we can review the statistics of the training set and the test set to find anomalies.

Outliers, Cleaning & Engineering

```
df.describe()
```

	fare_amount	pickup_longitude	pickup_latitude	dropoff_lo
count	3.999971e+06	3.999971e+06	3.999971e+06	3.999
mean	1.134389e+01	-7.250845e+01	3.991910e+01	-7.250
std	9.824417e+00	1.312180e+01	8.797877e+00	1.289
min	-6.200000e+01	-3.426609e+03	-3.488080e+03	-3.412
25%	6.000000e+00	-7.399207e+01	4.073491e+01	-7.399

Hosted on Jovian View File

```
test_df.describe()
```

	pickup_longitude	pickup_latitude	dropoff_longitude	dropo
count	9914.000000	9914.000000	9914.000000	99
mean	-73.974722	40.751041	-73.973657	
std	0.042774	0.033541	0.039072	
min	-74.252193	40.573143	-74.263242	
25%	-73.992501	40.736125	-73.991247	

Hosted on Jovian View File

there are outliers in the training dataset. Including these data points in the training set will prevent our model from running efficiently so it is best to handle them. The fare amount appears to range from -$62 to almost $1300 which makes no sense. The base fee for NYC taxi cabs is $2.50 so we will limit the fare amount to be at least $2.50 and below $500. The passenger count seems to go up to 208 which makes no sense so we will limit it to the maximum capacity which is 5 passengers.

```
df.drop(df[df['pickup_longitude'] == 0].index, axis
df.drop(df[df['pickup_latitude'] == 0].index, axis=
df.drop(df[df['dropoff_longitude'] == 0].index, axi
df.drop(df[df['dropoff_latitude'] == 0].index, axis
df.drop(df[df['passenger_count'] == 208].index, axi
```

Hosted on Jovian View File

Currently, Pandas is viewing the "**pickup_datetime**" feature as an object type in the dataframe. In order for object types to be utilized in our machine learning model, the feature needs to be transformed into a numeric type. This can be accomplished by converting the feature to a datetime type and then transforming the datetime object into multiple attributes using the Pandas datetime functionality. From the datetime object, we'll be able to create useful attributes like **Year**, **Month**, **Day**, **Day of Week**, and **Hour**.

```
df['key'] = pd.to_datetime(df['key'])
key = test_df.key
test_df['key'] = pd.to_datetime(test_df['key'])
df['pickup_datetime']  = pd.to_datetime(df['pickup_
```

Hosted on Jovian View File

The city of New York longitude ranges between -75 and -72. The latitude ranges between 40 and 42. There are a few points in the dataset that lie outside these bounds. These points will be removed since they are not within the boundaries of the city.

```
df.drop(df.index[(df.pickup_longitude < -75) |
            (df.pickup_longitude > -72) |
            (df.pickup_latitude < 40) |
            (df.pickup_latitude > 42)],inplace=True)
df.drop(df.index[(df.dropoff_longitude < -75) |
```

Hosted on Jovian View File

Along with the datetime attributes that were added, there are a few features that can be added using the pick-up and drop-off points from the data. The distance between the two points would be a common choice. I decided to use the **geopy geodesic** and **great_circle** distance calculations which uses both pick-up and drop-off points. The city is full of many notable locations that taxi travelers tend to go to so we will also introduce the distance between several of these notable locations and the drop-off point. The locations I decided to use were the JFK airport, the LaGuardia airport, the Newark airport, Times Square, Central Park, the Statue of Liberty, Grand Central, the MET museum, and the World Trade Center.

Model Training

Now that we have a clean dataset, we are ready to train a model for predicting taxi fare. Before we do that through, we will split the dataset into a train (80%) and test (20%).

```
X, y = df.drop('fare_amount', axis = 1), df['fare_a
X_train, X_test, y_train, y_test = train_test_split
```

Hosted on Jovian View File

XGBoost

The model I chose for this particular dataset was XGBoost. Another alternative would be LightGBM which many others have used as well in the Kaggle competition. XGBoost uses the DMatrix data structure for optimized and efficient computing so we will transform the training and test sets into a DMatrix. The parameters are already preset in my model from hyperparameter tuning using grid search. The preset parameters will

```
dtrain = xgb.DMatrix(X_train, label=y_train)
dvalid = xgb.DMatrix(X_test, label=y_test)
dtest = xgb.DMatrix(test_df)
watchlist = [(dtrain, 'train'), (dvalid, 'valid')]

xgb_params = {
    'min_child_weight': 1,
    'learning_rate': 0.05,
    'colsample_bytree': 0.7,
    'max_depth': 10,
    'subsample': 0.7,
    'n_estimators': 5000,
    'n_jobs': -1,
    'booster' : 'gbtree',
    'silent': 1,
    'eval_metric': 'rmse'}

model = xgb.train(xgb_params, dtrain, 700, watchlis
```

```
[01:27:37] WARNING: ../src/learner.cc:516:
Parameters: { n_estimators, silent } might not be
used.

  This may not be accurate due to some parameters
```

Hosted on Jovian View File

Prediction

After boosting the model, we are ready to test it against the real test dataset. This model which was trained and boosted on the CPU ended up achieving a score of 3.0325 which places us in the 80th percentile on the Kaggle leaderboard. In order to

GPU Based Training

It's very easy to run out of memory using the CPU when training the XGBoost model. Not only that, cleaning and training using 7% of the data took more than 8 hours. This makes it very inefficient to use a CPU when dealing with large datasets. GPUs on the other hand are extremely efficient when it comes to parallel computing. The perfect technology to use for this case would be **RAPIDS** which is an advanced library that runs on CUDA and utilizes **dask** for parallel computing. Using the same data cleaning processes earlier, I decided to put the XGBoost model to test using the entire dataset.

```
%%time
dtrain = xgb.dask.DaskDMatrix(client, X_train, y_tr
dvalid = xgb.dask.DaskDMatrix(client, X_test, y_tes
watchlist = [(dtrain, 'train'), (dvalid, 'valid')]

params = {
    'learning_rate': 0.05,
    'max_depth': 11,
    'objective': 'reg:squarederror',
    'subsample': 0.7,
    'colsample_bytree': 0.7,
    'min_child_weight': 1,
    'gamma': 1,
    'silent': True,
    'verbose_eval': True,
    'booster' : 'gbtree',
    'eval_metric': 'rmse',
    'tree_method':'gpu_hist',
    'n_gpus': 1
}

trained_model = xgb.dask.train(client, params, dtra
```

Hosted on Jovian View File

minutes! This shows the magnitude of using the GPU for training. It feels too good to be true but the numbers don't lie. I only used 1 GPU for this process and it's easily 10 times more efficient than using the CPU. Using the GPU trained model, the predictions turned out to be much better than the CPU trained model. The GPU trained model ended up achieving a score of 2.89185 which places us in the 94th percentile on the Kaggle leaderboard.

Conclusion

When we're given data, it's extremely important to be able to utilize all of it. Using 7% and 100% makes a huge difference and we have the technologies to be able to do that. The GPU offers better performance, higher efficiency and unmatched computing power compared to the CPU counterpart. Whenever you plan on your next data science project, just remember that the GPU can save you a whole day on pre-processing and training!

Learn More

- Project Code
- Kaggle Competition
- RAPIDS GitHub
- Dask

Data Science Machine Learning Artificial Intelligence Xgboost Gpu