New York City Taxi Trip Duration

Team Members Name and CIN:

Project Description

The main object of this project is to build a model to predict the total ride duration of taxi trips in New York City. We are provided a dataset available on Kaggle with trip duration information, and our goal is to build a machine learning model that can learn from the provided data samples, and predict the trip duration results for any new inputs containing required features.

The dataset was provided by the NYC Taxi and Limousine Commission. Data was collected from individual trips on the on board recorder and manually.

Individual responsibilities:

The team started with a group research on data preprocessing, and each team member provided ideas to preprocess data. After preprocessing, the team split into two groups to experiment different algorithms. Jiajun and Pruthvi worked on the regression algorithms, including Linear Regression, Lasso, Elastic Net and Ridge Regression. Andrew and Ruturaj worked on neural network related regression fields. After experimenting, Andrew and Jiajun finalized the coding details, Pruthvi provided data and result analysis, including graphic and comparison, and Ruturaj summarized, drew conclusions, and produced the final report and presentation.

Data Fields:

· id - a unique identifier for each trip.

· vendor_id - a code indicating the provider associated with the trip record

· pickup_datetime - date and time when the meter was engaged

· dropoff_datetime - date and time when the meter was disengaged

· passenger_count - the number of passengers in the vehicle (driver entered value)

· pickup_longitude - the longitude where the meter was engaged

· pickup_latitude - the latitude where the meter was engaged

· dropoff_longitude - the longitude where the meter was disengaged

· dropoff_latitude - the latitude where the meter was disengaged

· store_and_fwd_flag - This flag indicates whether the trip record was held in vehicle memory before sending to the vendor

· trip_duration - duration of the trip in second

Project Method

Data Loading

We start the process by reading the .csv file downloaded from Kaggle. As our main goal is to build a machine learning model, we need the data that have "correct answers". Therefore, we only imported the "train.csv", the training data that has the correct label. Using this data set, we are able to train a model that is available for any test input. The data set contains 10 columns and 1458644 rows of data entries.

```
(1458644, 11)
```

	id	vendor_id	pickup_datetime	dropoff_datetime	passenger_count	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	store_and_fwd_flag	trip_duration
0	id2875421	2	2016-03-14 17:24:55	2016-03-14 17:32:30	1	-73.982155	40.767937	-73.964630	40.765602	N	455
1	id2377394	1	2016-06-12 00:43:35	2016-06-12 00:54:38	1	-73.980415	40.738564	-73.999481	40.731152	N	663
2	id3858529	2	2016-01-19 11:35:24	2016-01-19 12:10:48	1	-73.979027	40.763939	-74.005333	40.710087	N	2124
3	id3504673	2	2016-04-06 19:32:31	2016-04-06 19:39:40	1	-74.010040	40.719971	-74.012268	40.706718	N	429
4	id2181028	2	2016-03-26 13:30:55	2016-03-26 13:38:10	1	-73.973053	40.793209	-73.972923	40.782520	N	435

Preprocessing

At the first discussion, we found out that some columns (features) may not have a significant impact on the reduction result, and we wanted to eliminate these columns for now (we may go back to these columns after initial prediction, and compare the result with these columns). The columns that we think are not import are:

- **"id"** - individual trip id should have nothing to do with the result.
- **"vendor_id"** - We believe that as both vendors are providing similar taxi service, the vendor identification should not play a big role.
- **"store_and_fwd_flag"** - The way this trip is reported, either through the forwarding system, or manually from the recording system, should not matter.

We then run the following code to eliminate these features for now:

```
taxi_new = taxi.drop(['id','vendor_id','store_and_fwd_flag'],axis=1)
```

	pickup_datetime	dropoff_datetime	passenger_count	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	trip_duration
0	2016-03-14 17:24:55	2016-03-14 17:32:30	1	-73.982155	40.767937	-73.964630	40.765602	455
1	2016-06-12 00:43:35	2016-06-12 00:54:38	1	-73.980415	40.738564	-73.999481	40.731152	663
2	2016-01-19 11:35:24	2016-01-19 12:10:48	1	-73.979027	40.763939	-74.005333	40.710087	2124
3	2016-04-06 19:32:31	2016-04-06 19:39:40	1	-74.010040	40.719971	-74.012268	40.706718	429
4	2016-03-26 13:30:55	2016-03-26 13:38:10	1	-73.973053	40.793209	-73.972923	40.782520	435

Pick-up Time

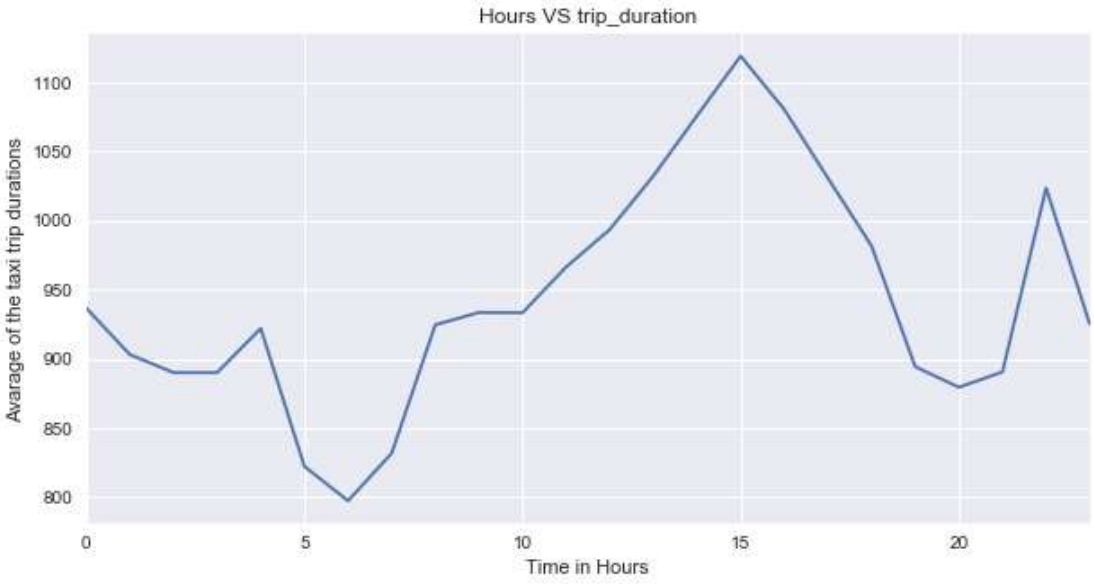
We find that “pickup_datetime” and dropoff_datatime” is highly correlated because the difference of them equals the “trip_duration”, which is our label. In other words, we need only one of these two features. We choose to retain “pickup_datetime” and eliminate the “dropoff_datatime” feature.

We also notice that pick-up time is important to trip durations because in the big city, different times of the day mean different traffic situations. We run an analysis on pick-up time over trip durations.

By importing matplotlib function and using line graph to represent the data for Aggregations (reductions) of finding mean based on Hour and trip_duration.

```
figure,axes = plt.subplots(figsize = (10, 5))
hours = taxi_new.groupby(["hour"]).agg(('mean'))["trip_duration"]
hours.plot(kind="line", ax=axes)
plt.title('Hours VS trip_duration')
axes.set_xlabel('Time in Hours')
axes.set_ylabel('Avarage of the taxi trip durations')
plt.show()
```

- **Representing data in Hours VS Trip Duration**



We can see that the trip duration has a global minimum at around 6am and a monotonous increase from 6am to 3pm. A global maximum is reached and then there's a decrease from 3pm to 8pm, reaching a local minimum. There's another increase again from 8pm to around 11pm, reaching a local maximum, and then there's a decrease from 11pm to 6pm. It is reasonable to think that because of these traffic situations, we need to further categorize the "pickup_datetime" feature.

We created a column to capture the "hour" only from pickup_datatime. We first use the following method to isolate "hour" from the date time:

taxi_new_h = pd.to_datetime(taxi_new["pickup_datetime"])

taxi_new['hour'] = taxi_new_h.map(lambda x: x.hour)

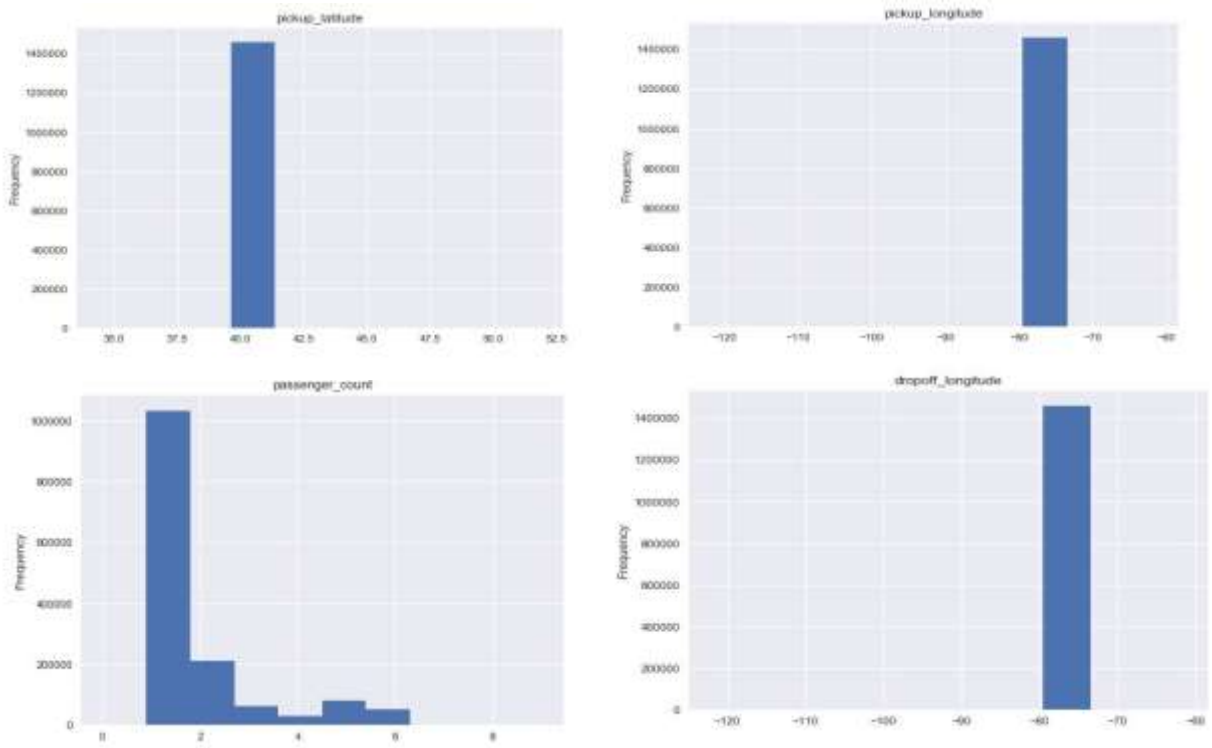
- **Table 1: Output after set hour from datetime**

	pickup_datetime	dropoff_datetime	passenger_count	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	trip_duration	hour
0	2016-03-14 17:24:55	2016-03-14 17:32:30	1	-73.982155	40.767937	-73.964630	40.765602	455	17
1	2016-06-12 00:43:35	2016-06-12 00:54:38	1	-73.980415	40.738564	-73.999481	40.731152	663	0
2	2016-01-19 11:35:24	2016-01-19 12:10:48	1	-73.979027	40.763939	-74.005333	40.710087	2124	11
3	2016-04-06 19:32:31	2016-04-06 19:39:40	1	-74.010040	40.719971	-74.012268	40.706718	429	19
4	2016-03-26 13:30:55	2016-03-26 13:38:10	1	-73.973053	40.793209	-73.972923	40.782520	435	13

To represent the each of the separate feature matrix list into Histogram plot graph by set Index trip duration:

```
time = taxi_new.set_index('trip_duration')
for f in feature_cols:
plt.figure()
plt.title(f)
time[f].plot(kind='hist')
plt.show()
```

- **Represents a Table 1 into Histogram plot graph**



We then run the following method to categorize the hour data base on our analysis:

```
def cateHours(x):
    if 0 <= x <= 5:
        return "EM"    # Early Morning, mono decrease
    elif 6 <= x <= 15:
        return "MP"    # Morning Peak, mono increase
    elif 16 <= x <= 19:
        return "AF"    # Afternoon to night time, mono decrease
    elif 20 <= x <= 22:
        return "AP"   # Night peak, mono increase
    elif 23 <= x <= 24:
        return "LN"   # Late Night, mono decrase
```

➢ **Output after categorize the hour into EM, MP, AF, AP, LN**

```
taxi_new['hour'] = taxi_new['hour'].apply(cateHours)
taxi_new.head()
```

	pickup_datetime	dropoff_datetime	passenger_count	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	trip_duration	hour
0	2016-03-14 17:24:55	2016-03-14 17:32:30	1	-73.982155	40.767937	-73.964630	40.765602	455	AF
1	2016-06-12 00:43:35	2016-06-12 00:54:38	1	-73.980415	40.738564	-73.999481	40.731152	663	EM
2	2016-01-19 11:35:24	2016-01-19 12:10:48	1	-73.979027	40.763939	-74.005333	40.710087	2124	MP
3	2016-04-06 19:32:31	2016-04-06 19:39:40	1	-74.010040	40.719971	-74.012268	40.706718	429	AF
4	2016-03-26 13:30:55	2016-03-26 13:38:10	1	-73.973053	40.793209	-73.972923	40.782520	435	MP

We then use One-Hot encoding to categorize this non-numeric feature:

taxi_new_onehotHour = pd.get_dummies(taxi_new['hour'])

taxi_new = pd.concat([taxi_new,taxi_new_onehotHour], axis=1)

➢ **Output after One-Hot encoding**

	AF	AP	EM	LN	MP
0	1	0	0	0	0
1	0	0	1	0	0
2	0	0	0	0	1
3	1	0	0	0	0
4	0	0	0	0	1

Distance

We also notice that the coordinates composed by "pickup_latitude", "pickup_longtitude", "dropoff_latitude", and "dropoff_longtitude" provide good information about this distance between the pick-up and drop-off points.

We use the following method to calculate the distance between the two points, and run an analysis for distance over trip duration:

```
from math import cos, asin, sqrt, sin, atan2, radians
def getDistanceFromLatLon(lat1,lon1,lat2,lon2):
    R = 3959 # Radius of the earth in miles
    dLat = radians(lat2-lat1)  #deg2rad below
    dLon = radians(lon2-lon1)
    a = sin(dLat/2) * sin(dLat/2) + cos(radians(lat1)) * cos(radians(lat2)) * sin(dLon/2) * sin(dLon/2)
    c = 2 * atan2(sqrt(a), sqrt(1-a))
    d = R * c; #Distance in km
    #print(d)
    return d

def f(x):
    return getDistanceFromLatLon(x[0], x[1], x[2], x[3])
```

```
taxi_new['distance'] = taxi_new[['pickup_latitude','pickup_longitude','dropoff_latitude','dropoff_longitude']].apply(f, axis =1)
```

- **Output after include the feature into our model.**

pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	trip_duration	hour	AF	AP	EM	LN	MP	distance
-73.982155	40.767937	-73.964630	40.765602	455	AF	1	0	0	0	0	0.931195
-73.980415	40.738564	-73.999481	40.731152	663	EM	0	0	1	0	0	1.121959
-73.979027	40.763939	-74.005333	40.710087	2124	MP	0	0	0	0	1	3.967761
-74.010040	40.719971	-74.012268	40.706718	429	AF	1	0	0	0	0	0.923103
-73.973053	40.793209	-73.972923	40.782520	435	MP	0	0	0	0	1	0.738600

Further Reduction

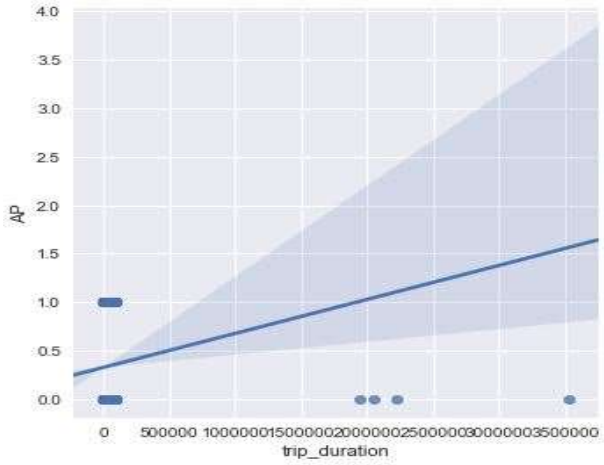
After the data processing, we think that “pickup_datetime” and its drop-off counter parts are not necessary in the feature list anymore, as well as the temporary column “hour”, so we drop them and isolate the label. The training data set then becomes:

- **Output after dropping temporary column hour**

	passenger_count	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	AF	AP	EM	LN	MP	distance
0	1	-73.982155	40.767937	-73.964630	40.765602	1	0	0	0	0	0.931195
1	1	-73.980415	40.738564	-73.999481	40.731152	0	0	1	0	0	1.121959
2	1	-73.979027	40.763939	-74.005333	40.710087	0	0	0	0	1	3.967761
3	1	-74.010040	40.719971	-74.012268	40.706718	1	0	0	0	0	0.923103
4	1	-73.973053	40.793209	-73.972923	40.782520	0	0	0	0	1	0.738600

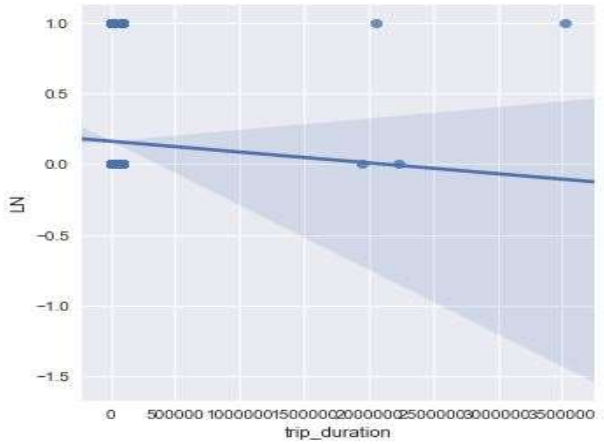
- **To represent the trip duration and AP into seaborn.lmolot**

```
import seaborn as sns
sns.lmplot(x='trip_duration', y='AP', data=taxi_new)
plt.show()
```



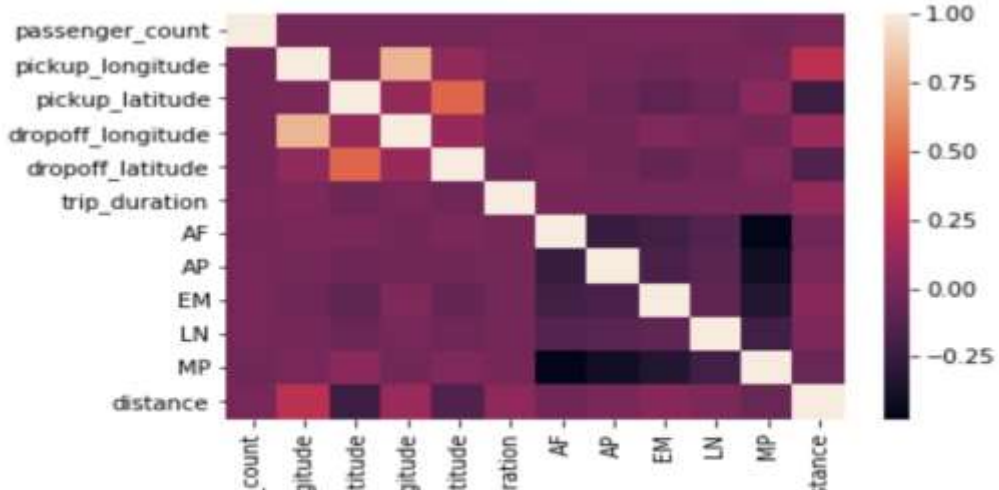
- **To represent the trip duration and LN into seaborn.lmolot**

```
import seaborn as sns
sns.lmplot(x='trip_duration', y='LN', data=taxi_new)
plt.show()
```



Correlation Heatmap

We then run the correlation heatmap on the dataset to see if any of the features are particularly correlated.



We see that except for the location coordinates, features are not particularly correlated to each other, so we are satisfied with the current features.

Normalization

In order to reduce the numeric relationships that may affect the accuracy of our prediction, we normalize the data using preprocessing.scale:

- **Output after normalization our data**

	0	1	2	3	4	5	6	7	8	9	10
0	-0.505637	-0.122261	0.517494	0.124369	0.384575	1.879925	-0.453426	-0.364997	-0.224157	-0.892739	-0.452072
1	-0.505637	-0.097727	-0.375819	-0.368970	-0.575303	-0.531936	-0.453426	2.739745	-0.224157	-0.892739	-0.380622
2	-0.505637	-0.078143	0.395910	-0.451805	-1.162220	-0.531936	-0.453426	-0.364997	-0.224157	1.120148	0.685258
3	-0.505637	-0.515558	-0.941274	-0.549976	-1.256071	1.879925	-0.453426	-0.364997	-0.224157	-0.892739	-0.455103
4	-0.505637	0.006112	1.286091	0.006974	0.855957	-0.531936	-0.453426	-0.364997	-0.224157	1.120148	-0.524207

Reducing Data Samples

After all preprocessing, our data set now contains 10 feature columns, and 1,458,644 rows of data. Due to computational complexity, 1.4 million data samples may take a decent amount of time. For efficiency, we want to first make the data samples smaller to pilot the prediction, and use the results to make adjustments. Then, we can run the prediction with the full data set.

In order to randomly and effectively split the data set into a smaller set, we use our test_split method, with the split size equal to 0.5 — making this a binary split. We split the data set and label half into the training and half into the testing data set, then we just use the training one as our new full data set.

X_train, X_test, y_train, y_test = train_test_split(X_train, y_train,

test_size=0.5, random_state=3)

In order to make splitting more controllable, we make a method as follows:

```
def shrinkDataSet (train,label,times, splitSize):   #times= how many times to split the original dataset
    X_train = train
    y_train = label
    for i in range (0,times):
        X_train, X_test, y_train, y_test = train_test_split(X_train, y_train, test_size=splitSize, random_state=3)
    return X_train, y_train
```

For example, if we split 5 times, this data set with 1.4 million data samples will be randomly reduced to 45582 data samples. With this method, we have more flexibility on running our predictions.

```
# use spliting 5 times as example.

taxi_reduced, label_reduced = shrinkDataSet (taxi_new,label,5,0.5)

print('original: ',taxi_new.shape)
print('After: ',taxi_reduced.shape)
```

```
original:  (1458644, 11)
After:  (45582, 11)
```

Algorithms

Since the dependent variable, trip duration, is a continuous variable, our first choice of algorithms are regression models. In this project, we import 4 different regression models from Sklearn library — Linear regression, Lasso, Ridge, and Elastic Net. We will run these algorithms in different sizes of the data set so that we can compare the size of the data sets and its effects.

We also want to use the ANN Regressor to be a secondary prediction method, so that we don't just use the models from one library.

To verify our prediction, we use RMSE (Root Mean Square Error) as our indicator for how good our model works.

We also use 10-fold cross validation for further RMSE comparison.

Predictions

Linear models

For our first choice, we choose Linear Regression, Lasso, Ridge and Elastic Net as our linear models to predict on different size of the data sets. To make things easier for repeated operations and comparison, we first randomly split the training dataset into 0.8-0.2 train-test ratio, and then run the following method:

```
def Regressions(feature,label):

    # split the dataset into training and testing sets by 80-20 ratio
    X_train, X_test, y_train, y_test = train_test_split(feature, label, test_size=0.2, rando

    #linear
    from sklearn.linear_model import LinearRegression
    myLinearReg = LinearRegression()
    myLinearReg.fit(X_train,y_train)
    y_predict = myLinearReg.predict(X_test)
    print('Linear ',RMSE(y_test,y_predict))

    y_predict = myLinearReg.predict(X_train)
    print('Linear Train',RMSE(y_train,y_predict),'\n')    ## Output the RMSE on the Training

    #Ridge
    from sklearn.linear_model import Ridge
    myRidge = Ridge()
    myRidge.fit(X_train,y_train)
    y_predict = myRidge.predict(X_test)
    print('Ridge ', RMSE(y_test,y_predict))

    y_predict = myRidge.predict(X_train)
    print('Ridge Train', RMSE(y_train,y_predict),'\n')    ## Output the RMSE on the Training

    #ElasticNet
    from sklearn.linear_model import ElasticNet
    myENet = ElasticNet()
    myENet.fit(X_train,y_train)
    y_predict = myENet.predict(X_test)
    print('ElasticNet ', RMSE(y_test,y_predict))

    y_predict = myENet.predict(X_train)
    print('ElasticNet Train', RMSE(y_train,y_predict),'\n')

    #Lasso
    from sklearn.linear_model import Lasso
    myLasso = Lasso()
    myLasso.fit(X_train,y_train)
    y_predict = myLasso.predict(X_test)
    print('Lasso ', RMSE(y_test,y_predict))

    y_predict = myLasso.predict(X_train)
    print('Lasso Train', RMSE(y_train,y_predict),'\n')
```

Using this method, we are able to predict the result, calculate RMSE, and compare the RMSE between testing set and the training set, and compare amongst different regression models.

Let k be the times we perform binary split on the original dataset, k = 0 means not performing the split, so we are using the full dataset.

```
def Run_compare(k):          # K = how many times to split

    taxi_reduced, label_reduced = shrinkDataSet (taxi_new,label, k ,0.5)
    print('Original shape: ',taxi_new.shape)
    print('After shape: ',taxi_reduced.shape,'\n')

    Regressions(taxi_reduced,label_reduced)
```

For k = 8:

```
k=8

Run_compare(k)
```

```
Original shape:  (1458644, 11)
After shape:  (5697, 11)

Linear  1677.38367183
Linear Train 2850.67764143

Ridge  1677.4026249
Ridge Train 2850.6776485

ElasticNet  1716.15264197
ElasticNet Train 2856.9494962

Lasso  1677.07258237
Lasso Train 2850.68223431

cross-validation 2021.76111409
```

For k = 5:

```
k=5

Run_compare(k)
```

```
Original shape:  (1458644, 11)
After shape:  (45582, 11)

Linear  2360.00212068
Linear Train 3139.21822558

Ridge  2359.9389808
Ridge Train 3139.21934679

ElasticNet  2360.50990752
ElasticNet Train 3141.12840396

Lasso  2359.43538113
Lasso Train 3139.65687642

cross-validation 3120.46497092
```

In k = 8, we are fitting the model with around 5000 data sample; in k = 5, we are fitting the model with around 45,000 data samples.

RMSE from k = 8 to k = 5 increases, meaning that error increases, but the RMSE between the test set and training set is getting closer.

We can almost conclude that more samples creates more error, but the model is more effective because the RMSE difference between the testing set and training set is smaller.

For k = 3:

```
k=3

Run_compare(k)
```

```
Original shape:  (1458644, 11)
After shape:  (182330, 11)

Linear  3122.76943374
Linear Train 2993.6234682

Ridge  3122.77069386
Ridge Train 2993.62347391

ElasticNet  3123.88277372
ElasticNet Train 2994.63109028

Lasso  3123.09736192
Lasso Train 2993.82103544

cross-validation 2980.58463961
```

In k = 3, we are fitting the model with 182,000 data samples. In this test, the RMSE between the training set and testing set is very close, both at the 3000 level, and the cross validation RMSE is also in the 3000 level.

We believe it is safe to conclude that at this level, the model is effective.

For k = 0 (full dataset):

```
k=0

Run_compare(k)
```

```
Original shape:  (1458644, 11)
After shape:  (1458644, 11)

Linear  4774.16988657
Linear Train 5317.03045721

Ridge  4774.16895939
Ridge Train 5317.03045751

ElasticNet  4772.98346594
ElasticNet Train 5317.98241472

Lasso  4773.79623661
Lasso Train 5317.15883643

cross-validation 4455.35744246
```

In k = 0, we are fitting the model with full dataset of 1.4 million samples. The RMSE for the testing is at around 4774, while for the training is around 5317, with cross validation RMSE at 4455.

We believe in this case they are still in a relatively close range and the model is still effective. However, it may not be as effective as that of k = 3.

In other words, our regression models, when running the full dataset, provide less effectiveness than running on the smaller dataset.

The possible reason could be that when the number of data samples increases, noise and overfitting increase as well.

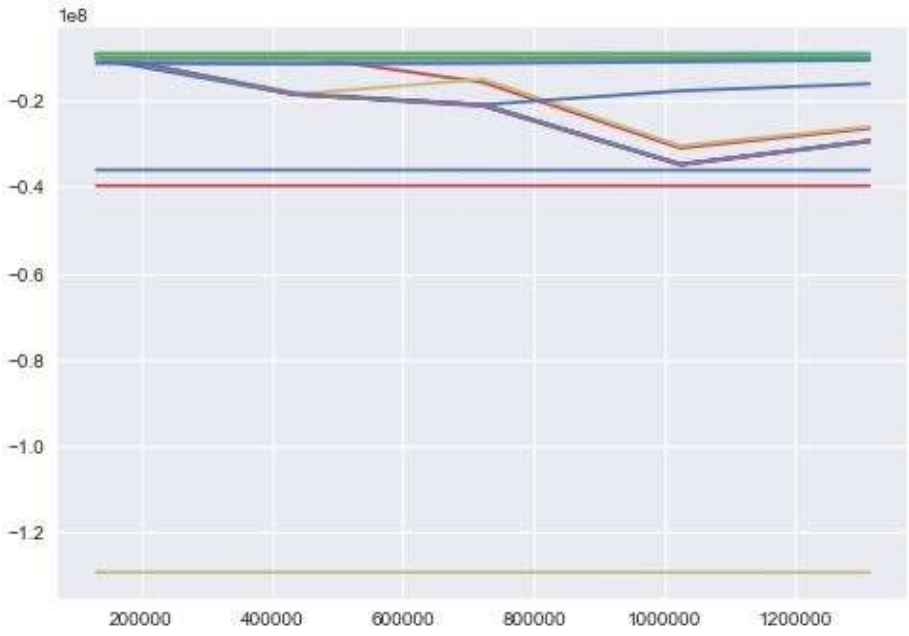
- **Plot Learning Curve for Cross validation:**

  #Plot learning curve

  from sklearn.model_selection import learning_curve

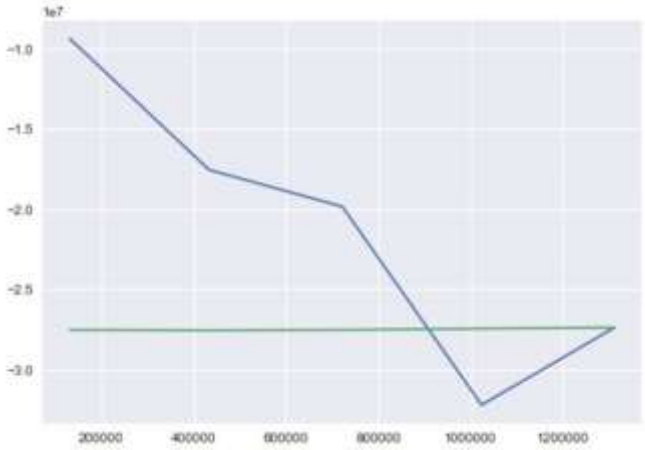
```
train_sizes_abs, train_scores, test_scores = learning_curve(my_linreg ,X, y, n_jobs=-1,cv=10, verbose=0, scoring='neg_mean_squared_error',train_sizes=np.array([ 0.1, 0.33, 0.55, 0.78, 1. ]))
plt.plot(train_sizes_abs, train_scores)
plt.plot(train_sizes_abs, test_scores)
plt.show()
```

- **Output for Plot Learning Curve for Cross validation:**



- **Mean Learning Curves for different Cross Validation Folds**

```
train_score_mean = np.mean(train_scores,axis=1)
test_score_mean = np.mean(test_scores,axis=1)
plt.plot(train_sizes_abs,train_score_mean)
plt.plot(train_sizes_abs, test_score_mean)
plt.show()
```



ANN Regressor

As a secondary method, we use ANN Regressor to run the model and compare the result. This time, we run this prediction with the full dataset.

```
# 1 hidden layer with :
my_ANN = MLPRegressor(activation= 'logistic',
                      solver='adam', alpha=1e-5, random_state=2,
                     learning_rate_init = 0.1)

# RMSE:
rmse_list = cross_val_score(my_ANN, X_train, y_train, cv=10, scoring='neg_mean_squared_error')
print(rmse_list)

# notification when code is done executing
os.system('say "all done"')
```

```
[ -1.08558429e+07  -1.09477374e+07  -9.73776346e+06  -9.44814716e+06
  -8.91627254e+06  -9.67572961e+06  -9.00812540e+06  -1.08122545e+07
  -1.07181409e+07  -1.17131493e+08]
0
```

```
# Notice that "cross_val_score" by default provides "negative" values for "mse" to clarify that
# in order to calculate root mean square error (rmse), we have to make them positive!
mse_list_positive = -rmse_list

# using numpy sqrt function to calculate rmse:
rmse_list = np.sqrt(mse_list_positive)
print(rmse_list)
```

```
[  3294.82061136   3308.73652008   3120.53896941   3073.78385124
   2986.01281626   3110.58348356   3001.35392852   3288.19927723
   3273.85719695  10822.73040848]
```

```
rmse_list.mean()
```

```
3928.0617063086065
```

This second method provides a small RMSE than that of in regression models when running the full dataset. The reason maybe that the ANN algorithms have slightly better ways to avoid overfitting.