Udacity's Machine Learning Engineer Nanodegreee Program
New York City Taxi Trip Duration
Capstone Report

Rodrigo S. Veiga

May 13, 2018

Contents

1 Introduction

The goal of this project is to present a solution for the Kaggle's playground competition New York City Taxi Trip Duration.

The competition dataset is based on the 2016 NYC Yellow Cab trip record data made available in Big Query on Google Cloud Platform. The data was originally published by the NYC Taxi and Limousine Commission (TLC). It was sampled and cleaned for the purposes of the playground competition. Based on individual trip attributes, we should predict the duration of each trip in the test set.

After accepting Kaggle's Terms, the data can downloaded from two zip files, one containing the training dataset and other the testing dataset. Additionally, a sample submission file is provided.

2 Dataset

The data is composed by two data files:

- `train.csv` - the training set (contains 1458644 trip records)
- `test.csv` - the testing set (contains 625134 trip records)

There are eleven features in the training data set.

- `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 because the vehicle did not have a connection to the server -`Y`=store and forward; `N`=not a store and forward trip
- `trip_duration` - duration of the trip in seconds

The last feature, `trip_duration` is the target variable.

Since the training data set provides the column `trip_duration`, the feature `dropoff_datetime` is irrelevant. Any investigation considering specific times and days in order to consider rush hours and holidays in New York City can be performed only with the feature `pickup_datetime`.

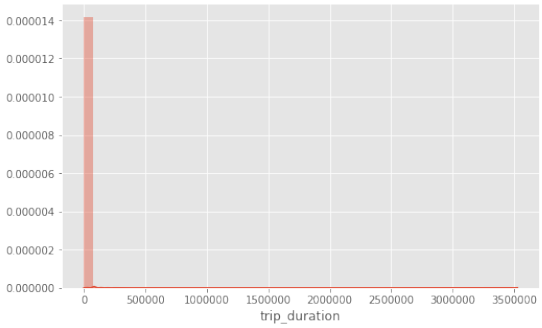
Below we tried to carry a visual analysis considering the dependence of trip duration with the day of the week the trip was take. For this purpose, the use of `datetime` module in the column `pickup_datetime` was crucial to extract interesting information about dates and hours.

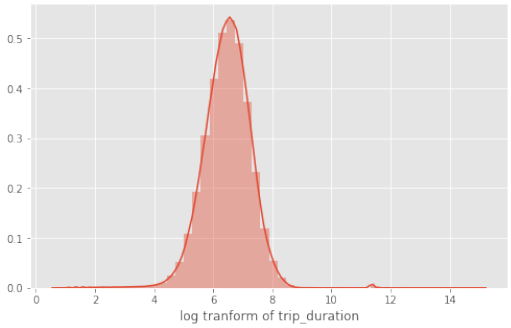
3 Data Exploration

First we take a look at the target variable.

3.1 Trip duration

Clearly the distribution of `trip_duration` is skewed, suggesting we should apply a logarithmic transformation before training.



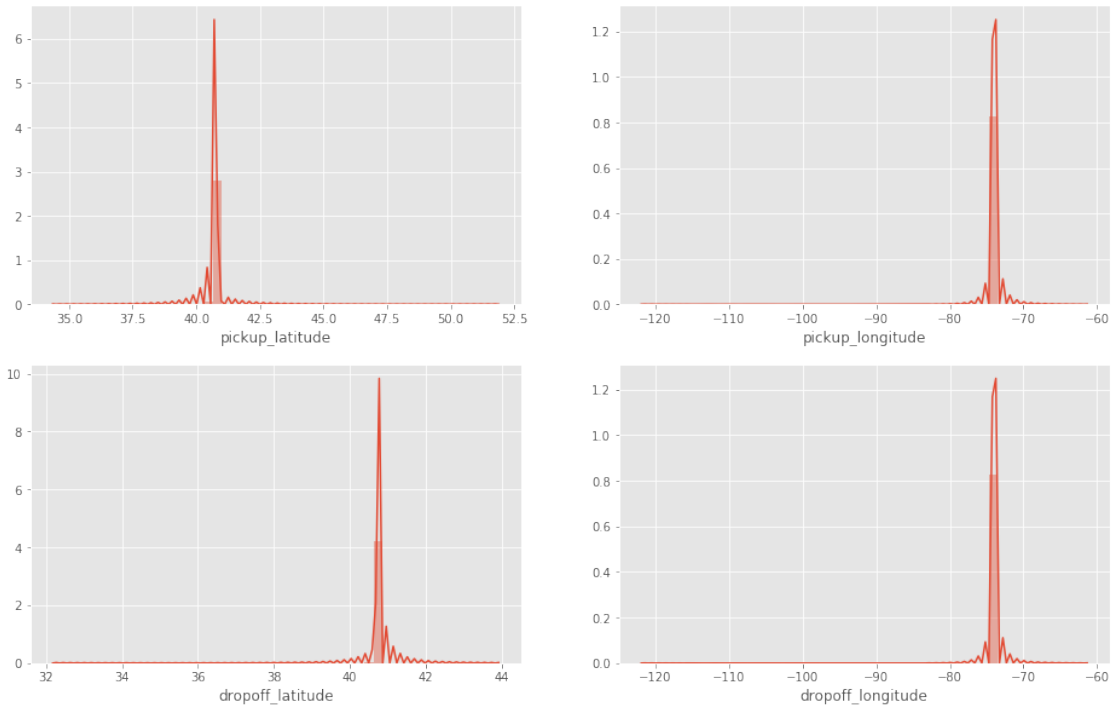


We also should be careful with some meaningless values (outliers), like the maximum value, which would represent a forty days taxi trip around New York. Rush hour cannot be that bad.

```
count    24310.733333
mean        15.991538
std         87.290529
min          0.016667
25%          6.616667
50%         11.033333
75%         17.916667
max      58771.366667
```

3.2 Localization coordinates and travel distances

It is also interesting to take a more careful look in the coordinates values.



Visually, we can infer that most of latitude coordinates vary from 40 to 42, while longitude from -80 to -70. Many interesting map visualization and improved analysis could be carried on with these coordinates, as done in many Kaggle kernels related with this competition). Nevertheless, at least for now, we will keep our model as simple as possible and the information from the four columns above will be merged in just one new feature.

4 Data Preprocessing

Certainly, we have to relate the variables `pickup_longitude`, `pickup_latitude`, `dropoff_longitude` and `dropoff_latitude` with `trip_duration`. First we create an L_1 distance function in two dimensions. This L_1 norm is known as taxicab metric or Manhattan distance.

The function also coverts the respective latitude and longitude differences from degrees to kilometers.

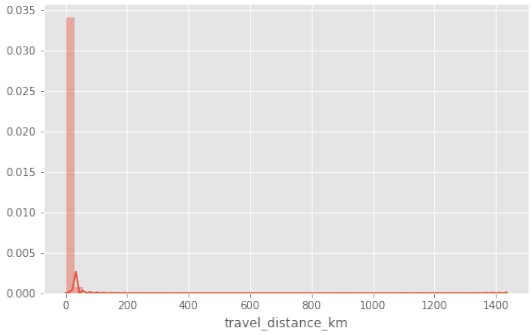
```
def L1_distance_2D(x1, y1, x2, y2):
    # Latitude approximate conversion: degrees to kilometers
    x_conversion = 111
    # Longitude approximate conversion: degrees to kilometers
    y_conversion =  111.321
    # Calculation of the Manhattan distance in kilometers
    dist = x_conversion*np.absolute(x2 - x1) + y_conversion*np.absolute(y2 - y1)
    return dist
```

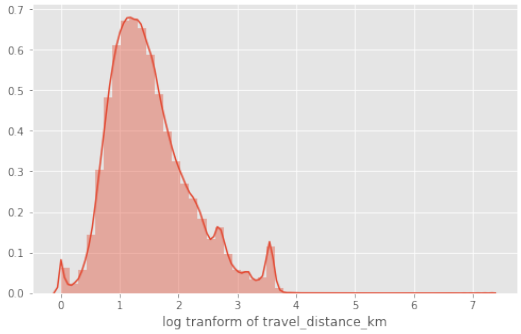
Then we calculate the Manhattan distance for each instance in `train_data` and store the information in a new variable called `travel_distance_km`. After that we will no longer need the coordinates features.

```
x1 = train_data['pickup_latitude']
y1 = train_data['pickup_longitude']
x2 = train_data['dropoff_latitude']
y2 = train_data['dropoff_longitude']

train_data = train_data.drop(columns=['pickup_latitude', 'pickup_longitude',
                                      'dropoff_latitude', 'dropoff_longitude'])

train_data['travel_distance_km'] = L1_distance_2D(x1, y1, x2, y2)
```

Naturally, we do the same for for `test_data`.

Once we have created the new feature `travel_distance_km`, we can analyze its distribution, just like we did for `trip_duration`. Clearly the distribution is skewed, suggesting we should apply a logarithmic transformation before training.





4.1 Meaningless examples

The figures above indicate we should be careful with outliers. From the application of the `describe` method on `train_data` we can see that, strangely, at least one instance has the value zero for `travel_distance_km`.

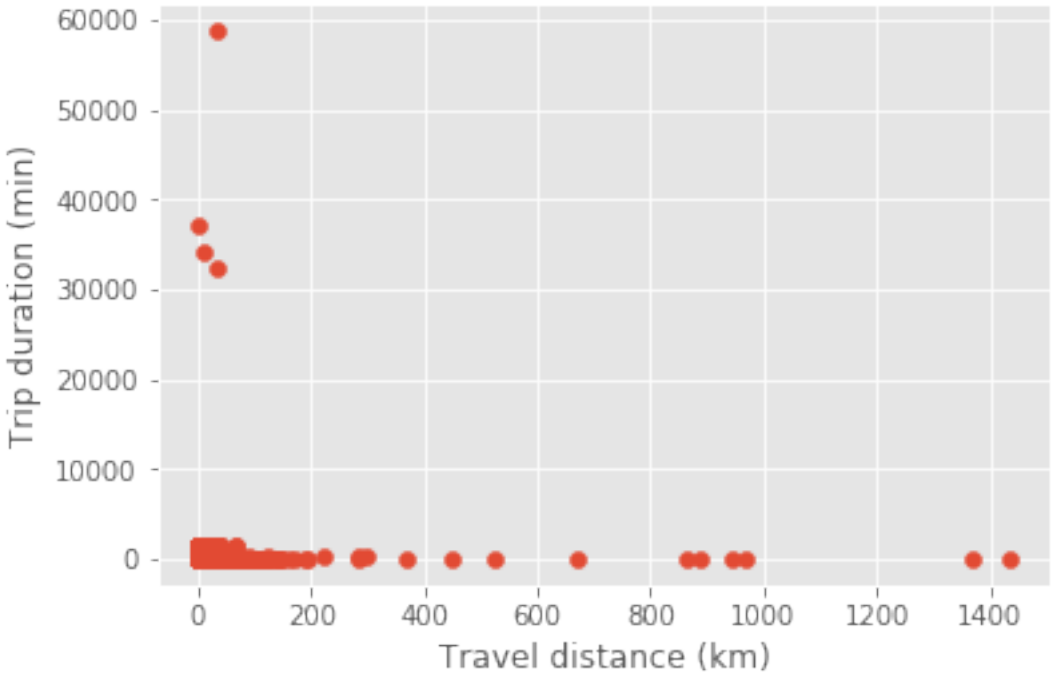
```
       travel_distance_km
count        1.458644e+06
mean         5.102859e+00
std          6.637813e+00
min          0.000000e+00
25%          1.788463e+00
50%          3.043757e+00
75%          5.610539e+00
max          1.435183e+03
```

Taking a careful look on those cases, we note there are several rows with `travel_distance_km` equals to zero.

```
zero_travel_distance = train_data.loc[ train_data['travel_distance_km'] == 0 ]

print(zero_travel_distance.shape[0])
```

```
5897
```

Since those trips do not make sense we will throw them away. However, some data points in the train set are still not reliable, since the magnitude of their values for the feature `trip_duration` is of order of ten thousands minutes.



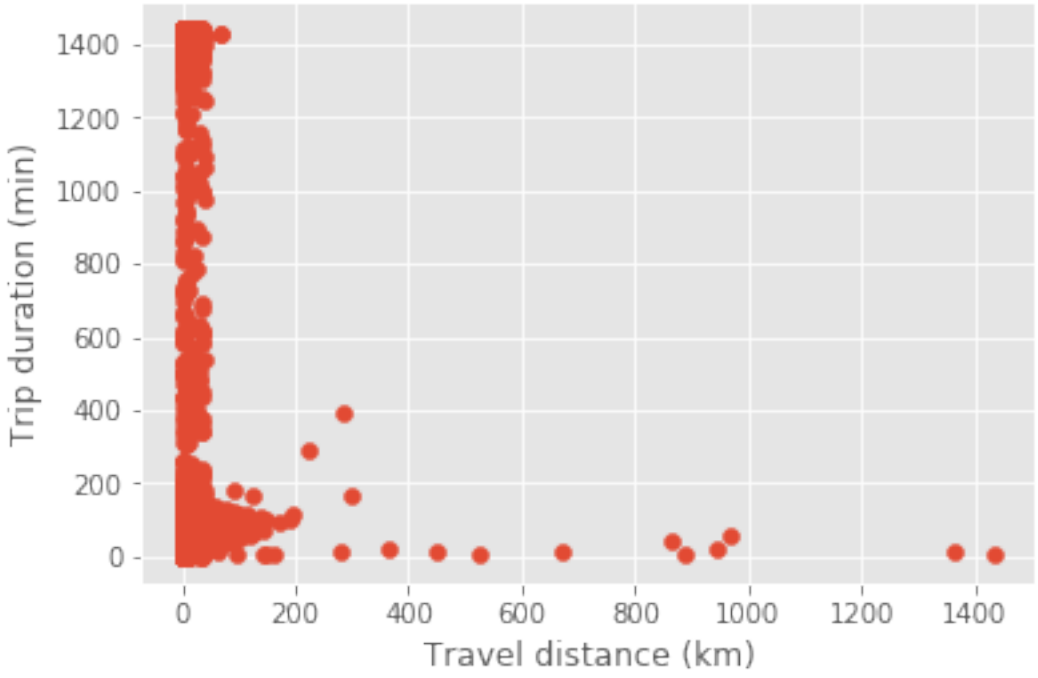
Although the taxi mean speed might hide important information about the city traffic, it is certainly suspicious when its value is less then 0.1 km/h in travels up to 33 km.

```
trip_duration_outliers = train_data.loc[ train_data['trip_duration'] > 10000*60 ]
# Mean speed (km/h)
3600*trip_duration_outliers['travel_distance_km'] / trip_duration_outliers['trip_durati
```

```
id
id1864733    0.061220
id0369307    0.017153
id1325766    0.002934
id0053347    0.033599
```

Because of that we remove from `train_data` the lines shown in `trip_duration_outliers`.

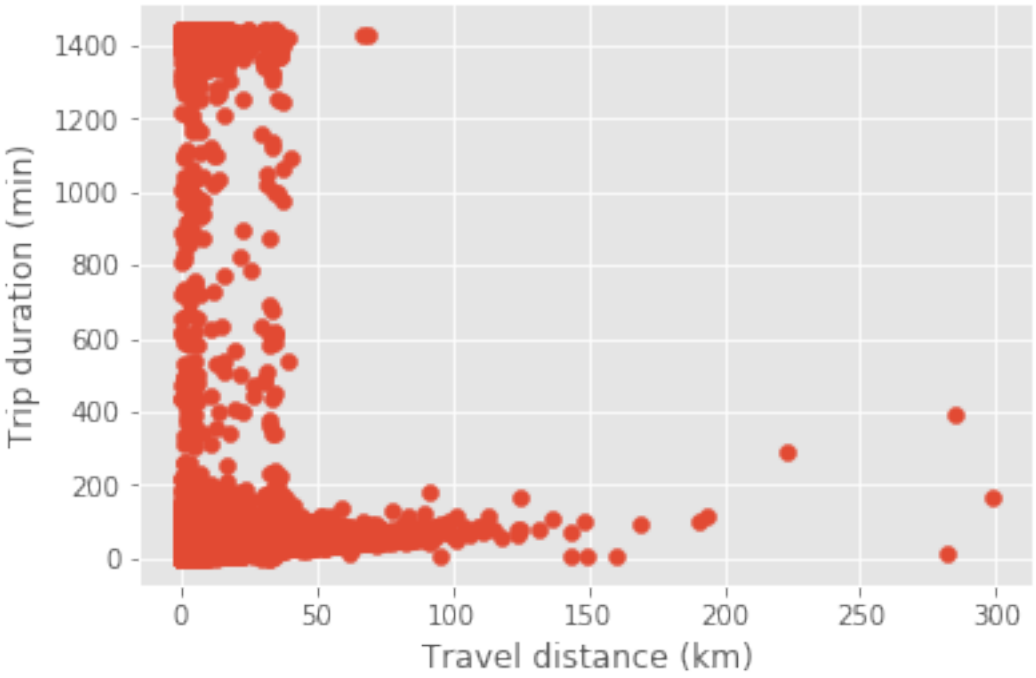
The same analysis can be done looking for non-reliable data points from the point of view of the travel distance, because the mean speed for those point does not make sense either.




```
travel_distance_outliers = train_data.loc[  train_data['travel_distance_km'] > 300]
 # Mean speed (km/h)
 3600*travel_distance_outliers['travel_distance_km'] / travel_distance_outliers['trip_du
```

```
id
 id2306955     6211.686039
 id0978162     1851.813416
 id0116374     6710.694096
 id0982904     1235.223321
 id1146400    10567.491153
 id1001696     2421.114225
 id1510552     8456.072300
 id3626673     2589.658842
 id0838705     1168.140557
 id2644780     1065.878866
```

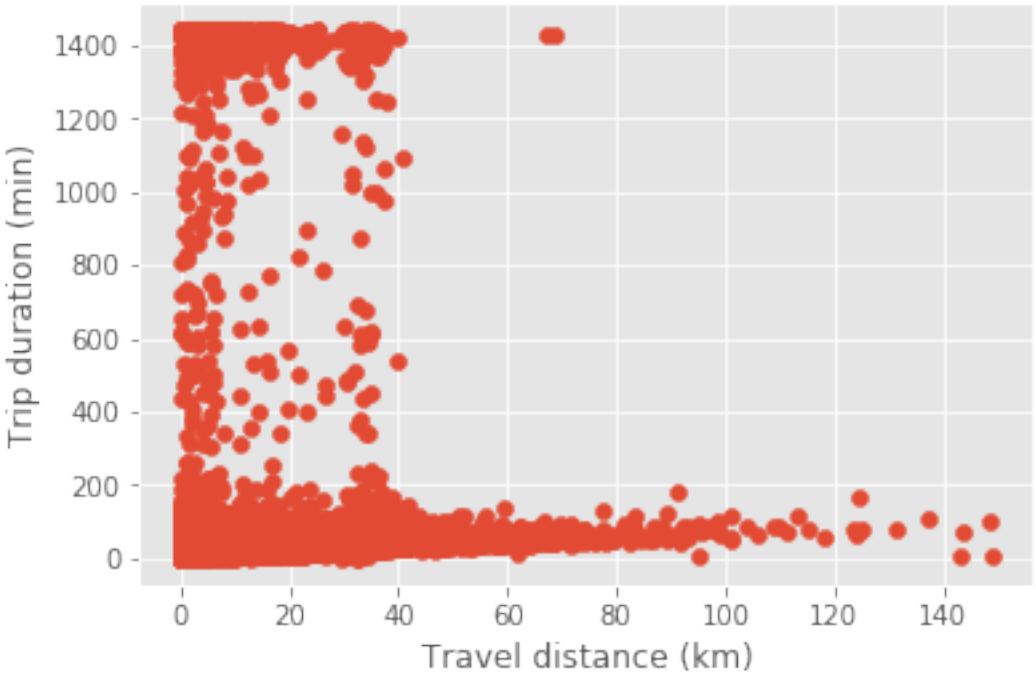
Removing from `train_data` the lines shown in `travel_distance_outliers` we have the new scatter plot.



There are more outliers with inconsistent mean speed values which can be dropped from the training data.

```
more_distance_outliers = train_data.loc[  train_data['travel_distance_km'] > 150]
 # Mean speed (km/h)
 mean_speed_more = 3600*more_distance_outliers['travel_distance_km'] / more_distance_out
 mean_speed_more

id
 id1092161      97.220917
 id1311087      43.787064
 id2778014    1351.784060
 id2066082     109.625331
 id0401529    1261.852560
 id0687776     116.109127
 id1216866     108.340772
 id3795134      46.182152
```

After throwing them away we can another look at the remaining points.



4.2 Rush hour and days in New York City

Naturally, rush hour periods and weekends could change taxi trip duration. Since `pickup_datetime` feature is a `datetime` object, it is not difficult to label each instance with the week day the trip was taken.

```
train_data['week_day'] = train_data['pickup_datetime'].dt.weekday_name
test_data['week_day'] = test_data['pickup_datetime'].dt.weekday_name
```

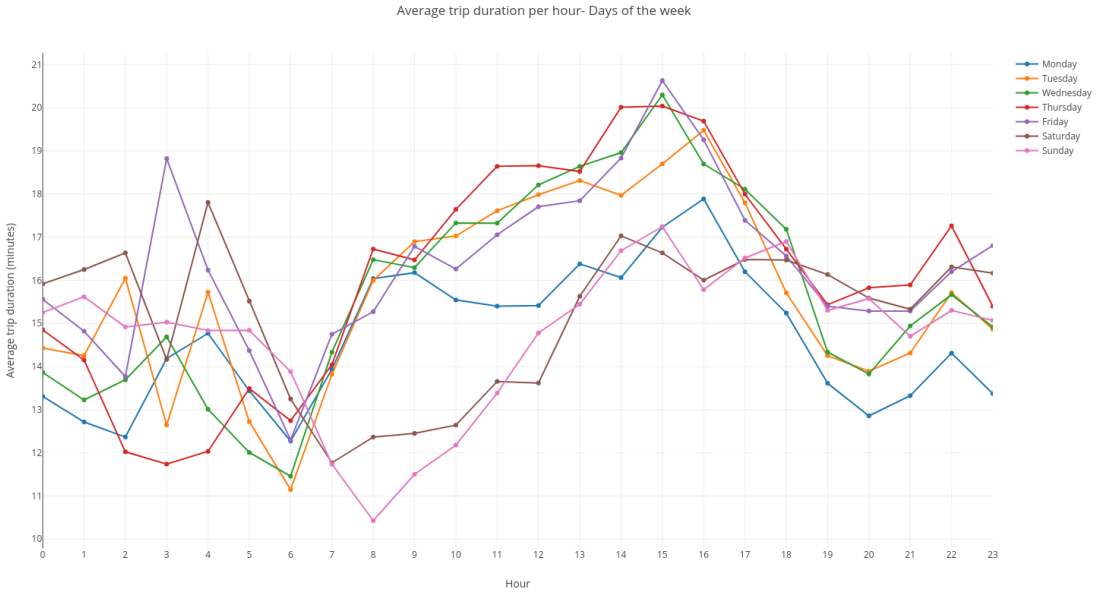
In many cities around the world there are reasonable well defined periods of time called rush hours, on which one should not (if possible) drive into the city. Usually rush hours take place take early in the morning and in the end of afternoon.

In New York City, however, there are longer periods of rush hours. Some people even claim that all times are rush hours. Naturally, another important feature would be the name of the street where the taxi trip begins. Rush hours periods near airports, for example, could be longer. Such deeper analysis could be performed from latitude and longitude coordinates or even with the help of additional datasets, but it will not be carried here.

In order to eventually define a rush hour period and to analyze the behavior of the data for different days of the week, we seek to calculate the average trip duration per hour. First, we extract the hour in which each trip started from the `datetime` objects, and create a new variable called `pickup_hour`.

```
train_data['pickup_hour'] = train_data['pickup_datetime'].dt.hour
test_data['pickup_hour'] = test_data['pickup_datetime'].dt.hour
```

Using pandas `groupby` method we calculate the average values of `trip_duration` for each value of `pickup_hour` and organize them considering each day of the week. The results are then organized in a nice visualization plot using the `plotly` library.



The usual times attributed as rush hours are not clearly depicted in the visualization plot. Actually, one could define rush hour as the period between 8 am and 3 pm, on which there is a tendency to the taxi trip get longer, on average, regardless the day of the week. However, that would be quite an arbitrary choice, since it is based just on a trend. Another one could come up with a different definition.

Although some exciting behaviors could be inferred from the figure, such the ones in the early hours of Fridays and Saturdays, at the dawn of Saturdays and Sundays, the every day peak around 10 pm, we choose to keep all the days and hour variables and do not define any additional label such as rush hour. It appears there is an important correlation between all this features and we hope the chosen regression algorithms are able to grab it.

Since our analysis judges that most of the relevant information provided by the feature `pickup_datetime` can be compressed in `week_day` and `pickup_hour`, we just drop `pickup_datetime`.

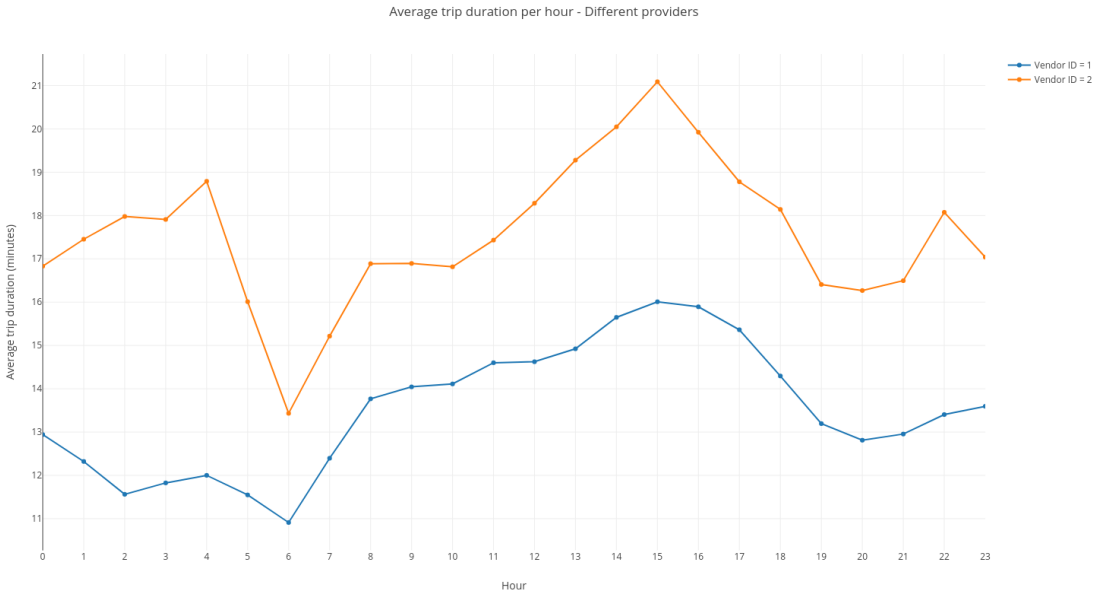
```
train_data = train_data.drop(columns=['pickup_datetime'])
train_data.head()
```

4.2.1 Provider associated with the trip record: `vendor_id`

A naive thinking could conclude that the taxi company choice does not matter for taxi trip duration, since we are counting just the interval between pickup and drop off. Some years ago this could have been true, but nowadays (data is from 2016) there are many technological resources such as Google Maps and Waze that a provider may or may not take advantage. Additionally,

providers which accept payment directly on an smartphone app, similar to Uber, do not loose time after the car reached its final destination.

A similar inspection to one made for `week_day` can be carried for the feature `vendor_id`, which stores an id of the provider associated with the trip record.

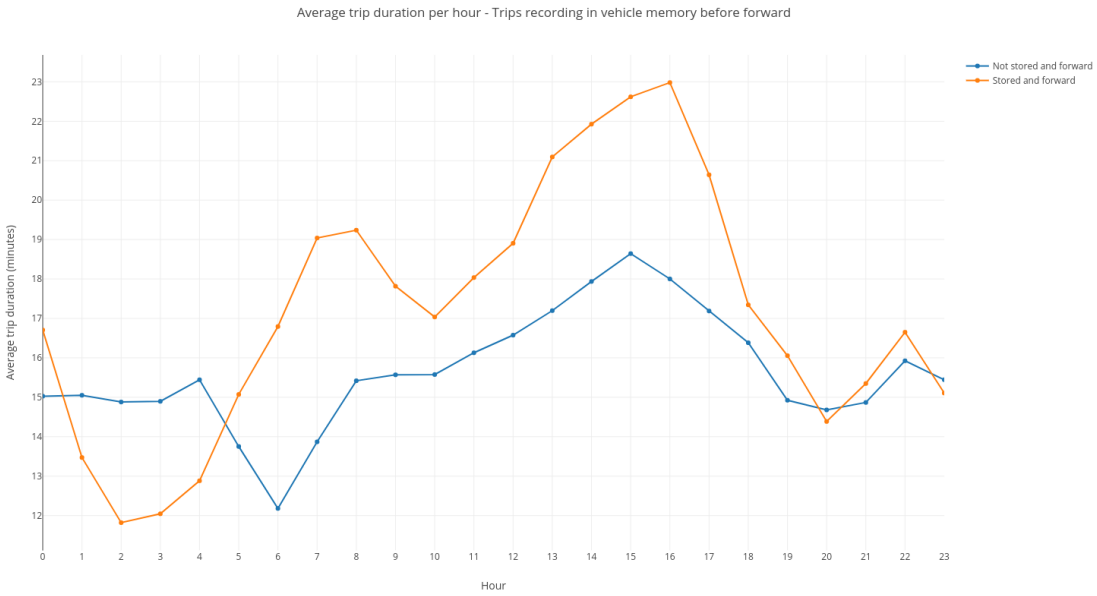


On average it is clear that taxi trips on cars provided by the company identified with number 2 take a little longer, which suggests `vendor_id` feature can play a relevant role in the prediction. Then this feature will be kept.

4.2.2 Trips recorded in vehicle memory before forward: `store_and_fwd_flag`

The feature `store_and_fwd_flag` is a flag indicating whether the trip record was held in vehicle memory before sending to the vendor because the vehicle did not have a connection to the server: `Y` means store and forward while `N`not a store and forward trip.

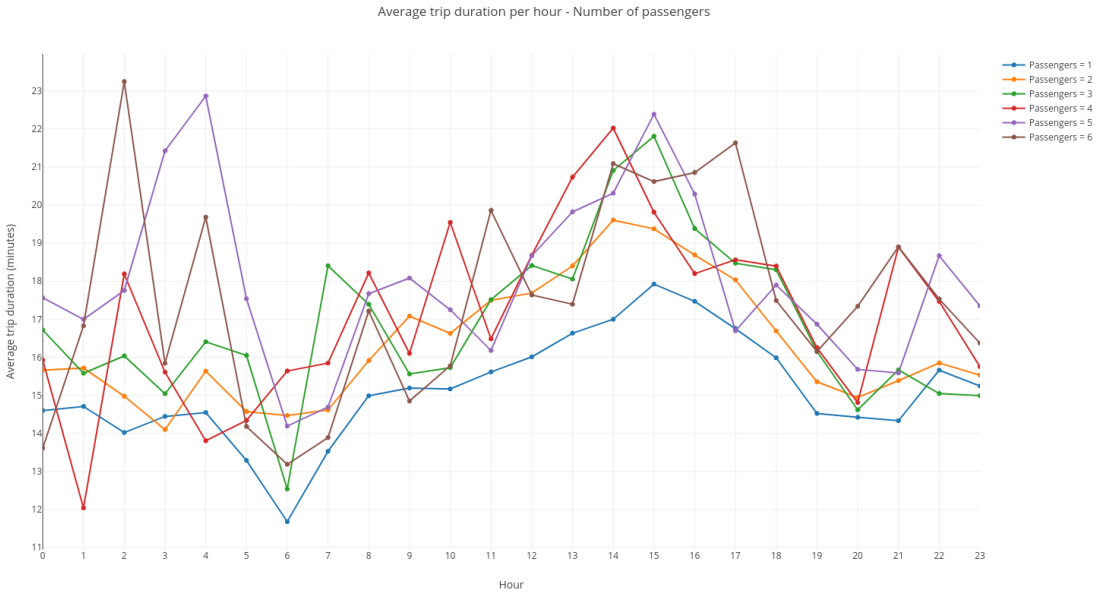
Similarly to the discussion of the previous section, this feature could indicate whether or not a car has technological resources available. A resembling examination can be done for `store_and_fwd_flag`.



Most of the trips, on average, are faster for those cars which do not store information, perhaps indicating they have more technological gadgets available, like wireless connection to the vendor. This suggests `store_and_fwd_flag` may be related with `vendor_id` and may play a relevant role in the prediction. Then this feature will be kept.

4.2.3 The number of passengers in the vehicle: `passenger_count`

Now we perform a similar investigation for the feature which counts the number of passengers the taxi cab was carrying.



The figure above shows a visual indication that there is indeed a correlation between trip duration and number of passengers. Though some fluctuations, the curves display reasonable behaviors, such as the one for single passengers. It is fair to think that a single passenger is faster to get in and get off a car.

4.3 One-hot encoding scheme for categorical variables

The one-hot encoding scheme should be applied on categorical variables. It is important to note even `passenger_count` should be one-hot encoded, because if its values are treated as numbers, the algorithm could consider during training non-integer values for the number of passengers.

```
train_data = pd.get_dummies(train_data, columns=['vendor_id', 'passenger_count',
                                                 'store_and_fwd_flag', 'week_day', 'pickup_hour'])

print(train_data.columns)
```

```
Index(['trip_duration', 'travel_distance_km', 'vendor_id_1', 'vendor_id_2',
       'passenger_count_0', 'passenger_count_1', 'passenger_count_2',
       'passenger_count_3', 'passenger_count_4', 'passenger_count_5',
       'passenger_count_6', 'store_and_fwd_flag_N', 'store_and_fwd_flag_Y',
       'week_day_Friday', 'week_day_Monday', 'week_day_Saturday',
       'week_day_Sunday', 'week_day_Thursday', 'week_day_Tuesday',
       'week_day_Wednesday', 'pickup_hour_0', 'pickup_hour_1', 'pickup_hour_2',
       'pickup_hour_3', 'pickup_hour_4', 'pickup_hour_5', 'pickup_hour_6',
       'pickup_hour_7', 'pickup_hour_8', 'pickup_hour_9', 'pickup_hour_10',
       'pickup_hour_11', 'pickup_hour_12', 'pickup_hour_13', 'pickup_hour_14',
       'pickup_hour_15', 'pickup_hour_16', 'pickup_hour_17', 'pickup_hour_18',
       'pickup_hour_19', 'pickup_hour_20', 'pickup_hour_21', 'pickup_hour_22',
       'pickup_hour_23'],
      dtype='object')
```

Naturally the same is done for `test_data`.

4.4 Logarithmic transformation

Accordingly to the data exploration part, before training we apply a logarithmic transformation in the numerical variables.

```
train_data['trip_duration'] = np.log(train_data['trip_duration'] + 1)
train_data['travel_distance_km'] = np.log(train_data['travel_distance_km'] + 1)

test_data['travel_distance_km'] = np.log(test_data['travel_distance_km'] + 1)
```

To avoid mistakes, we then rename the columns pointing explicitly they are been measured on a logarithmic scale.

4.5 Creating `X_train`, `y_train`

Before training the algorithms, we create the `X_train` and `y_train` objects, while the later is defined to be the training target variable.

```
y_train = train_data['log_trip_duration']
X_train = train_data.drop(columns=['log_trip_duration'])
```

5 Training: estimating Kaggle score using `train_test_split` from `sklearn.model_selection`

The competition does not provide direct access to the test targets, then we are not able to calculate here the root mean square logarithmic error (RMSLE) ϵ, which is given by

$$\epsilon = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\log(p_i+1) - \log(a_i+1))^2}\,,$$

where n is the total number of observations in the test set, p_i is the i-th prediction of trip duration, and a_i is the i-th actual trip duration.

The RMSLE is similar to square root of the mean square error, which is the simplest evaluation metric and it has a straightforward interpretation; it estimates the variance between the predicted instances and correct ones. The lower this variance, the better the prediction. Mean square logarithmic error is basically the same measure but in a logarithmic scale. It is interesting if one does not want to penalize too much huge differences between predictions and actual values.

Our score is reveled only when we make a submission to Kaggle.

Naturally, it is not practical to make cross validation submitting files in the competition's web site. Using `train_test_split` from `sklearn.model_selection` we can estimate the final Kaggle score by selecting a fictitious test target from the training data.

```
from sklearn.model_selection import train_test_split

XX_train, XX_test, yy_train, yy_test = train_test_split(X_train,
                                                        y_train,
                                                        test_size = 0.3,
                                                        random_state = 0)
```

Next we define a score function to perform cross validation.

```
def score_kaggle_log(y_true, y_pred):
    y_pred_t = np.exp(y_pred) - 1
    y_true_t = np.exp(y_true) - 1
    e_i = np.square( np.log(y_pred_t + 1) - np.log( y_true_t + 1) )
    score = np.sqrt( (1/len(y_true_t)) * np.sum(e_i) )
    return score
```

The `score_kaggle_log` function just applies the formula for the root mean square logarithmic error (RMSLE), which is the evaluation metrics considered in the competition. Note that, since we had applied a logarithmic transformation in the numerical variables, we have put the inverse log-transformation inside the score function.

5.1 Linear Regression

It is convenient to begin training models as simple as possible. Usually this approach is helpful to give insights about which more complicated path is worth to follow. Here we choose to start with the usual multivariate linear regression with its standard parameters.

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
clf = model.fit(XX_train, yy_train)
print('Estimating RMSLE on the test set with LinearRegression')
print(score_kaggle_log( yy_test, clf.predict(XX_test) ) )
```

```
Estimating RMSLE on the test set with LinearRegression
0.49269888205163326
```

If we take a look also in the RMSLE on the training set, there is no sign of overfitting, suggesting that regularization methods will not considerably improve the results.

```
print(score_kaggle_log( yy_train, clf.predict(XX_train) ) )
```

```
0.4940908409128837
```

5.2 Linear regression with regularization: `Lasso`

However, since some generalized linear models are straightforward, we will take a quick look at some regularized models. We begin by a multivariate linear model with L1 penalty, which is a penalty equals to the absolute value of the magnitude of coefficients. In scikit-learn this modified model is imported as `Lasso`.

```
from sklearn.linear_model import Lasso
model = Lasso(random_state=0)
clf = model.fit(XX_train, yy_train)
print('Estimating RMSLE on the test set with Lasso')
print(score_kaggle_log( yy_test, clf.predict(XX_test) ) )
```

```
Estimating RMSLE on the test set with Lasso
0.7835251024842801
```

Since L1 regularization limits the size of the coefficients, some of them can become zero, therefore being eliminated. That is why this type of model usually works well on sparse data, i.e. data with few relevant coefficients. That is not the case here. Definitely, `Lasso` is eliminating relevant coefficients during the training procedure, leading to a poor performance.

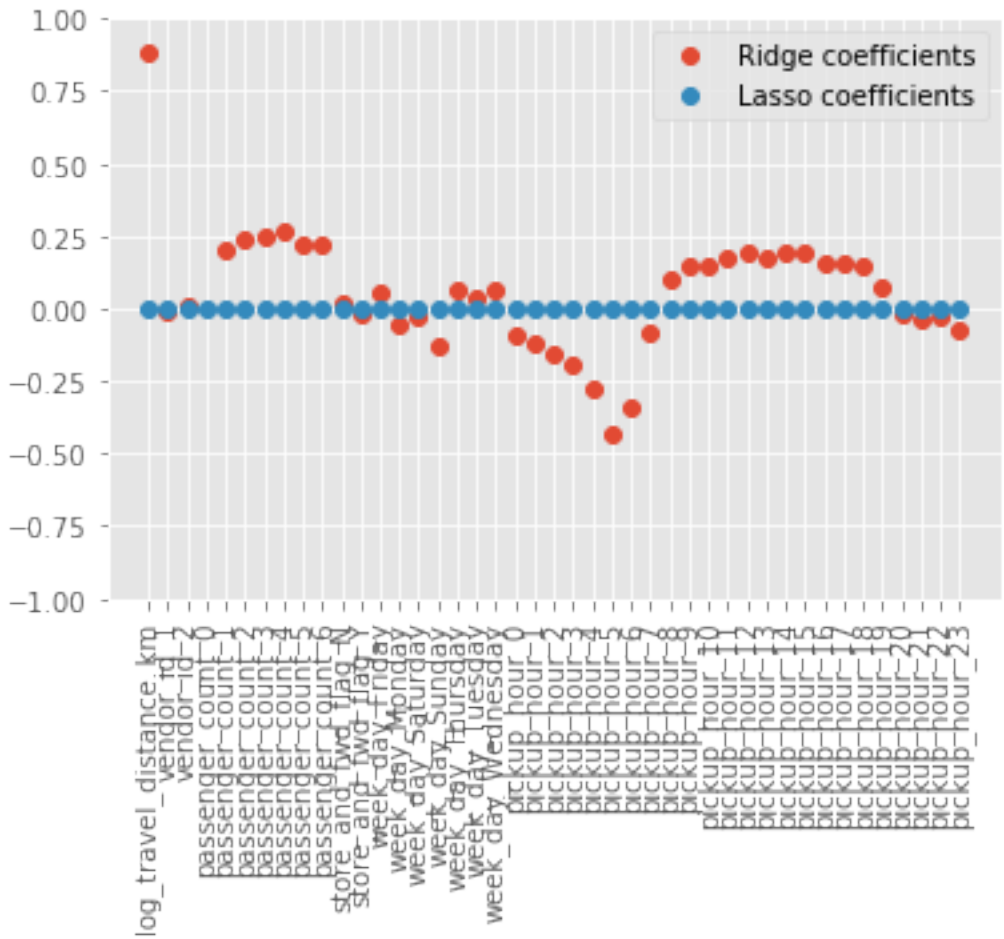
5.3 Linear regression with regularization: `Ridge`

Now we considerer L2 regularization, which is a penalty equals to the square of the magnitude of coefficients. As the square of a small number is a smaller number, we do not expect to have as much null coefficients as we had obtained with L1 regression. In scikit-learn a linear model with L2 regularization is imported as `Ridge`.

```
In [57]: from sklearn.linear_model import Ridge
         model = Ridge(random_state=0)
         clf = model.fit(XX_train, yy_train)
         print('Estimating RMSLE on the test set with Ridge')
         print(score_kaggle_log( yy_test, clf.predict(XX_test) ) )
```

```
Estimating RMSLE on the test set with Ridge
0.4926978071042146
```

With the method `.coef_` we can compare Ridge and Lasso coefficients. Regularization L1 leads to zero coefficients for all features, while L2 regularization is able to construct a model with more complexity.



Between `Lasso` and `Ridge`, it is clean we should go on with the later.

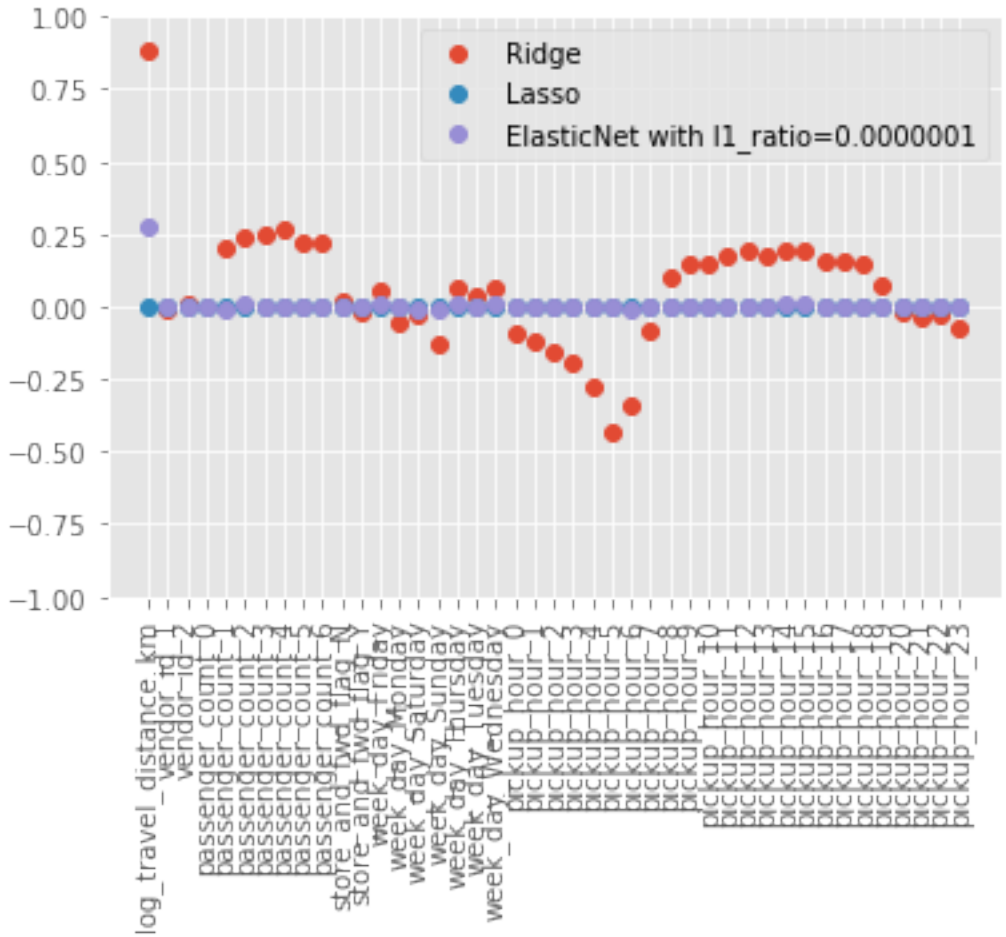
5.4 Linear regression with regularization: `ElasticNet`

If we had to chose one kind of regularization, there is no doubt about what penalty should be chosen. Just for pedagogical purposes, we also present the method `ElasticNet` which combines L1 and L2 regularizations. An interesting parameter is `l1_ratio` which is a mixing parameter; for `l1_ratio = 0` the penalty is an L2 penalty. For `l1_ratio = 1` it is an L1 penalty and for `0 < l1_ratio < 1`, the penalty is a combination of L1 and L2.

Interestingly, a tiny contribution of an L1 term is enough to a poor generalization.

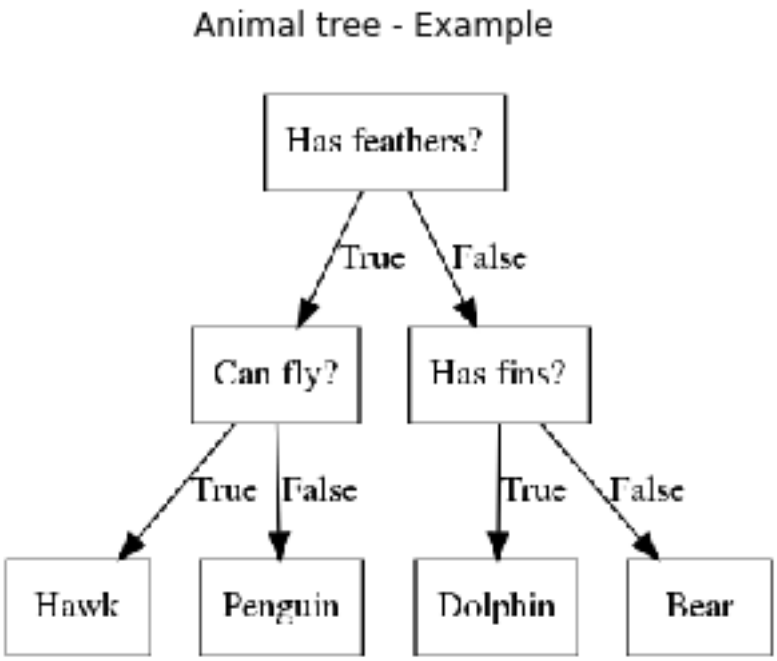
```
from sklearn.linear_model import ElasticNet
model = ElasticNet(random_state=0, l1_ratio=0.0000001)
clf = model.fit(XX_train, yy_train)
print('Estimating RMSLE on the test set with ElasticNet')
print(score_kaggle_log( yy_test, clf.predict(XX_test) ) )
```

```
Estimating RMSLE on the test set with ElasticNet
0.651556924312904
```

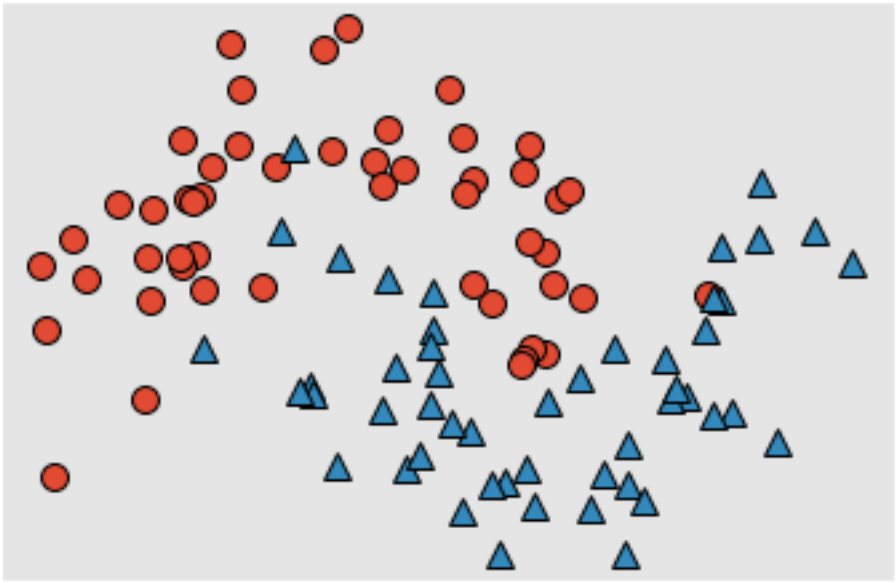


5.5 Decisions Trees

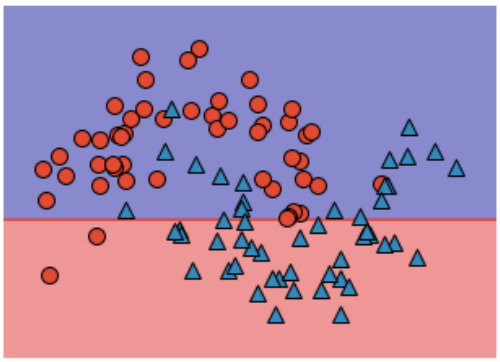
Essentially, decision trees learn a hierarchy of if-else questions, leading to a decision. In the following illustration, taken from the book Introduction to Machine Learning with Python by Andreas Mueller and Sarah Guido each node in the tree either represents a question or a terminal node (also called a leaf) which contains the answer.

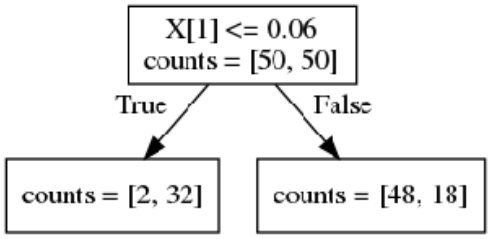


Decision trees may be very intuitive for classification tasks, but one may be wondering how are the questions for continuous data, such as ours. In this they are of the form "is feature i larger than value a?". To build a tree, the algorithm searches over all possible questions and find the most informative one about the target variable. The book Introduction to Machine Learning with Python by Andreas Mueller and Sarah Guido provides a nice visualization of this procedure.

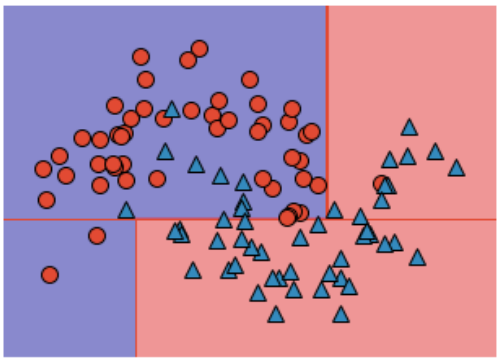


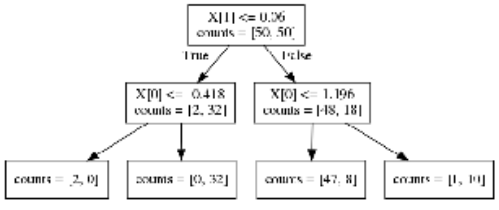
depth = 1

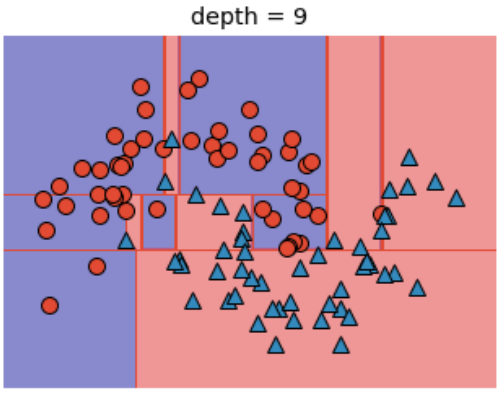


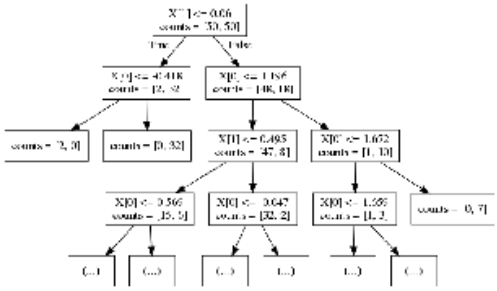


depth = 2









Typically, building a tree as described and continuing until all leaves are pure leads to very complex models and highly overfit the training data. There are to ways to prevent overfitting:

- Pre-pruning: stop the creation of the tree before it is ended.
- Post-pruning or pruning: build the tree, but removing or collapsing nodes with little information.

The scikit-learn library implements only pre-pruning, on which one could limit he maximum depth of the tree by the parameter `max_depth` or even limit the maximum number of leaves with `max_leaf_nodes`.

As decision trees methods are widely used for classification and even regression tasks, we will give a shot here with `DecisionTreeRegressor` algorithm. There are several important parameters such as `max_depth`, mentioned above

As decision trees methods are widely used for classification and even regression tasks, we will give a shot here with `DecisionTreeRegressor` algorithm. There are several important parameters such as `max_depth` mentioned above and `min_samples_split`, which controls the minimum number of samples required to split an internal node. For now, we will keep all parameters in their default values.

```
from sklearn.tree import DecisionTreeRegressor
model = DecisionTreeRegressor(random_state= 0,max_depth= None, min_samples_split= 2)
clf = model.fit(XX_train, yy_train)
print('Estimating RMSLE on the test set with DecisionTreeRegressor')
print(score_kaggle_log( yy_test, clf.predict(XX_test) ) )
```

```
Estimating RMSLE on the test set with DecisionTreeRegressor
0.661230882441467
```

We note `Ridge` regression is still better. As $\approx$0.66 is quite far form $\approx$0.49, instead of try to tune the hyperparameters of `DecisionTreeRegressor`, we decide to analyze ensemble methods, which combine multiple models to create more powerful models.

5.5.1 Ensembles of Decision Tress

There are some models in this category, but three of them have proven to be effective on a wide range of data set for classification and regression and they use decision trees on the building block: Random Forests, Gradient Boosted Decision Trees and Extreme Gradient Boosting.

Random Forests Random forests are essentially a collection of decision trees, where each tree is slightly different from the others. The idea: each tree might do a relatively good prediction job, but will likely overfit on part of the data. If many trees are built, all of which working well and overfitting in different ways, the amount of overfitting can be reduced by averaging the results.

The parameter `n_estimator` controls the number of trees in the random forest.

Each tree is built with a bootstrap sample of the data. A bootstrap sample is constructed by random sampling with replacement of the data points. Then bootstrapping creates a data set as big as the original one, but some data points will be missing from it and some will be repeated. Next, a decision tree is built based on each new data set created by bootstrapping. However,in this context, the decision tree algorithm is slightly modified. Before, it searched for the best test for each node. In this context, it randomly selects in each node a subset of the features and looks for the best possible test involving one of these features. The quantity of selected features is controlled by the `max_features` parameter. This selection is repeated in each node, so each of them make a decision using a different features subset.

Note `max_features` is a critical parameter. If `max_features = n_features` no randomness is injected, while if `max_features = 1` the algorithm has no choice of which feature to test and can only search for thresholds for the randomly selected feature.

Below we aplly `RandomForestRegressor`. For now, we will keep all parameters in their default values.

```
from sklearn.ensemble import RandomForestRegressor
model = RandomForestRegressor(random_state=0)
clf = model.fit(XX_train, yy_train)
print('Estimating RMSLE on the test set with RandomForestRegressor')
print(score_kaggle_log( yy_test, clf.predict(XX_test) ) )
```

```
Estimating RMSLE on the test set with RandomForestRegressor
0.5239454227464584
```

The result provided by `RandomForestRegressor` with its default hyperparameters is already much better than the one obtained by `DecisionTreeRegressor`. Perhaps we can get smaller score values if we try to tune the random forest parameters using cross-validation.

Gradient Boosted Regression Trees Gradient boosting models were first proposed in the paper Greedy Function Approximation: A Gradient Boosting Machine, by Jerome H. Friedman. Both Gradient Boosting Regression Trees and XGBoost mentioned in this work are based on this original proposal.

Gradient Boosted Regression Trees (or Gradient Boosted Machines) is another ensemble method combining multiple decision trees to a more powerful model. Despite its name, it can be used for regression and classification.

It works by building trees in a serial manner, where each tree tries to corrected the mistakes of the previous one. There is no randomization. Instead, strong pre-pruning is used. It often uses very shallow trees, of depth one to five, making a faster model with less memory use.

The main idea of gradient boosted: combine many simple models (known as weak learners) like shallow trees. Each tree can provide good predictions only on part of the data, and more and more trees are added to iteratively improve performance.

They are generally a bit more sensitive to parameter settings than random forests, but can provide better accuracy if the parameters are set correctly (pre-pruning, number of trees in the ensemble and learning rate are the model most important features).

The `learning_rate` parameter controls how strongly each tree tries to correct the mistakes of the previous ones. High `learning_rate` means each tree make strong corrections, allowing for a more complex model. Similarly, adding more trees to the ensemble, `n_estimators` also increases model complexity.

Below we aplly `GradientBoostingRegressor`. For now, we will keep all parameters in their default values.

```
from sklearn.ensemble import GradientBoostingRegressor
model = GradientBoostingRegressor(random_state=0)
clf = model.fit(XX_train, yy_train)
print('Estimating RMSLE on the test set with GradientBoostingRegressor')
print(score_kaggle_log( yy_test, clf.predict(XX_test) ) )
```

```
Estimating RMSLE on the test set with GradientBoostingRegressor
0.471361235487172
```

Without changing any of the hyperparameters default values, `GradientBoostingRegressor` achieved the lowest score value so far. Certainly, we will do cross-validation with this algorithm.

Extreme Gradient Boosting: XGBoost XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way. The same code runs on major distributed environment (Hadoop, SGE, MPI) and can solve problems beyond billions of examples.

The `xgboost` package is worth looking to apply gradient boosting to a large scale problem, because it is usually faster and sometimes easier to tune than the scikit-learn implementation of gradient boosting on many data sets. Below we first apply `XGBRegressor` with its parameters set in their default values.

```
import xgboost as xgb
model = xgb.XGBRegressor(random_state=0)
clf = model.fit(XX_train, yy_train)
print('Estimating RMSLE on the test set with XGBRegressor')
print(score_kaggle_log( yy_test, clf.predict(XX_test) ) )
```

```
Estimating RMSLE on the test set with XGBRegressor
0.47136019069697366
```

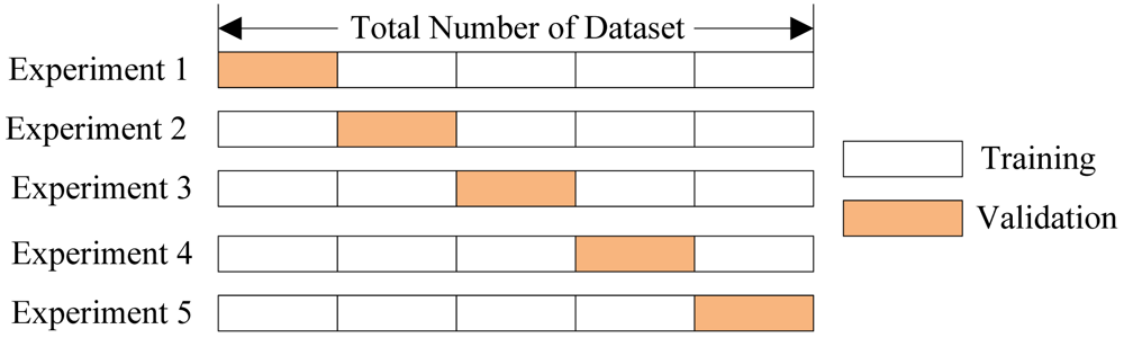
The performance of `XGBRegressor` is almost the same of `GradientBoostingRegressor`, which indicates it is worth tuning the hyperparameters of these two models. Since `Ridge` did not do a bad job either, we will work on tuning its parameters as well.

5.6 Cross-validation and Grid Search

Cross-validation is a model validation technique for assessing how the results of a statistical analysis will generalize to an independent data set, preventing overfitting. Note that cross-validation does not measure generalization, it just tries to estimate from the training set itself how well the model will generalize.

In cross-validation, the same model is run several times using a fraction of the training set and storing what is left as a validation set, which is used to evaluated the performance of the model. This is procedure is run several times, until all the training data have played the part of a validation set. The cross-validation resulting model is then the average of the ones computed in the loop.

The figure below, taken from this tutorial by DanB illustrates the procedure for 5 folds.



Additionally to cross-validation, we use grid-search to estimate the best hyperparameters to generalization. It is simply an exhaustive searching through a manually specified subset of the hyperparameter space of a learning algorithm. A grid search algorithm must be guided by some performance metric, typically measured by cross-validation on the training set or evaluation on a held-out validation set.

For example, let us suppose a machine learning model has at least two hyperparameters that need to be tuned for good performance on unseen data, namely A and B. To perform grid search, suppose one selects a finite set of "reasonable" values for each, say $A \in \{a_1, a_2, a_3\}$ and $b \in \{b_1, b_2, b_3\}$. Grid search then trains the model with each pair (A, B) in the Cartesian product of these two sets and evaluates their performance on a held-out validation set (or by internal cross-validation on the training set). Finally,the grid search algorithm outputs the settings that achieved the highest score in the validation procedure.

In scikit-learn grid-search and cross-validation are applied simultaneously with `GridSearchCV`.

5.6.1 Cross-validation and Grid-search with Ridge Regression

First we perform cross-validation with the Ridge Regression by changing the hyperparameter `alpha`, which controls the regularization strength (default=1.0). We also investigate the performance for different choices of `solver`:

- `auto`: chooses the solver automatically based on the type of data.
- `svd`: uses a Singular Value Decomposition of X to compute the Ridge coefficients.
- `lsqr`: uses the dedicated regularized least-squares routine scipy.sparse.linalg.lsqr. It is the fastest but may not be available in old scipy versions. It also uses an iterative procedure.
- `sag`: uses a Stochastic Average Gradient descent.

```
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer
# greater_is_better= False: we want to minimize the root mean square logarithmic error
scorer = make_scorer(score_kaggle_log, greater_is_better= False)
model = Ridge(random_state=0)
parameters={'alpha': [0.01, 0.1, 1.0, 10.0], 'solver': ['auto', 'lsqr', 'sag', 'svd']}
clf = GridSearchCV(model, param_grid= parameters, scoring= scorer,  cv= 3)
grid_fit = clf.fit(XX_train, yy_train)
print(grid_fit.best_params_)
best_clf = grid_fit.best_estimator_
print(score_kaggle_log( yy_test, best_clf.predict(XX_test) ) )
```

```
Fitting 3 folds for each of 16 candidates, totalling 48 fits

[Parallel(n_jobs=-1)]: Done  48 out of  48 | elapsed:  1.7min finished

{'alpha': 0.1, 'solver': 'auto'}
0.492698026491787
```

The cross-validation procedure suggests we should have set `alpha=0.1` instead of `alpha=1.0` (0.4926978071042146) in order to improve the performance based on the root mean square logarithmic error. The gain is quite small, but in a Kaggle competition could represent several positions on the leaderboard.

5.6.2 Cross-validation and Grid-search with Random Forest Regression

Now we check the behavior of `RandomForestRegressor` by changing the following hyperparameters:

- `max_depth` : the maximum depth of the tree (default=`None`). If `None`, then nodes are expanded until all leaves are pure or until all leaves contain less than `min_samples_split` samples.
- `n_estimators` : the number of trees in the forest (default=10).
- `min_samples_split`: the minimum number of samples required to split an internal node (default=2)
- `max_features` : the number of features to consider when looking for the best split (default=`auto`). If `auto`, then `max_features=n_features`, if `sqrt`, then `max_features=sqrt(n_features)`.

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer
# greater_is_better= False: we want to minimize the root mean square logarithmic error
```

```
scorer = make_scorer(score_kaggle_log, greater_is_better= False)
model = RandomForestRegressor(random_state=0)
parameters = {'max_depth': [10, 15], 'n_estimators': [20, 30],
              'min_samples_split': [6, 10], 'max_features': ['auto', 'sqrt']}
clf = GridSearchCV(model, param_grid= parameters, scoring= scorer, verbose= 1, cv= 2)
grid_fit = clf.fit(XX_train, yy_train)
print(grid_fit.best_params_)
best_clf = grid_fit.best_estimator_
print(score_kaggle_log( yy_test, best_clf.predict(XX_test) ) )
```

```
Fitting 2 folds for each of 16 candidates, totalling 32 fits

[Parallel(n_jobs=1)]: Done  32 out of  32 | elapsed: 19.3min finished

{'max_depth': 15, 'max_features': 'auto', 'min_samples_split': 10, 'n_estimators': 30}
0.4724815086016824
```

Varying three hyperparameters considerably improved the previous score (0.5239454227464584). Although cross-validation has indicated we should keep `max_features = n_features` (no randomness is injected), the other tree hyperparameters should have been changed: the maximum depth of tree increase limited to 15, the minimum number of samples required to split an internal node raised from 2 to 10 (more complexity) and the number of trees in the forest increased from 10 to 20. The RMSLE score for the fictitious test set obtained by the random forest algorithm with these hyperparameters is the best we have obtained so far.

5.6.3 Cross-validation and Grid-search with Gradient Boosting Regression

Now we check the behavior of `GradientBoostingRegressor` by changing the following hyperparameters:

- `max_depth` : maximum depth of the individual regression estimators (default=3). The maximum depth limits the number of nodes in the tree. The best value depends on the interaction of the input variables.
- `n_estimators` : the number of boosting stages to perform (default=100). Gradient boosting is fairly robust to overfitting so a large number usually results in better performance.
- `min_samples_split`: the minimum number of samples required to split an internal node (default=2)
- `leanirng_rate`: shrinks the contribution of each tree (default0.1). There is a trade-off between `learning_rate` and `n_estimators`.

```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer
# greater_is_better= False: we want to minimize the root mean square logarithmic error
scorer = make_scorer(score_kaggle_log, greater_is_better= False)
model = GradientBoostingRegressor(random_state=0)
parameters = {'max_depth': [3, 5], 'n_estimators': [100, 200],
              'min_samples_split': [2, 6], 'learning_rate': [0.1, 1.0]}
clf = GridSearchCV(model, param_grid= parameters, scoring= scorer, verbose= 1, cv= 2)
```

```
        grid_fit = clf.fit(XX_train, yy_train)
        print(grid_fit.best_params_)
        best_clf = grid_fit.best_estimator_
        print(score_kaggle_log( yy_test, best_clf.predict(XX_test) ) )
```

```
Fitting 2 folds for each of 16 candidates, totalling 32 fits

[Parallel(n_jobs=1)]: Done  32 out of  32 | elapsed: 86.5min finished

{'learning_rate': 0.1, 'max_depth': 5, 'min_samples_split': 2, 'n_estimators': 200}
0.46219359665138765
```

Varying two hyperparameters slightly improved (≈1.95%) the previous score (0.471361235487172). As mentioned before, such a gain could not appear to be worthwhile, especially if we compare the training times so far (note `GradientBoostingRegressor` took much more time to train). Yet 2% in a competition could represent several positions in the leaderboard.

Although cross-validation has indicated we should keep `learning_rate=0.1` and `min_samples_split=2`, the other two hyperparameters should have been changed: the maximum depth of the individual regression estimators (limits the number of nodes in the tree) increased from 3 to 5 and the number of boosting stages to perform from 100 to 200. The RMSLE score for the fictitious test set obtained by the gradient boosting algorithm with these hyperparameters is the best we have obtained so far.

5.6.4 Cross-validation and Grid-search with XGBoost Regression

Now we check the behavior of `XGBRegressor` by changing the following hyperparameters:

- `max_depth` : maximum depth of a tree (default=3), increase this value will make the model more complex/likely to be overfitting.
- `n_estimators` : number of boosted trees to fit (default=100).
- `leanirng_rate`: shrinks the contribution of each tree (default=0.1).
- `min_samples_split`: the minimum number of samples required to split an internal node (default=2)
- `reg_lambda`: L2 regularization term on weights (default=1.0).

```
In [84]: import xgboost as xgb
         from sklearn.model_selection import GridSearchCV
         from sklearn.metrics import make_scorer
         # greater_is_better= False: we want to minimize the root mean square logarithmic error
         scorer = make_scorer(score_kaggle_log, greater_is_better=False)
         model = xgb.XGBRegressor(random_state=0)
         parameters = {'max_depth': [5, 10, 20], 'n_estimators': [100, 200],
                       'learning_rate': [0.1, 1.0], 'reg_lambda': [0.1, 1.0] }
         clf = GridSearchCV(model, param_grid= parameters, scoring= scorer, cv=2)
         grid_fit = clf.fit(XX_train, yy_train)
         print(grid_fit.best_params_)
         best_clf = grid_fit.best_estimator_
         print(score_kaggle_log( yy_test, best_clf.predict(XX_test) ) )
```

```
Fitting 2 folds for each of 24 candidates, totalling 48 fits
```

```
[Parallel(n_jobs=-1)]: Done  48 out of  48 | elapsed: 112.6min finished

{'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 200, 'reg_lambda': 1.0}
0.46214027536016206
```

Juke like `GradientBoostingRegression`, the variation is suggested only on maximum depth (from 3 to 5) and number of estimators (from 100 to 200). The performance is also very similar (≈0.01% in favor of `XGBRegressor`). The major advantage in favor to extreme boosting in this case is the training time. While each fit using the usual gradient boosting took, on average, 2.7 minutes, extreme boosting took 2.35 minutes (on average) to complete each fit. This would represent more than an hour if we had more than 170 fits.

6 Training with full training data using the cross-validation hyperparameters and submitting to Kaggle

Although up to now boosting methods have led to better results, we still have not trained any algorithm with full training data in order to make predictions on the real test set. Since ridge regression and random forests are quite fast to train and did not lead to terrible estimators, we are still using them in this final section.

As mentioned before, performance of predictions on the real test set cannot be evaluated here. The final score is achieved only when a submission to Kaggle is made.

The winner team, composed by Francesco Palma, Aldo Podestá, TomBoy and W.R. Lemes de Oliveira obtained the score 0.28976 on the private leaderboard, which is calculated with approximately 70% of the test data. Naturally, their solution is the benchmark model. However, it seems clear to us that, to begin with, much more sophisticated data preprocessing and future extraction, perhaps using clusters and principal component analysis, should be carried on in order to achieve some score close to theirs. Since our purpose is learning, not competing, we define our goal as reaching a score less than 0.5 on the private learderboard.

6.1 Ridge Regression

We start by training a Ridge regressor with regularization strength `alpha` equals to 0.1, following the cross-validation analysis.

```
from sklearn.linear_model import Ridge
model = Ridge(random_state= 0, alpha= 0.1)
clf = model.fit(X_train, y_train)
test_data_pred =  np.exp( clf.predict(test_data)) - 1
```

The submission file constructed with Ridge regression achieved the score 0.51054 on private leaderboard.

6.2 Random Forest Regression

We now train using random forest regression with `max_depth=15`, `min_samples_split= 10` and `n_estimators=30`, following the cross-validation analysis.

```
from sklearn.ensemble import RandomForestRegressor
model = RandomForestRegressor(random_state=0, max_depth=15, min_samples_split= 10,
                              n_estimators=30)
clf = model.fit(X_train, y_train)
test_data_pred =  np.exp( clf.predict(test_data)) - 1
```

The submission file constructed with Random Forest regression achieved the score 0.51063 on private leaderboard. Ridge regression, though its simplicity, is still better.

6.3 Extreme Gradient Boosting

We now train using extreme gradient boosting with `learning_rate=0.1`, `max_depth=5`, `n_estimators=200` following the cross-validation analysis.

```
import xgboost as xgb
 model = xgb.XGBRegressor(random_state= 0, learning_rate= 0.1, max_depth= 5,
                          n_estimators= 200, reg_lambda= 1.0)
 clf = model.fit(X_train, y_train)
 test_data_pred =  np.exp( clf.predict(test_data)) - 1
```

The submission file constructed with Extreme Gradient Boosting achieved the score 0.49961 on private leaderboard and this result fulfills our needs for now, since we have defined our goal as a score less than 0.5.

6.4 Gradient Boosting

Finishing, we now train using gradient boosting with `learning_rate=0.1`, `max_depth=5`, `min_samples_split=2`, `n_estimators=200` following the cross-validation analysis.

```
from sklearn.ensemble import GradientBoostingRegressor
 model = GradientBoostingRegressor(random_state=0, max_depth=5,min_samples_split=2,
                                   n_estimators=200, learning_rate=0.1)
 clf = model.fit(X_train, y_train)
 test_data_pred =  np.exp( clf.predict(test_data)) - 1
```

The submission file above, constructed with the usual Gradient Boosting regression achieved the score 0.49960 on private leaderboard. Similarly to our estimation analysis, it is almost the same result as the one obtained using extreme boosting.