Capstone Project

Hotel Feature Labeling and Rating Prediction

AI-Jhe(Tom) Lin

8th April, 2018

Content of Table

Definition

Project Overview

This project mainly focuses on how we analyze and summarize hotel's feature and its ratings from the feedback survey of customers who've stayed.

More specifically, the first goal is to find out what trait best describes the impression customers have toward each hotel. The second goal is to try building up a prediction model on hotel ratings.

The project's dataset is originally from Booking.com and now published on Kaggle[1]. It contains abundant text, categorical and numerical information. Hence, it is quite suitable to be used for text modeling or prediction modeling.

Now speaking of modeling, in order to find out the impression customers have to each hotel, text topic modeling or Latent Dirichlet Allocation(LDA) model[2] will be used. It helps identify the most important impression or topic in text, and hence if counting up the topic in all reviews, we can tell what is the most important topic or impression the public has to each hotel. Following the other model, regression model, is to take in information on hotel past ratings, review text, again, to predict the customer's rating on hotel.

Problem Statement

Today we have a lot of reviews on hotels, but it is always difficult to summarize and identify the most important topic out of a bunch of messy texts. Therefore, to extract out simplified insight, I will use LDA to classify each hotel review into one topic(class), the target variable. Afterwards, I will assign the best topic for each hotel based on the most frequent one being reflected in reviews.

Following the second step, in order to further leverage the information we have in the text, I will build up a sentiment model to classify each input review into binary positive/negative classes, which will later be used as another input for prediction model.

The last step is to come up with predicted rating for hotel, the target variable. The idea for the model is to take other existing information such as past ratings along with positive/negative classes derived from sentiment model to build up a regression model for customer's continuous rating prediction.

[1] 515K Hotel Reviews Data in Europe
[2] Latent Dirichlet Allocation Model

Metrics

In the text sentiment model, usually the **accuracy** is the most commonly used metric for model evaluation. Nonetheless, the dataset we have at hand is quite imbalanced where positive scores have a large share and few negative cases. Instead, I will use **F1 score**[3] and **recall**[4] as the evaluation metrics, so that we can balance out the effect. In prediction model, owing to the target variable is in numerical-continuous scale, I will use **R^2**[5] as the major evaluation metrics, which is not only commonly used and also easy for interpretation.

Analysis

Data Exploration

The dataset contains around 515,000 customer reviews on more than 1,400 hotels. Each review record has 17 features as described below:

Column Definition from Kaggle:

Column Name	**Data type**	**Description**
Hotel Address	String	
Additional Number of Scoring	Integer	For the hotel, it indicates how many records have only scores and no text review. As sometimes customers just make scores and don't leave text review.
Review Date	Date d/m/y	
Average Score	Float	Average score of the hotel, calculated based on the latest comment in the last year.
Hotel Name	String	
Reviewer Nationality	String	
Negative Review	Text	If the reviewer does not give the negative review, then it should be: 'No Negative'.
Review Total Negative Word Counts	Int	Total number of words in the negative review.
Total Number	Int	Total number of reviews hotel has.

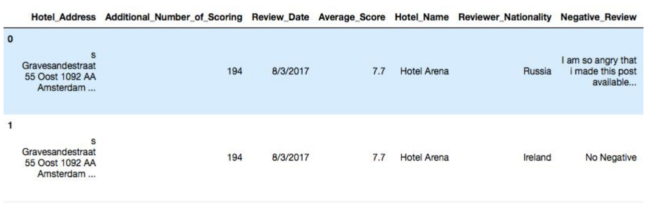
[3] F1 score
[4] Recall
[5] R^2

of Reviews		
Positive Review	Text	If the reviewer does not give the positive review, then it should be: 'No Positive'.
Review Total Positive Word Counts	Int	Total number of words in the positive review.
Total Number of Reviews Reviewer Has Given	Int	Number of Reviews the reviewers has given in the past.
Reviewer Score	Float	
Tags	List of String	Tags reviewer gave the hotel. For example: [' Leisure trip ', ' Solo traveler ', ' Duplex Double Room ', ' Stayed 3 nights ']
Days Since Review	String	Duration between the review date and scrape date. For example: 0 days
Lat	Float	Latitude of the hotel
Ing	Float	Longitude of the hotel

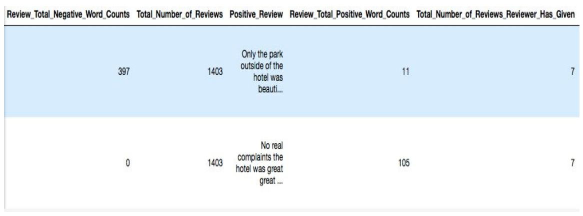
The dataset is different from other review data. It explicitly separates the positive and negative review, and comes up with the respective word counts for each review type. Besides, it also contains the past average score, total reviews the hotel receives and the date when the review is made.

For the general understanding, let's have a look at the dataset.

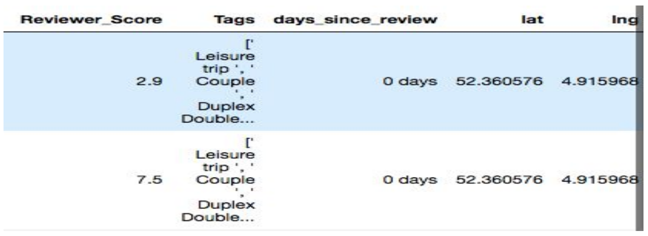
Preview of the dataset:



	Hotel_Address	Additional_Number_of_Scoring	Review_Date	Average_Score	Hotel_Name	Reviewer_Nationality	Negative_Review
0	s Gravesandestraat 55 Oost 1092 AA Amsterdam ...	194	8/3/2017	7.7	Hotel Arena	Russia	I am so angry that i made this post available...
1	s Gravesandestraat 55 Oost 1092 AA Amsterdam ...	194	8/3/2017	7.7	Hotel Arena	Ireland	No Negative



Review_Total_Negative_Word_Counts	Total_Number_of_Reviews	Positive_Review	Review_Total_Positive_Word_Counts	Total_Number_of_Reviews_Reviewer_Has_Given
397	1403	Only the park outside of the hotel was beauti...	11	7
0	1403	No real complaints the hotel was great great ...	105	7



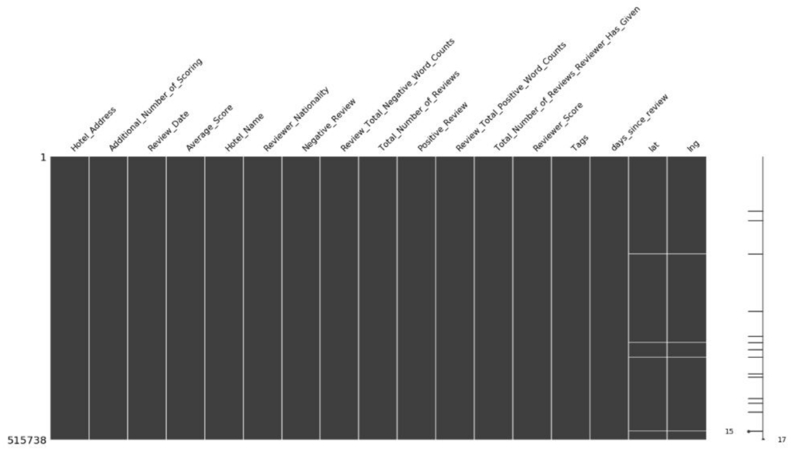
Reviewer_Score	Tags	days_since_review	lat	lng
2.9	[' Leisure trip ', ' Couple ', ' Duplex Double...	0 days	52.360576	4.915968
7.5	[' Leisure trip ', ' Couple ', ' Duplex Double...	0 days	52.360576	4.915968

The comprehensive descriptive statistical summary is in **Appendix**, below are some brief key findings in the overall dataset.

1. The date range of dataset spans from 2015-08-04 to 2017-08-04.
2. *Britannia International Hotel Canary* receives the most reviews.
3. The top one country for reviewers' nationality is UK.
4. The lowest reviewer score is 2.5 while the highest reviewer score is 10.
5. In the dataset, averagely each reviewer gives 7.1 reviews.
6. Averagely, word counts for positive review is 17.7 words.
7. Averagely, word counts for negative review is 18.5 words.
8. The minimal number of reviews hotel receives is 43 reviews and the largest number of reviews hotel receives is 16670.

Additionally, let's have a look at the missing value in each column. For there is a very convenient module called **missingno**, it can help visualize the distribution of missing value in each column, as follows:

Flag of missing values in rows and columns:



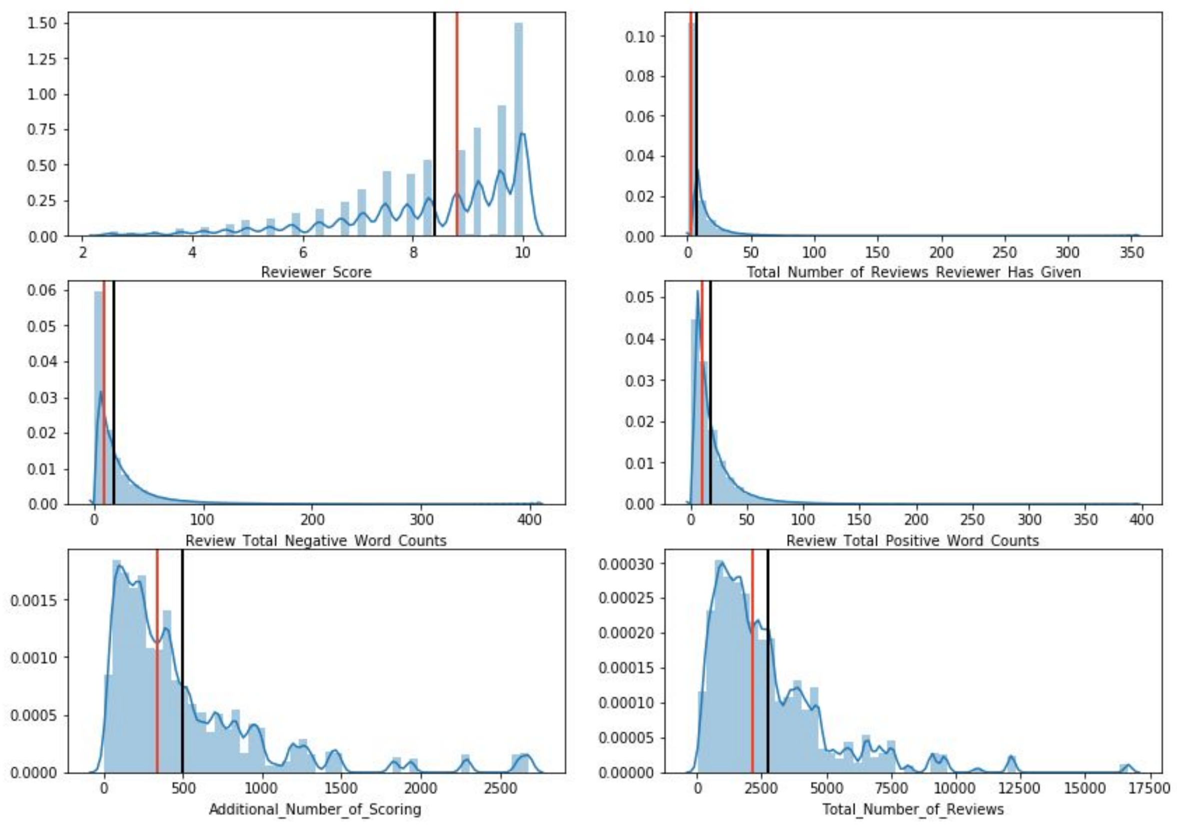
From the chart, we can tell all missing values only occur in the column **Lat** and **Ing**. As we see, the horizontal white line in the chart represents the row position containing missing value and on what columns. The total row number containing missing value is 3268, which only accounts for a pretty small portion of dataset. Hence for the brevity of analysis, I will drop out all the rows.

Finally, let's check up if there exist duplicated rows, since it's quite common to have duplicated data, especially in large dataset. Using *df.drop_duplicates()* method from pandas, I remove 526 rows of duplicated data as well.

Exploratory Visualization

In the step of visualization, I will first use **Histogram** for all numerical features to better understand their distribution. Second is **Bar plot** for categorical feature, which is *Reviewer Nationality* only, so that we can understand the proportion of reviewers around the world. Also plus the **Box plot** of *Reviewer Nationality* on *Reviewer Score*. This way we can tell if the distribution of *Reviewer Score* is different among nationalities. In addition, it's worth noted that *Average Score* and *Total Number of Reviews* are in hotel level that means for this dataset of review level, we will see the frequency of Average Score and Total Number of Reviews inflated as there are many reviews on the same hotel. I will filter the unique hotel and see the distribution of these features on hotel level as well. Finally, I will create **Correlation Matrix** of all numerical features. This way, we can figure out what features are highly correlated to *Reviewer Score* and hence they are well suited as explanatory variables.

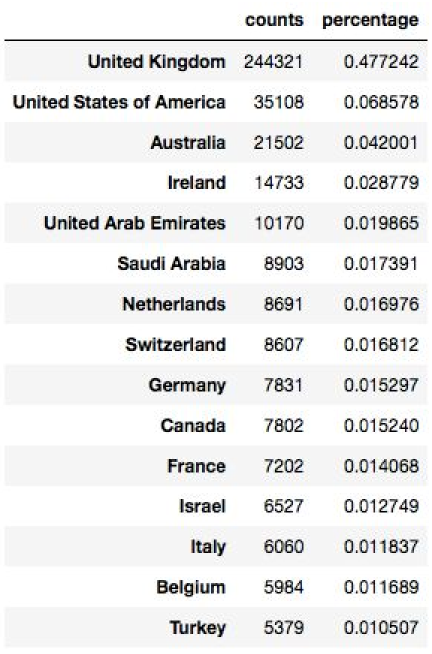
Histogram on numerical features:



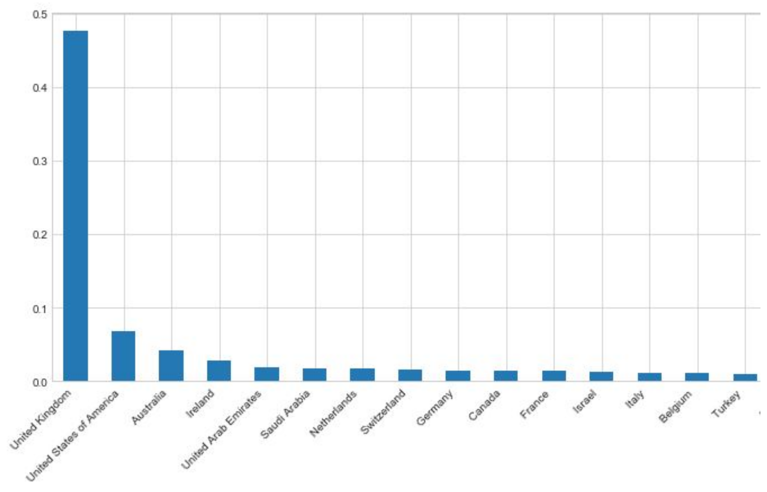
The chart above is the histogram of all numerical features. The **black vertical line** is the mean value and the **red vertical line** is the median value in that respective feature. The *Reviewer Score* is clearly left skewed where the mean is less than the median, and the others are rather right skewed. As we can tell Reviewer Scores mostly gather around at high scores while there are only few cases of low scores. Because the Reviewer Score is the target variable, I will later implement **BOX-COX transformation** to better separate the target variable more evenly, hoping this way will improve the performance on modeling.

Next comes the overview of the proportion of nationalities among reviewers. As shown below, UK reviewers account for almost half of the reviews in this dataset, following by US and Australia, although they only take up around 7% and 4% of total review respectively. After the top 15 countries, remaining countries each take up less than 1% share, and the top 15 countries take up in total around 75% share.

Bar plot on Reviewer Nationality:

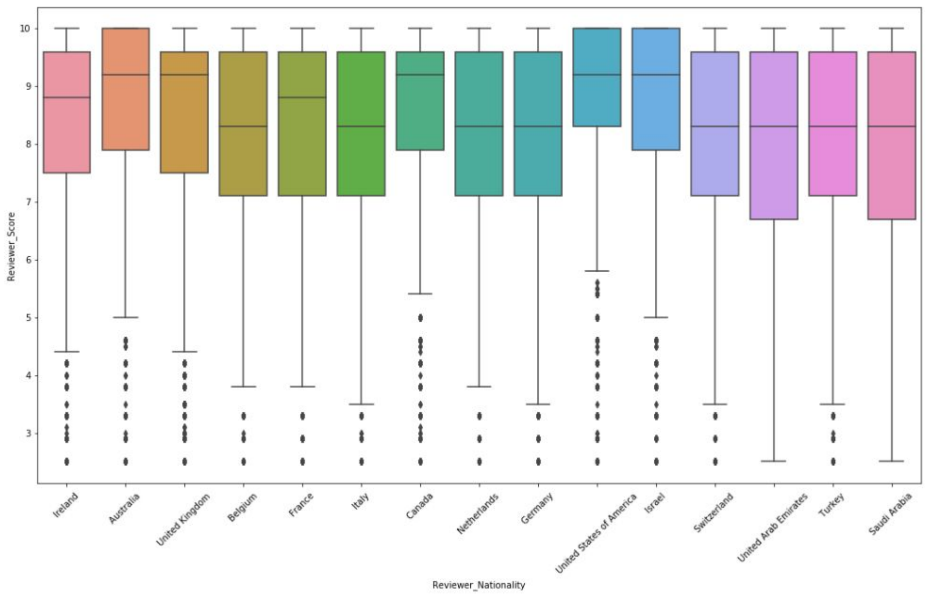


	counts	percentage
United Kingdom	244321	0.477242
United States of America	35108	0.068578
Australia	21502	0.042001
Ireland	14733	0.028779
United Arab Emirates	10170	0.019865
Saudi Arabia	8903	0.017391
Netherlands	8691	0.016976
Switzerland	8607	0.016812
Germany	7831	0.015297
Canada	7802	0.015240
France	7202	0.014068
Israel	6527	0.012749
Italy	6060	0.011837
Belgium	5984	0.011689
Turkey	5379	0.010507



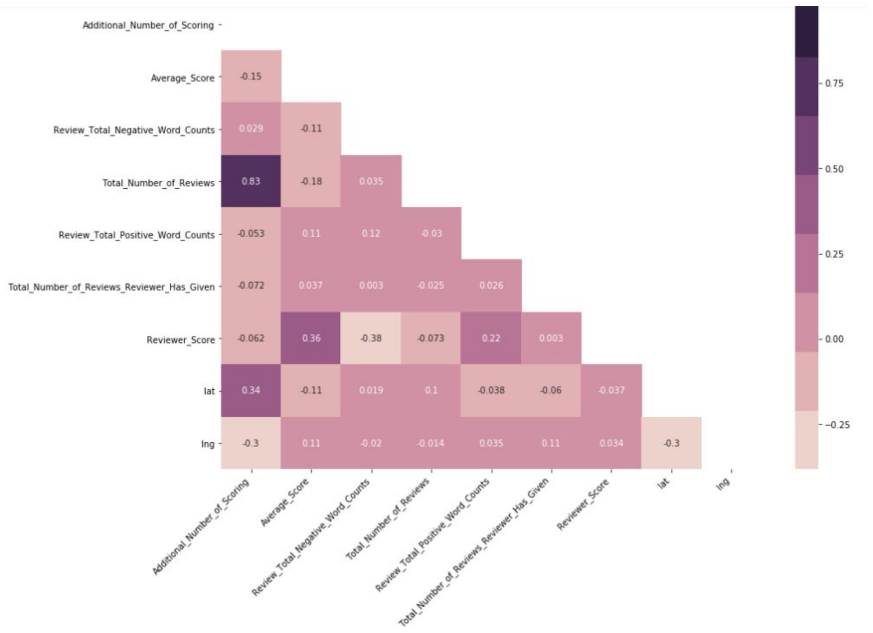
If we dig a little further on the distribution of *Reviewer Score* among all different nationalities. We can tell there are only mild differences on the median Reviewer Score among each countries. For there are too many countries, in order to show up the distribution of Reviewer Score clearly on each country, I will choose only the top 15 countries with highest share on Reviewer Nationality. In the chart below, it shows some countries having higher median score like Australia, USA, Israel, but the difference between high and low median score among countries is just around 1. Hence, overall speaking, the *Reviewer Nationality* seems not influencing the distribution of *Reviewer Score* very much.

Box plot on Reviewer Score over Reviewer Nationality:



Last, we can have a look at the **Correlation matrix** of all numerical features. This is a very useful chart to reveal potential key explanatory variables. Nevertheless, from the chart we can discover that almost there are no features highly correlated to our target variable - **Reviewer Score**. The most correlated ones are *Review Total Negative Word Counts(-0.38)* and next are *Average Score(0.36)* and *Review Total Positive Word Counts(0.22)*. But these correlation coefficients are still quite small in the eyes of general situation. However, the chart still serves as a guide for us to include these features in our modeling building in the end.

Correlation matrix on all numerical features:



Algorithms and Techniques

In addressing the task on text topic modeling. I will use LDA model as mentioned before. However, the input for the LDA model needs to be properly transformed in advance. That is using the **Tokenizer(Bag of Words Model)** to tokenize every word in the text, so that LDA model can take in the input data and train the model. In sentiment modeling, I will use Naive Bayes and Multi Layer Perceptron model. These two are very popular algorithms when working on text sentiment analysis. Again, the **tokenized(Bag of Words)** input data will be used for this model training. The parameters for LDA, Naive Bayes, Multi-layer Perceptron, Tokenizer are listed below:

Parameters for models from Sklearn and Keras module:

Model	**Parameter**	**Description**
LDA	n_components	Number of topics
	max_iter	The maximum number of iterations
Naive Bayes	alpha	Additive smoothing parameter
Multi-layer Perceptron	dense	Nodes for the layer
	dropout	Fraction of nodes to be excluded
	activation	Activation function to the output
Tokenizer	tokenizer	Advanced lemmatizer
	max_df	Ignore terms that have a document frequency higher than the threshold
	min_df	Ignore terms that have a document frequency lower than the threshold
	max_features	Only consider the top max_features ordered by term frequency across the corpus
	stop_words	The stop words to be removed from tokens

	ngram_range	Range of number of words to be considered as one token

In prediction modeling, **Linear Lasso Model** and **Support Vector Machine** will be used. The input data will contain the target numerical variable - *Reviewer Score*, and the explanatory features will include mostly numerical ones such as *Review Total Negative Word Counts.* All being transformed by **BOX-COX transformation** if necessary. However, one binary variable will be included especially - the predicted 0/1 sentiment variable. The variable is the new feature coming from the just-built text sentiment model. I hope the newly added feature will help leverage the information from review text and thus enhance the prediction power for the model.

Parameters for models from Sklearn module:

Model	**Parameter**	**Description**
Lasso	alpha	Regularization parameter
	max_iter	The maximum number of iterations
SVR (support vector regression)	C	Regularization parameter
	gamma	Inverse of the influence from each single sample

Benchmark

For text sentiment modeling, it comes to relatively straightforward when evaluating the classification model. In two classes scenario, random guessing will have only half chance to get the right class given two classes have equal probability. Therefore, as long as the model's classification power is greater than ½, it outperforms the random guessing. If the two classes are not equally alike, then the precision of random guessing for one class will be that class' probability. To put it in mathematical term, if the accuracy of the model is higher than that class' probability, then the model outperforms random guessing. In hindsight, the accuracy for this text sentiment model should be greater than **0.89**, because the probability for the positive review class is 0.89. (The positive review class and negative review class is separated apart on subjective criterion, more details will be discussed in **Implementation.**)

For rating prediction modeling on the other hand, the benchmark for evaluating model on continuous target variable is a bit of blurred. It's difficult to tell how large the gap between the predicted value and actual value will be considered imprecise. Though R^2 greater than **0.5** might be a good benchmark for saying the model's performance acceptable.

Methodology

Data Preprocessing

Discretize *Reviewer Score* into binary output for text sentiment analysis

In order to flag out which text review is positive, which is negative, I use *Reviewer Score* as guideline to separate review records. For Review Score <= 6, the text review will be tagged as negative review, and review with score greater than 6 will be tagged as positive review. Then the new feature is named as *Review Sentiment*.

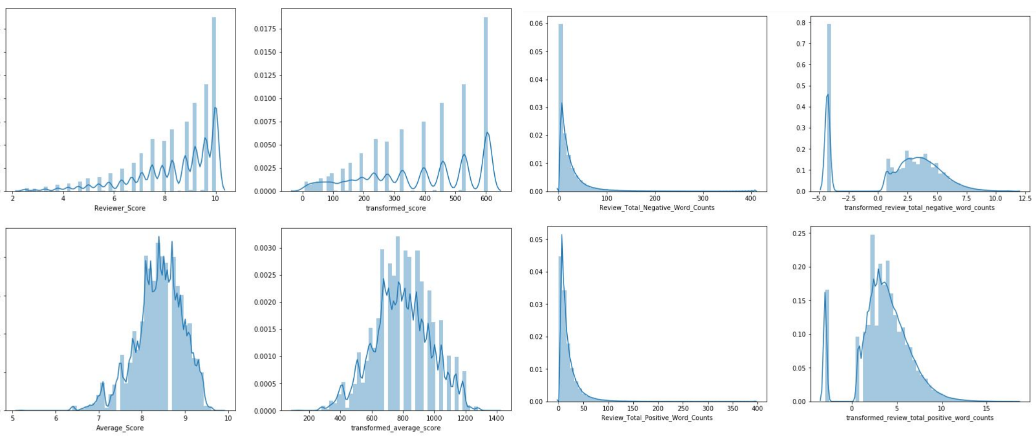
Implement Box-Cox transformation to all the skewed target and explanatory variables

Features being transformed:

1. Reviewer Score
2. Average Score
3. Review Total Negative Word Counts
4. Review Total Positive Word Counts

In side note, for Box-Cox transformation requires all values to be positive, but some values in Review Total Negative Word Counts and Review Total Positive Word Counts are zero. Hence, I jitter around the value and replace all 0 to 0.0001.

Comparison of distribution before/after box-cox transformation

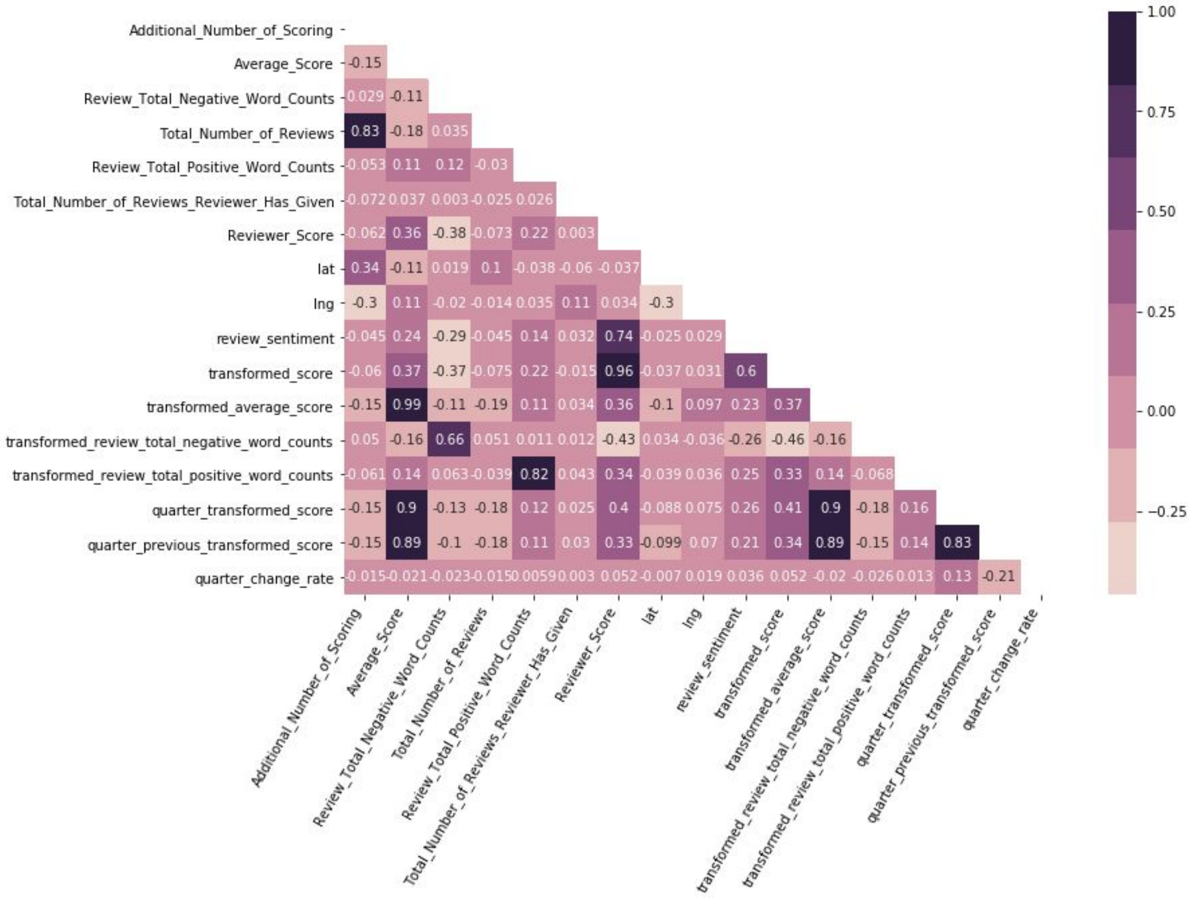


Implement feature engineering

In the section for feature engineering, I will take advantage of **Date** information. Because in the linear modeling for prediction, I will only use the **transformed score** as the target variable, it is better to compute the derived score features all from transformed score for the sake of consistency. First, I calculate the quarterly average rating for the hotel prior to the date that review is made and add it in as extra feature on the review record. Besides, the derived change-rate of attitude rating will also be calculated, which, I believe, will serve as a useful indicator for predicting attitude rating on each review record as well.

In brief, the derived features include:

1. Quarter Transformed Score: This is the quarterly average of transformed score.
2. Quarter Previous Transformed Score: This is the previous quarterly average of transformed score.
3. Quarter Change Rate: This is the change rate of quarterly average transformed score.



Brief Discussion on Correlations

Quite surprisingly, the derived features seem not to have high correlation with transformed score as I expect. The correlations are listed below:

1. Quarter Transformed Score vs Transformed Score: **0.41**
2. Quarter Previous Transformed Score vs Transformed Score: **0.34**
3. Quarter Change Rate vs Transformed Score: **0.05**

Nevertheless, on the positive side, compared to the original features in dataset where the most correlated feature (*Review Total Negative Word Counts*) to the target variable (*Reviewer Score*) is **-0.38**, the *Quarter Transformed Score* achieves a slightly higher correlation to the target variable (*Transformed Score*) with **0.41**.

On the other aspect, if we look at the new binary *Text Sentiment* feature's correlation to Transformed Score, it reaches **0.6**.

Evenfurther, the BOX-COX transformed features all have relatively high correlations to Transformed Score.

1. Transformed Average Score vs Transformed Score: **0.37**
2. Transformed Review Total Positive Word Counts vs Transformed Score: **0.33**
3. Transformed Review Total Negative Word Counts vs Transformed Score: **-0.46**

Hence, maybe the box-cox features are more suitable to be included as explanatory variables for rating prediction model.

Overview of Target and Predictor in Text Sentiment Model:

Feature	**Notes**
Target variable	
Review Sentiment	
Predictor	
Combined Review	Will combine positive and negative review into one column

Overview of Target and Explanatory Variables in Rating Prediction Model:

Feature	**Notes**
Target variable	
Transformed Score	
Explanatory variable	
Text Sentiment	Will use the **Predicted Text Sentiment** from text sentiment model
Transformed Review Total Negative Word Counts	
Transformed Review Total Positive Word Counts	
Transformed Average Score	

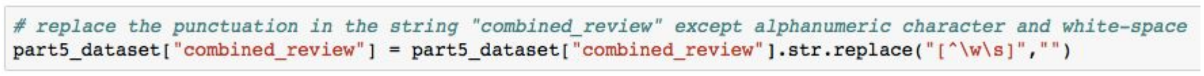
Quarter Transformed Score	
Quarter Previous Transformed Score	

Implementation 01 - Building up Latent Dirichlet Allocation Model

Step 01: come up with single text review column - combined_review

In the original dataset, the positive and negative review are separated. For the simplicity of conducting text topic modeling, I combine the two text columns into one. Following processing includes removing the punctuations, so that they will not influence the outcome of bag of words modeling.

Code Snippet of Step 01:



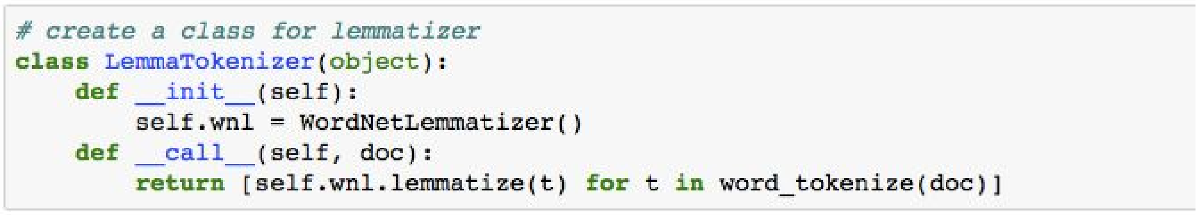
```
# replace the punctuation in the string "combined_review" except alphanumeric character and white-space
part5_dataset["combined_review"] = part5_dataset["combined_review"].str.replace("[^\w\s]","")
```

Step 02: create customized lemmatizer and print_top_words function

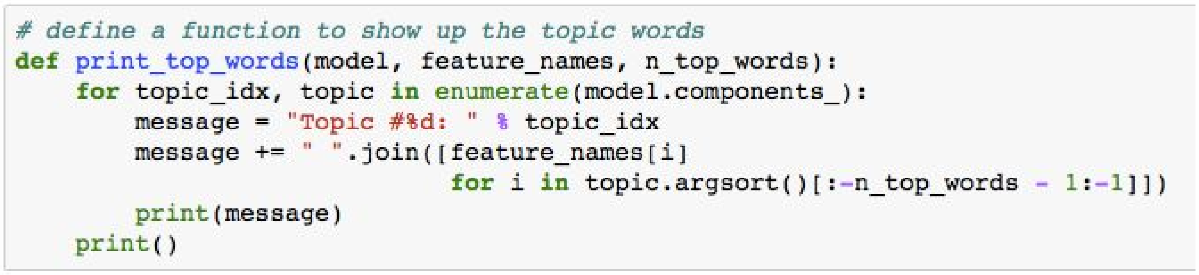
In order to further clean up the words in the text, I take reference from sklearn examples[6], which shows up how to implement advanced lemmatizer to tidy up the word and its derivative forms. The print_top_words functions is used when I need to examine the most important words in each text topic.

[6] Customized lemmatizer and print_top_words function

Code Snippet of Step 02:



```
# create a class for lemmatizer
class LemmaTokenizer(object):
    def __init__(self):
        self.wnl = WordNetLemmatizer()
    def __call__(self, doc):
        return [self.wnl.lemmatize(t) for t in word_tokenize(doc)]
```



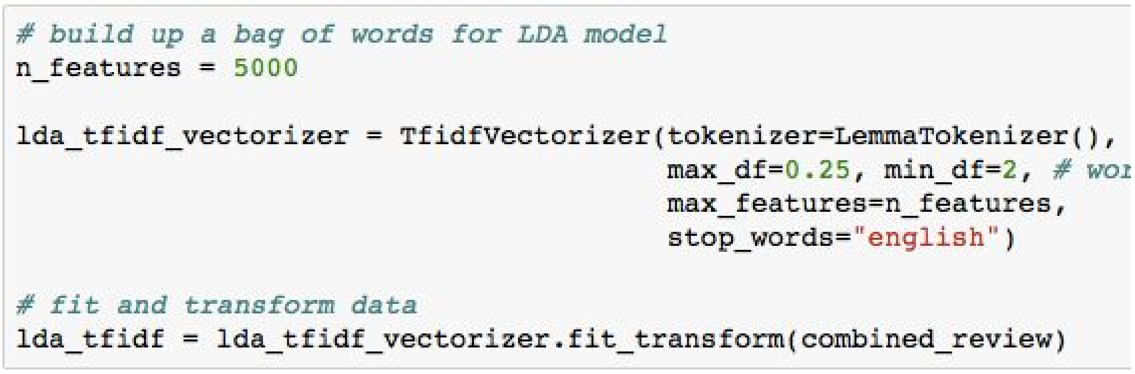
```
# define a function to show up the topic words
def print_top_words(model, feature_names, n_top_words):
    for topic_idx, topic in enumerate(model.components_):
        message = "Topic #%d: " % topic_idx
        message += " ".join([feature_names[i]
                             for i in topic.argsort()[:-n_top_words - 1:-1]])
        print(message)
    print()
```

Step 03: create bag of words model for solely for LDA

Model specifications as follow:

1. Include all dataset (511944 rows)
2. Using Term frequency - Inverse Document Frequency Tokenizer
3. Only pick up top frequent 5000 words as features
4. Use advanced Lemmatization
5. Only pick up words with frequency less than 25% and shows at least in 2 documents
6. Exclude English stop words

Code Snippet of Step 03:



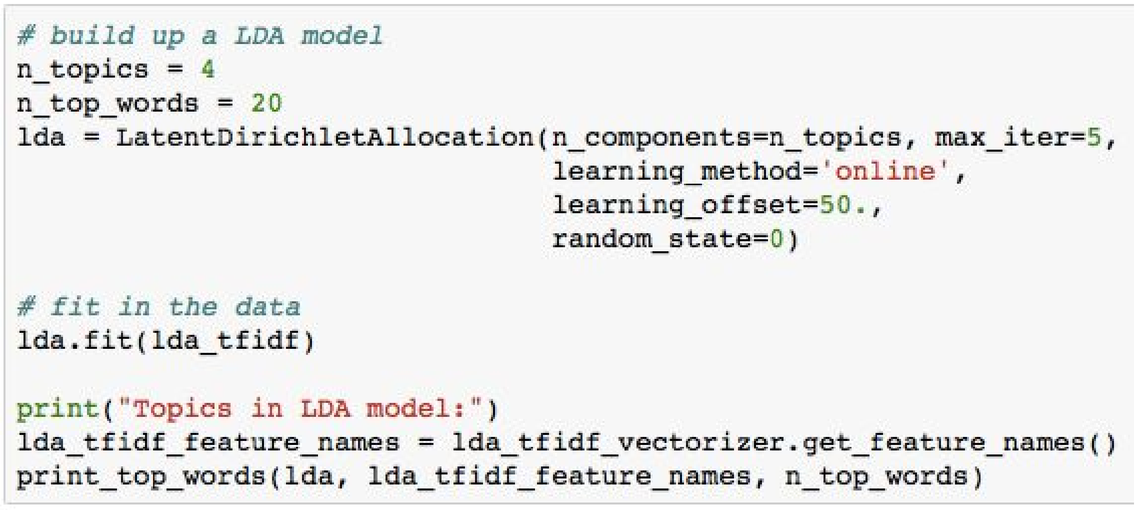
```
# build up a bag of words for LDA model
n_features = 5000

lda_tfidf_vectorizer = TfidfVectorizer(tokenizer=LemmaTokenizer(),
                                       max_df=0.25, min_df=2, # wor
                                       max_features=n_features,
                                       stop_words="english")

# fit and transform data
lda_tfidf = lda_tfidf_vectorizer.fit_transform(combined_review)
```

Step 04: build up LDA

Model specifications as follow:

1. Choose only 4 topics
2. Set up model iterations to 5

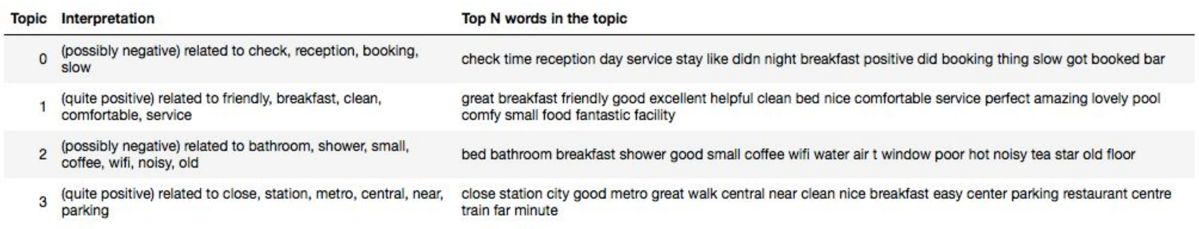
Code Snippet of Step 04:



```
# build up a LDA model
n_topics = 4
n_top_words = 20
lda = LatentDirichletAllocation(n_components=n_topics, max_iter=5,
                                learning_method='online',
                                learning_offset=50.,
                                random_state=0)

# fit in the data
lda.fit(lda_tfidf)

print("Topics in LDA model:")
lda_tfidf_feature_names = lda_tfidf_vectorizer.get_feature_names()
print_top_words(lda, lda_tfidf_feature_names, n_top_words)
```

Step 05: interpretation of each topic

From the table below, we can tell **Topic 0** is negative tone and related to check-in. **Topic 1** is positive tone and related to service. **Topic 2** is negative tone and related to facility. **Topic 3** is positive tone and related to location and traffic.



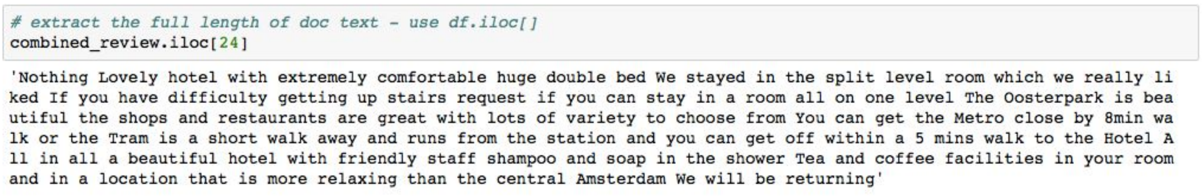
Topic	Interpretation	Top N words in the topic
0	(possibly negative) related to check, reception, booking, slow	check time reception day service stay like didn night breakfast positive did booking thing slow got booked bar
1	(quite positive) related to friendly, breakfast, clean, comfortable, service	great breakfast friendly good excellent helpful clean bed nice comfortable service perfect amazing lovely pool comfy small food fantastic facility
2	(possibly negative) related to bathroom, shower, small, coffee, wifi, noisy, old	bed bathroom breakfast shower good small coffee wifi water air t window poor hot noisy tea star old floor
3	(quite positive) related to close, station, metro, central, near, parking	close station city good metro great walk central near clean nice breakfast easy center parking restaurant centre train far minute

Step 06: calculate the topic probabilities for each document

Omitted.

Step 07: have a look at the doc topic assigned and the doc text

Let's see if the topic being assigned matches up the doc text review. For example, the **document 24** is being assigned to **topic 3** which is positive tone and related to traffic. It seems like the LDA model doing a relatively good job, as it captures the major tone in this document.



```
# extract the full length of doc text - use df.iloc[]
combined_review.iloc[24]
```

```
'Nothing Lovely hotel with extremely comfortable huge double bed We stayed in the split level room which we really li
ked If you have difficulty getting up stairs request if you can stay in a room all on one level The Oosterpark is bea
utiful the shops and restaurants are great with lots of variety to choose from You can get the Metro close by 8min wa
lk or the Tram is a short walk away and runs from the station and you can get off within a 5 mins walk to the Hotel A
ll in all a beautiful hotel with friendly staff shampoo and soap in the shower Tea and coffee facilities in your room
and in a location that is more relaxing than the central Amsterdam We will be returning'
```

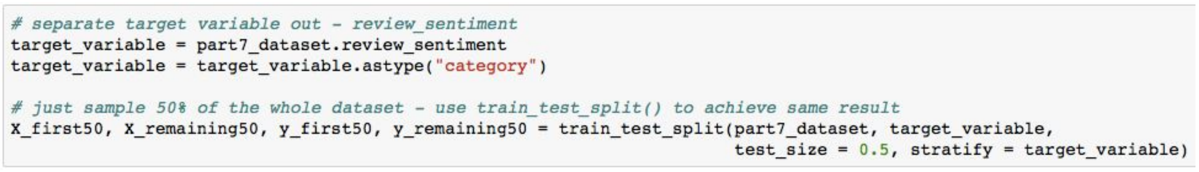
Implementation 02 - Building up Text Sentiment Model

Step 01: stratified sampling 50% of the dataset

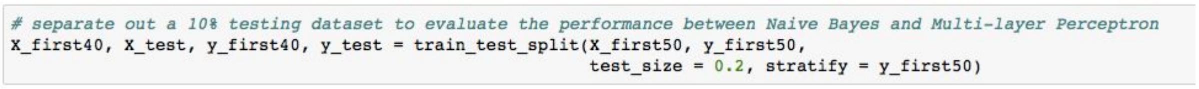
For the second model - Text Sentiment Model, I will only use **the first 50%** of dataset as training, validation and testing dataset, which can be achieved by conducting **train_test_split()** to separate out each sub-dataset and all **classes are stratified** as well.

Then especially, **10%** of the first 50% dataset will be reserved for use as **testing dataset**, so that I can better evaluate the performance between **Naive Bayes** and **Multi-layer Perceptron** model. This manner also avoids data leakage, because I don't use the remaining 50% of the population dataset for guiding what model should be chosen.

Code Snippet of Step 01:



```
# separate target variable out - review_sentiment
target_variable = part7_dataset.review_sentiment
target_variable = target_variable.astype("category")

# just sample 50% of the whole dataset - use train_test_split() to achieve same result
X_first50, X_remaining50, y_first50, y_remaining50 = train_test_split(part7_dataset, target_variable,
                                                                      test_size = 0.5, stratify = target_variable)
```



```
# separate out a 10% testing dataset to evaluate the performance between Naive Bayes and Multi-layer Perceptron
X_first40, X_test, y_first40, y_test = train_test_split(X_first50, y_first50,
                                                        test_size = 0.2, stratify = y_first50)
```

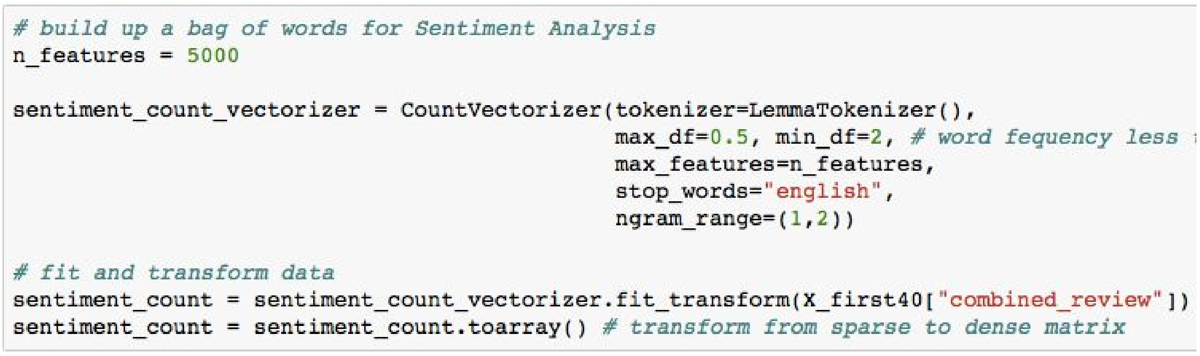
Step 02: create customized lemmatizer for sentiment analysis

Again, I will use the same customized lemmatizer being defined in the phase of LDA model implementation to create a bag of words model.

Step 03: create a bag of words solely for sentiment analysis

The text data source again is from column - **combined_review**. Because of later on I will use *MultinomialNB* model, it's better to use **CountVectorizer** in this bag of word model, contrary to the **TfidfVectorizer** being used in LDA model. Top 5000 words will be chosen only for the bag, similar to what I conduct in LDA modeling. One difference is that this time I experiment with **both one and bi grams** to see if potential influential word patterns exist.

Code Snippet of Step 03:

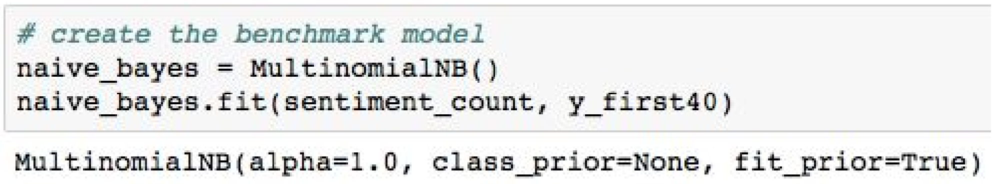


```
# build up a bag of words for Sentiment Analysis
n_features = 5000

sentiment_count_vectorizer = CountVectorizer(tokenizer=LemmaTokenizer(),
                                             max_df=0.5, min_df=2, # word fequency less
                                             max_features=n_features,
                                             stop_words="english",
                                             ngram_range=(1,2))

# fit and transform data
sentiment_count = sentiment_count_vectorizer.fit_transform(X_first40["combined_review"])
sentiment_count = sentiment_count.toarray() # transform from sparse to dense matrix
```

Step 04: build up the benchmark model (naive bayes) for sentiment analysis

No argument adjustment in this step.

Code Snippet of Step 04:



```
# create the benchmark model
naive_bayes = MultinomialNB()
naive_bayes.fit(sentiment_count, y_first40)
```

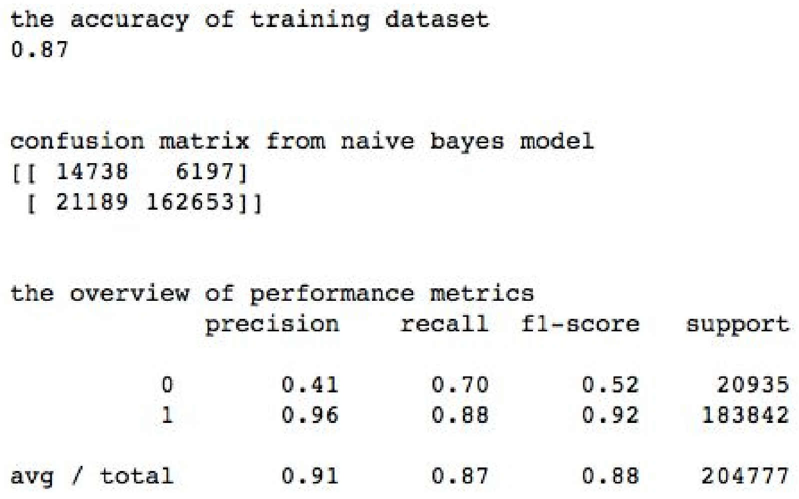
```
MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)
```

Step 05: evaluate the naive bayes model performance

As mentioned earlier (in **Metrics** and **Benchmark** section), in classification case, the **accuracy, F1 and recall** will be used for model evaluation.

One thing worth reminding is that ideally it's better to have a model with accuracy higher than **0.89** because the probability for the positive review class is 0.89 and probability for the negative review class is (1-0.89) in the population dataset. Thus, now we can check if the Naive Bayes model outperforms blind guessing.

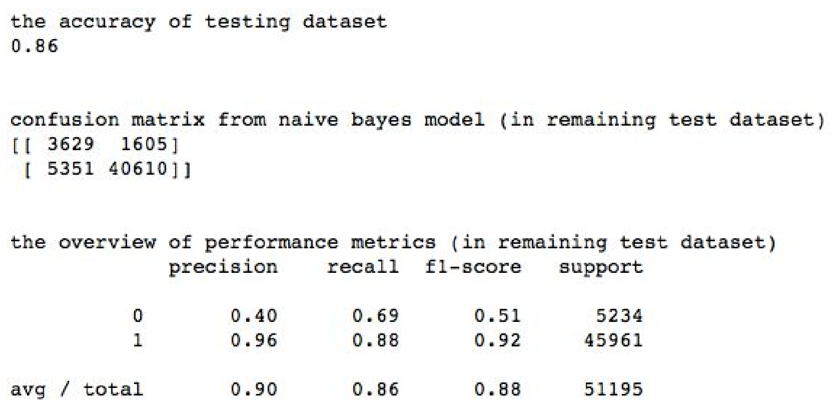
Output of the model prediction and confusion matrix on training dataset(nb model):



```
the accuracy of training dataset
0.87


confusion matrix from naive bayes model
[[ 14738   6197]
 [ 21189 162653]]


the overview of performance metrics
             precision    recall  f1-score   support

          0       0.41      0.70      0.52     20935
          1       0.96      0.88      0.92    183842

avg / total       0.91      0.87      0.88    204777
```

Output of the model prediction and confusion matrix on testing dataset(nb model):

REMINDING: the testing dataset for text sentiment model is **X_test** and **y_test**, not *X_remaining50* and *y_remaining50*.



```
the accuracy of testing dataset
0.86


confusion matrix from naive bayes model (in remaining test dataset)
[[ 3629  1605]
 [ 5351 40610]]


the overview of performance metrics (in remaining test dataset)
             precision    recall  f1-score   support

          0       0.40      0.69      0.51      5234
          1       0.96      0.88      0.92     45961

avg / total       0.90      0.86      0.88     51195
```

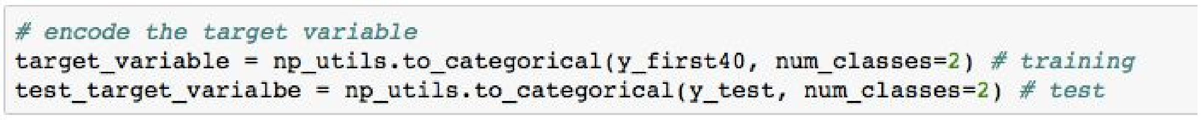
It seems like the Naive Bayes model's performance doesn't outperform random guessing. Its accuracy is only **0.86** on testing dataset, less than **0.89** threshold.

Step 06: build up the comparison model - multi-layer perceptron model

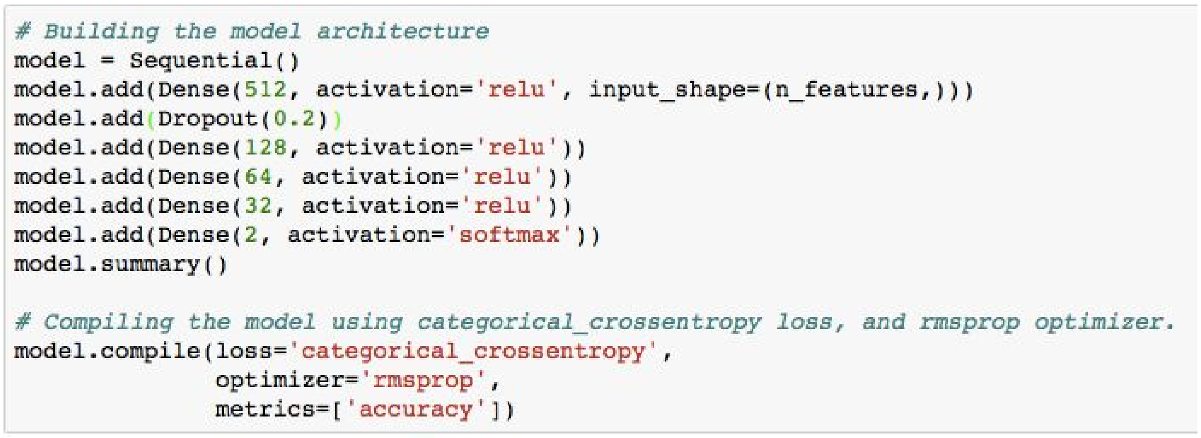
Here I will build up a comparing MLP model on text sentiment identification. The target variable should be transformed into acceptable format first and the architecture of MLP also needs to be specified accordingly as well. In classification case, it's best to use **softmax** as final activation function and the loss method is set **categorical_crossentropy**. The **relu** activation is a common recommended activation function since it avoids zero gradients. I also adopt these in this model. The **number of layers**, **dense nodes** and **dropout rate** are being experimentally adjusted and tested for several times. The following architecture is the one with acceptable performance I can get for now.

During model fitting, the **early-stopping** is implemented and batch_size for training is set **50**, meaning not every sample is used within each epoch. Last, one-fourth of the training dataset is reserved for validation.

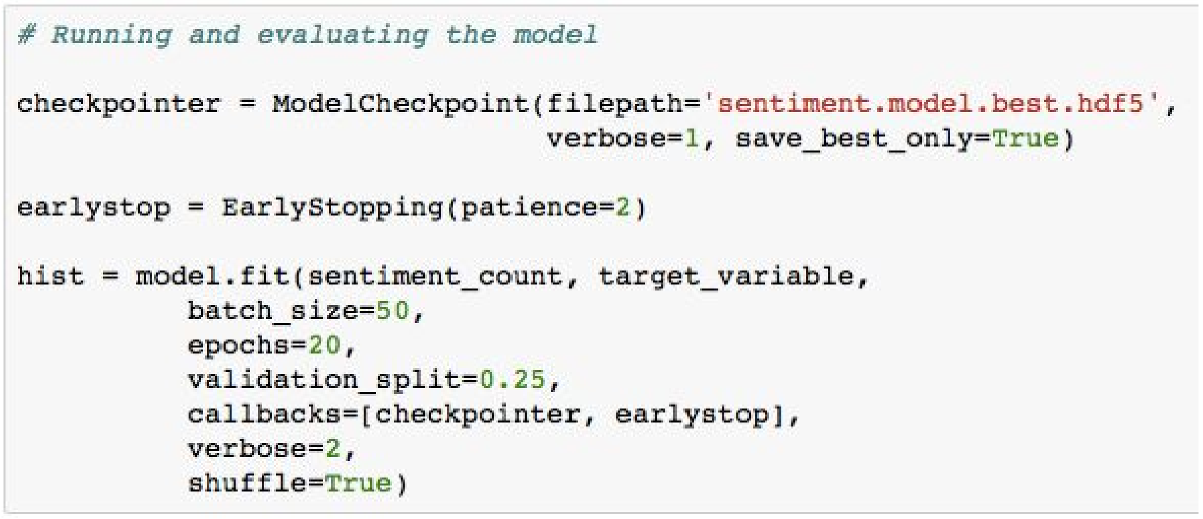
Code Snippet of Step 06:



```
# encode the target variable
target_variable = np_utils.to_categorical(y_first40, num_classes=2) # training
test_target_varialbe = np_utils.to_categorical(y_test, num_classes=2) # test
```



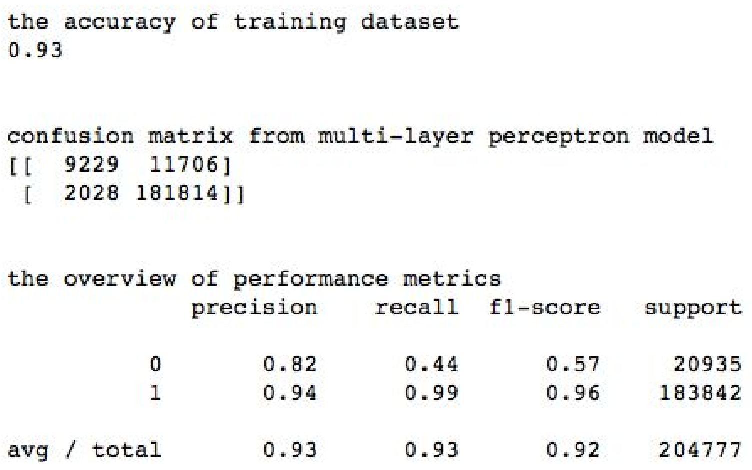
```
# Building the model architecture
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(n_features,)))
model.add(Dropout(0.2))
model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(2, activation='softmax'))
model.summary()

# Compiling the model using categorical_crossentropy loss, and rmsprop optimizer.
model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```



```
# Running and evaluating the model

checkpointer = ModelCheckpoint(filepath='sentiment.model.best.hdf5',
                               verbose=1, save_best_only=True)

earlystop = EarlyStopping(patience=2)

hist = model.fit(sentiment_count, target_variable,
          batch_size=50,
          epochs=20,
          validation_split=0.25,
          callbacks=[checkpointer, earlystop],
          verbose=2,
          shuffle=True)
```

Step 07: evaluate the MLP model performance

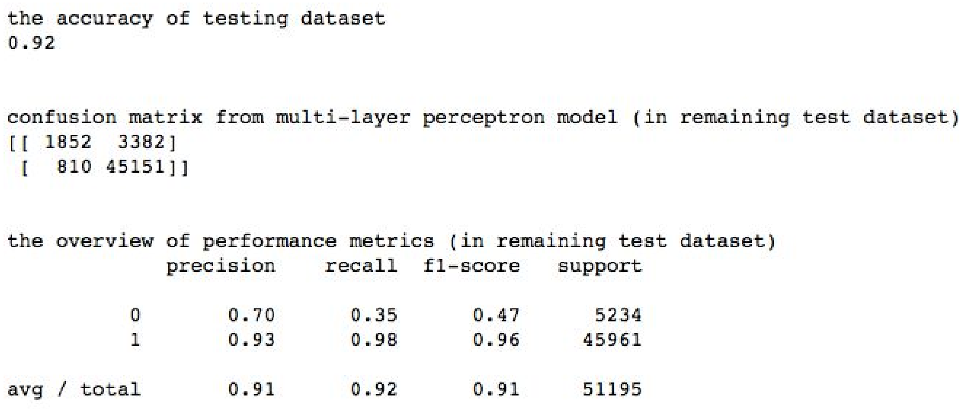
Output of the model prediction and confusion matrix on training dataset(mlp model):



```
the accuracy of training dataset
0.93


confusion matrix from multi-layer perceptron model
[[  9229  11706]
 [  2028 181814]]


the overview of performance metrics
             precision    recall  f1-score   support

          0       0.82      0.44      0.57     20935
          1       0.94      0.99      0.96    183842

avg / total       0.93      0.93      0.92    204777
```

Output of the model prediction and confusion matrix on testing dataset(mlp model):

REMINDING: the testing dataset for text sentiment model is **X_test** and **y_test**, not *X_remaining50* and *y_remaining50*.



```
the accuracy of testing dataset
0.92


confusion matrix from multi-layer perceptron model (in remaining test dataset)
[[ 1852  3382]
 [  810 45151]]


the overview of performance metrics (in remaining test dataset)
             precision    recall  f1-score   support

          0       0.70      0.35      0.47      5234
          1       0.93      0.98      0.96     45961

avg / total       0.91      0.92      0.91     51195
```

Apparently, MLP model's performance passes the threshold for **0.89** accuracy. Its accuracy is **0.92** on testing dataset.

Step 08: choose better model and predict text sentiment for remaining 50% dataset

The performance result on testing dataset between NB and MLP:

Naive Bayes	Metrics	Multi-layer Perceptron
0.86	**Accuracy**	0.92
0.88	**F1-score**	0.91
0.86	**Recall**	0.92

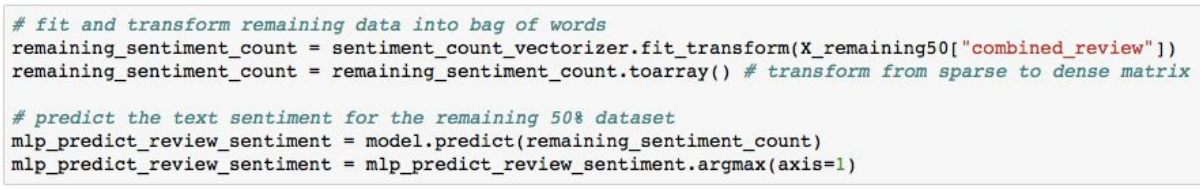
- Class 0: negative review

- Class 1: positive review

As we can tell, the MLP model seems to outperform the Naive Bayes model a little bit on all three metrics. But if we look closer, on the unbalanced(fewer) class 0, of all class 0, Naive Bayes can identify correctly 69% in testing dataset while the MLP model can only identify 35% in testing dataset. That means Naive Bayes model has a better performance of recall and f1 on the small class (that is the 0 group). If identifying class 0 is very important, then Naive Bayes model should be chosen instead.

However, in my case, I will weight class 0 and 1 equally, and choose the model with higher overall performance - that is Multi-layer Perceptron Model. The code for creating new predicted feature on text sentiment is listed below. This new feature is named **mlp_predict_review_sentiment** and will be included as one of the explanatory variables for rating prediction model in the next phase.

Code Snippet of Step 08:



```
# fit and transform remaining data into bag of words
remaining_sentiment_count = sentiment_count_vectorizer.fit_transform(X_remaining50["combined_review"])
remaining_sentiment_count = remaining_sentiment_count.toarray() # transform from sparse to dense matrix

# predict the text sentiment for the remaining 50% dataset
mlp_predict_review_sentiment = model.predict(remaining_sentiment_count)
mlp_predict_review_sentiment = mlp_predict_review_sentiment.argmax(axis=1)
```

Implementation 03 - Building up Rating Prediction Model

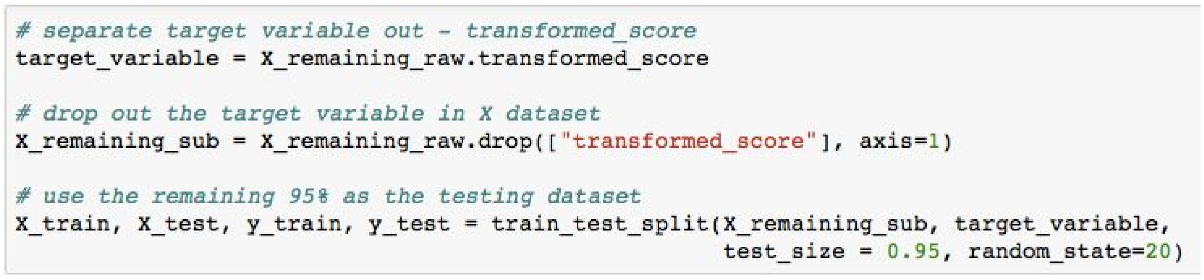
Step 01: shuffle and sampling the remaining dataset

In this section, it's time to start building up our final model - rate prediction model. The benchmark model will be Lasso model and the comparing model will be SVM, which under continuous prediction case would be Support Vector Regression. The target variable is **transformed_score** and the predictors, which all have higher correlation with transformed_score, are

1. mlp_predict_review_sentiment
2. transformed_review_total_negative_word_counts
3. transformed_review_total_positive_word_counts
4. transformed_average_score
5. quarter_transformed_score
6. quarter_previous_transformed_score

After some preliminary experiments, it turns out if the number of samples goes beyond **10,000** in Support Vector Machine, *the training time will be super long*. Thus, in order to avoid long training hours, I will universally use only **5%** (that is around **11728** samples) as training and validation dataset for both **Lasso** and **SVM** model. Hence, I use the remaining 95% of dataset as testing dataset for rating prediction model.

Code Snippet of Step 01:

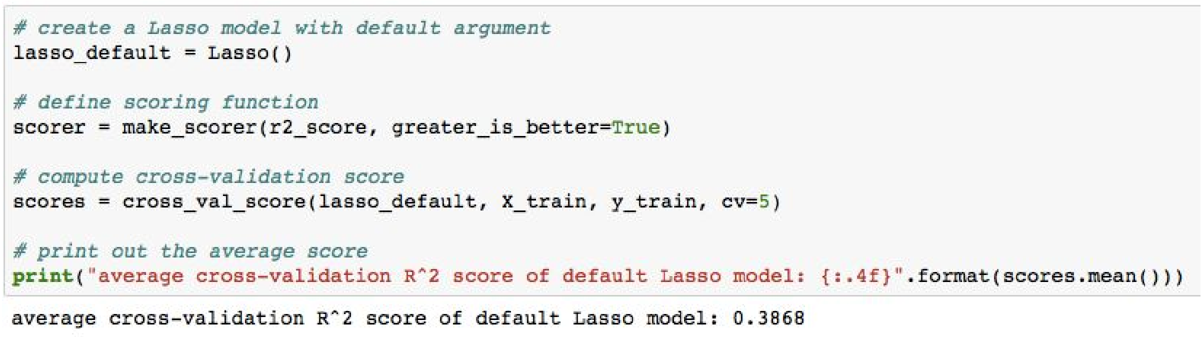


```
# separate target variable out - transformed_score
target_variable = X_remaining_raw.transformed_score

# drop out the target variable in X dataset
X_remaining_sub = X_remaining_raw.drop(["transformed_score"], axis=1)

# use the remaining 95% as the testing dataset
X_train, X_test, y_train, y_test = train_test_split(X_remaining_sub, target_variable,
                                                    test_size = 0.95, random_state=20)
```

Step 02: create Lasso model as benchmark model - using default parameter

There is only one default parameter for Lasso model, **alpha** = **1**. As we can tell the average cross-validation **R^2** score for default Lasso is **0.38**.

Code Snippet of Step 02:



```
# create a Lasso model with default argument
lasso_default = Lasso()

# define scoring function
scorer = make_scorer(r2_score, greater_is_better=True)

# compute cross-validation score
scores = cross_val_score(lasso_default, X_train, y_train, cv=5)

# print out the average score
print("average cross-validation R^2 score of default Lasso model: {:.4f}".format(scores.mean()))
```

```
average cross-validation R^2 score of default Lasso model: 0.3868
```

Step 03: create Lasso model as benchmark model - using grid-search

There are two parameters being fine tuned in grid-search. One is **alpha**, when alpha goes down, the model will become more complicated, hence overfitting. In the meantime, **max_iter** will need to go up responding to smaller alpha. **The range of alpha** being searched is between **0.001** and **80**. **The range of max_iter** being searched is between **100** and **10,000**.

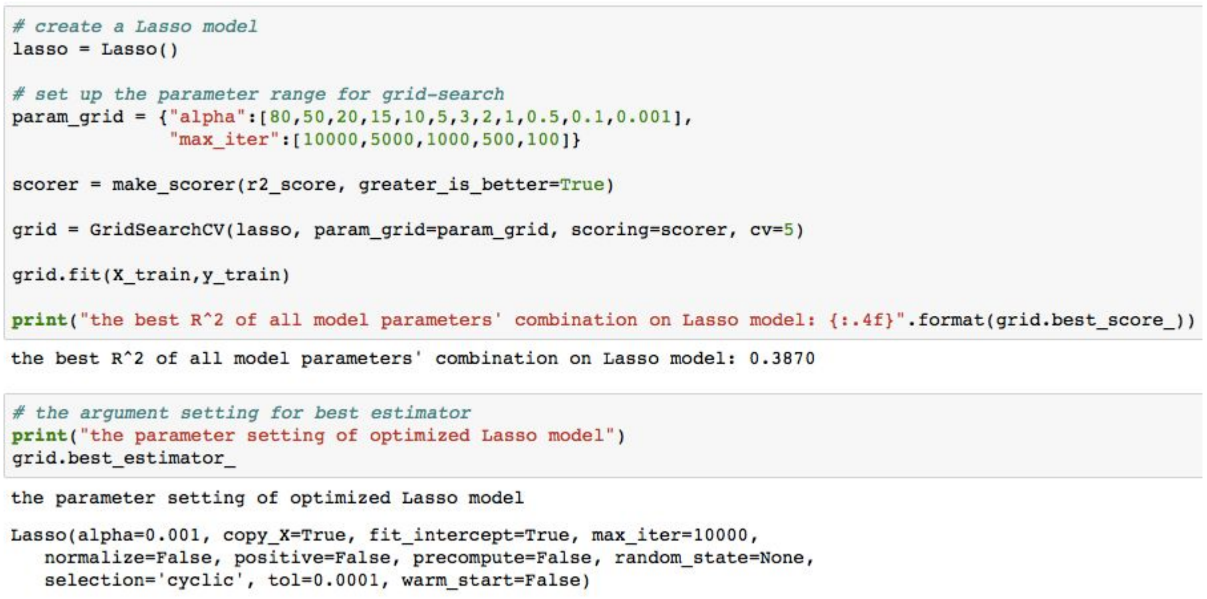
Now, let's comparing the **R^2** between the default parameter and the optimized parameter:

Before - default parameter	**After - grid-search optimization**
0.3868 Parameter setting: alpha: 1	0.3867 Parameter setting: alpha: 0.001 Max_iter: 10,000

BRIEF RESULT:

As we can tell, the **GridSearchCV** function chose to have smaller alpha and larger max_iter, comparing to the default parameters, *implying that default model is still a bit of underfitting*.

Code Snippet of Step 03:



```
# create a Lasso model
lasso = Lasso()

# set up the parameter range for grid-search
param_grid = {"alpha":[80,50,20,15,10,5,3,2,1,0.5,0.1,0.001],
              "max_iter":[10000,5000,1000,500,100]}

scorer = make_scorer(r2_score, greater_is_better=True)

grid = GridSearchCV(lasso, param_grid=param_grid, scoring=scorer, cv=5)

grid.fit(X_train,y_train)

print("the best R^2 of all model parameters' combination on Lasso model: {:.4f}".format(grid.best_score_))
```

```
the best R^2 of all model parameters' combination on Lasso model: 0.3870
```

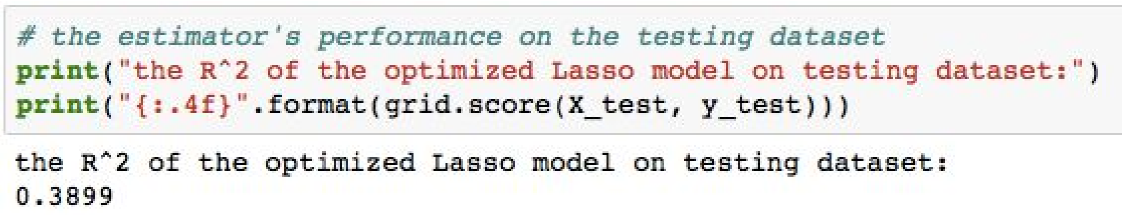
```
# the argument setting for best estimator
print("the parameter setting of optimized Lasso model")
grid.best_estimator_
```

```
the parameter setting of optimized Lasso model

Lasso(alpha=0.001, copy_X=True, fit_intercept=True, max_iter=10000,
   normalize=False, positive=False, precompute=False, random_state=None,
   selection='cyclic', tol=0.0001, warm_start=False)
```

Step 04: optimized Lasso model's performance on testing dataset

The **R^2** score for testing dataset is **0.3899**, quite similar to the validation score on training dataset. In addition, the size of testing dataset is 222,842.

Code Snippet of Step 04:



```
# the estimator's performance on the testing dataset
print("the R^2 of the optimized Lasso model on testing dataset:")
print("{:.4f}".format(grid.score(X_test, y_test)))
```

```
the R^2 of the optimized Lasso model on testing dataset:
0.3899
```

Step 05: create comparing model support vector regression - using default parameter

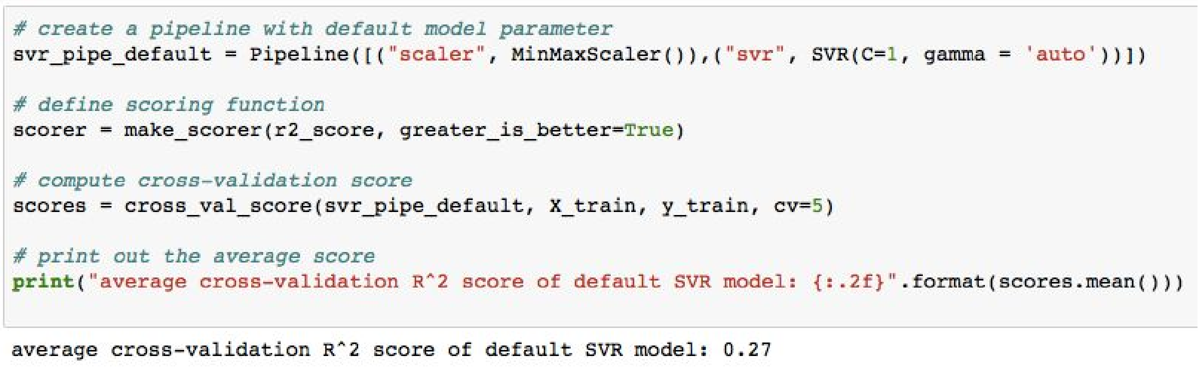
Now comes the comparing model - SVR. When training Support Vector Machine, it requires to have **similar scale on all features**. For I have dummy variables(0/1), I will use **MinMaxScaler()** to scale all numeric features into range 0/1 as well.

Moreover, in the following model evaluation, for the proper use of train and validation dataset in cross-validation, it's better to **create a pipeline** for it. For each fold of cross-validation, pipeline enables to use the training data of current fold only and create scaler based on it. **It avoids data information leakage from validation dataset**.

Lastly, in training Support Vector Machine, the samples of dataset is recommended not to go beyond 100,000 rows[7], the time it takes for training the model goes exponentially. In my model training process, I will choose samples around 11,728 rows.

The default parameters for SVR are **C** = **1** and **gamma** = **1/number of feature**. The cross-validation **R^2** score is only **0.27**, quite undesirable.

Code Snippet of Step 05:



```
# create a pipeline with default model parameter
svr_pipe_default = Pipeline([("scaler", MinMaxScaler()),("svr", SVR(C=1, gamma = 'auto'))])

# define scoring function
scorer = make_scorer(r2_score, greater_is_better=True)

# compute cross-validation score
scores = cross_val_score(svr_pipe_default, X_train, y_train, cv=5)

# print out the average score
print("average cross-validation R^2 score of default SVR model: {:.2f}".format(scores.mean()))
```

```
average cross-validation R^2 score of default SVR model: 0.27
```

Step 06: create comparing model support vector regression - using grid-search

This turn, I try to fine tune the parameters and see if resulting better model. For the grid search process, in order to speed up the model training process, I will try to use **Randomized SearchCV** function this time, instead of GridSearchCV. The RandomizedSearchCV will only select some combinations of parameters, hence reducing training time. One thing worth notice, RandomizedSearchCV uses **distribution** instead of **hard-code number** for specifying parameters.

Again, when building up SVR model, two parameters are being fine tuned. One is **C**, which is similar to the L1 regularization in Lasso model. When C goes up, the model will move toward overfitting. The other is **gamma**, which means how far each sample's influence is. If gamma goes up, every sample will have shorter range of influence, hence the model will move toward overfitting. **The range of C** being searched is between **0** and **10**. **The range of gamma** being searched is between **0** and **10**.

Now again, let's compare the **R^2** between the default parameter and the optimized parameter.

Before - default parameter	**After - randomized grid search optimization**
0.27 Parameter setting: C: 1	0.41 Parameter setting: C: 8.91

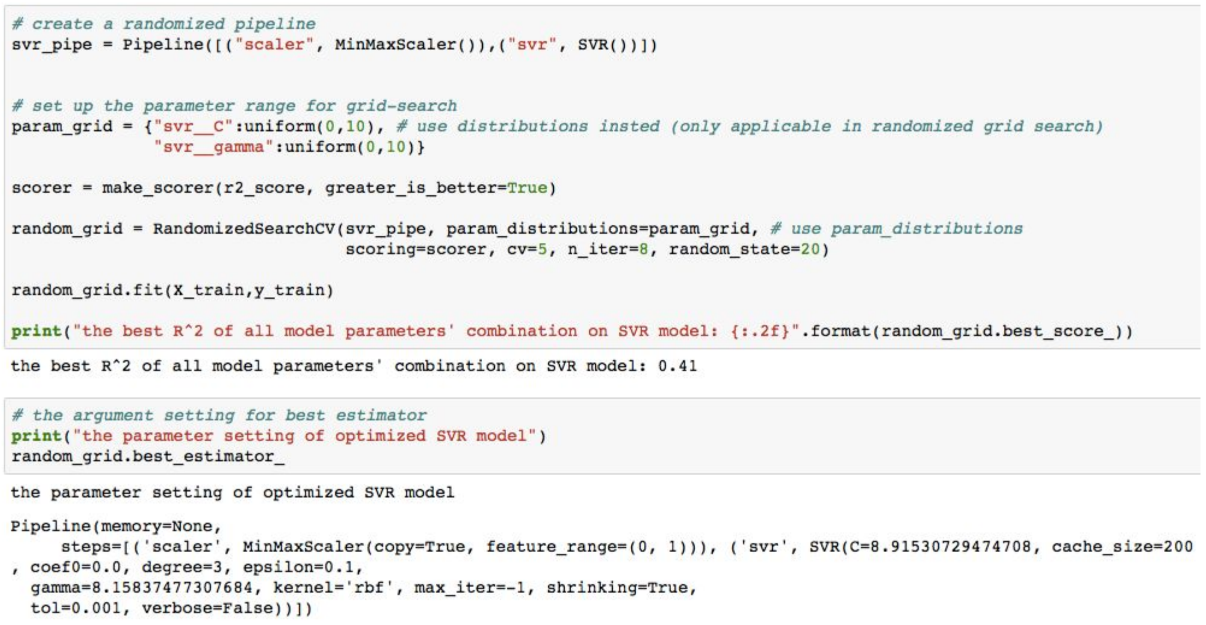
[7] Advice from Introduction to Machine Learning with Python

gamma: 0.16	gamma: 8.15

BRIEF RESULT:

As we can tell, the **RandomizedSearchCV** chose to have larger C and gamma, comparing to the default parameters, *implying that default model is still a bit of underfitting, the same result as we found in Lasso model*.

Code Snippet of Step 06:



```
# create a randomized pipeline
svr_pipe = Pipeline([("scaler", MinMaxScaler()),("svr", SVR())])


# set up the parameter range for grid-search
param_grid = {"svr__C":uniform(0,10), # use distributions insted (only applicable in randomized grid search)
              "svr__gamma":uniform(0,10)}

scorer = make_scorer(r2_score, greater_is_better=True)

random_grid = RandomizedSearchCV(svr_pipe, param_distributions=param_grid, # use param_distributions
                                 scoring=scorer, cv=5, n_iter=8, random_state=20)

random_grid.fit(X_train,y_train)

print("the best R^2 of all model parameters' combination on SVR model: {:.2f}".format(random_grid.best_score_))
```

```
the best R^2 of all model parameters' combination on SVR model: 0.41
```

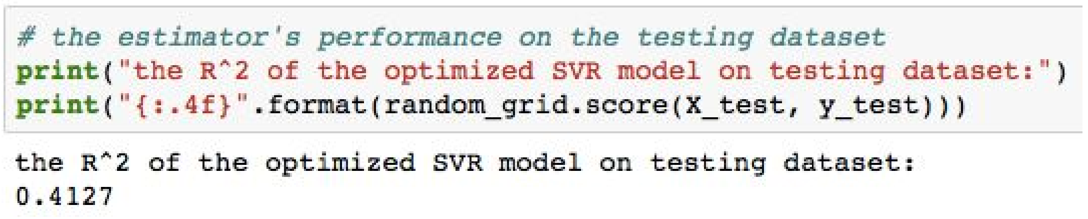
```
# the argument setting for best estimator
print("the parameter setting of optimized SVR model")
random_grid.best_estimator_
```

```
the parameter setting of optimized SVR model

Pipeline(memory=None,
     steps=[('scaler', MinMaxScaler(copy=True, feature_range=(0, 1))), ('svr', SVR(C=8.91530729474708, cache_size=200
, coef0=0.0, degree=3, epsilon=0.1,
  gamma=8.15837477307684, kernel='rbf', max_iter=-1, shrinking=True,
  tol=0.001, verbose=False))])
```

Step 07: optimized SVR model's performance on testing dataset

The **R^2** score for testing dataset for SVR model is **0.41**, roughly the same as we see in cross-validation R^2 score. In addition, the testing dataset is the same one used in Lasso model.

Code Snippet of Step 07:



```
# the estimator's performance on the testing dataset
print("the R^2 of the optimized SVR model on testing dataset:")
print("{:.4f}".format(random_grid.score(X_test, y_test)))
```

```
the R^2 of the optimized SVR model on testing dataset:
0.4127
```

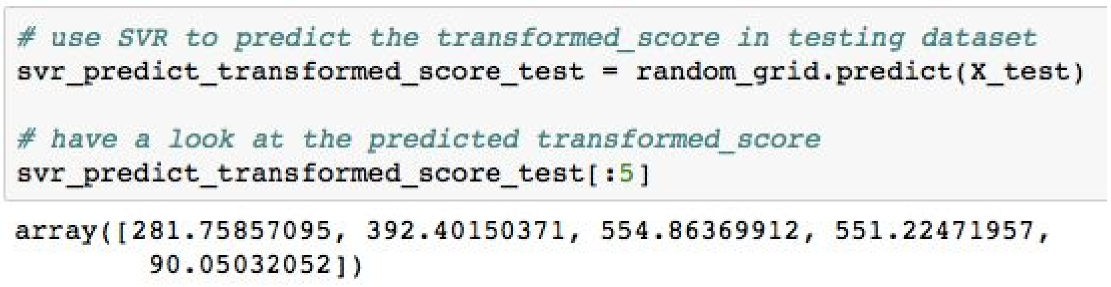
Step 08: choose better model and predict the rating for the testing dataset

As I use **R^2** as the criterion to decide which model(*Lasso or SVR*) performs better, it turns out **SVR** has a higher R^2 by **a relatively small margin**. I will use SVR as the final rating model and predict the **transformed_score** for the testing dataset, which I can compare with the true scores in later **Justification** section

The brief performance result on testing dataset between Lasso and SVR:

Lasso Model	Metrics	SVR model
0.38	**R^2 score**	0.41

Code Snippet of Step 08:



```
# use SVR to predict the transformed_score in testing dataset
svr_predict_transformed_score_test = random_grid.predict(X_test)

# have a look at the predicted transformed_score
svr_predict_transformed_score_test[:5]
```

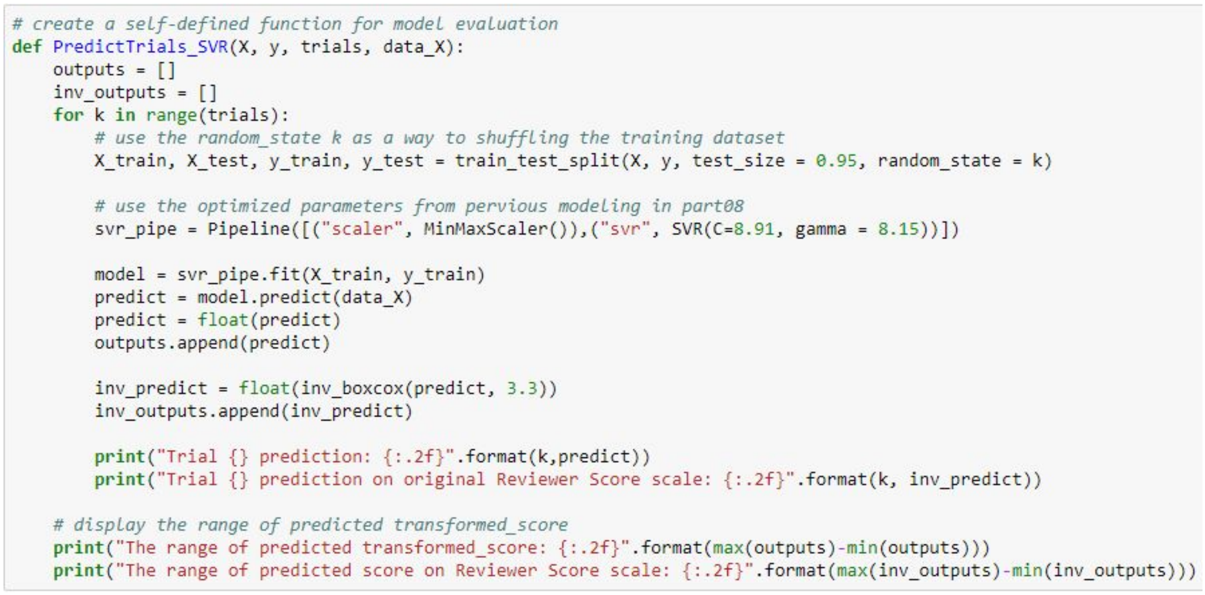
```
array([281.75857095, 392.40150371, 554.86369912, 551.22471957,
        90.05032052])
```

Results

Model Evaluation and Validation

In this section, I will implement sensitivity analysis on SVR rate prediction model. First, I take reference from the model evaluation function being used in the course assignment of *Predicting Boston Housing Prices*. The function - **PredictTrials**, uses different training dataset to build up the model and always predicts on the same data point in order to observe the variation. Therefore, from the predicted output of all models in different trials, we can evaluate the stability and validity of the model. I specifically modify the code and create another version to fit in this analysis case. My function - **PredictTrials_SVR** is defined as following:

Code for Self-defined PredictTrials_SVR Function



```python
# create a self-defined function for model evaluation
def PredictTrials_SVR(X, y, trials, data_X):
    outputs = []
    inv_outputs = []
    for k in range(trials):
        # use the random_state k as a way to shuffling the training dataset
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.95, random_state = k)

        # use the optimized parameters from pervious modeling in part08
        svr_pipe = Pipeline([("scaler", MinMaxScaler()),("svr", SVR(C=8.91, gamma = 8.15))])

        model = svr_pipe.fit(X_train, y_train)
        predict = model.predict(data_X)
        predict = float(predict)
        outputs.append(predict)

        inv_predict = float(inv_boxcox(predict, 3.3))
        inv_outputs.append(inv_predict)

        print("Trial {} prediction: {:.2f}".format(k,predict))
        print("Trial {} prediction on original Reviewer Score scale: {:.2f}".format(k, inv_predict))

    # display the range of predicted transformed_score
    print("The range of predicted transformed_score: {:.2f}".format(max(outputs)-min(outputs)))
    print("The range of predicted score on Reviewer Score scale: {:.2f}".format(max(inv_outputs)-min(inv_outputs)))
```

Along the process, I first extract out one data sample and also exclude the data sample from the training dataset. Then, I will implement for **5** trials to see the predicted value on both **boxcox transformed_score** and **score in original Reviewer Score scale**. The **PredictTrials_SVR** function also presents the variation range of score among these trials. Last, I also include the **actual transformed_score** and **its score in original scale**, so that we can tell how accurately the model's prediction does. The following table is the brief result of the sensitivity analysis.

Sensitivity Analysis on SVR Rate Prediction Model

Trial	**Predicted box-cox transformed_score**	**Predicted score in original scale**	**The actual box-cox transformed_score**	**The actual score in original scale**
1	315.57	8.21	233.96	7.50
2	313.27	8.20		
3	329.59	8.32		
4	327.72	8.31		
5	312.46	8.19		
The range of predicted transformed_score: 17.13				
The range of predicted score on Reviewer Score scale: 0.13				

Hence, from the table above, we can come to some conclusions.

1. From the trial result, it seems that the model's prediction is quite consistent. **The variation range(17.13)** of the predicted box-cox transformed_score only accounts for around **5.5%** of the average predicted transformed score. If we look at the score in original scale, **the variation range(0.13)** of the predicted score only accounts for around **1.6%** of the average predicted score in original scale. Both prove that SVR model provides quite consistent prediction, not influenced by different training dataset used for model building.
2. Nonetheless, if we compare the prediction to the **actual score**, we can tell that **SVR model seems a little over-estimating this data point**, where the prediction for the data point is around 310 to 330 in transformed scale and 8.1 to 8.3 in original scale. The actual value in transformed scale is 233 and 7.5 in original scale.

Justification

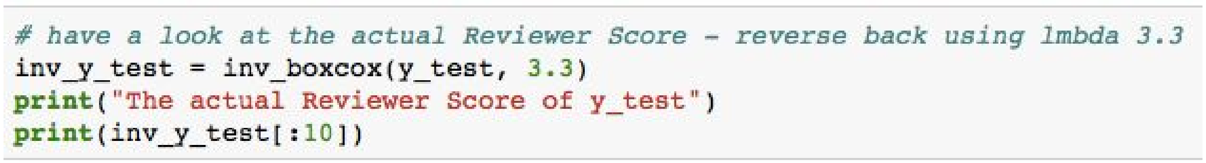
Before we compare the predicted value from SVR model to the actual values on score, let's review the evaluation benchmark of R^2. The range of R^2 spans from 0 to 1. Though there is no explicit criterion for judging the performance of continuous prediction model, in this project I choose **0.5**, the middle point of R^2, as the benchmark for model evaluation.

Apparently, both **Lasso(0.38)** and **SVR(0.41)** model fail to achieve R^2 of benchmark standard, although SVR still has a higher R^2.

Since SVR is my final model, now we can compare the predicted score from SVR to the actual score, to see how the model performs. Following first table is the first 5 sample data from **X_test**, **y_test** and its corresponding predicted value from SVR - **svr_predict_transformed_score_test**. Second table is the comparison specifically focused on the **lowest/highest score points(extreme points) in original scale**.

For a side note, the inverse transformation of transformed_score is done via **inv_boxcox()** from scipy module. As I know the **lmbda** for boxcox is **3.3**, I use it as the function parameter for inverse transformation back to original scale. The brief code as follows.

Code Snippet for Inverse Box-cox Transformation:



```
# have a look at the actual Reviewer Score - reverse back using lmbda 3.3
inv_y_test = inv_boxcox(y_test, 3.3)
print("The actual Reviewer Score of y_test")
print(inv_y_test[:10])
```

Comparison of Predicted and Actual Value

y_test transformed_score	SVR predict transformed _score	y_test in original scale	SVR predict in original scale

459.47	281.75	9.2	7.9
459.47	392.40	9.2	8.7
396.73	554.86	8.8	9.7
396.73	551.22	8.8	9.7
195.18	90.05	7.1	5.6

Comparison on Lowest/Highest Score Points

y_test lowest score points	SVR predict lowest score points	y_test highest score points	SVR predict highest score points
2.9	5.2	9.2	7.9
2.5	6.6	9.2	8.7
2.9	9.5	8.8	9.7
2.5	7.9	8.8	9.7
2.9	8.8	7.1	5.6

Throughout all these comparisons, I conclude with some judgements on the model performance.

1. It turns out that the SVR model seems not able to identify **low score** data points. **It largely over-estimates these data points with higher score.** As we see for those data points with actual score below 2(Reviewer Score), the SVR predictions still remain around score 5, even to score 9.
2. On the other hand, SVR model predictions seem quite consistent with data points which are actually high scores. As we see when the actual score goes down to 7 around, the SVR prediction also catches up the trend and echos to the result(prediction score 5.6).

Feature Importance

I use Lasso model as the benchmark model, so it comes to have bonus of insights on each explanatory feature's influence provided by the model. Besides, Lasso model has quite a competing explanatory power to SVR model, and thus its judgement on feature importance should be pretty persuasive as well. For reminding, Lasso model will suppress the coefficients of unimportant features close to 0. Hence, we can have a look at the table below.

Coefficients From Lasso Model:

```
# filter out the nonzero coefficients
nonzero_index = flatnonzero(grid.best_estimator_.coef_) # return the index for nonzero coefficients
nonzero_feature_name = X_train.columns[nonzero_index] # index out the feature name
nonzero_coef = grid.best_estimator_.coef_[nonzero_index] # index out the coefficient's value
```

```
# display the coefficients as sorted dataframe
value = nonzero_coef[nonzero_coef.argsort()[::-1]]
value = np.round(value, 6) # avoid scientific notation
feature = nonzero_feature_name[nonzero_coef.argsort()[::-1]]
lasso_coefficient = pd.DataFrame({"feature":feature, "value":value})
display(lasso_coefficient)
```

	feature	value
0	transformed_review_total_positive_word_counts	16.342751
1	quarter_transformed_score	0.684411
2	transformed_average_score	0.019337
3	quarter_previous_transformed_score	0.009739
4	mlp_predict_review_sentiment_positive	0.000000
5	transformed_review_total_negative_word_counts	-17.836231
6	mlp_predict_review_sentiment_negative	-40.343272

1. It is obvious that **transformed review total positive word counts** is quite influential to higher scores.
2. On the contrary, **transformed review total negative word counts** and **mlp predict review sentiment negative** are significantly correlated to lower scores.
3. One interesting discovery is that the **mlp predict review sentiment negative** seems quite useful in identifying low scores, *proving the value of text sentiment model to come up with derived feature - predicted review sentiment* . Nonetheless, it's not so powerful in identifying high score instead.
4. The remaining features such as **quarter transformed score** and **quarter previous transformed score**, although they have been created by feature engineering and sophisticated transformed, they still don't have much explanatory power in predicting the score. The result surprises me for originally I thought these derived features should outperform the original ones. In other words, the original features, for example, **positive/negative word counts** seem to already serve as better predictors, while the ones I created are not so useful in the end.

NOTICE:

Owing to the target variable and explanatory variables are being box-cox re-scaled, we can only interpret the importance of variables based on coefficients. The absolute value of coefficient doesn't represent its effect on the original target variable.

Conclusion

Reflection

Overviewing the whole project picture, first, I use the text review to build up LDA model, during the process, advance lemmatization and 4 topic parameter are configured. The interpretation of 4 topics are **related checkin - negative attitude**, **related service - positive attitude**, **related utility - negative attitude**, **related location - positive attitude**. In addition, I also assign each hotel to one of the topics based on the most frequent topic being referred to that hotel. While having a look at the review and its topic label, I think LDA model does quite a good job grasping the main tone of the review.

Secondly, in order to further taking advantage of the text review, I build up a text sentiment classifier using MLP model. In data pre-processing, the score rate is discretized to 0/1 to serve as target variable, and word token is liberated up to including bi-grams. Based on the performance on testing dataset, MLP reaches quite a high **accuracy, F1-score, and Recall**, all above **0.9**, exceeding the 0.89 threshold. However, the classes of the dataset is largely imbalanced. Thus, if we look at MLP's performance on minor class, its F1-score is only around **0.5**, and its recall is even lower - barely **0.35**. Therefore, we can judge that MLP may be good at identifying positive review, while not so good at identifying negative review.

Thirdly, I include the predicted text sentiment label from second model and start on building score rate prediction model. I use SVR, branch of SVM, as the final model. During the process, the predictors and target variable are box-cox transformed, and optimized grid-search is applied where the parameters being fine tuned are **C** and **gamma**. Again, the SVR model only reaches around **0.41 R^2**, which is unsatisfactory. Furthermore, comparing the predicted score to actual score, the model seems a bit of over-estimating scores and thus it is not able to splitting the actual low score samples out.

The funny part is that in the phase of **Data Preprocessing - Brief Discussion of Correlations**, these predictors are correlated to the target variable(transformed score) in a range around 0.3, 0.4 to 0.6. Originally I though with the combination of these features, I can achieve a more powerful prediction model. However, the R^2 of the final model still doesn't come to a **satisfactory level(above 0.5)**.

Hence, I figure maybe SVR model is not suitable in this case and dataset. I experiment with some other common models including **DecisionTreeRegressor**, **AdaBoosting** and even deep learning **Multi-layer Perceptron** again. It turns out these models still don't perform well for score rate prediction. The **R^2** for these models also lie around **0.3 to 0.42**, pretty close to SVR model's performance actually. The brief result of these model can be found in **Appendix**.

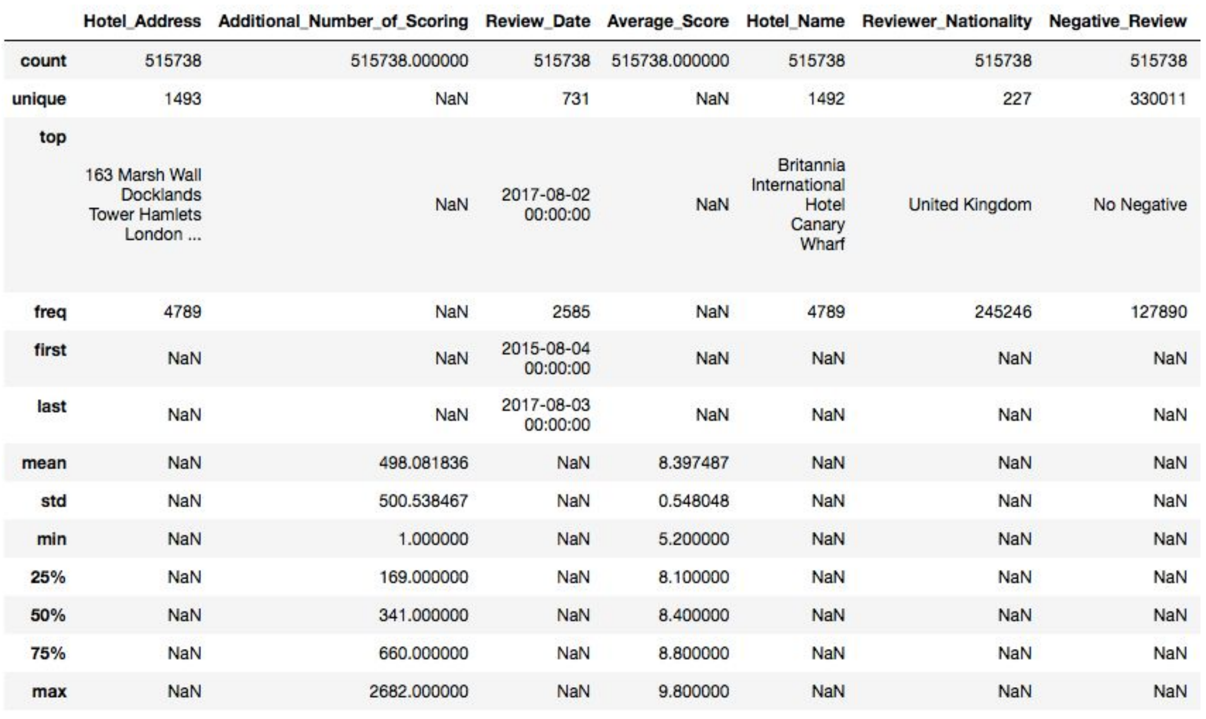
Therefore, in the current scenario, I suppose this is the finest model I can achieve for now.

Improvement

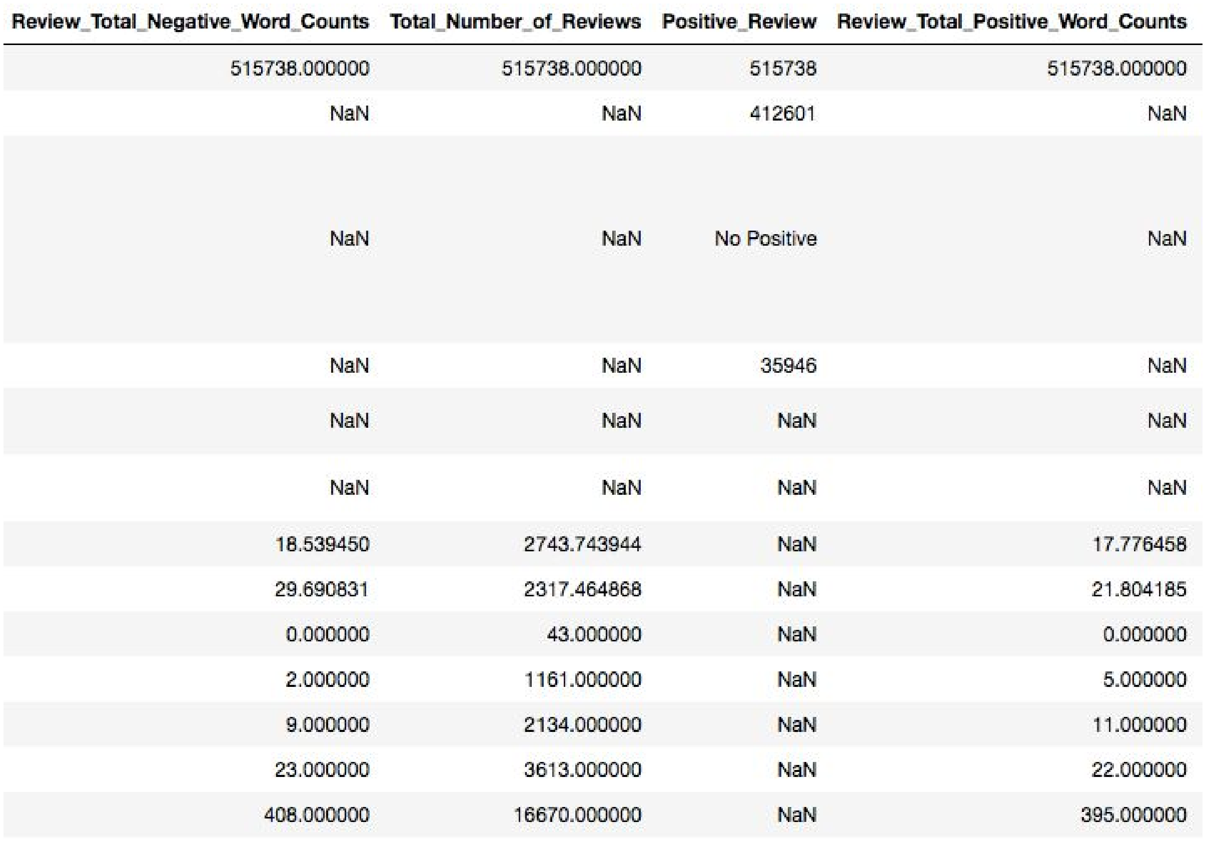
In the future experiment, maybe I can try **not to directly** build up a model predicting the continuous score variable. The score range only spans from 0 to 10, possibly it's better to **discretize them into 10 ordinal classes** and therefore build up a **multi-classes model**. Using this approach, perhaps I can come up with a model more capable of splitting samples between the low score and high score.

Appendix

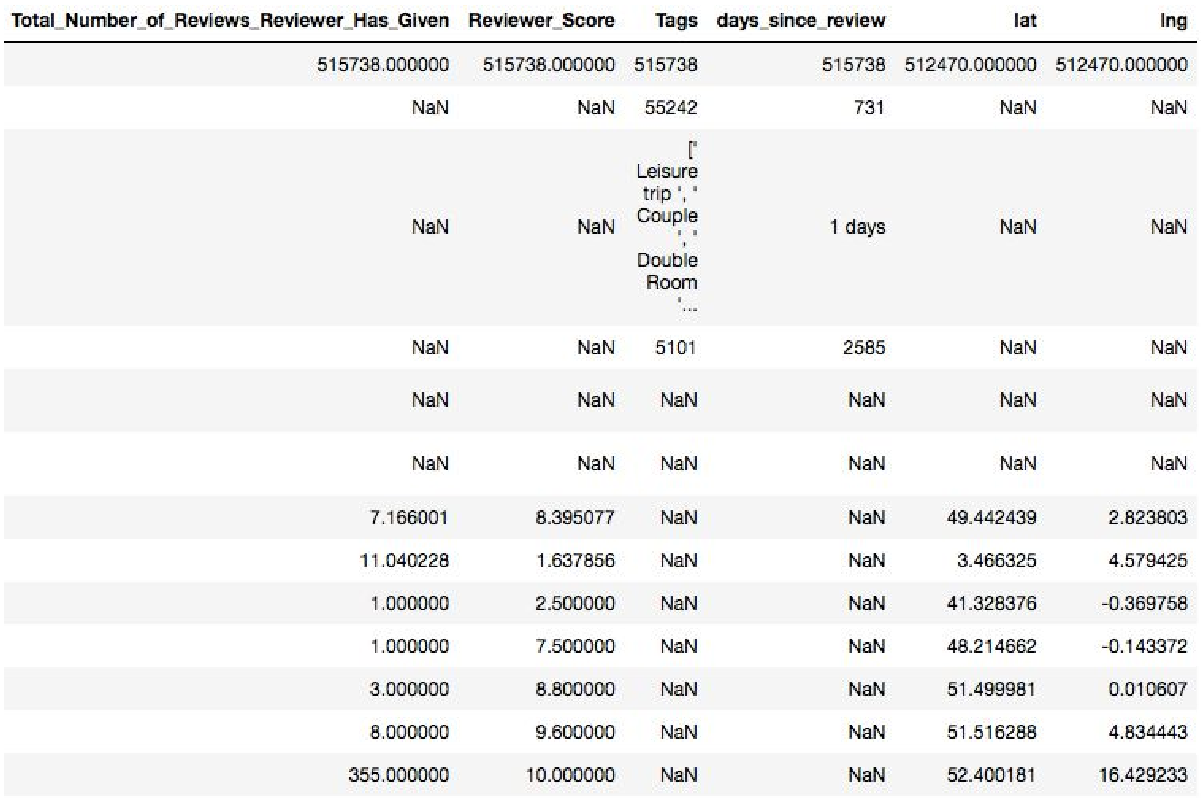
The comprehensive summary of descriptive statistics on each column:



	Hotel_Address	Additional_Number_of_Scoring	Review_Date	Average_Score	Hotel_Name	Reviewer_Nationality	Negative_Review
count	515738	515738.000000	515738	515738.000000	515738	515738	515738
unique	1493	NaN	731	NaN	1492	227	330011
top	163 Marsh Wall Docklands Tower Hamlets London ...	NaN	2017-08-02 00:00:00	NaN	Britannia International Hotel Canary Wharf	United Kingdom	No Negative
freq	4789	NaN	2585	NaN	4789	245246	127890
first	NaN	NaN	2015-08-04 00:00:00	NaN	NaN	NaN	NaN
last	NaN	NaN	2017-08-03 00:00:00	NaN	NaN	NaN	NaN
mean	NaN	498.081836	NaN	8.397487	NaN	NaN	NaN
std	NaN	500.538467	NaN	0.548048	NaN	NaN	NaN
min	NaN	1.000000	NaN	5.200000	NaN	NaN	NaN
25%	NaN	169.000000	NaN	8.100000	NaN	NaN	NaN
50%	NaN	341.000000	NaN	8.400000	NaN	NaN	NaN
75%	NaN	660.000000	NaN	8.800000	NaN	NaN	NaN
max	NaN	2682.000000	NaN	9.800000	NaN	NaN	NaN

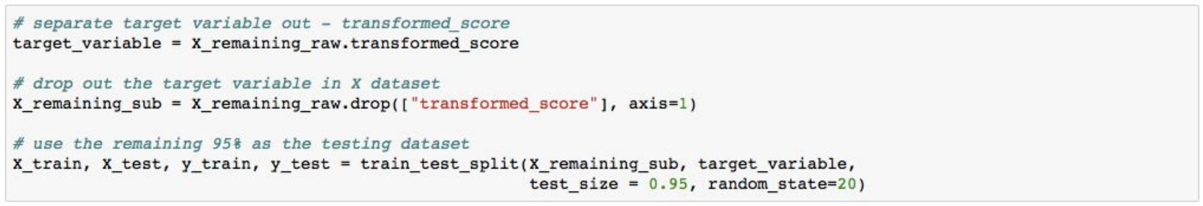


Review_Total_Negative_Word_Counts	Total_Number_of_Reviews	Positive_Review	Review_Total_Positive_Word_Counts
515738.000000	515738.000000	515738	515738.000000
NaN	NaN	412601	NaN
NaN	NaN	No Positive	NaN
NaN	NaN	35946	NaN
NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN
18.539450	2743.743944	NaN	17.776458
29.690831	2317.464868	NaN	21.804185
0.000000	43.000000	NaN	0.000000
2.000000	1161.000000	NaN	5.000000
9.000000	2134.000000	NaN	11.000000
23.000000	3613.000000	NaN	22.000000
408.000000	16670.000000	NaN	395.000000

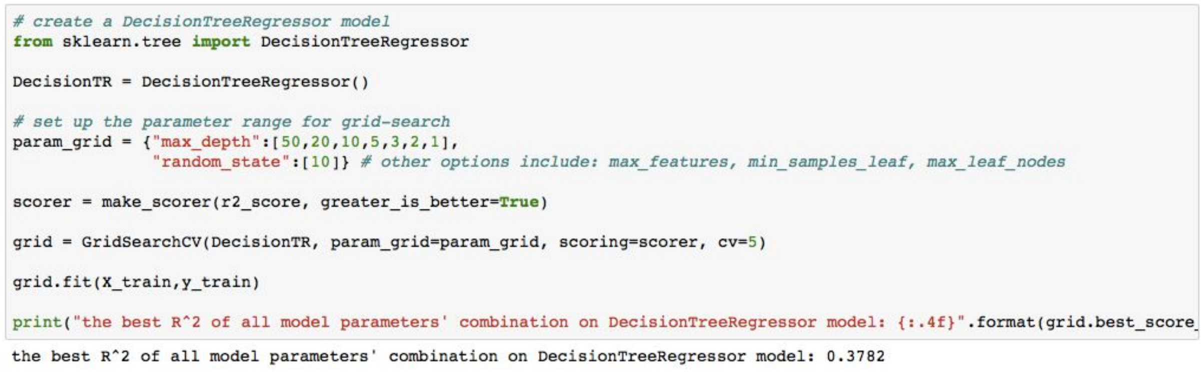


Total_Number_of_Reviews_Reviewer_Has_Given	Reviewer_Score	Tags	days_since_review	lat	lng
515738.000000	515738.000000	515738	515738	512470.000000	512470.000000
NaN	NaN	55242	731	NaN	NaN
NaN	NaN	[' Leisure trip ', ' Couple ', ' Double Room '...	1 days	NaN	NaN
NaN	NaN	5101	2585	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN
7.166001	8.395077	NaN	NaN	49.442439	2.823803
11.040228	1.637856	NaN	NaN	3.466325	4.579425
1.000000	2.500000	NaN	NaN	41.328376	-0.369758
1.000000	7.500000	NaN	NaN	48.214662	-0.143372
3.000000	8.800000	NaN	NaN	51.499981	0.010607
8.000000	9.600000	NaN	NaN	51.516288	4.834443
355.000000	10.000000	NaN	NaN	52.400181	16.429233

Try DecisionTreeRegressor for rate prediction model

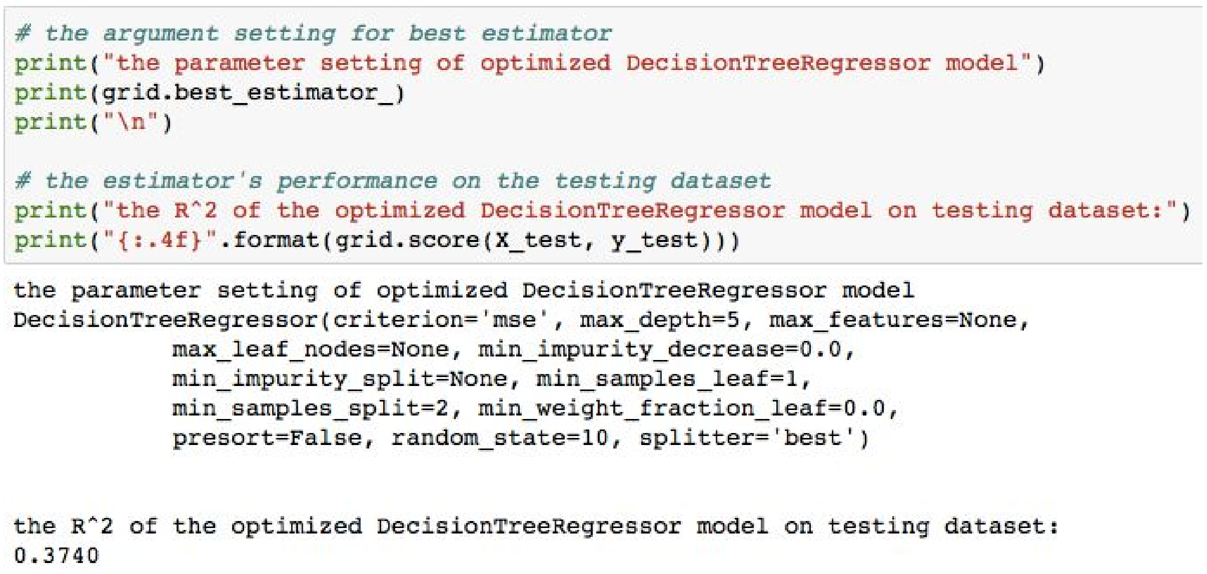


```
# separate target variable out - transformed_score
target_variable = X_remaining_raw.transformed_score

# drop out the target variable in X dataset
X_remaining_sub = X_remaining_raw.drop(["transformed_score"], axis=1)

# use the remaining 95% as the testing dataset
X_train, X_test, y_train, y_test = train_test_split(X_remaining_sub, target_variable,
                                                    test_size = 0.95, random_state=20)
```



```
# create a DecisionTreeRegressor model
from sklearn.tree import DecisionTreeRegressor

DecisionTR = DecisionTreeRegressor()

# set up the parameter range for grid-search
param_grid = {"max_depth":[50,20,10,5,3,2,1],
              "random_state":[10]} # other options include: max_features, min_samples_leaf, max_leaf_nodes

scorer = make_scorer(r2_score, greater_is_better=True)

grid = GridSearchCV(DecisionTR, param_grid=param_grid, scoring=scorer, cv=5)

grid.fit(X_train,y_train)

print("the best R^2 of all model parameters' combination on DecisionTreeRegressor model: {:.4f}".format(grid.best_score_
```

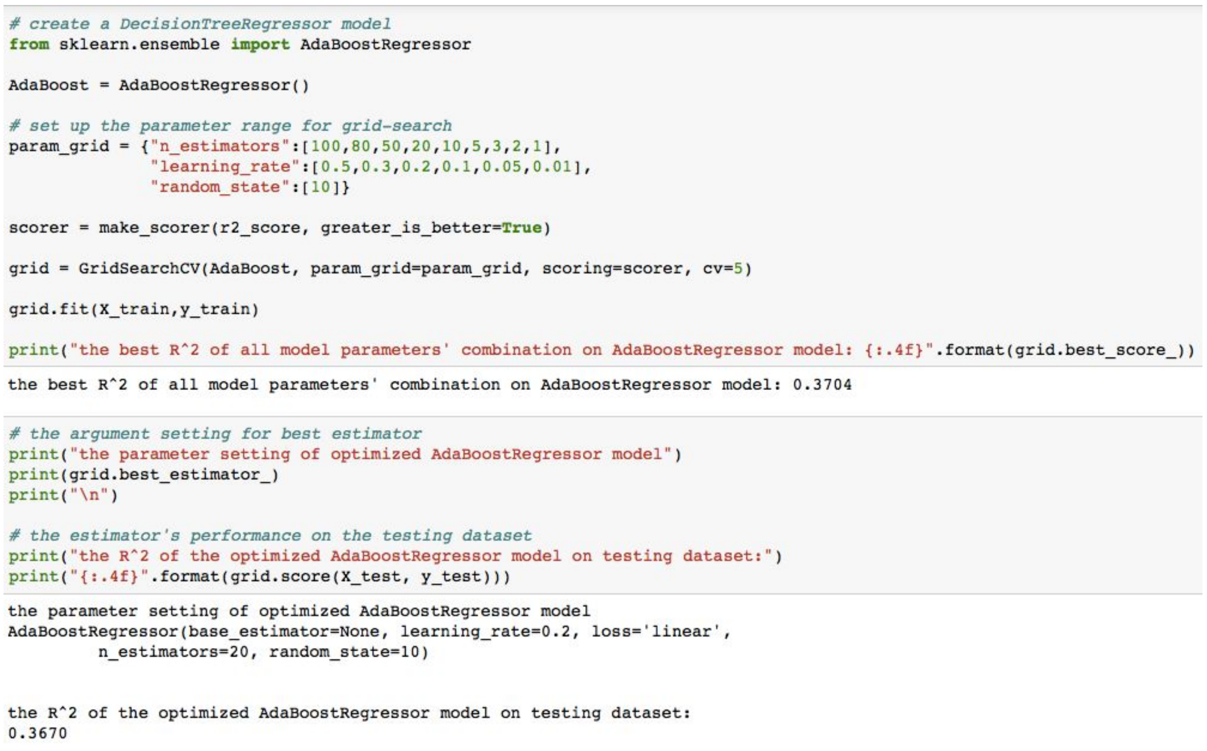
```
the best R^2 of all model parameters' combination on DecisionTreeRegressor model: 0.3782
```



```
# the argument setting for best estimator
print("the parameter setting of optimized DecisionTreeRegressor model")
print(grid.best_estimator_)
print("\n")

# the estimator's performance on the testing dataset
print("the R^2 of the optimized DecisionTreeRegressor model on testing dataset:")
print("{:.4f}".format(grid.score(X_test, y_test)))
```

```
the parameter setting of optimized DecisionTreeRegressor model
DecisionTreeRegressor(criterion='mse', max_depth=5, max_features=None,
           max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           presort=False, random_state=10, splitter='best')


the R^2 of the optimized DecisionTreeRegressor model on testing dataset:
0.3740
```

Try AdaBoosting for rate prediction model



```
# create a DecisionTreeRegressor model
from sklearn.ensemble import AdaBoostRegressor

AdaBoost = AdaBoostRegressor()

# set up the parameter range for grid-search
param_grid = {"n_estimators":[100,80,50,20,10,5,3,2,1],
              "learning_rate":[0.5,0.3,0.2,0.1,0.05,0.01],
              "random_state":[10]}

scorer = make_scorer(r2_score, greater_is_better=True)

grid = GridSearchCV(AdaBoost, param_grid=param_grid, scoring=scorer, cv=5)

grid.fit(X_train,y_train)

print("the best R^2 of all model parameters' combination on AdaBoostRegressor model: {:.4f}".format(grid.best_score_))
```

```
the best R^2 of all model parameters' combination on AdaBoostRegressor model: 0.3704
```

```
# the argument setting for best estimator
print("the parameter setting of optimized AdaBoostRegressor model")
print(grid.best_estimator_)
print("\n")

# the estimator's performance on the testing dataset
print("the R^2 of the optimized AdaBoostRegressor model on testing dataset:")
print("{:.4f}".format(grid.score(X_test, y_test)))
```

```
the parameter setting of optimized AdaBoostRegressor model
AdaBoostRegressor(base_estimator=None, learning_rate=0.2, loss='linear',
         n_estimators=20, random_state=10)


the R^2 of the optimized AdaBoostRegressor model on testing dataset:
0.3670
```

Try Multi-layer Perceptron for rate prediction model

```
from keras.utils import np_utils # encode categorical variable
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.callbacks import ModelCheckpoint, EarlyStopping
```

```
Using TensorFlow backend.
```

REF:custom R^2 score for keras

```
# custom R2-score metrics for keras backend
from keras import backend as K

def r2_keras(y_true, y_pred):
    SS_res =  K.sum(K.square(y_true - y_pred))
    SS_tot = K.sum(K.square(y_true - K.mean(y_true)))
    return ( 1 - SS_res/(SS_tot + K.epsilon()) )
```

```
# Building the model architecture
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(X_train.shape[1],)))
model.add(Dropout(0.2))
model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(1))
model.summary()

# Compiling the model using categorical_crossentropy loss, and rmsprop optimizer.
model.compile(loss='mean_squared_error',
              optimizer='rmsprop',
              metrics=['mae',r2_keras])
```

```python
# normalize input data
mean = X_train.mean(axis=0)
X_train = X_train - mean
std = X_train.std(axis=0)
X_train = X_train/std
```

```python
X_test = X_test - mean
X_test = X_test/std
```

```python
# Running and evaluating the model

checkpointer = ModelCheckpoint(filepath='rate.model.best.hdf5',
                               verbose=1, save_best_only=True)

earlystop = EarlyStopping(patience=2)

hist = model.fit(X_train.as_matrix(), y_train.as_matrix(),
          batch_size=50,
          epochs=20,
          validation_split=0.25,
          callbacks=[checkpointer, earlystop],
          verbose=2,
          shuffle=True)
```

```
Train on 8796 samples, validate on 2932 samples
Epoch 1/20
Epoch 00000: val_loss improved from inf to 21623.96487, saving model to rate.model.best.hdf5
1s - loss: 44850.7644 - mean_absolute_error: 164.9602 - r2_keras: -3.8778e-01 - val_loss: 21623.9649 - val_mean_absol
ute_error: 119.9491 - val_r2_keras: 0.3093
Epoch 2/20
Epoch 00001: val_loss improved from 21623.96487 to 20046.13426, saving model to rate.model.best.hdf5
1s - loss: 21537.4712 - mean_absolute_error: 119.9884 - r2_keras: 0.3451 - val_loss: 20046.1343 - val_mean_absolute_e
rror: 116.8699 - val_r2_keras: 0.3574
Epoch 3/20
Epoch 00002: val_loss improved from 20046.13426 to 19074.28822, saving model to rate.model.best.hdf5
1s - loss: 20558.0097 - mean_absolute_error: 116.4715 - r2_keras: 0.3746 - val_loss: 19074.2882 - val_mean_absolute_e
rror: 113.4209 - val_r2_keras: 0.3909
Epoch 4/20
Epoch 00003: val_loss did not improve
0s - loss: 20344.6784 - mean_absolute_error: 115.8818 - r2_keras: 0.3865 - val_loss: 19285.6775 - val_mean_absolute_e
rror: 116.2771 - val_r2_keras: 0.3847
Epoch 5/20
Epoch 00004: val_loss did not improve
0s - loss: 20139.2609 - mean_absolute_error: 114.9282 - r2_keras: 0.3913 - val_loss: 19343.6485 - val_mean_absolute_e
rror: 112.5561 - val_r2_keras: 0.3835
Epoch 6/20
Epoch 00005: val_loss did not improve
0s - loss: 20082.9943 - mean_absolute_error: 114.8177 - r2_keras: 0.3918 - val_loss: 20111.7924 - val_mean_absolute_e
rror: 113.5873 - val_r2_keras: 0.3597
```