Snap-it Evaluation-2

~Anurag Singh Naurka (IMT2020093)
~Rakshit Bang (IMT2020105)

Data Overview-

There are two files:
train.csv : 1260154 x 8
test.csv: 222381 x 7

The feature columns are-

- PRODUCT_ID - Id of the product
- PRODUCT_NAME - Name of the product
- PRODUCT_CONDITION - Condition of item on a scale of 5
- PRODUCT_DESCRIPTION - Description of the Product
- PRODUCT_BRAND - Brand of the product
- SHIPPING_AVAILABILITY - 1 if shipping fee paid by seller and 0 if paid by the seller
- CATEGORY
- PRODUCT_PRICE - the price of the item was sold (target variable)

Pre-processing (EDA+Text Processing)-

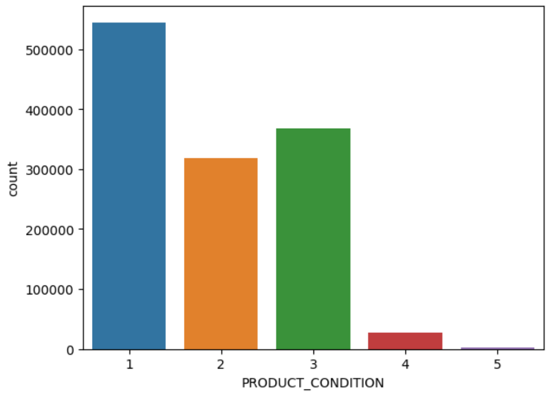
EDA-

1) PRODUCT_NAME -

```
count       1260154
unique      1052486
top          Bundle
freq           1873
Name: PRODUCT_NAME, dtype: object
```

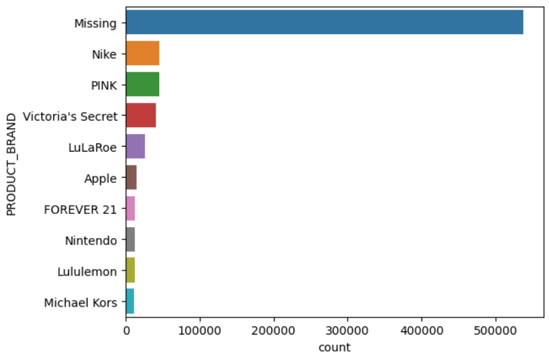
There are 10,52,486 unique product names and the most common name is *"Bundle"* which has a frequency of 1873.

2) PRODUCT_CONDITION -



The product condition is rated on a scale of 1 to 5. It can be seen that very few items i.e only 2023 items have a condition of 5.

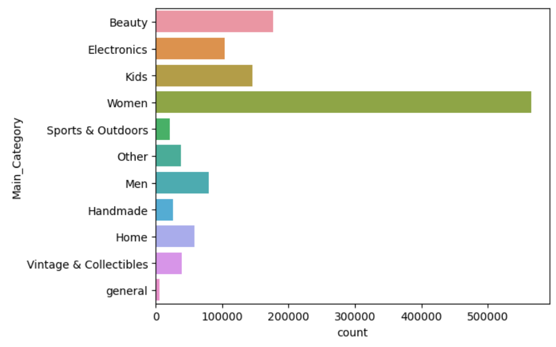
3) PRODUCT_BRAND -

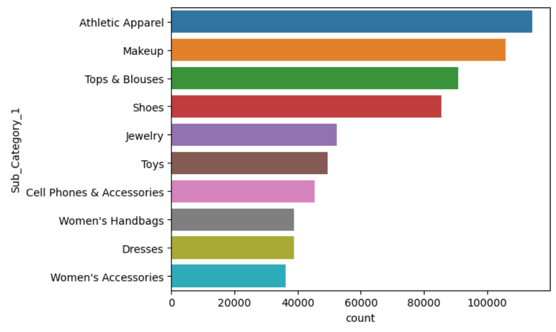


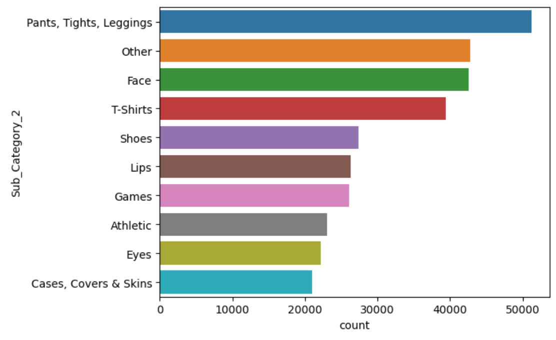
We can see that there are 537885 missing values of product brand which is very large with respect to the dataset. Hence, we cannot remove these NaN values. To tackle this we replace the missing values with *"local"*. After this we perform **one-hot encoding** on brand feature.

4) CATEGORY -

With observation we deduce that there are mainly three 3 sub-categories within this feature separated by *"/"*. So, we create 3 new features using this category feature called `"Main_Category" , "Sub_Category_1","Sub_Category_2"`. By this we get 11 unique values in Main_Category, 114 unique values in Sub_Category_1 and 864 unique values in Sub_Category_2.

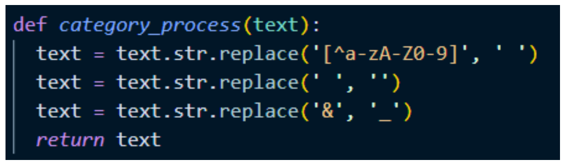






The most common method for converting a categorical variable into a numeric variable is one hot encoding because label encoding gives the impression that values are ranked. Thus, we use **one-hot encoding** on the category columns.

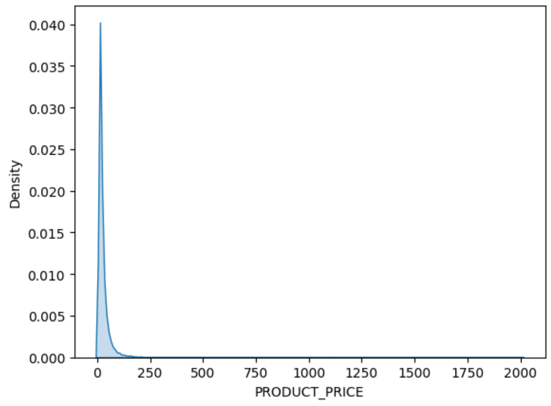
We also process the category column using the following function to refine the data-



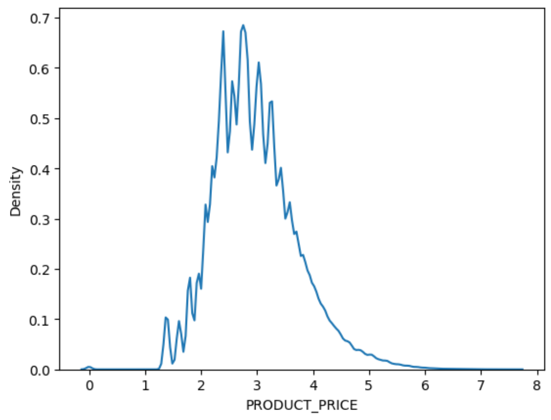
```
def category_process(text):
    text = text.str.replace('[^a-zA-Z0-9]', ' ')
    text = text.str.replace(' ', '')
    text = text.str.replace('&', '_')
    return text
```

5) PRODUCT_PRICE -

It can be seen from the plot below that the price range lies between 0 to 2000 but most of the items have a price less than 250.



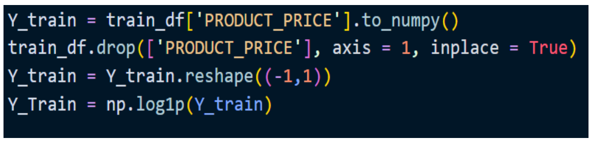
Distribution of price variable



Plot of price variable after log transformation.

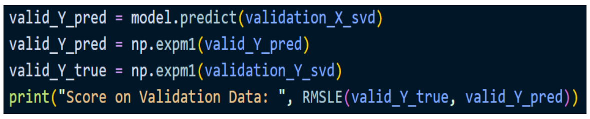
The plot above shows that the price feature has a right-skewed distribution. We know that because of points on the other side of the distribution, a skewed distribution would produce high Mean Squared Error (MSE) values but, if the data are normally distributed, the MSE is constrained. Thus, we take log transformation of price variables using log1p for training models and further analysis. We use expm1 to recover the original price values.

log1p(): The log1p() function returns the natural logarithm (base e) of 1 + x , where x is the argument.



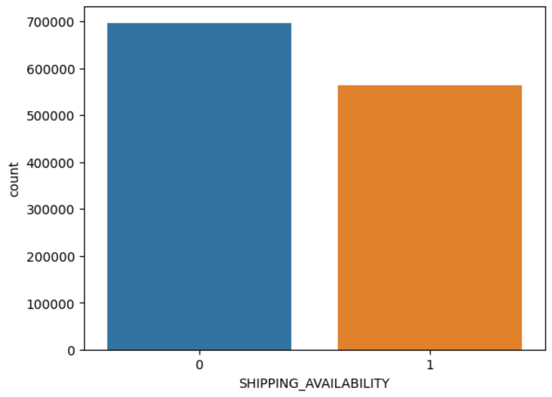
```
Y_train = train_df['PRODUCT_PRICE'].to_numpy()
train_df.drop(['PRODUCT_PRICE'], axis = 1, inplace = True)
Y_train = Y_train.reshape((-1,1))
Y_Train = np.log1p(Y_train)
```

expm1(): The Math.expm1() function returns e raised to the power of a number, subtracted by 1.



```
valid_Y_pred = model.predict(validation_X_svd)
valid_Y_pred = np.expm1(valid_Y_pred)
valid_Y_true = np.expm1(validation_Y_svd)
print("Score on Validation Data: ", RMSLE(valid_Y_true, valid_Y_pred))
```

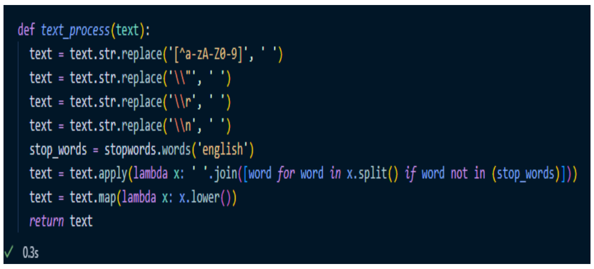
6) SHIPPING_AVAILABILITY-
Shipping cost is evenly shared between buyers and sellers, with the sellers covering more than half of the shipping costs (55%).



Text Processing:

There are 2 text fields i.e. PRODUCT_NAME and PRODUCT_DESCRIPTION.
To refine our data we process this text in the following ways:

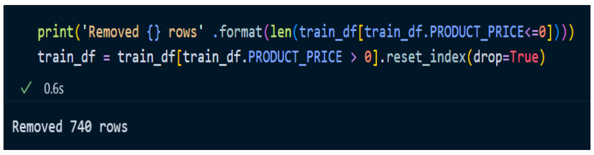
- Removal of non-alphanumeric characters
- Removing regular expressions
- Stopwords removal



```
def text_process(text):
    text = text.str.replace('[^a-zA-Z0-9]', ' ')
    text = text.str.replace('\\"', ' ')
    text = text.str.replace('\\r', ' ')
    text = text.str.replace('\\n', ' ')
    stop_words = stopwords.words('english')
    text = text.apply(lambda x: ' '.join([word for word in x.split() if word not in (stop_words)]))
    text = text.map(lambda x: x.lower())
    return text
✓ 0.3s
```

Lemmatization: Lemmatization is the process of grouping together the different inflected forms of a word so they can be analysed as a single item.
We have not been able to find an optimised way to lemmatize such a large dataset. Using lemmatization was giving insufficient memory error.

Negative price handling -
In the training data, price feature has values which are less than 0, which is invalid as the price of a
product cannot be negative. There are 740 such values and hence we remove these rows.



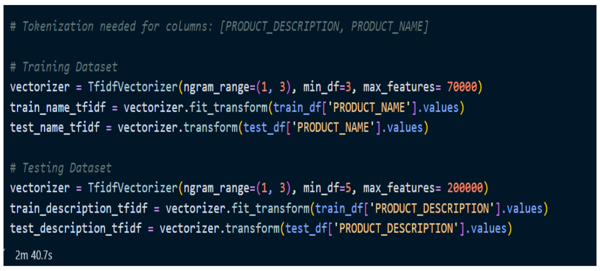
```
print('Removed {} rows' .format(len(train_df[train_df.PRODUCT_PRICE<=0])))
train_df = train_df[train_df.PRODUCT_PRICE > 0].reset_index(drop=True)
✓ 0.6s

Removed 740 rows
```

Tokenization: Tf-Idf

Tokenization is the process of tokenizing or splitting a string, text into a list of tokens. We used Tf-Idf for tokenization.

Term Frequency–inverse Document Frequency says that if a word is used extensively in all documents, its existence within a specific document will not be able to provide us much specific information about the document itself. So the second term could be seen as a penalty term that penalises common words such as "a", "the", "and", etc. tf-idf can therefore be seen as a weighting scheme for word relevancy in a specific document.

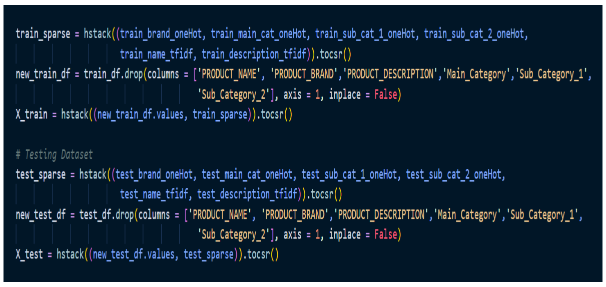
In our scenario we can use a TfidfVectorizer with an n_gram range of 1 to 3 and the maximum features of 70,000 for product name and same n_gram with maximum features of 2,00,000 for product description.



```
# Tokenization needed for columns: [PRODUCT_DESCRIPTION, PRODUCT_NAME]

# Training Dataset
vectorizer = TfidfVectorizer(ngram_range=(1, 3), min_df=3, max_features= 70000)
train_name_tfidf = vectorizer.fit_transform(train_df['PRODUCT_NAME'].values)
test_name_tfidf = vectorizer.transform(test_df['PRODUCT_NAME'].values)

# Testing Dataset
vectorizer = TfidfVectorizer(ngram_range=(1, 3), min_df=5, max_features= 200000)
train_description_tfidf = vectorizer.fit_transform(train_df['PRODUCT_DESCRIPTION'].values)
test_description_tfidf = vectorizer.transform(test_df['PRODUCT_DESCRIPTION'].values)
 2m 40.7s
```

Sparse Matrix

Sparse matrices store data that contains a large number of zero-valued elements. Sparse matrices have significant advantages in terms of computational efficiency. Unlike operations with full matrices, operations with sparse matrices do not perform unnecessary low-level arithmetic, such as zero-adds. The resulting efficiencies can lead to dramatic improvements in execution time for programs working with large amounts of sparse data. Thus, we convert our training data into a sparse matrix.



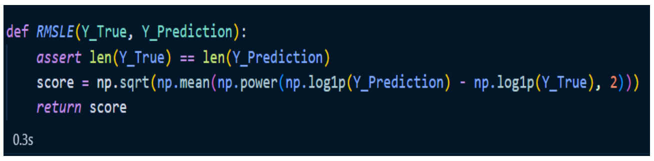
```
train_sparse = hstack((train_brand_oneHot, train_main_cat_oneHot, train_sub_cat_1_oneHot, train_sub_cat_2_oneHot,
                       train_name_tfidf, train_description_tfidf)).tocsr()
new_train_df = train_df.drop(columns = ['PRODUCT_NAME', 'PRODUCT_BRAND','PRODUCT_DESCRIPTION','Main_Category','Sub_Category_1',
                                        'Sub_Category_2'], axis = 1, inplace = False)
X_train = hstack((new_train_df.values, train_sparse)).tocsr()

# Testing Dataset
test_sparse = hstack((test_brand_oneHot, test_main_cat_oneHot, test_sub_cat_1_oneHot, test_sub_cat_2_oneHot,
                       test_name_tfidf, test_description_tfidf)).tocsr()
new_test_df = test_df.drop(columns = ['PRODUCT_NAME', 'PRODUCT_BRAND','PRODUCT_DESCRIPTION','Main_Category','Sub_Category_1',
                                        'Sub_Category_2'], axis = 1, inplace = False)
X_test = hstack((new_test_df.values, test_sparse)).tocsr()
```

- We have one-hot encoded the columns brand, main category, sub category 1 and sub category 2.
- We have tokenized name and description column.
- Both of the above-mentioned matrices contain mostly zeros. Thus, their concatenation gives us a sparse matrix of our training data.

After concatenation the shape of the train data becomes, `(1259414, 276163)`.

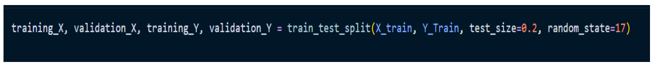
Performance Metric-

The evaluation metric for this competition is Root Mean Squared Logarithmic Error (RMSLE).



```
def RMSLE(Y_True, Y_Prediction):
    assert len(Y_True) == len(Y_Prediction)
    score = np.sqrt(np.mean(np.power(np.log1p(Y_Prediction) - np.log1p(Y_True), 2)))
    return score
 0.3s
```

Train and Test data Splitting-



```
training_X, validation_X, training_Y, validation_Y = train_test_split(X_train, Y_Train, test_size=0.2, random_state=17)
```

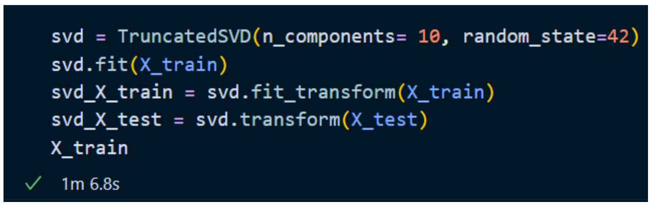
We split our training data into training and validation dataset so we can evaluate our model efficiency against unknown data points.

Selecting Model-

As our target feature is price we are dealing with a regression problem. We have learned in our course we can apply the following models -

1) **Linear Regression**
2) **Random Forest Regression**
3) **Ridge Regression**
4) Neural Networks

SVD - Singular Value Decomposition (SVD) is a widely used technique to decompose a matrix into several component matrices, exposing many of the useful and interesting properties of the original matrix.

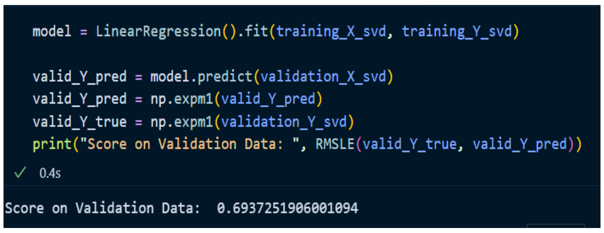


```
svd = TruncatedSVD(n_components= 10, random_state=42)
svd.fit(X_train)
svd_X_train = svd.fit_transform(X_train)
svd_X_test = svd.transform(X_test)
X_train
✓ 1m 6.8s
```

We use SVD reduction because most models are inefficient for large feature set.

Linear Regression -

Linear Regression is a machine learning algorithm based on **supervised learning**. It performs a **regression task**. Regression models a target prediction value based on independent variables.



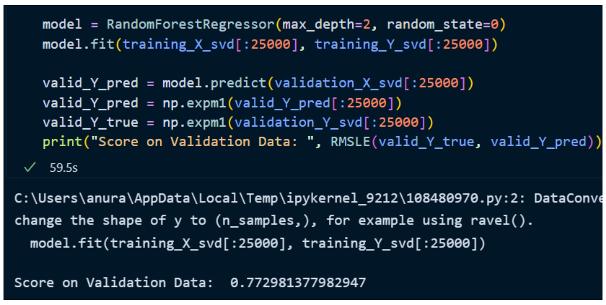
```
model = LinearRegression().fit(training_X_svd, training_Y_svd)

valid_Y_pred = model.predict(validation_X_svd)
valid_Y_pred = np.expm1(valid_Y_pred)
valid_Y_true = np.expm1(validation_Y_svd)
print("Score on Validation Data: ", RMSLE(valid_Y_true, valid_Y_pred))
✓ 0.4s

Score on Validation Data:  0.6937251906001094
```

Here we use linear regression on training data with reduced features (n=10) using **SVD** because directly using linear regression on original training data is highly memory inefficient.

RMSLE on Validation Data:
`0.6937251906001094`

Random Forest Regression -

Random Forest algorithm combines ensemble learning methods with the decision tree framework to create multiple randomly drawn decision trees from the data, averaging the results to output a new result that often leads to strong predictions/classifications.



```
model = RandomForestRegressor(max_depth=2, random_state=0)
model.fit(training_X_svd[:25000], training_Y_svd[:25000])

valid_Y_pred = model.predict(validation_X_svd[:25000])
valid_Y_pred = np.expm1(valid_Y_pred[:25000])
valid_Y_true = np.expm1(validation_Y_svd[:25000])
print("Score on Validation Data: ", RMSLE(valid_Y_true, valid_Y_pred))
✓ 59.5s

C:\Users\anura\AppData\Local\Temp\ipykernel_9212\108480970.py:2: DataConve
change the shape of y to (n_samples,), for example using ravel().
  model.fit(training_X_svd[:25000], training_Y_svd[:25000])

Score on Validation Data:  0.772981377982947
```

The concept of random forest is to group a lot of very deep trees. But growing deep trees eats a lot of resources.Playing with parameters like max.depth and num.trees help to reduce computational time. Still, they are not ideal.

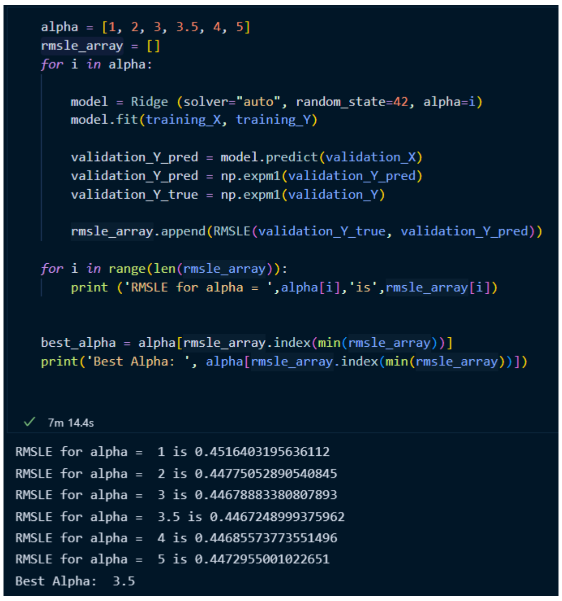
On using this model on our training data it leads to memory crash. Thus, we train the model for a sample of 25,000 data points and we get the below result.

RMSLE on Validation Data:
`0.772981377982947`

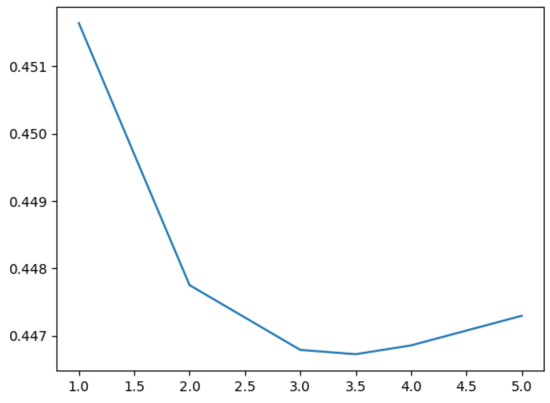
Ridge Regression (L2 Norm)-

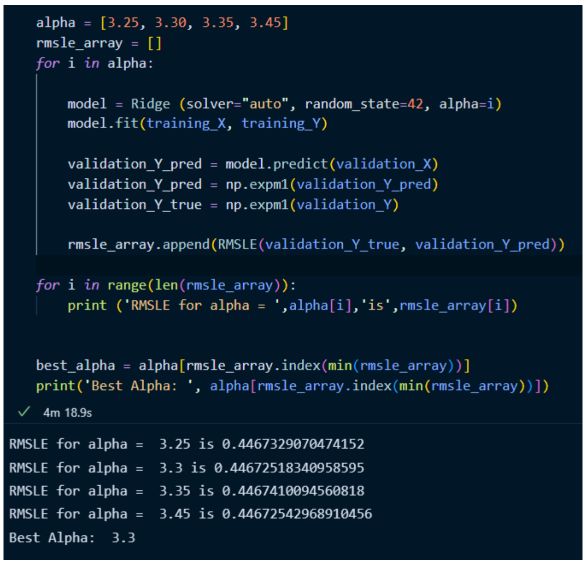
Ridge regression is a regularised regression algorithm that performs **L2 regularisation** that adds an L2 penalty, which equals the square of the magnitude of coefficients. All coefficients are shrunk by the same factor i.e none are eliminated. L2 regularisation will not result in sparse models. Ridge regression adds bias to make the estimates reliable approximations to true population values.

α is a constant that multiplies the L2 term, controlling regularisation strength. Here we try to find an optimal value of alpha that provides the best fitting.



```
alpha = [1, 2, 3, 3.5, 4, 5]
rmsle_array = []
for i in alpha:

    model = Ridge (solver="auto", random_state=42, alpha=i)
    model.fit(training_X, training_Y)

    validation_Y_pred = model.predict(validation_X)
    validation_Y_pred = np.expm1(validation_Y_pred)
    validation_Y_true = np.expm1(validation_Y)

    rmsle_array.append(RMSLE(validation_Y_true, validation_Y_pred))

for i in range(len(rmsle_array)):
    print ('RMSLE for alpha = ',alpha[i],'is',rmsle_array[i])


best_alpha = alpha[rmsle_array.index(min(rmsle_array))]
print('Best Alpha: ', alpha[rmsle_array.index(min(rmsle_array))])

✓ 7m 14.4s

RMSLE for alpha =  1 is 0.4516403195636112
RMSLE for alpha =  2 is 0.44775052890540845
RMSLE for alpha =  3 is 0.44678883380807893
RMSLE for alpha =  3.5 is 0.4467248999375962
RMSLE for alpha =  4 is 0.44685573773551496
RMSLE for alpha =  5 is 0.4472955001022651
Best Alpha:  3.5
```





```
alpha = [3.25, 3.30, 3.35, 3.45]
rmsle_array = []
for i in alpha:

    model = Ridge (solver="auto", random_state=42, alpha=i)
    model.fit(training_X, training_Y)

    validation_Y_pred = model.predict(validation_X)
    validation_Y_pred = np.expm1(validation_Y_pred)
    validation_Y_true = np.expm1(validation_Y)

    rmsle_array.append(RMSLE(validation_Y_true, validation_Y_pred))

for i in range(len(rmsle_array)):
    print ('RMSLE for alpha = ',alpha[i],'is',rmsle_array[i])


best_alpha = alpha[rmsle_array.index(min(rmsle_array))]
print('Best Alpha: ', alpha[rmsle_array.index(min(rmsle_array))])
```

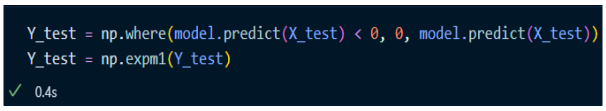
✓ 4m 18.9s

```
RMSLE for alpha =  3.25 is 0.4467329070474152
RMSLE for alpha =  3.3 is 0.446725183409585 95
RMSLE for alpha =  3.35 is 0.4467410094560818
RMSLE for alpha =  3.45 is 0.44672542968910456
Best Alpha:  3.3
```

From the above analysis we find that the best alpha for our model comes out to be **3.30**

RMSLE on Validation Data:
`0.407926547419 32827`

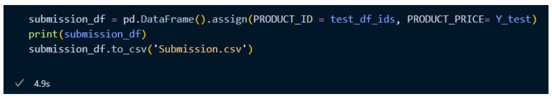
Despite our training data only contains positive values for price, Ridge Regressor might predict negative values and in our case we are getting one negative value
So, we handle this value by replacing them with 0.



```
Y_test = np.where(model.predict(X_test) < 0, 0, model.predict(X_test))
Y_test = np.expm1(Y_test)
```

✓ 0.4s

Creating Submission.csv:



```
submission_df = pd.DataFrame().assign(PRODUCT_ID = test_df_ids, PRODUCT_PRICE= Y_test)
print(submission_df)
submission_df.to_csv('Submission.csv')
```

✓ 4.9s