Mercari Price Suggestion Challenge

Given product details, Predict the sales price of the product.

Mercari: Your Marketplace

Created By: Samarjit Banerjee

Created On- 26-02-2022

Contents:

—--

Introduction

Mercari, Japan's biggest community-powered shopping app, knows this problem deeply. They'd like to offer pricing suggestions to sellers, but this is tough because their sellers are enabled to put just about anything, or any bundle of things, on Mercari' s marketplace. This provides a platform where customers can sell items that are no longer useful/unused products. It tries to make all the processes hassle-free by providing at-home pickups, same-day delivery, and many other advantages. The company website displays more than 350k items listed every day on the website which reflects its popularity among users.

Business Problem

The problem is easy to understand where, given the details of the product, the price for the product should be the output. When we pose this as a machine learning problem we call this out as a Regression Problem as the output is the real number(price). It can be treated as a Price Prediction Challenge for a product given its details.

Prerequisites

Here are the assumptions, reader should be having the understanding of Machine Learning, Deep Learning tool-kits and basic understanding of the ML/DL algorithms. As this is a regression problem predicting selling price of the products. Here are some ML algorithms understanding must have to continue with the rest of the content.

1. Linear Regression with L1&L2 Regularisation.
2. Decision Tree Regressor
3. Basic understanding of Clustering with K-Means Algorithm
4. Basic understanding of MLP, Embedding Laye

Evaluation

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

The RMSLE is calculated as

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

Where:

epsilon is the RMSLE(Root Mean Square Log Error) value (score)

n is the total number of observations in the (public/private) data set,

pi is your prediction of price, and

ai is the actual sale price for i.

log(x) is the natural logarithm of (x)

Data Source

Data Source : Here

Understanding the Data

- **train_id** or **test_id** — the id of the listing
- **name** — the title of the listing. Note that we have cleaned the data to remove text that
- looks like prices (e.g. $20) to avoid leakage. These removed prices are represented as [rm]
- **item_condition_id** — the condition of the items provided by the seller
- **category_name** — category of the listing
- **brand_name** — brand name of the product
- **price** — the price that the item was sold for. This is the target variable that you will predict. The unit is USD. This column doesn't exist in test.tsv since that is what you will predict.
- **shipping** — 1 if shipping fee is paid by seller and 0 by buyer
- **item_description** — the full description of the item. Note that we have cleaned the data to remove text that looks like prices (e.g. $20) to avoid leakage. These removed prices are represented as [rm]

Exploratory Data Analysis

This step includes to understand the data behaviour and its trends, different analysis between dependent and independent variables and try to gain as much knowledge about data which will help to build a good ML solution.

Missing values should be handled before fitting the data in any model. Analysis of the missing values and tackling these data-points are important steps before commencing data analysis.

Handling Duplicate Products

Duplicate values are redundant, and don't add any value. Requirement to identify those data points and removal of those points are important.

```
duplicate_Row_perc = (df_train[df_train.duplicated() == True].shape[0]/df_train.shape[0])*100
print('{} % products are duplicates.'.format(round(duplicate_Row_perc,5)))
```

```
0.00331 % products are duplicates.
```

```
df_train.drop_duplicates(keep = 'first',inplace=True)
```

```
df_train.shape
```

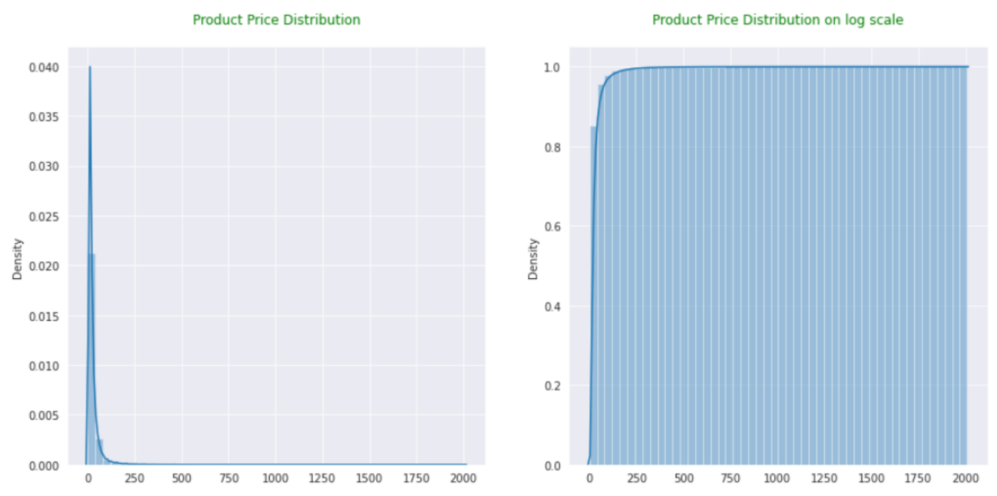
```
(1482486, 7)
```

Data Summary after removing duplicate values.

Analysis of the distribution of Product Selling Price

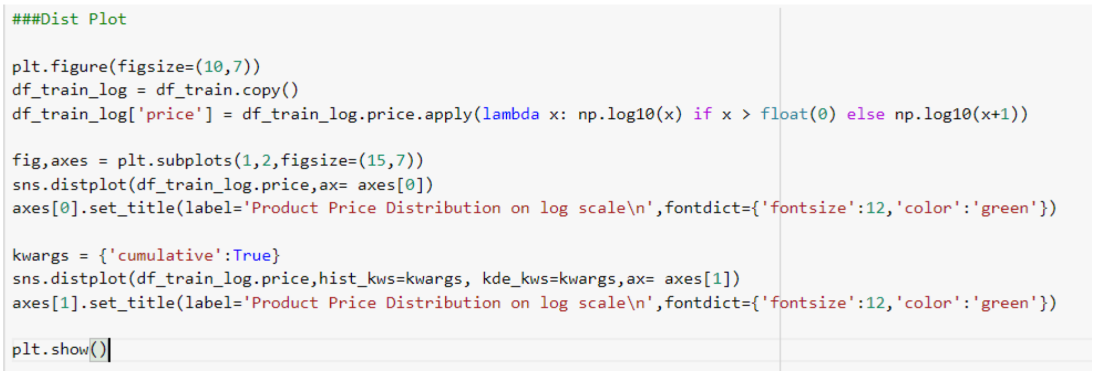
Selling price is the target variable and below is the analysis regarding the distribution of target variable. We can observe the distribution is target price left skewed.

```
###Dist Plot

fig,axes = plt.subplots(1,2,figsize=(15,7))
sns.distplot(df_train.price,ax=axes[0])
axes[0].set_title(label='Product Price Distribution\n',fontdict={'fontsize':12,'color':'green'})

kwargs = {'cumulative':True}
sns.distplot(df_train.price,hist_kws=kwargs, kde_kws=kwargs,ax= axes[1])
axes[1].set_title(label='Product Price Distribution on log scale\n',fontdict={'fontsize':12,'color':'green'})
plt.show()
```

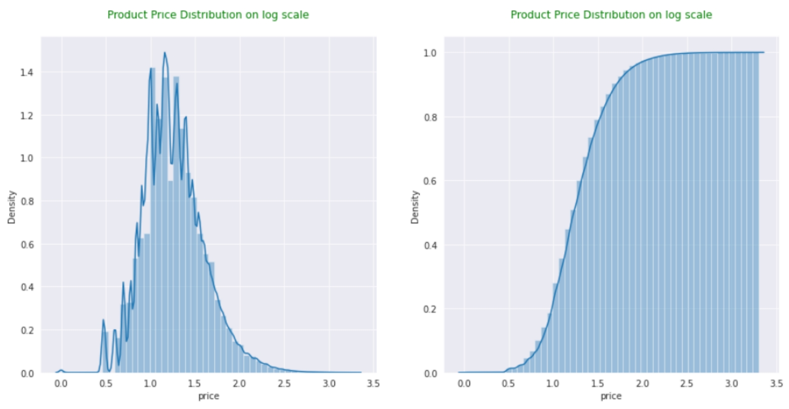


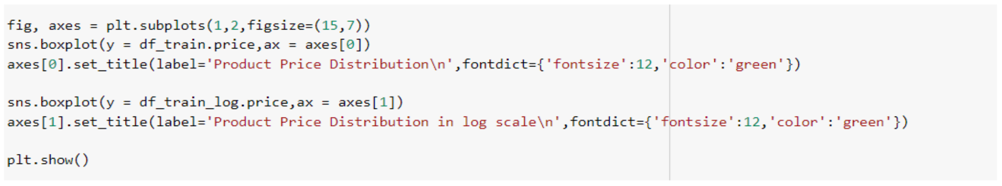
We could analyse that most products are having selling price less than 100$

- Distribution of price in log scale is similar to Gaussian(Normal) Distribution. Hence, Product price distribution is similar to log-normal.

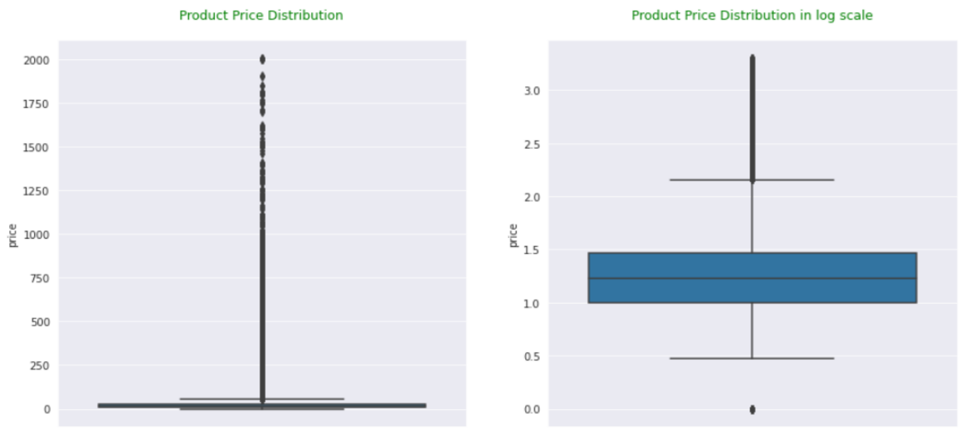


```
###Dist Plot

plt.figure(figsize=(10,7))
df_train_log = df_train.copy()
df_train_log['price'] = df_train_log.price.apply(lambda x: np.log10(x) if x > float(0) else np.log10(x+1))

fig,axes = plt.subplots(1,2,figsize=(15,7))
sns.distplot(df_train_log.price,ax= axes[0])
axes[0].set_title(label='Product Price Distribution on log scale\n',fontdict={'fontsize':12,'color':'green'})

kwargs = {'cumulative':True}
sns.distplot(df_train_log.price,hist_kws=kwargs, kde_kws=kwargs,ax= axes[1])
axes[1].set_title(label='Product Price Distribution on log scale\n',fontdict={'fontsize':12,'color':'green'})

plt.show()
```





```
fig, axes = plt.subplots(1,2,figsize=(15,7))
sns.boxplot(y = df_train.price,ax = axes[0])
axes[0].set_title(label='Product Price Distribution\n',fontdict={'fontsize':12,'color':'green'})

sns.boxplot(y = df_train_log.price,ax = axes[1])
axes[1].set_title(label='Product Price Distribution in log scale\n',fontdict={'fontsize':12,'color':'green'})

plt.show()
```



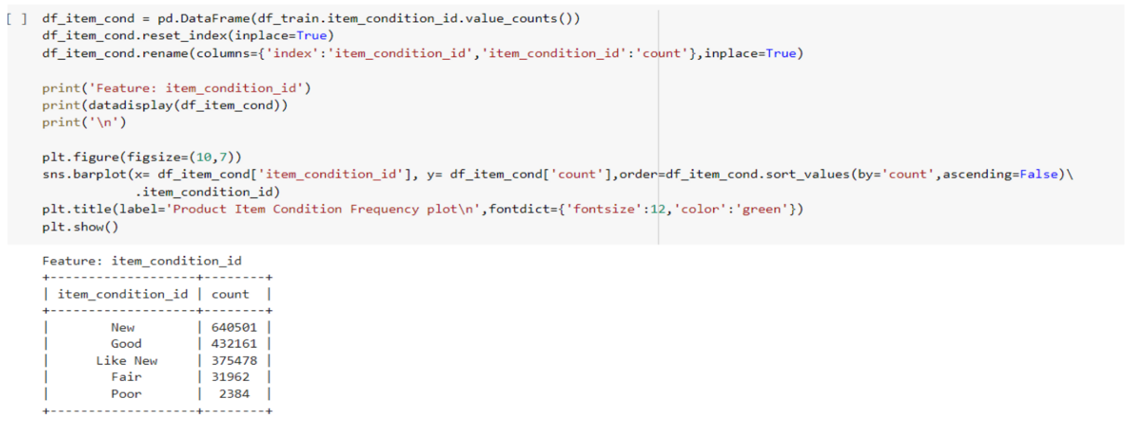
Key- Notes:

- price range from 0 to more than 3.5 in log scale.—(0—$2009 in actual scale)
- The 50% item selling price in log scale is around 1.2(USD).
- 25th percentile of items sold out @1.0(USD) and 75th percentile products sold out @1.45(USD). IQR(InterQuartile Range)—0.45.
- Zero price and very high price products are visible as outliers.

Feature: item_condition_id:

This Feature describes the condition of the item which is stepped into the platform for selling.

{1:’New’,2:’Like New’,3:’Good’,4:’Fair’,5:’Poor’}

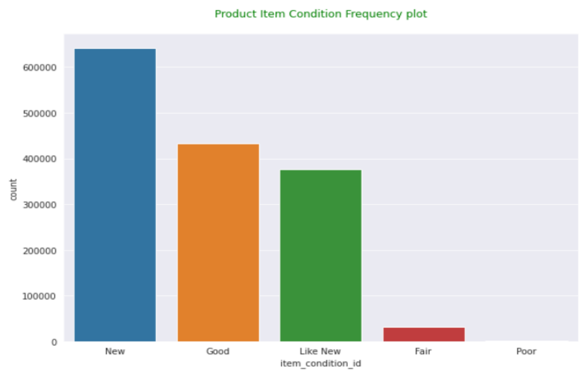


```
[ ] df_item_cond = pd.DataFrame(df_train.item_condition_id.value_counts())
    df_item_cond.reset_index(inplace=True)
    df_item_cond.rename(columns={'index':'item_condition_id','item_condition_id':'count'},inplace=True)

    print('Feature: item_condition_id')
    print(datadisplay(df_item_cond))
    print('\n')

    plt.figure(figsize=(10,7))
    sns.barplot(x= df_item_cond['item_condition_id'], y= df_item_cond['count'],order=df_item_cond.sort_values(by='count',ascending=False)\
              .item_condition_id)
    plt.title(label='Product Item Condition Frequency plot\n',fontdict={'fontsize':12,'color':'green'})
    plt.show()
```

```
    Feature: item_condition_id
    +-------------------+--------+
    | item_condition_id | count  |
    +-------------------+--------+
    |        New        | 640501 |
    |        Good       | 432161 |
    |      Like New     | 375478 |
    |        Fair       | 31962  |
    |        Poor       |  2384  |
    +-------------------+--------+
```

Key- Note:

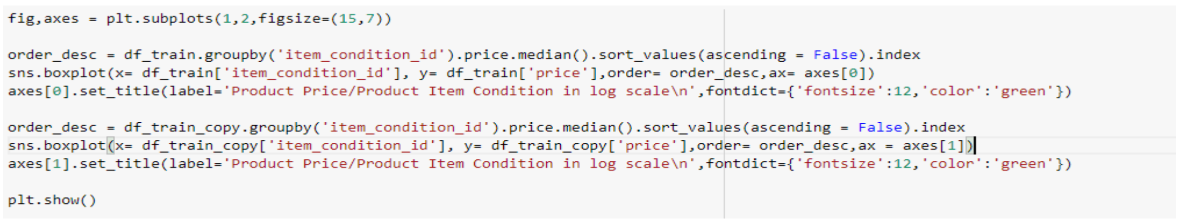
- New Products are launched in more quantity than other products, whereas Poor and Fair products are very less in quantity. Listed Products are imbalanced based upon product item conditions.



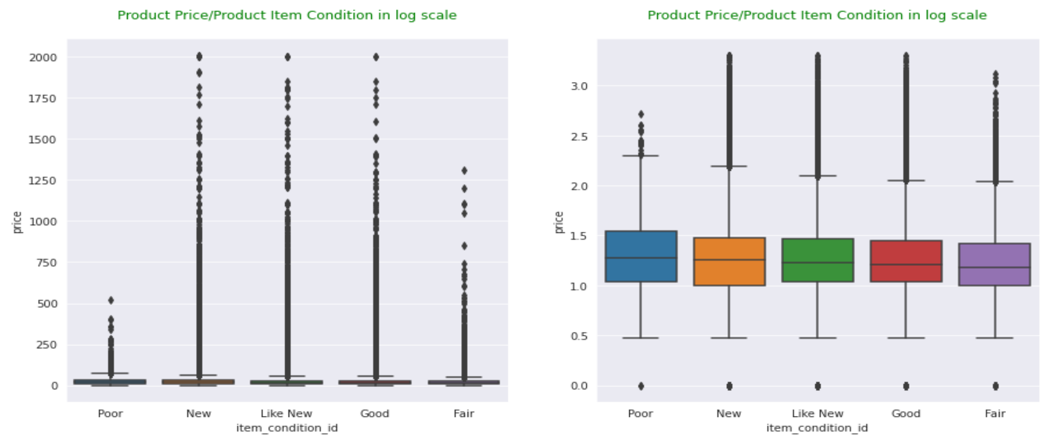
Key- Notes:

- Variation in frequency of different item condition id’s.
- Products in count with New condition are sold out way higher than products with Poor condition which should be expected.
- Small variation in products frequency between Good and Like New as products with Good/Like New are similar in nature.

Item Condition is not taking any part to identify the price, i.e. Item Condition and Item Price are not correlated.

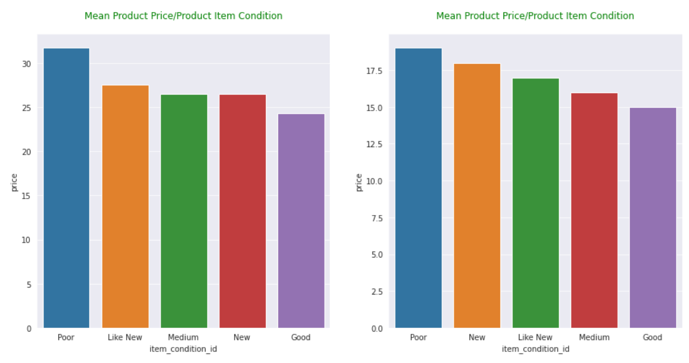


```
fig,axes = plt.subplots(1,2,figsize=(15,7))

order_desc = df_train.groupby('item_condition_id').price.median().sort_values(ascending = False).index
sns.boxplot(x= df_train['item_condition_id'], y= df_train['price'],order= order_desc,ax= axes[0])
axes[0].set_title(label='Product Price/Product Item Condition in log scale\n',fontdict={'fontsize':12,'color':'green'})

order_desc = df_train_copy.groupby('item_condition_id').price.median().sort_values(ascending = False).index
sns.boxplot(x= df_train_copy['item_condition_id'], y= df_train_copy['price'],order= order_desc,ax = axes[1])
axes[1].set_title(label='Product Price/Product Item Condition in log scale\n',fontdict={'fontsize':12,'color':'green'})

plt.show()
```



Key- Note:

- Products with Poor condition are sold out in higher avg. price than others which is odd. This might be a reason of product with poor conditions are of different categories like (eg. Electronics) which we will analyse more on it. From New products, avg. selling prices look in order.

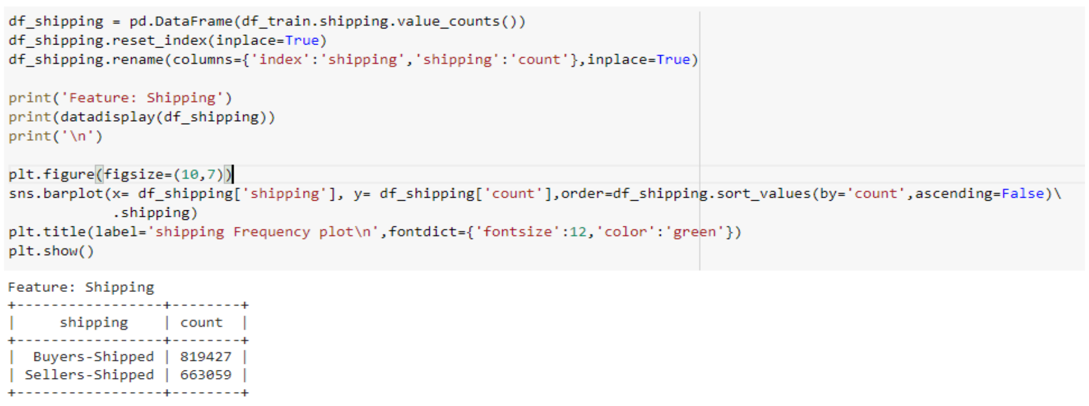


Key- Note:

- Items with Poor Condition are having a higher average price than others. Though condition id is an ordinal feature, The price for items with these conditions doesn't vary with the item conditions. As stated earlier, This might be a reason of product with poor conditions are of different categories like (eg. Electronics) which we will analyse more towards it.

Feature: shipping:

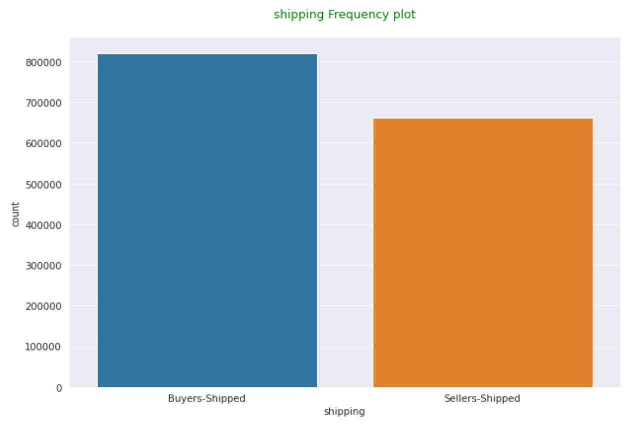
This feature states if the seller has taken the charge towards shipping the product to the buyer/not.

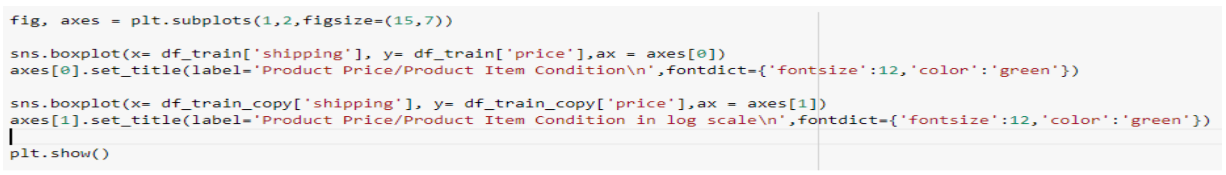


```
df_shipping = pd.DataFrame(df_train.shipping.value_counts())
df_shipping.reset_index(inplace=True)
df_shipping.rename(columns={'index':'shipping','shipping':'count'},inplace=True)

print('Feature: Shipping')
print(datadisplay(df_shipping))
print('\n')

plt.figure(figsize=(10,7))
sns.barplot(x= df_shipping['shipping'], y= df_shipping['count'],order=df_shipping.sort_values(by='count',ascending=False)\
            .shipping)
plt.title(label='shipping Frequency plot\n',fontdict={'fontsize':12,'color':'green'})
plt.show()
```

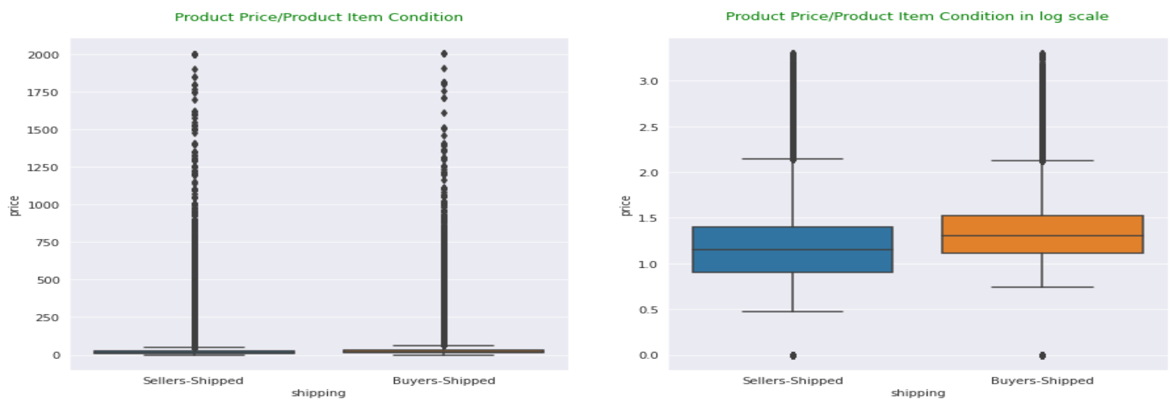
```
Feature: Shipping
+----------------+--------+
|    shipping    | count  |
+----------------+--------+
|  Buyers-Shipped | 819427 |
| Sellers-Shipped | 663059 |
+----------------+--------+
```

From the observation, we can conclude more buyers than sellers have taken charges towards the shipment of the product.





```
fig, axes = plt.subplots(1,2,figsize=(15,7))

sns.boxplot(x= df_train['shipping'], y= df_train['price'],ax = axes[0])
axes[0].set_title(label='Product Price/Product Item Condition\n',fontdict={'fontsize':12,'color':'green'})

sns.boxplot(x= df_train_copy['shipping'], y= df_train_copy['price'],ax = axes[1])
axes[1].set_title(label='Product Price/Product Item Condition in log scale\n',fontdict={'fontsize':12,'color':'green'})

plt.show()
```

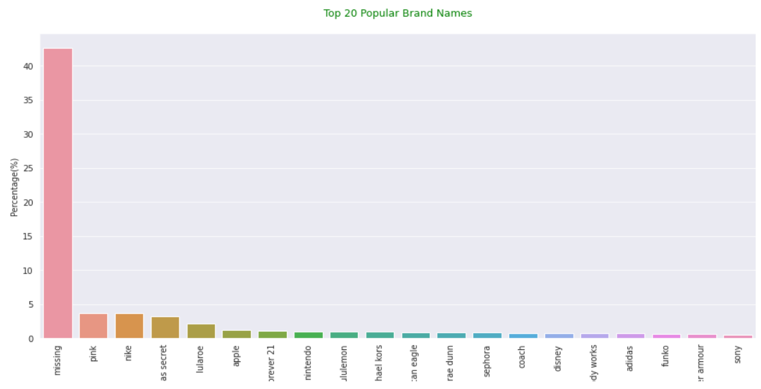


Key- Note:

- Buyers shipped products have higher avg. selling price.

Feature: brand_name:

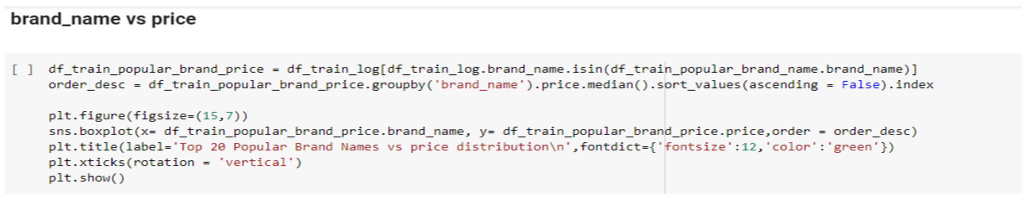
A brand name is **the name of the distinctive product**. Branding is the process of creating and disseminating the brand name.



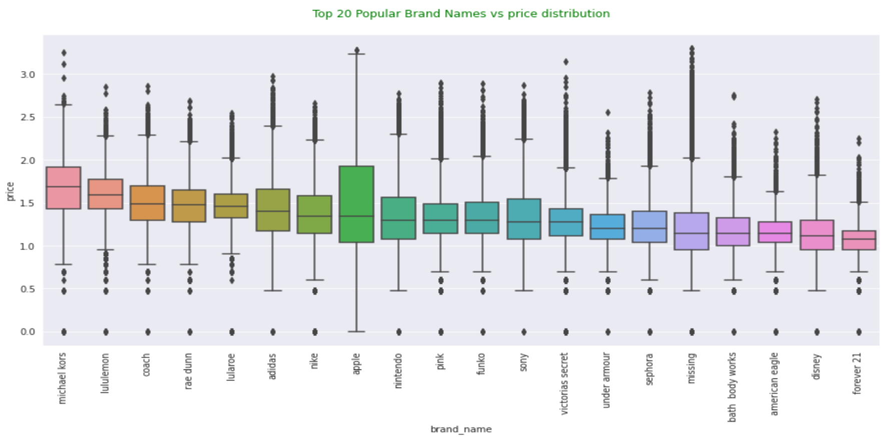
key- Note:

- Most of the products sold out without any brand name mentioned.
- Pink, Nike, Victoria’s Secret, Apple are among the top popular brands.

Feature: brand_name vs price(Multivariate Analysis)

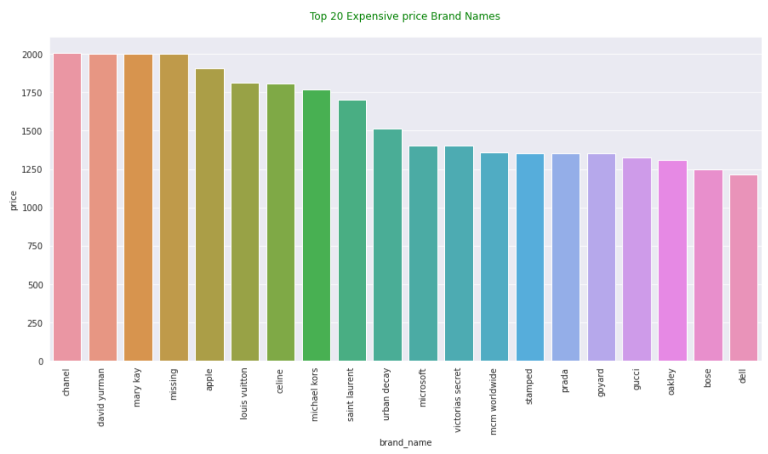

brand_name vs price

```
[ ] df_train_popular_brand_price = df_train_log[df_train_log.brand_name.isin(df_train_popular_brand_name.brand_name)]
    order_desc = df_train_popular_brand_price.groupby('brand_name').price.median().sort_values(ascending = False).index

    plt.figure(figsize=(15,7))
    sns.boxplot(x= df_train_popular_brand_price.brand_name, y= df_train_popular_brand_price.price,order = order_desc)
    plt.title(label='Top 20 Popular Brand Names vs price distribution\n',fontdict={'fontsize':12,'color':'green'})
    plt.xticks(rotation = 'vertical')
    plt.show()
```



Key- Note:

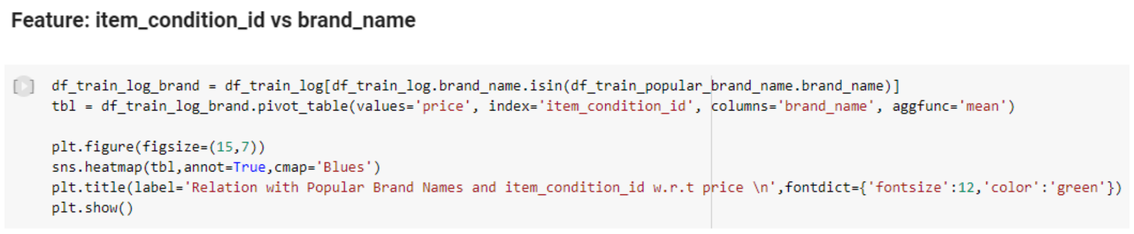
- Out of top 20 brands, Michael Kors is having higher avg. selling price than others.



Key- Note:

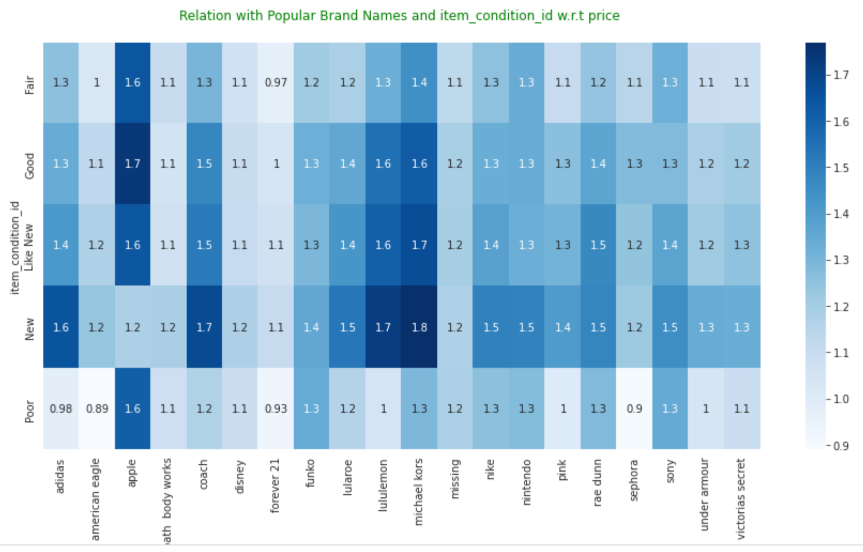
- Chanel, David Yurman, Mary Kay are among top expensive selling brands.

Feature: item_condition_id vs brand_name(Multivariate Analysis)



Feature: item_condition_id vs brand_name

```
df_train_log_brand = df_train_log[df_train_log.brand_name.isin(df_train_popular_brand_name.brand_name)]
tbl = df_train_log_brand.pivot_table(values='price', index='item_condition_id', columns='brand_name', aggfunc='mean')

plt.figure(figsize=(15,7))
sns.heatmap(tbl,annot=True,cmap='Blues')
plt.title(label='Relation with Popular Brand Names and item_condition_id w.r.t price \n',fontdict={'fontsize':12,'color':'green'})
plt.show()
```

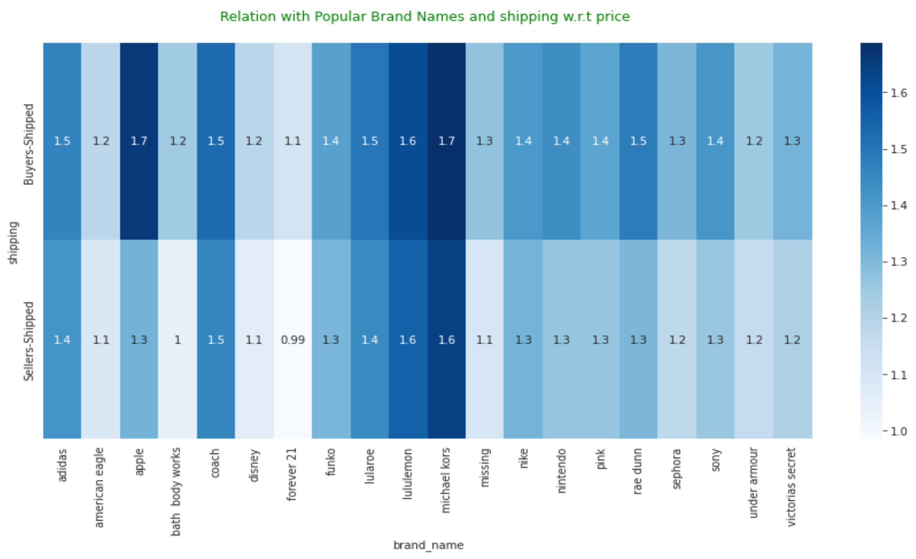


Key- Note:

- Michael Kors, Apple, coach products among the top popular brands are sold at higher prices irrespective of their condition.
- New Adidas, lululemon products sold out at a higher price.

Feature: shipping vs brand_name(Multivariate Analysis)

```
df_train_log_brand = df_train_log[df_train_log.brand_name.isin(df_train_popular_brand_name.brand_name)]
tbl = df_train_log_brand.pivot_table(values='price', index='shipping', columns='brand_name', aggfunc='mean')

plt.figure(figsize=(15,7))
sns.heatmap(tbl,annot=True,cmap='Blues')
plt.title(label='Relation with Popular Brand Names and shipping w.r.t price \n',fontdict={'fontsize':12,'color':'green'})
plt.show()
```



Key- Note:

- Michale Kors, Apple, coach, Adidas products among the top popular brands are sold at higher prices irrespective of their shipment type.

Feature: Category_name

Category name is the hierarchical structure of the product representing an entire category of items for sale. for eg. If you have a women's online clothing store, you might have a unique category description on the pages for tops, bottoms, dresses, sportswear, and accessories.

```
df_train_top20_category_name = pd.DataFrame(df_train.category_name.value_counts().sort_values(ascending = False).head(20))
df_train_top20_category_name.reset_index(inplace = True)
df_train_top20_category_name.rename(columns={'index':'category_name','category_name':'Percentage(%)'},inplace=True)
df_train_top20_category_name['Percentage(%)'] = df_train_top20_category_name['Percentage(%)'].apply(lambda x: \
                                                ((x/df_train.shape[0])*100))
print('Feature: category_name')
print('-'*20)
print('Total categories: {}'.format(len(list(df_train.category_name.unique()))))
print('Top 20 Popular categories:')
print(datadisplay(df_train_top20_category_name))
print('\n')

plt.figure(figsize=(15,7))
sns.barplot(x = df_train_top20_category_name.category_name,y = df_train_top20_category_name['Percentage(%)'])
plt.xticks(rotation = 'vertical')
plt.title(label='Top 20 Popular categories:\n',fontdict={'fontsize':12,'color':'green'})
plt.show()
```

Key- Note:

- Women's Athletic apparels(Pants, Tights, Leggings) are sold out in the highest quantity.

Splitting Category in Hierarchy(Feature Engineering)

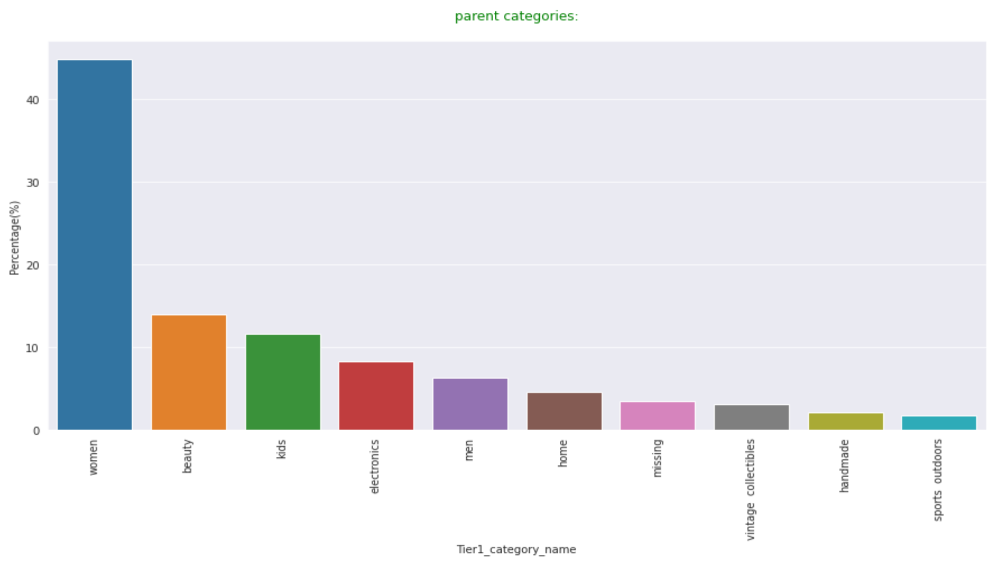
```
df_train['Tier1_category_name'] = df_train.category_name.apply(lambda x: x.split('/')[0].lower() if len(x.split('/')) > 0 else x)
df_train['Tier2_category_name'] = df_train.category_name.apply(lambda x: x.split('/')[1].lower() if len(x.split('/')) > 1 else x)
df_train['Tier3_category_name'] = df_train.category_name.apply(lambda x: x.split('/')[2].lower() if len(x.split('/')) > 2 else x)
df_train.drop(columns = 'category_name', inplace = True)
```

```
df_train_log['Tier1_category_name'] = df_train_log.category_name.apply(lambda x: x.split('/')[0].lower() if len(x.split('/')) > 0 else x)
df_train_log['Tier2_category_name'] = df_train_log.category_name.apply(lambda x: x.split('/')[1].lower() if len(x.split('/')) > 1 else x)
df_train_log['Tier3_category_name'] = df_train_log.category_name.apply(lambda x: x.split('/')[2].lower() if len(x.split('/')) > 2 else x)
df_train_log.drop(columns = 'category_name', inplace = True)
```

```
df_train_copy['Tier1_category_name'] = df_train_copy.category_name.apply(lambda x: x.split('/')[0].lower() if len(x.split('/')) > 0 else x)
df_train_copy['Tier2_category_name'] = df_train_copy.category_name.apply(lambda x: x.split('/')[1].lower() if len(x.split('/')) > 1 else x)
df_train_copy['Tier3_category_name'] = df_train_copy.category_name.apply(lambda x: x.split('/')[2].lower() if len(x.split('/')) > 2 else x)
df_train_copy.drop(columns = 'category_name', inplace = True)
```

```
df_train_Tier1_category_name = pd.DataFrame(df_train.Tier1_category_name.value_counts().sort_values(ascending = False))
df_train_Tier1_category_name.reset_index(inplace = True)
df_train_Tier1_category_name.rename(columns={'index':'Tier1_category_name','Tier1_category_name':'Percentage(%)'},inplace=True)
df_train_Tier1_category_name['Percentage(%)'] = df_train_Tier1_category_name['Percentage(%)'].apply(lambda x: \
                                                    ((x/df_train.shape[0])*100))
print('Feature: parent_category_name')
print('-'*20)
print(datadisplay(df_train_Tier1_category_name))
print('\n')

plt.figure(figsize=(15,7))
sns.barplot(x = df_train_Tier1_category_name.Tier1_category_name,y = df_train_Tier1_category_name['Percentage(%)'])
plt.xticks(rotation = 'vertical')
plt.title(label='parent categories:\n',fontdict={'fontsize':12,'color':'green'})
plt.show()
```

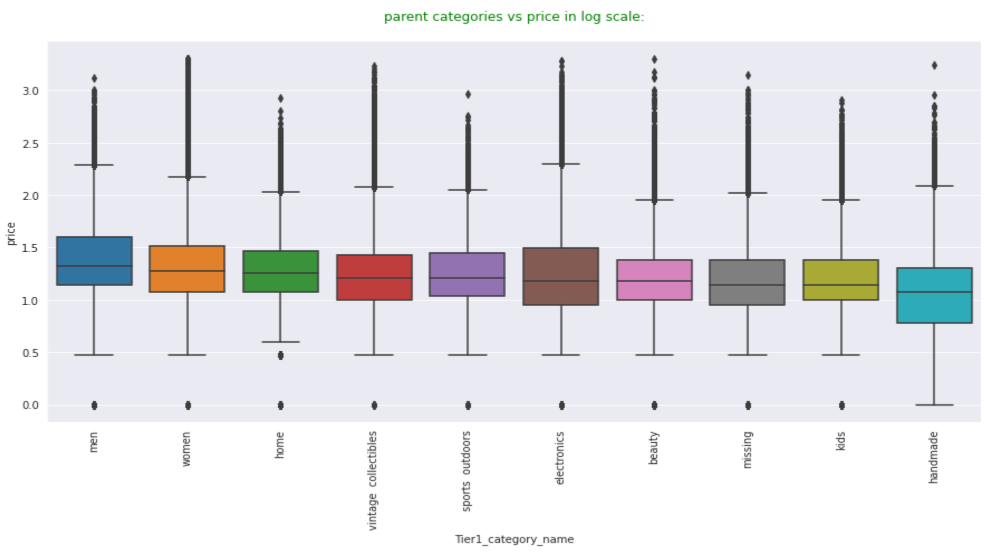
```
Feature: parent_category_name
--------------------
+----------------------+-------------------+
|  Tier1_category_name |   Percentage(%)   |
+----------------------+-------------------+
|        women         | 44.81526301091545 |
|        beauty        | 14.01787268142836 |
|         kids         | 11.581087443658827 |
|     electronics      | 8.275491303121917 |
|         men          | 6.318912961066749 |
|         home         | 4.578120805188042 |
|       missing        | 3.4849570248892734 |
| vintage  collectibles | 3.138646840509792 |
|       handmade       | 2.080222005469192 |
|    sports  outdoors   | 1.7094259237523997 |
+----------------------+-------------------+
```



Key- Note:

- Women products are sold out in the highest quantity.

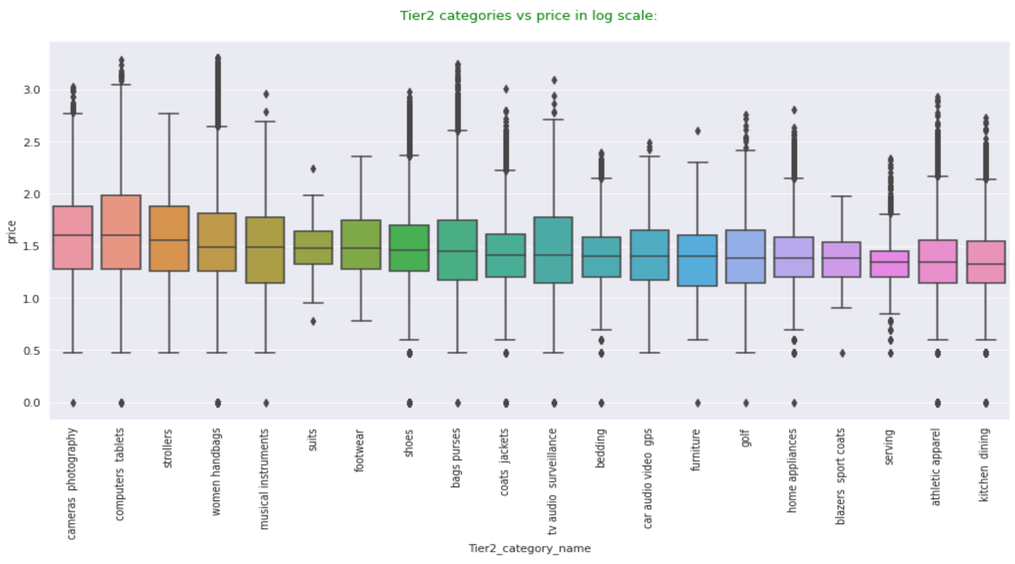
```
plt.figure(figsize=(15,7))
order_desc = df_train_log.groupby('Tier1_category_name').price.median().sort_values(ascending = False).index
sns.boxplot(x = df_train_log.Tier1_category_name,y = df_train_log.price, order = order_desc)
plt.title(label='parent categories vs price in log scale:\n',fontdict={'fontsize':12,'color':'green'})
plt.xticks(rotation = 'vertical')
plt.show()
```



Key- Note:

- Men products have the highest median selling price.

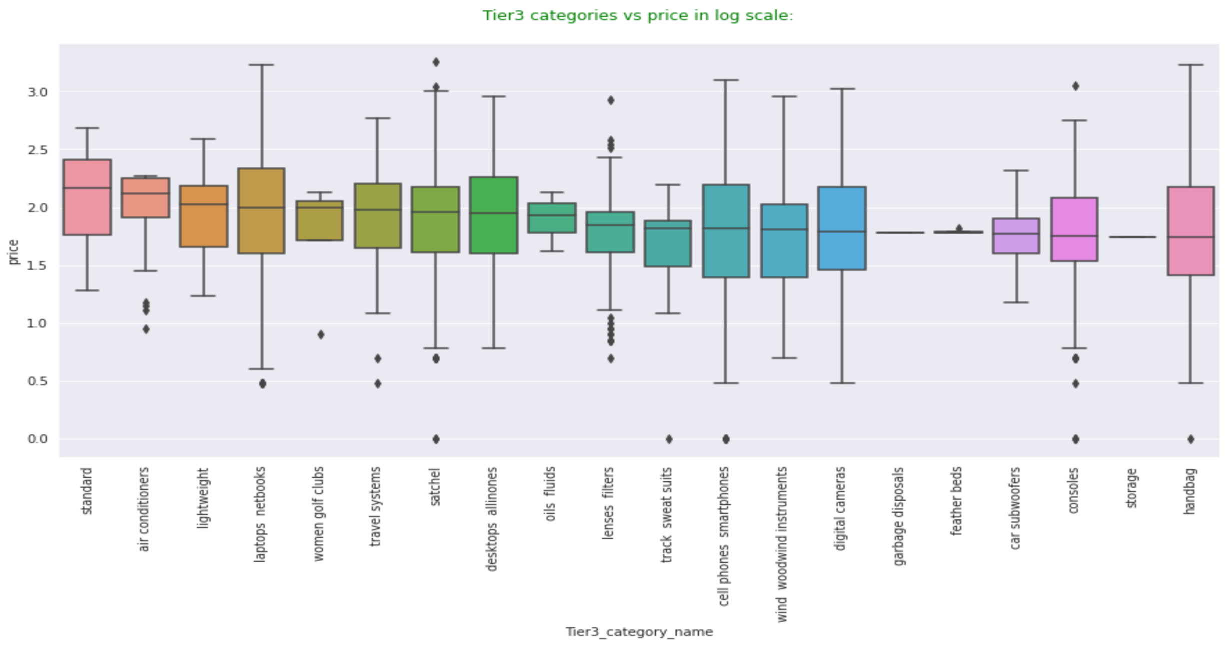
```
plt.figure(figsize=(15,7))
df_train_log_tier2 = df_train_log[df_train_log.Tier2_category_name.isin(df_price_parent_cat.Tier2_category_name)]
order_desc = df_train_log_tier2.groupby('Tier2_category_name').price.median().sort_values(ascending = False).index
sns.boxplot(x = df_train_log_tier2.Tier2_category_name,y = df_train_log_tier2.price, order = order_desc)
plt.title(label='Tier2 categories vs price in log scale:\n',fontdict={'fontsize':12,'color':'green'})
plt.xticks(rotation = 'vertical')
plt.show()
```



Key- Note:

- Computer & Tablets, Cameras and Photography category products are having the highest median selling price.

```
plt.figure(figsize=(15,7))
df_train_log_tier3 = df_train_log[df_train_log.Tier3_category_name.isin(df_price_parent_cat.Tier3_category_name)]
order_desc = df_train_log_tier3.groupby('Tier3_category_name').price.median().sort_values(ascending = False).index
sns.boxplot(x = df_train_log_tier3.Tier3_category_name,y = df_train_log_tier3.price, order = order_desc)
plt.title(label='Tier3 categories vs price in log scale:\n',fontdict={'fontsize':12,'color':'green'})
plt.xticks(rotation = 'vertical')
plt.show()
```

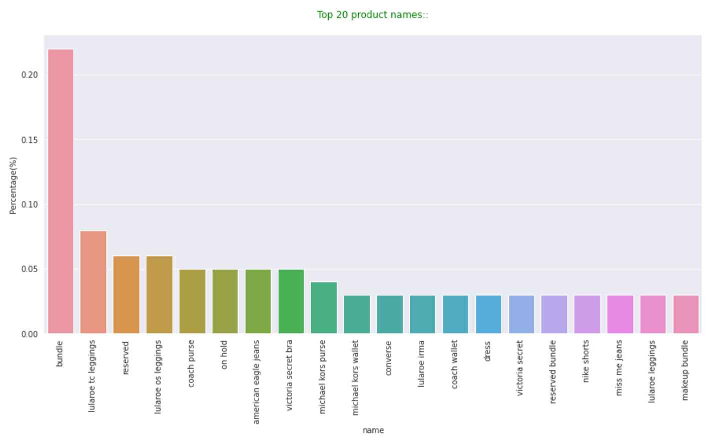


Key- Note:

- Standard products have the highest median selling price.

Feature: name:

It contains a brief description about the product.

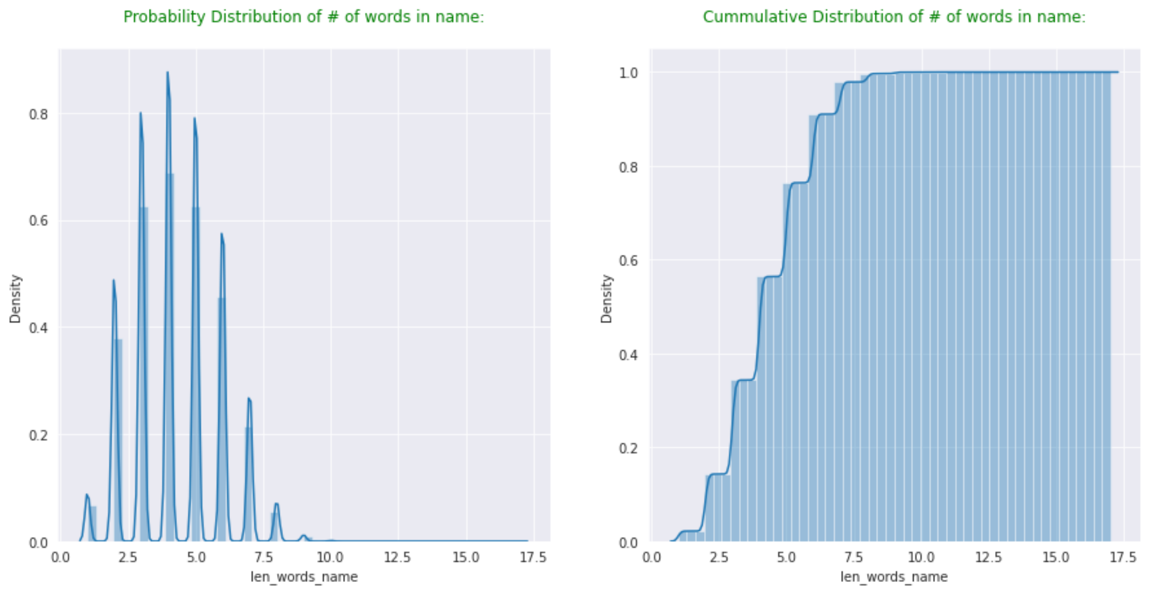


Key- Note: Products with the name ‘bundle’ are sold out in the highest quantity.

Feature Engineering

```
df_train['len_words_name']= df_train.name.apply(lambda x: len(x.split(' ')))
df_train_copy['len_words_name']= df_train_copy.name.apply(lambda x: len(x.split(' ')))
df_train_log['len_words_name']= df_train_log.name.apply(lambda x: len(x.split(' ')))
```

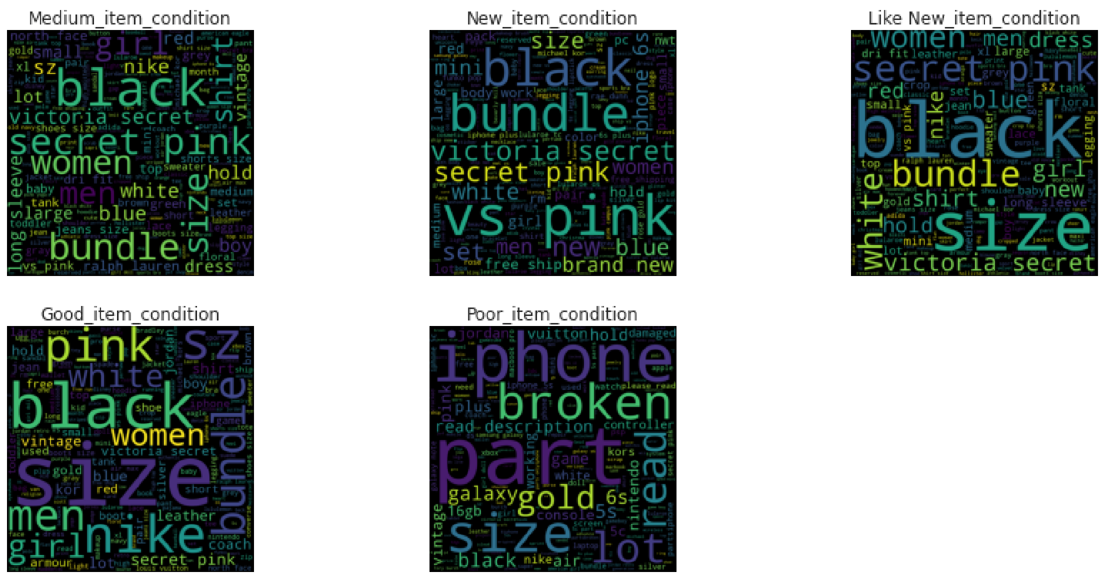
```
fig, axes = plt.subplots(1,2,figsize=(15,7))
sns.distplot(df_train_copy['len_words_name'],ax=axes[0]) #pdf
axes[0].set_title(label='Probability Distribution of # of words in name:\n',fontdict={'fontsize':12,'color':'green'})
#df_train_copy['len_words_name'].plot(kind = 'hist', cumulative=True, density = True,ax = axes[1])
kwargs = {'cumulative': True}
sns.distplot(df_train_copy['len_words_name'],hist_kws=kwargs, kde_kws=kwargs,ax=axes[1]) #cdf
axes[1].set_title(label='Cummulative Distribution of # of words in name:\n',fontdict={'fontsize':12,'color':'green'})
plt.show()
```



Key- Note:

- From the above Distribution of # of words in name, 97.88% of product names are having less than/equal to 7 words.
- Count of most number of words in a product name is 17
- Count of avg. number of words in a product name is 4

name vs item_condition_id(Multivariate Analysis)

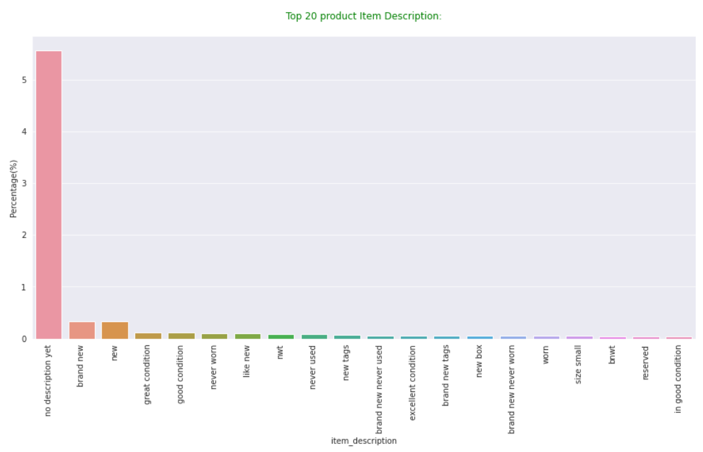


Key- Note:

- New/Like New Products are having ‘new’ as important/common words in name.
- Poor Products are having ‘broken’ as an important/common word in name.

Feature: item_description:

A product description is a form of marketing copy used to describe and explain the benefits of your product. In other words, it provides all the information and details of your product on your ecommerce site.



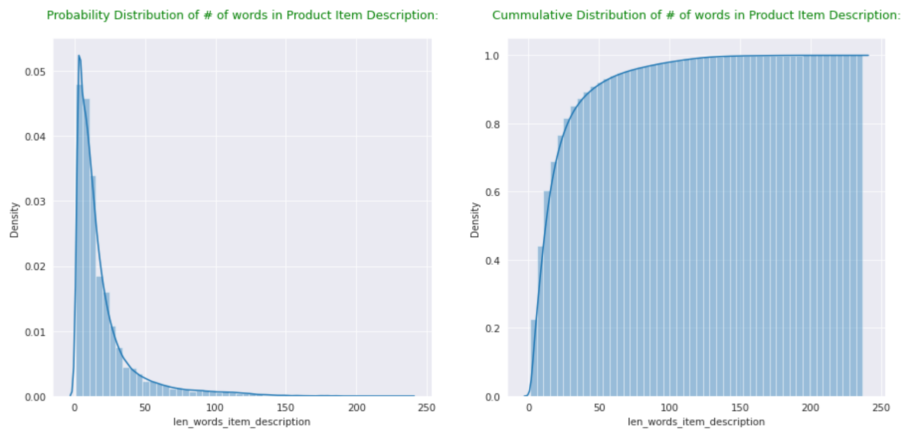
Key- Note:

- Most products sold with Item Description as ‘No description yet’.

Feature Engineering

```
df_train['len_words_item_description']= df_train.item_description.apply(lambda x: len(x.split(' ')))
df_train_copy['len_words_item_description']= df_train_copy.item_description.apply(lambda x: len(x.split(' ')))
df_train_log['len_words_item_description']= df_train_log.item_description.apply(lambda x: len(x.split(' ')))
```

```
fig, axes = plt.subplots(1,2,figsize=(15,7))
sns.distplot(df_train_copy['len_words_item_description'],ax=axes[0]) #pdf
axes[0].set_title(label='Probability Distribution of # of words in Product Item Description:\n',fontdict={'fontsize':12,'color':'green'})
#df_train_copy['len_words_name'].plot(kind = 'hist', cumulative=True, density = True,ax = axes[1])
kwargs = {'cumulative': True}
sns.distplot(df_train_copy['len_words_item_description'],hist_kws=kwargs, kde_kws=kwargs,ax=axes[1]) #cdf
axes[1].set_title(label='Cummulative Distribution of # of words in Product Item Description:\n',fontdict={'fontsize':12,'color':'green'})
plt.show()
```

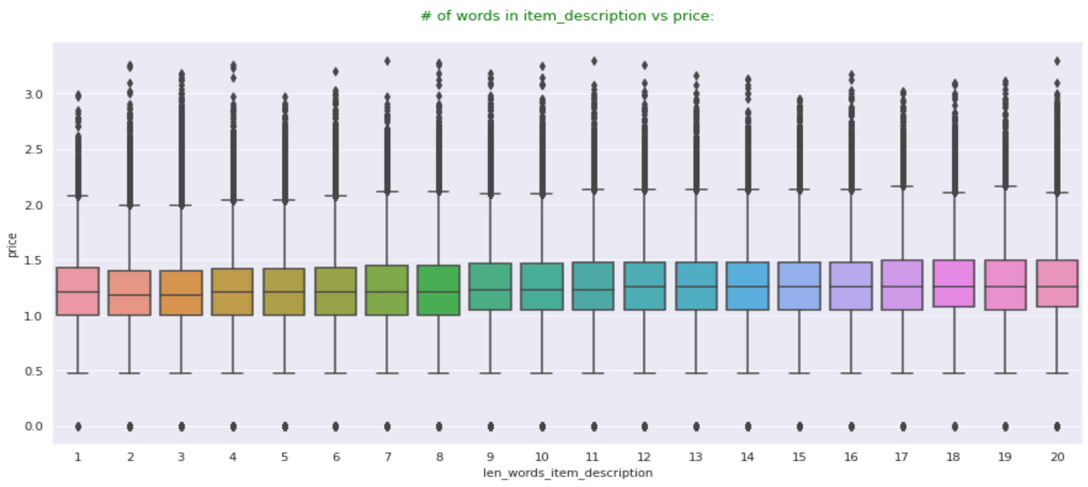


Key- Note:

- From the above Distribution of # of words in name, 99.19% of product Item Description are having less than/equal to 120 words.
- Count of most number of words in a product item description is 237
- Count of avg. number of words in a product item description is 19

#of words in item_description vs selling price(Multivariate Analysis)

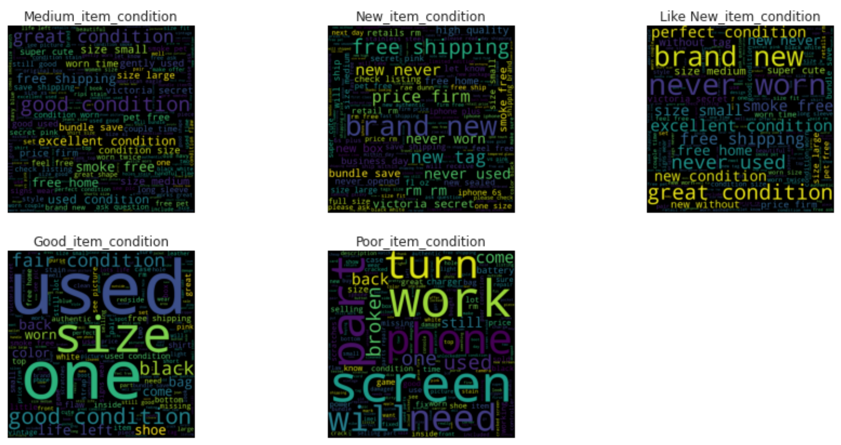
```
plt.figure(figsize=(15,7))
top20 = df_train_log.groupby('len_words_item_description')['price'].mean().head(20).index
sns.boxplot(df_train_log['len_words_item_description'],df_train_log['price'], order = top20)
plt.title(label='# of words in item_description vs price:\n',fontdict={'fontsize':12,'color':'green'})
plt.show()
```



key- Note:

- There is no such relation between price and length of words available in item_description.

item_description vs item_condition_id(Multivariate Analysis)

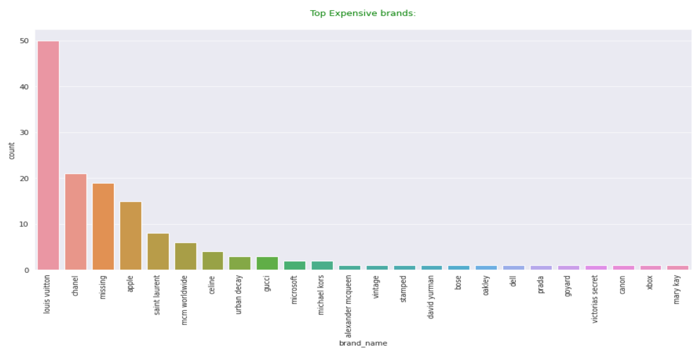


Key- Note:

- New/Like New Products are having ‘new’, ’perfect’, ’brand’ as important/common words in item description.
- Poor Products are having ‘broken’ as an important/common word in item description.

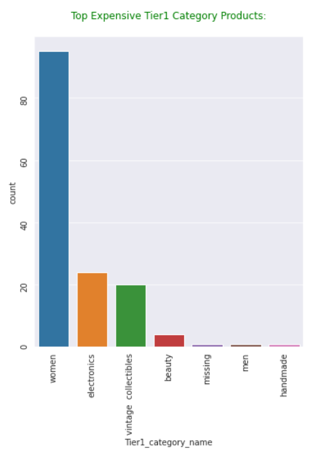
Analysis on Higher Selling Price Products:

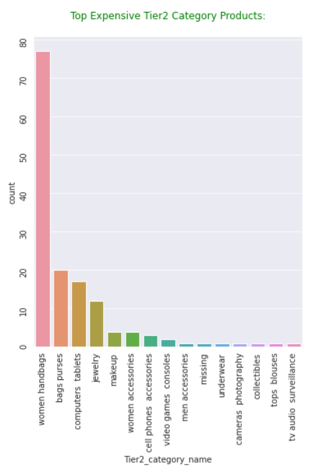
Many products sold at higher price, can visualize these products and related features separately to determine any trend on the data based upon features.

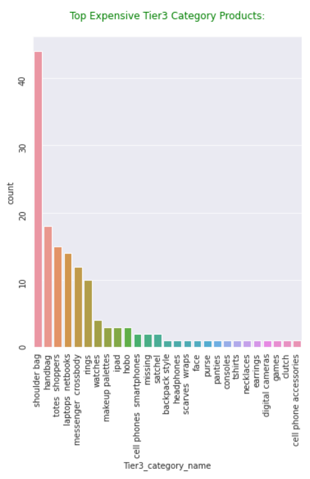


Key- Note:

- ‘Louis Vuitton’ is among top expensive brands whose products are sold out more in higher selling price
- Electronic product brands like Apple, Dell, Bose, Canon are among top list.





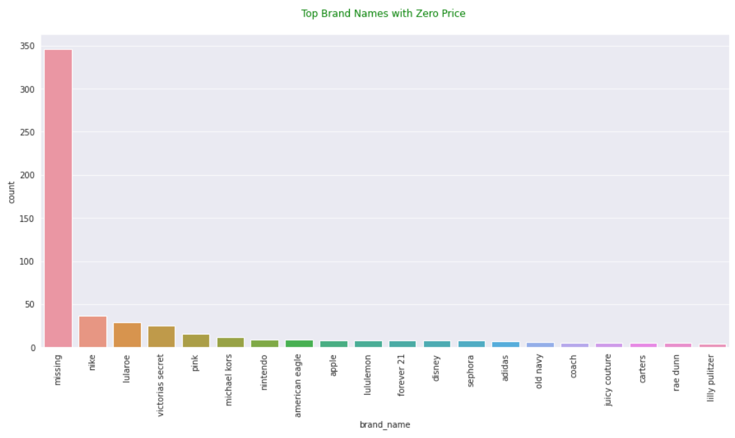


Key- Note:

- Women Shoulder bags are the top most expensive category of products sold.
- Electronic products(Laptops & Netbooks, iPad), jewellery(Rings, Necklaces) are among top expensive category products.

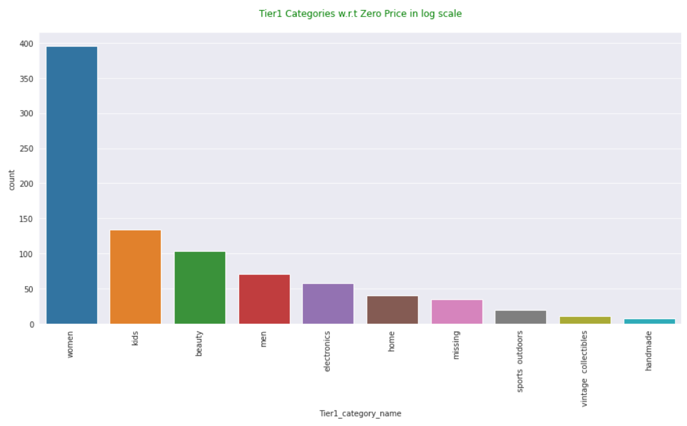
Analysis of Zero-Price Product:

As we analysed, more products sold at zero price, it is always better to identify the products and analyse these for better feature engineering.



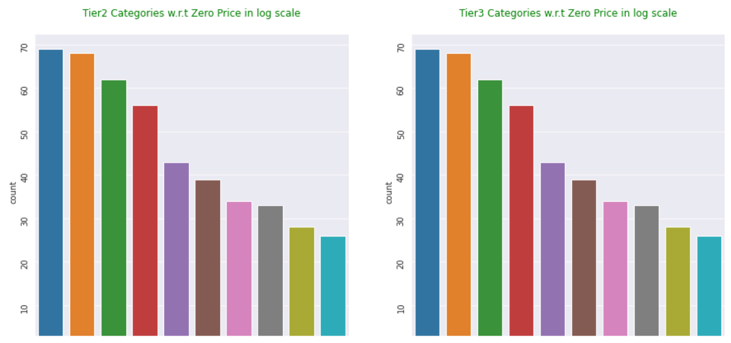
Key- Note:

- Most products with Missing brand name are sold in $0



Key- Note:

- Most products with Women category are sold in $0



Most products with Tier2 category as 'Tops & Blouses' and Tier3 category as 'T-Shirts' are sold in $0

Key- Note:

- Most products with Tier2 category as 'Tops & Blouses' and Tier3 category as 'T-Shirts' are sold in $0

Data Analysis Conclusion:

- Most sold products have a selling price between 0-$250.
- 'PINK', 'Nike' are among top most sold out Medium brands
- Women Athletic Apparel like Shirts, Pants, Tights, Leggings are amongst top medium category products sold.
- Electronic Products/Jewelries are not part of the top medium categories sold.
- Women Shoulder bags, jewelries, Electronic gadgets are amongst expensive sold out products.
- Michale Kors, Apple products among the top expensive popular brands are sold at higher prices irrespective of their condition.
- Most products with Missing brand name are sold in $0
- Most products with selling price as $0 are in Women Category and shipment category as Buyers-Shipped.

Split the data in Train-Test(70–30)

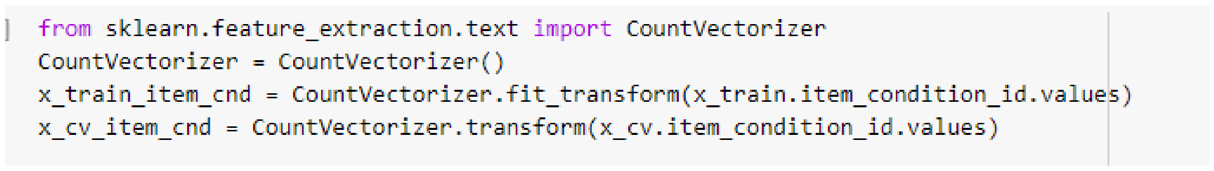
x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.3,random_state=42)

Split the data in Train-CV(70–30)

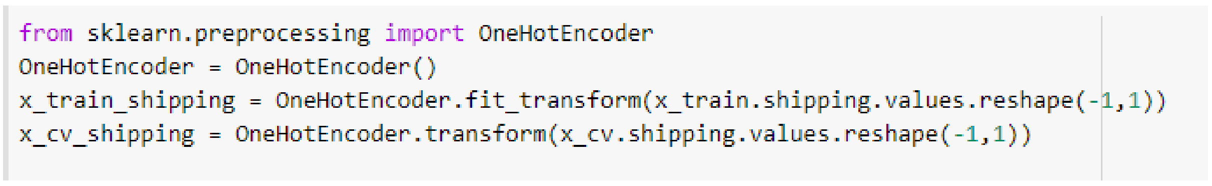
x_train,x_cv,y_train,y_cv = train_test_split(x_train,y_train,test_size=0.3,random_state=42)

Data Pre-processing:

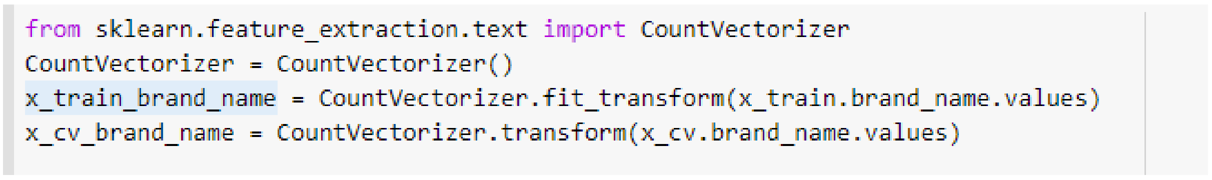
Countvectorizer(BOW) used for categorical features, TF-IDF-Vectorizer for text features. As Length of Item name words and item description is based on numerical feature, minmaxscaler is used to scale the feature between [0,1].



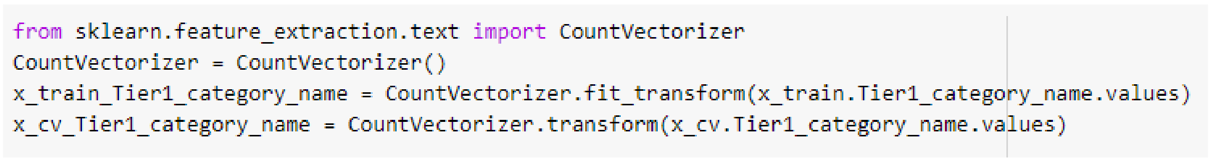
```
from sklearn.feature_extraction.text import CountVectorizer
CountVectorizer = CountVectorizer()
x_train_item_cnd = CountVectorizer.fit_transform(x_train.item_condition_id.values)
x_cv_item_cnd = CountVectorizer.transform(x_cv.item_condition_id.values)
```



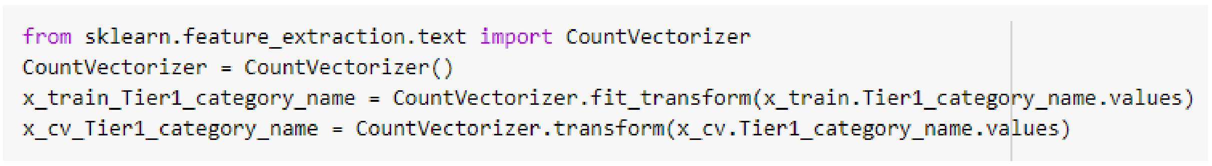
```
from sklearn.preprocessing import OneHotEncoder
OneHotEncoder = OneHotEncoder()
x_train_shipping = OneHotEncoder.fit_transform(x_train.shipping.values.reshape(-1,1))
x_cv_shipping = OneHotEncoder.transform(x_cv.shipping.values.reshape(-1,1))
```



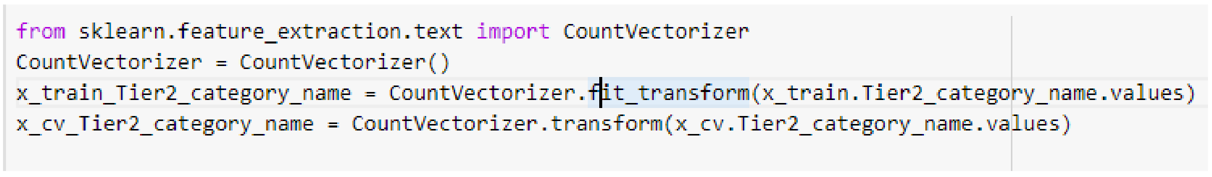
```
from sklearn.feature_extraction.text import CountVectorizer
CountVectorizer = CountVectorizer()
x_train_brand_name = CountVectorizer.fit_transform(x_train.brand_name.values)
x_cv_brand_name = CountVectorizer.transform(x_cv.brand_name.values)
```



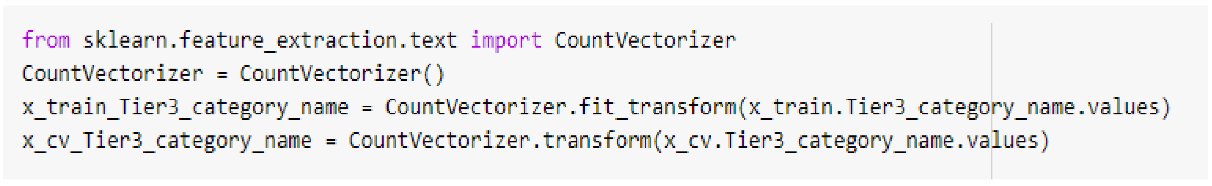
```
from sklearn.feature_extraction.text import CountVectorizer
CountVectorizer = CountVectorizer()
x_train_Tier1_category_name = CountVectorizer.fit_transform(x_train.Tier1_category_name.values)
x_cv_Tier1_category_name = CountVectorizer.transform(x_cv.Tier1_category_name.values)
```



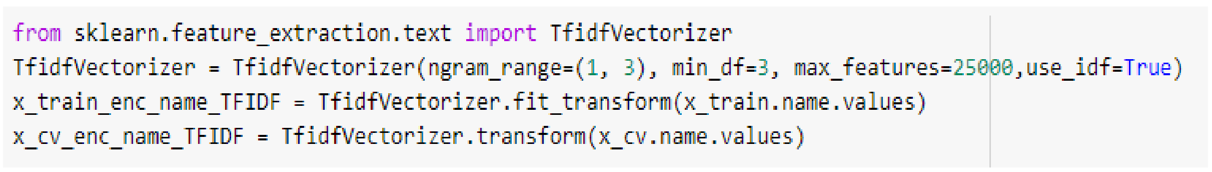
```
from sklearn.feature_extraction.text import CountVectorizer
CountVectorizer = CountVectorizer()
x_train_Tier1_category_name = CountVectorizer.fit_transform(x_train.Tier1_category_name.values)
x_cv_Tier1_category_name = CountVectorizer.transform(x_cv.Tier1_category_name.values)
```



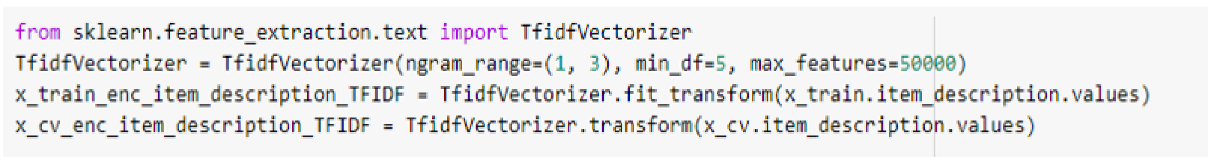
```python
from sklearn.feature_extraction.text import CountVectorizer
CountVectorizer = CountVectorizer()
x_train_Tier2_category_name = CountVectorizer.fit_transform(x_train.Tier2_category_name.values)
x_cv_Tier2_category_name = CountVectorizer.transform(x_cv.Tier2_category_name.values)
```



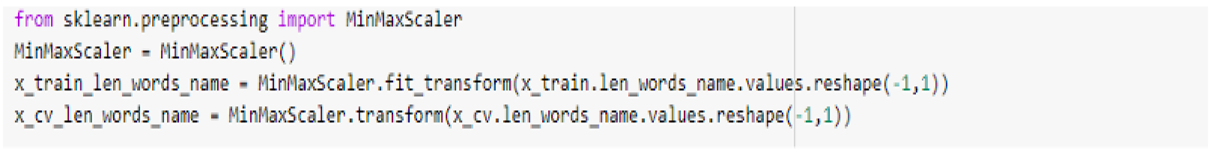
```python
from sklearn.feature_extraction.text import CountVectorizer
CountVectorizer = CountVectorizer()
x_train_Tier3_category_name = CountVectorizer.fit_transform(x_train.Tier3_category_name.values)
x_cv_Tier3_category_name = CountVectorizer.transform(x_cv.Tier3_category_name.values)
```



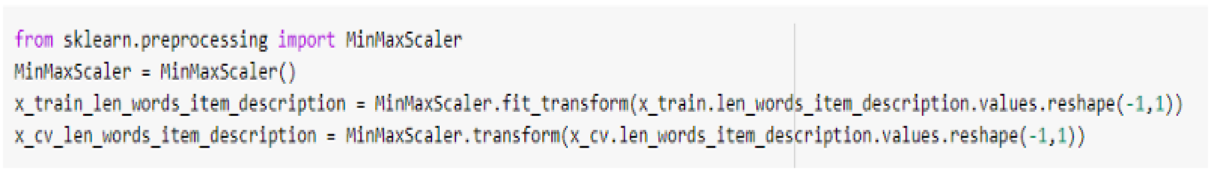
```python
from sklearn.feature_extraction.text import TfidfVectorizer
TfidfVectorizer = TfidfVectorizer(ngram_range=(1, 3), min_df=3, max_features=25000,use_idf=True)
x_train_enc_name_TFIDF = TfidfVectorizer.fit_transform(x_train.name.values)
x_cv_enc_name_TFIDF = TfidfVectorizer.transform(x_cv.name.values)
```



```python
from sklearn.feature_extraction.text import TfidfVectorizer
TfidfVectorizer = TfidfVectorizer(ngram_range=(1, 3), min_df=5, max_features=50000)
x_train_enc_item_description_TFIDF = TfidfVectorizer.fit_transform(x_train.item_description.values)
x_cv_enc_item_description_TFIDF = TfidfVectorizer.transform(x_cv.item_description.values)
```



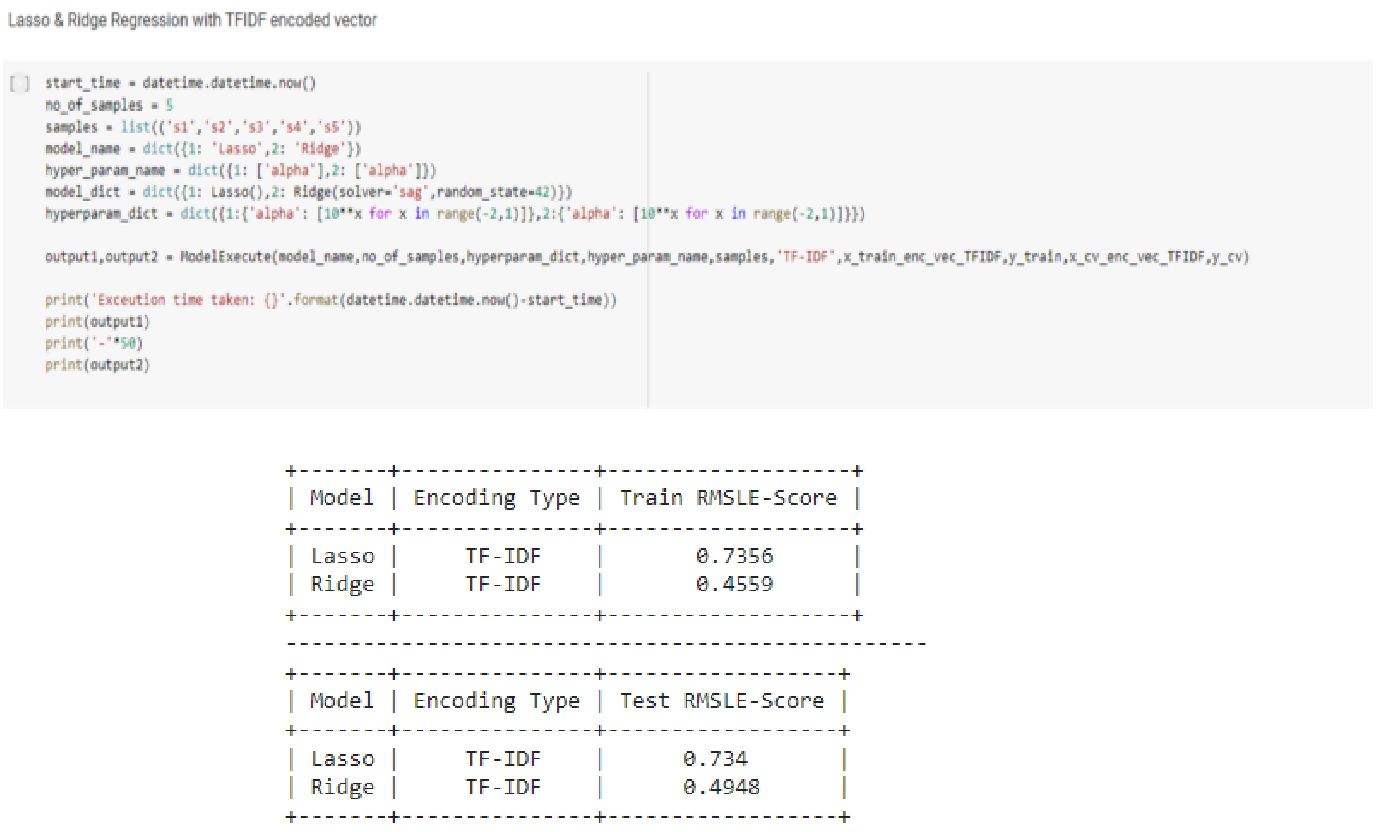
```python
from sklearn.preprocessing import MinMaxScaler
MinMaxScaler = MinMaxScaler()
x_train_len_words_name = MinMaxScaler.fit_transform(x_train.len_words_name.values.reshape(-1,1))
x_cv_len_words_name = MinMaxScaler.transform(x_cv.len_words_name.values.reshape(-1,1))
```



```python
from sklearn.preprocessing import MinMaxScaler
MinMaxScaler = MinMaxScaler()
x_train_len_words_item_description = MinMaxScaler.fit_transform(x_train.len_words_item_description.values.reshape(-1,1))
x_cv_len_words_item_description = MinMaxScaler.transform(x_cv.len_words_item_description.values.reshape(-1,1))
```

First Cut Solution:

Applying TF-IDF encoding technique along with Ridge Regression(Linear Regression with L2 Regularizer) performed well which significantly stands out of other non-linear/tree based algorithms.



Lasso & Ridge Regression with TFIDF encoded vector

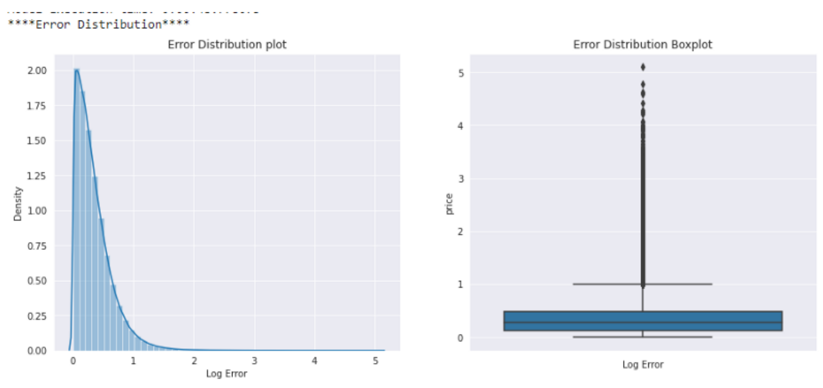
```
[ ] start_time = datetime.datetime.now()
    no_of_samples = 5
    samples = list(('s1','s2','s3','s4','s5'))
    model_name = dict({1: 'Lasso',2: 'Ridge'})
    hyper_param_name = dict({1: ['alpha'],2: ['alpha']})
    model_dict = dict({1: Lasso(),2: Ridge(solver='sag',random_state=42)})
    hyperparam_dict = dict({1:{'alpha': [10**x for x in range(-2,1)]},2:{'alpha': [10**x for x in range(-2,1)]}})

    output1,output2 = ModelExecute(model_name,no_of_samples,hyperparam_dict,hyper_param_name,samples,'TF-IDF',x_train_enc_vec_TFIDF,y_train,x_cv_enc_vec_TFIDF,y_cv)

    print('Exceution time taken: {}'.format(datetime.datetime.now()-start_time))
    print(output1)
    print('-'*50)
    print(output2)
```

```
+-------+---------------+-------------------+
| Model | Encoding Type | Train RMSLE-Score |
+-------+---------------+-------------------+
| Lasso |     TF-IDF    |       0.7356      |
| Ridge |     TF-IDF    |       0.4559      |
+-------+---------------+-------------------+
--------------------------------------------------
+-------+---------------+------------------+
| Model | Encoding Type | Test RMSLE-Score |
+-------+---------------+------------------+
| Lasso |     TF-IDF    |      0.734       |
| Ridge |     TF-IDF    |      0.4948      |
+-------+---------------+------------------+
```

Ridge Regression Error Analysis:

Error Analysis performed a First cut solution to extract the erroneous points and try to fit in different algorithms, which didn’t perform well with others, so to branch this dataset and fit into different models to create a custom ensemble approach wasn’t best fit for this use-case. This might be due to a small number of data points being available in the erroneous dataset. Tried to fit inside the DL neural network model to see if it performs well, but there also it failed. The above said experiments can be viewed in the GitHub Repository.

```python
start_time = datetime.datetime.now()
model = Ridge(alpha = 1,solver='sag',random_state=42)
model.fit(x_train_enc_vec_TFIDF,y_train)
y_pred = model.predict(x_train_enc_vec_TFIDF)

error = abs(y_train - y_pred)

print('Model Execution time: {}'.format(datetime.datetime.now()-start_time))
print('****Error Distribution****')

fig, axes = plt.subplots(1,2,figsize = (15,6))

sns.distplot(error, ax = axes[0])
axes[0].set_xlabel('Log Error')
axes[0].set_title('Error Distribution plot')

sns.boxplot(y = error, ax = axes[1])
axes[1].set_xlabel('Log Error')
axes[1].set_title('Error Distribution Boxplot')

plt.show()
```



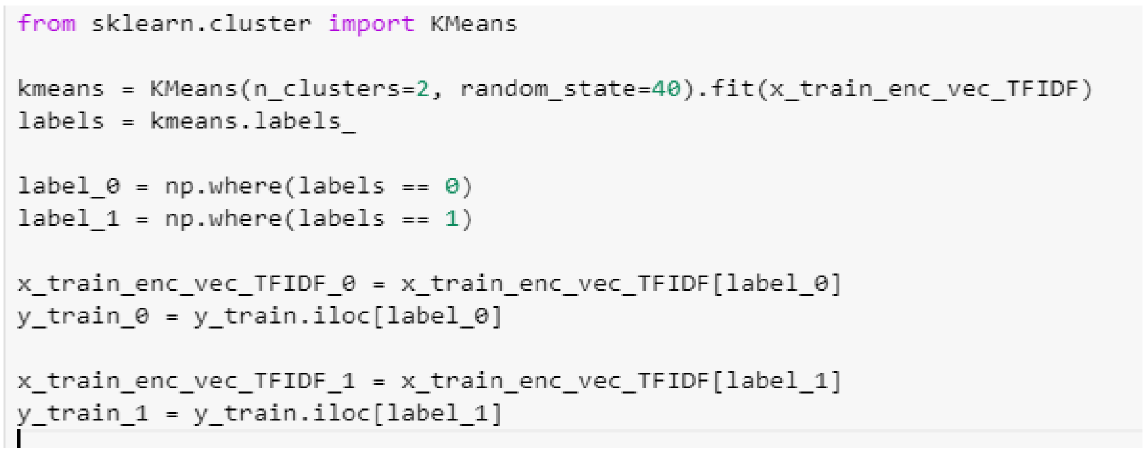
```
****Error Distribution****
```

```python
for i in range(90,101,1):
  print('{} th percentile of Error is: {}'.format(i,round(np.percentile(error.values, i),4)))
```

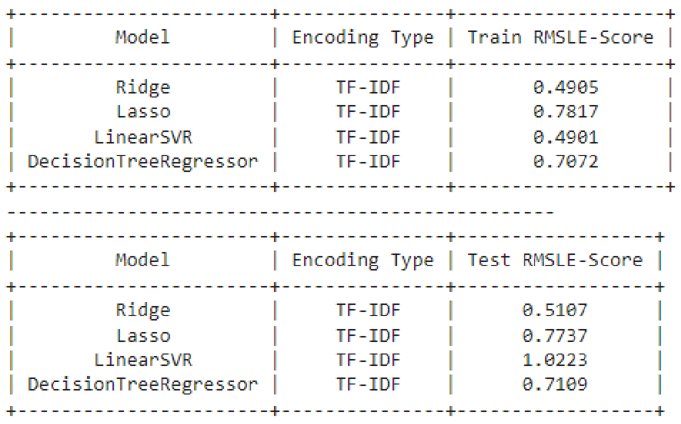
```
90 th percentile of Error is: 0.7242
91 th percentile of Error is: 0.7519
92 th percentile of Error is: 0.7837
93 th percentile of Error is: 0.8194
94 th percentile of Error is: 0.8597
95 th percentile of Error is: 0.9083
96 th percentile of Error is: 0.9698
97 th percentile of Error is: 1.0479
98 th percentile of Error is: 1.1628
99 th percentile of Error is: 1.3675
100 th percentile of Error is: 5.0992
```

Advance Modelling(Ensemble):

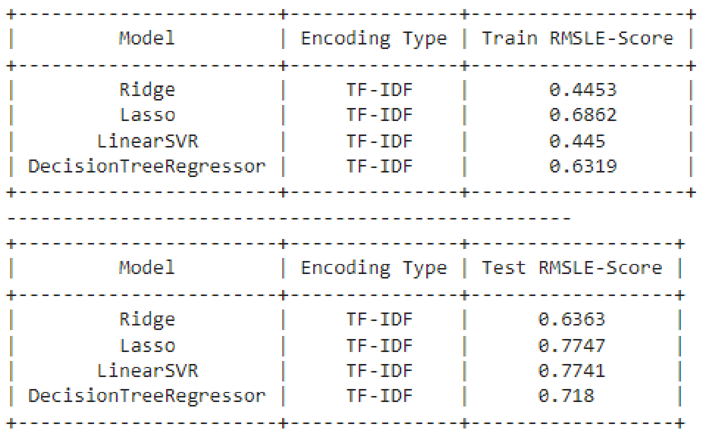
Approach is to cluster all the data points into two clusters and try to fit different models into each cluster of points to check if the model performance increases. But each cluster gave good results in Ridge Regression with TF-IDF text encoder. So this approach didn't work pretty well. Performed error Analysis on each cluster and by extracting the erroneous points both the clusters are working fine with Ridge Regression and Erroneous points didn't perform well with any sort of algorithms due to lack of data-space. Below is the result of the experiment being performed.



```python
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=2, random_state=40).fit(x_train_enc_vec_TFIDF)
labels = kmeans.labels_

label_0 = np.where(labels == 0)
label_1 = np.where(labels == 1)

x_train_enc_vec_TFIDF_0 = x_train_enc_vec_TFIDF[label_0]
y_train_0 = y_train.iloc[label_0]

x_train_enc_vec_TFIDF_1 = x_train_enc_vec_TFIDF[label_1]
y_train_1 = y_train.iloc[label_1]
```



```
+-----------------------+---------------+------------------+
|         Model         | Encoding Type | Train RMSLE-Score |
+-----------------------+---------------+------------------+
|         Ridge         |     TF-IDF    |      0.4905      |
|         Lasso         |     TF-IDF    |      0.7817      |
|       LinearSVR       |     TF-IDF    |      0.4901      |
| DecisionTreeRegressor |     TF-IDF    |      0.7072      |
+-----------------------+---------------+------------------+
--------------------------------------------------
+-----------------------+---------------+-----------------+
|         Model         | Encoding Type | Test RMSLE-Score |
+-----------------------+---------------+-----------------+
|         Ridge         |     TF-IDF    |      0.5107     |
|         Lasso         |     TF-IDF    |      0.7737     |
|       LinearSVR       |     TF-IDF    |      1.0223     |
| DecisionTreeRegressor |     TF-IDF    |      0.7109     |
+-----------------------+---------------+-----------------+
```



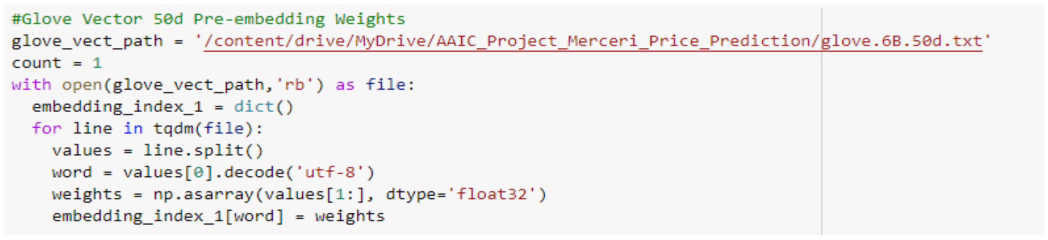
```
+-------------------------+---------------+-------------------+
|          Model          | Encoding Type | Train RMSLE-Score |
+-------------------------+---------------+-------------------+
|          Ridge          |     TF-IDF    |       0.4453      |
|          Lasso          |     TF-IDF    |       0.6862      |
|        LinearSVR        |     TF-IDF    |       0.445       |
| DecisionTreeRegressor   |     TF-IDF    |       0.6319      |
+-------------------------+---------------+-------------------+
--------------------------------------------------
+-------------------------+---------------+------------------+
|          Model          | Encoding Type | Test RMSLE-Score |
+-------------------------+---------------+------------------+
|          Ridge          |     TF-IDF    |      0.6363      |
|          Lasso          |     TF-IDF    |      0.7747      |
|        LinearSVR        |     TF-IDF    |      0.7741      |
| DecisionTreeRegressor   |     TF-IDF    |      0.718       |
+-------------------------+---------------+------------------+
```

Deep Learning Model-Final:

Embedding Layer used on text features along with a predefined glove vector matrix. The weights of the embedding layer associated with text features didn't train as part of the network- experimentally done.

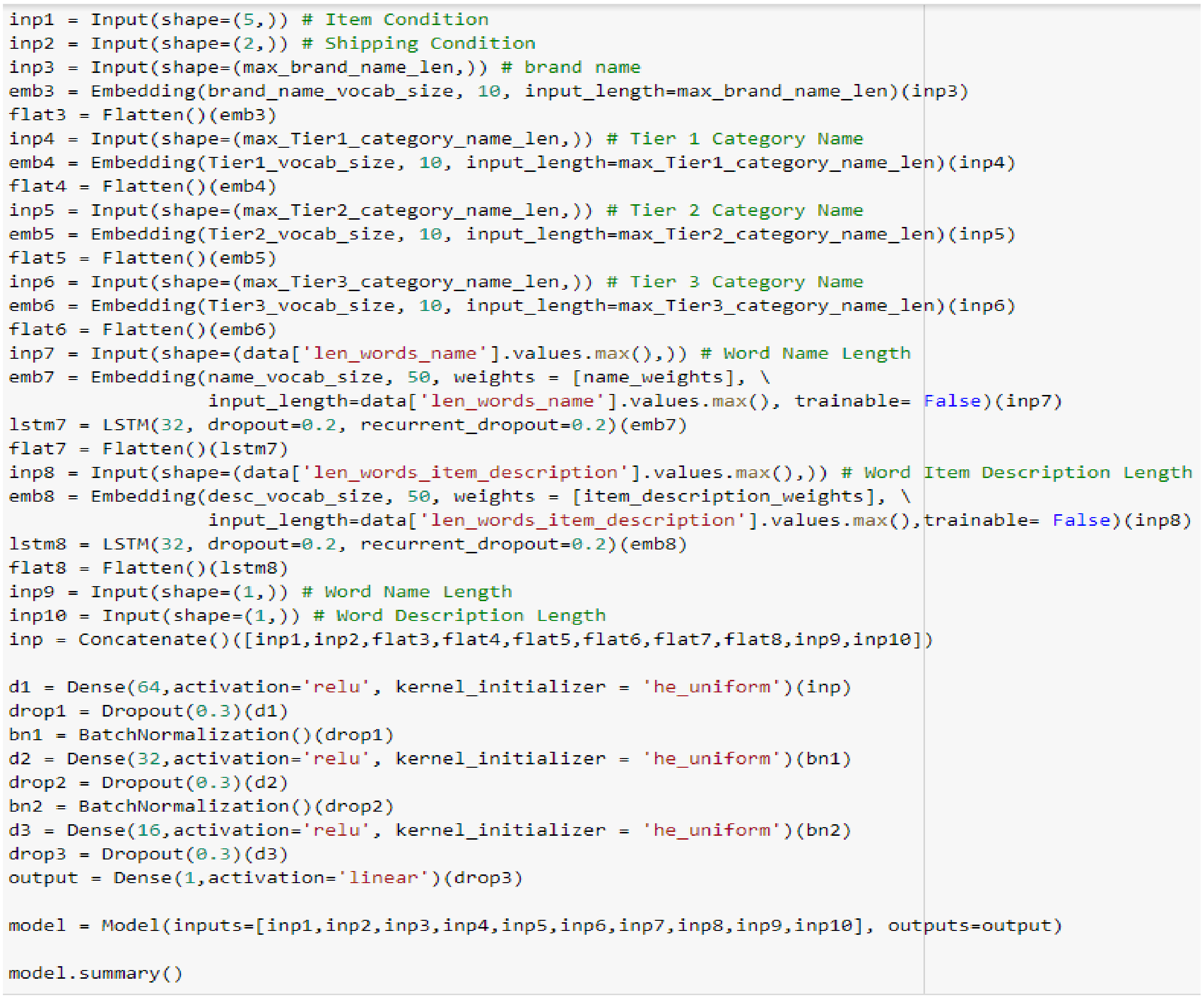
Custom RMSLE error function embedded during model fit. Adam optimizer along with callbacks with early stopping has been used.

LSTM's have been used as part of the network in regards to the text features.

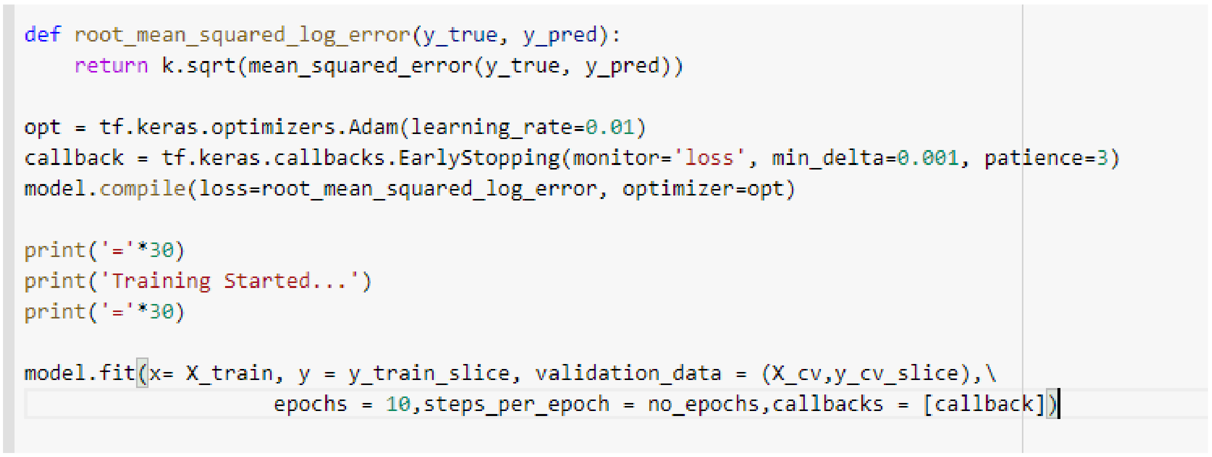


```
#Glove Vector 50d Pre-embedding Weights
glove_vect_path = '/content/drive/MyDrive/AAIC_Project_Merceri_Price_Prediction/glove.6B.50d.txt'
count = 1
with open(glove_vect_path,'rb') as file:
  embedding_index_1 = dict()
  for line in tqdm(file):
    values = line.split()
    word = values[0].decode('utf-8')
    weights = np.asarray(values[1:], dtype='float32')
    embedding_index_1[word] = weights
```

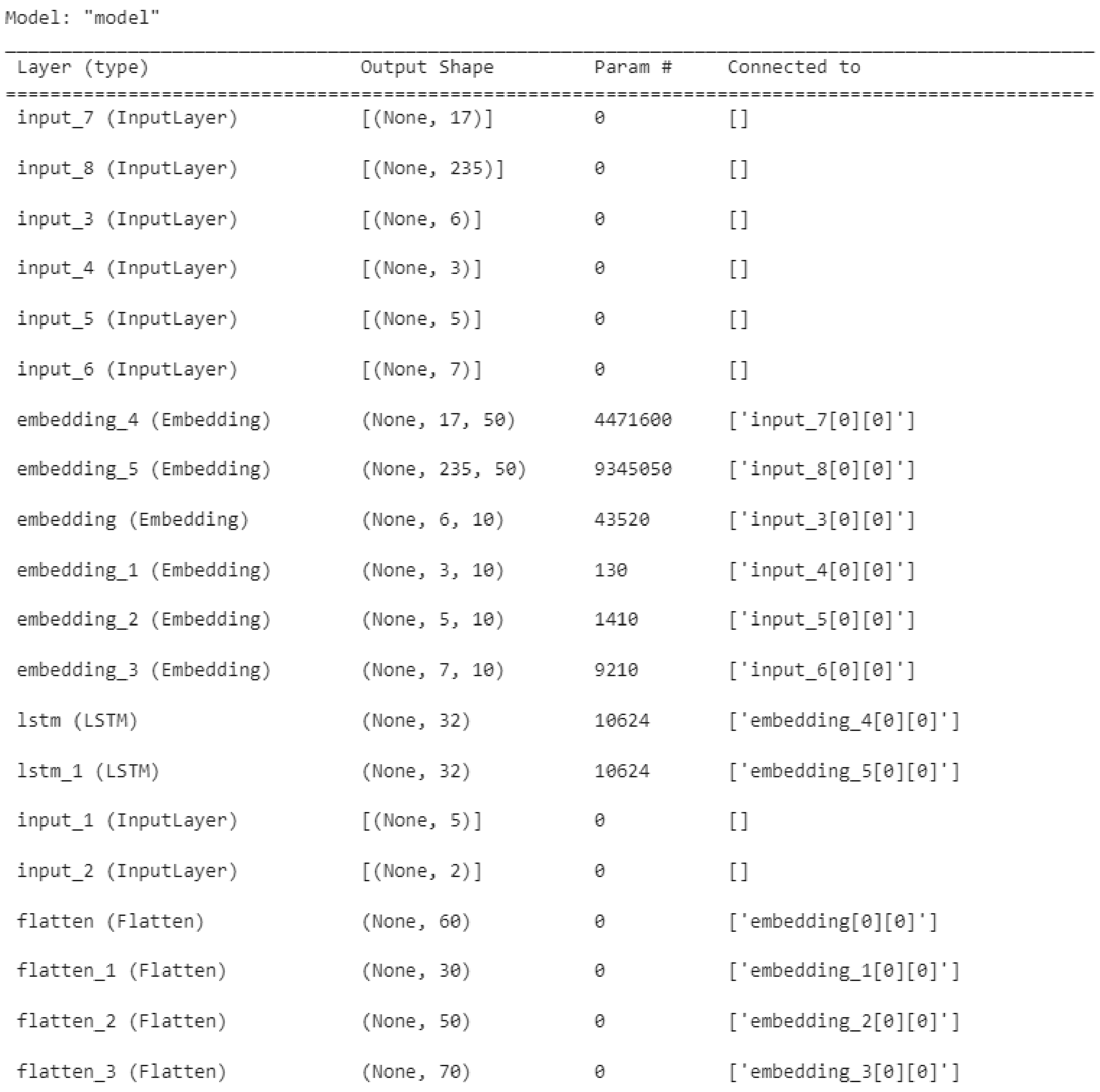
This embedding matrix contains all the words associated with a 50 dimensional vector. These weights are used in embedding layers of text features.



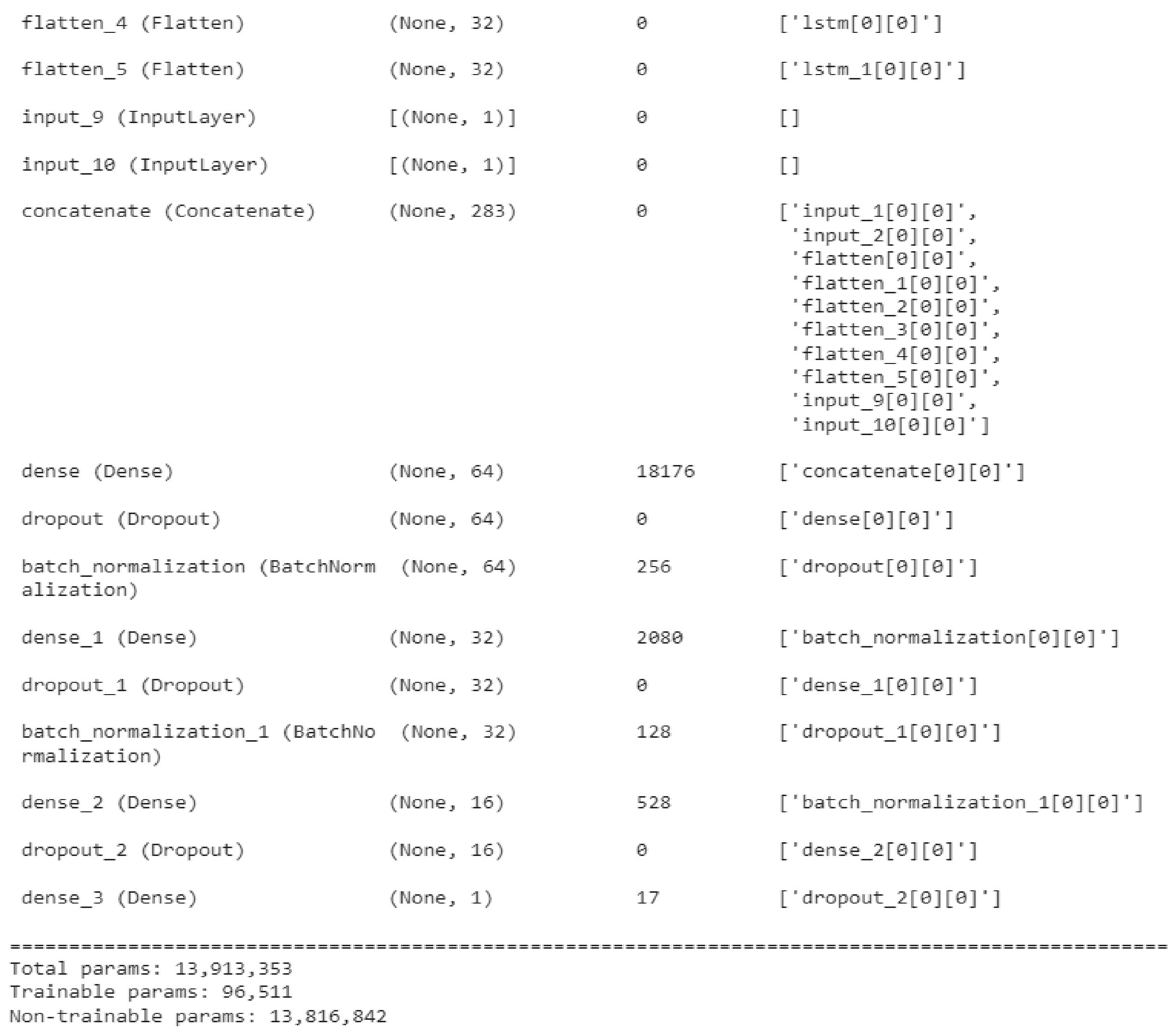
```
inp1 = Input(shape=(5,)) # Item Condition
inp2 = Input(shape=(2,)) # Shipping Condition
inp3 = Input(shape=(max_brand_name_len,)) # brand name
emb3 = Embedding(brand_name_vocab_size, 10, input_length=max_brand_name_len)(inp3)
flat3 = Flatten()(emb3)
inp4 = Input(shape=(max_Tier1_category_name_len,)) # Tier 1 Category Name
emb4 = Embedding(Tier1_vocab_size, 10, input_length=max_Tier1_category_name_len)(inp4)
flat4 = Flatten()(emb4)
inp5 = Input(shape=(max_Tier2_category_name_len,)) # Tier 2 Category Name
emb5 = Embedding(Tier2_vocab_size, 10, input_length=max_Tier2_category_name_len)(inp5)
flat5 = Flatten()(emb5)
inp6 = Input(shape=(max_Tier3_category_name_len,)) # Tier 3 Category Name
emb6 = Embedding(Tier3_vocab_size, 10, input_length=max_Tier3_category_name_len)(inp6)
flat6 = Flatten()(emb6)
inp7 = Input(shape=(data['len_words_name'].values.max(),)) # Word Name Length
emb7 = Embedding(name_vocab_size, 50, weights = [name_weights], \
                 input_length=data['len_words_name'].values.max(), trainable= False)(inp7)
lstm7 = LSTM(32, dropout=0.2, recurrent_dropout=0.2)(emb7)
flat7 = Flatten()(lstm7)
inp8 = Input(shape=(data['len_words_item_description'].values.max(),)) # Word Item Description Length
emb8 = Embedding(desc_vocab_size, 50, weights = [item_description_weights], \
                 input_length=data['len_words_item_description'].values.max(),trainable= False)(inp8)
lstm8 = LSTM(32, dropout=0.2, recurrent_dropout=0.2)(emb8)
flat8 = Flatten()(lstm8)
inp9 = Input(shape=(1,)) # Word Name Length
inp10 = Input(shape=(1,)) # Word Description Length
inp = Concatenate()([inp1,inp2,flat3,flat4,flat5,flat6,flat7,flat8,inp9,inp10])

d1 = Dense(64,activation='relu', kernel_initializer = 'he_uniform')(inp)
drop1 = Dropout(0.3)(d1)
bn1 = BatchNormalization()(drop1)
d2 = Dense(32,activation='relu', kernel_initializer = 'he_uniform')(bn1)
drop2 = Dropout(0.3)(d2)
bn2 = BatchNormalization()(drop2)
d3 = Dense(16,activation='relu', kernel_initializer = 'he_uniform')(bn2)
drop3 = Dropout(0.3)(d3)
output = Dense(1,activation='linear')(drop3)

model = Model(inputs=[inp1,inp2,inp3,inp4,inp5,inp6,inp7,inp8,inp9,inp10], outputs=output)

model.summary()
```



```
def root_mean_squared_log_error(y_true, y_pred):
    return k.sqrt(mean_squared_error(y_true, y_pred))

opt = tf.keras.optimizers.Adam(learning_rate=0.01)
callback = tf.keras.callbacks.EarlyStopping(monitor='loss', min_delta=0.001, patience=3)
model.compile(loss=root_mean_squared_log_error, optimizer=opt)

print('='*30)
print('Training Started...')
print('='*30)

model.fit(x= X_train, y = y_train_slice, validation_data = (X_cv,y_cv_slice),\
                    epochs = 10,steps_per_epoch = no_epochs,callbacks = [callback])
```



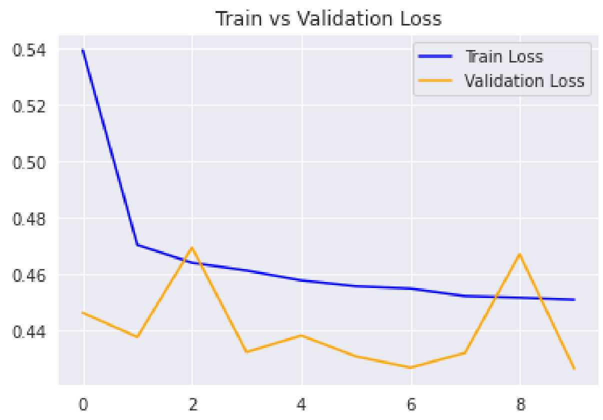
```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_7 (InputLayer)           [(None, 17)]         0           []

 input_8 (InputLayer)           [(None, 235)]        0           []

 input_3 (InputLayer)           [(None, 6)]          0           []

 input_4 (InputLayer)           [(None, 3)]          0           []

 input_5 (InputLayer)           [(None, 5)]          0           []

 input_6 (InputLayer)           [(None, 7)]          0           []

 embedding_4 (Embedding)        (None, 17, 50)       4471600     ['input_7[0][0]']

 embedding_5 (Embedding)        (None, 235, 50)      9345050     ['input_8[0][0]']

 embedding (Embedding)          (None, 6, 10)        43520       ['input_3[0][0]']

 embedding_1 (Embedding)        (None, 3, 10)        130         ['input_4[0][0]']

 embedding_2 (Embedding)        (None, 5, 10)        1410        ['input_5[0][0]']

 embedding_3 (Embedding)        (None, 7, 10)        9210        ['input_6[0][0]']

 lstm (LSTM)                    (None, 32)           10624       ['embedding_4[0][0]']

 lstm_1 (LSTM)                  (None, 32)           10624       ['embedding_5[0][0]']

 input_1 (InputLayer)           [(None, 5)]          0           []

 input_2 (InputLayer)           [(None, 2)]          0           []

 flatten (Flatten)              (None, 60)           0           ['embedding[0][0]']

 flatten_1 (Flatten)            (None, 30)           0           ['embedding_1[0][0]']

 flatten_2 (Flatten)            (None, 50)           0           ['embedding_2[0][0]']

 flatten_3 (Flatten)            (None, 70)           0           ['embedding_3[0][0]']
```



```
 flatten_4 (Flatten)            (None, 32)           0           ['lstm[0][0]']

 flatten_5 (Flatten)            (None, 32)           0           ['lstm_1[0][0]']

 input_9 (InputLayer)           [(None, 1)]          0           []

 input_10 (InputLayer)          [(None, 1)]          0           []

 concatenate (Concatenate)      (None, 283)          0           ['input_1[0][0]',
                                                                  'input_2[0][0]',
                                                                  'flatten[0][0]',
                                                                  'flatten_1[0][0]',
                                                                  'flatten_2[0][0]',
                                                                  'flatten_3[0][0]',
                                                                  'flatten_4[0][0]',
                                                                  'flatten_5[0][0]',
                                                                  'input_9[0][0]',
                                                                  'input_10[0][0]']

 dense (Dense)                  (None, 64)           18176       ['concatenate[0][0]']

 dropout (Dropout)              (None, 64)           0           ['dense[0][0]']

 batch_normalization (BatchNorm  (None, 64)          256         ['dropout[0][0]']
 alization)

 dense_1 (Dense)                (None, 32)           2080        ['batch_normalization[0][0]']

 dropout_1 (Dropout)            (None, 32)           0           ['dense_1[0][0]']

 batch_normalization_1 (BatchNo  (None, 32)          128         ['dropout_1[0][0]']
 rmalization)

 dense_2 (Dense)                (None, 16)           528         ['batch_normalization_1[0][0]']

 dropout_2 (Dropout)            (None, 16)           0           ['dense_2[0][0]']

 dense_3 (Dense)                (None, 1)            17          ['dropout_2[0][0]']

==================================================================================================
Total params: 13,913,353
Trainable params: 96,511
Non-trainable params: 13,816,842
```

```
______________________________________________________________________________________________________
==============================
Training Started...
==============================
Epoch 1/10
2180/2180 [==============================] - 942s 429ms/step - loss: 0.5392 - val_loss: 0.4464
Epoch 2/10
2180/2180 [==============================] - 927s 425ms/step - loss: 0.4705 - val_loss: 0.4378
Epoch 3/10
2180/2180 [==============================] - 945s 434ms/step - loss: 0.4641 - val_loss: 0.4695
Epoch 4/10
2180/2180 [==============================] - 936s 429ms/step - loss: 0.4614 - val_loss: 0.4325
Epoch 5/10
2180/2180 [==============================] - 936s 429ms/step - loss: 0.4579 - val_loss: 0.4384
Epoch 6/10
2180/2180 [==============================] - 930s 427ms/step - loss: 0.4558 - val_loss: 0.4310
Epoch 7/10
2180/2180 [==============================] - 915s 420ms/step - loss: 0.4550 - val_loss: 0.4270
Epoch 8/10
2180/2180 [==============================] - 906s 416ms/step - loss: 0.4523 - val_loss: 0.4322
Epoch 9/10
2180/2180 [==============================] - 909s 417ms/step - loss: 0.4518 - val_loss: 0.4671
Epoch 10/10
2180/2180 [==============================] - 908s 416ms/step - loss: 0.4510 - val_loss: 0.4267
<keras.callbacks.History at 0x7f766752bf10>
```

```python
plt.plot(model.history.history['loss'],color = 'Blue',label = 'Train Loss')
plt.plot(model.history.history['val_loss'],color = 'Orange',label = 'Validation Loss')
plt.legend()
plt.title('Train vs Validation Loss')
plt.show()
```



Model didn't overfit though the network structure was complex

```python
X_test = preprocess(x_train_slice,x_test)
results = model.evaluate(X_test,y_test, batch_size=1000)
print("test RMSLE", results)
```

```
445/445 [==============================] - 120s 267ms/step - loss: 0.4288
test RMSLE 0.4287871718406677
```

```
Predicted Price for 1th item in test data is: 10.74 with Actual Price as: 15.00
Predicted Price for 2th item in test data is: 23.02 with Actual Price as: 109.00
Predicted Price for 3th item in test data is: 13.04 with Actual Price as: 12.00
Predicted Price for 4th item in test data is: 24.04 with Actual Price as: 26.00
Predicted Price for 5th item in test data is: 27.60 with Actual Price as: 19.00
Predicted Price for 6th item in test data is: 17.75 with Actual Price as: 42.00
Predicted Price for 7th item in test data is: 14.92 with Actual Price as: 10.00
Predicted Price for 8th item in test data is: 7.47 with Actual Price as: 7.00
Predicted Price for 9th item in test data is: 21.28 with Actual Price as: 24.00
Predicted Price for 10th item in test data is: 23.75 with Actual Price as: 24.00
```

Key-Note:

- It predicts the price close for the low-middle priced products but for higher range products the model fails to predict the close by price.

Conclusion:

Ridge regression showed descent performance with respect to RMSLE(Root Mean Square Log Error) score and later a complex deep learning model with embedding layer and LSTM enhanced the model performance by a huge margin. Test RMSLE score is **0.4287** which is also considered in Kaggle.

Deployment and Predictions:

Deployed the model in AWS EC2 using Flask API. Created an HTML/JavaScript based web application with Predicted Sales Price button enabled which calls flask API internally which calls the model stored in .h5 format. Data pre-processing and vectorization is occurring inside the flask. Please let me know your thoughts for any modifications.

Future Work:

1. To try on CNN based approach.
2. To try with advanced text encoding techniques like- Bert. Using Hugging Face Library.
3. To try with different network architecture which will reduce the RMSLE error further.

GitHub:

https://github.com/samarjitbanerjee/Merceri-Price-Prediction-Challenge

Technical Blog:

https://medium.com/@samarjitbanerjee/given-product-details-predict-the-sales-price-of-the-product-2575972b106b

References

- https://www.appliedroots.com/course/6/diploma-in-ml-ai-with-uoh
- https://www.kaggle.com/c/mercari-price-suggestion-challenge
- https://www.youtube.com/watch?v=QFR0IHbzA30&t=2010s