In [1]:
```
# for numerical computing
import numpy as np

# for dataframes
import pandas as pd

# for easier visualization
import seaborn as sns

# for visualization and to display plots
from matplotlib import pyplot as plt
%matplotlib inline

# import color maps
from matplotlib.colors import ListedColormap

# Ignore Warnings
import warnings
warnings.filterwarnings("ignore")

from math import sqrt

# to split train and test set
from sklearn.model_selection import train_test_split

# to perform hyperparameter tuning
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV

from sklearn.linear_model import Ridge  # Linear Regression + L2 regularization
from sklearn.linear_model import Lasso  # Linear Regression + L1 regularization
from sklearn.svm import SVR # Support Vector Regressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor

# Evaluation Metrics
from sklearn.metrics import mean_squared_error as mse
from sklearn.metrics import r2_score as rs
from sklearn.metrics import mean_absolute_error as mae
```

```
#import xgboost
import os
mingw_path = 'C:\\Program Files\\mingw-w64\\x86_64-7.2.0-posix-seh-rt_v5-rev0\\mingw64\\bin'
os.environ['PATH'] = mingw_path + ';' + os.environ['PATH']
from xgboost import XGBRegressor
from xgboost import plot_importance  # to plot feature importance

# to save the final model on disk
from sklearn.externals import joblib
```

In [2]:
```
np.set_printoptions(precision=2, suppress=True) #for printing floating point numbers upto  precision 2
```

Load real estate data from CSV

In [3]:
```
df = pd.read_csv('BlackFriday 2 (1).csv')
```

In [4]:
```
df.shape
```

Out[4]:
```
(537577, 12)
```

Columns of the dataset

In [5]:
```
df.columns
```

Out[5]:
```
Index(['User_ID', 'Product_ID', 'Gender', 'Age', 'Occupation', 'City_Category',
       'Stay_In_Current_City_Years', 'Marital_Status', 'Product_Category_1',
       'Product_Category_2', 'Product_Category_3', 'Purchase'],
      dtype='object')
```

Display the first 5 rows to see example observations.

In [6]:
```
pd.set_option('display.max_columns', 12) ## display max 20 columns
df.head()
```

Out[6]:

	User_ID	Product_ID	Gender	Age	Occupation	City_Category	Stay_In_Current_City_Years	Marital_Status	Product_Category_1	Pro
0	1000001	P00069042	F	0-17	10	A	2	0	3	
1	1000001	P00248942	F	0-17	10	A	2	0	1	
2	1000001	P00087842	F	0-17	10	A	2	0	12	
3	1000001	P00085442	F	0-17	10	A	2	0	12	
4	1000002	P00285442	M	55+	16	C	4+	0	8	

Some feaures are numeric and some are categorical

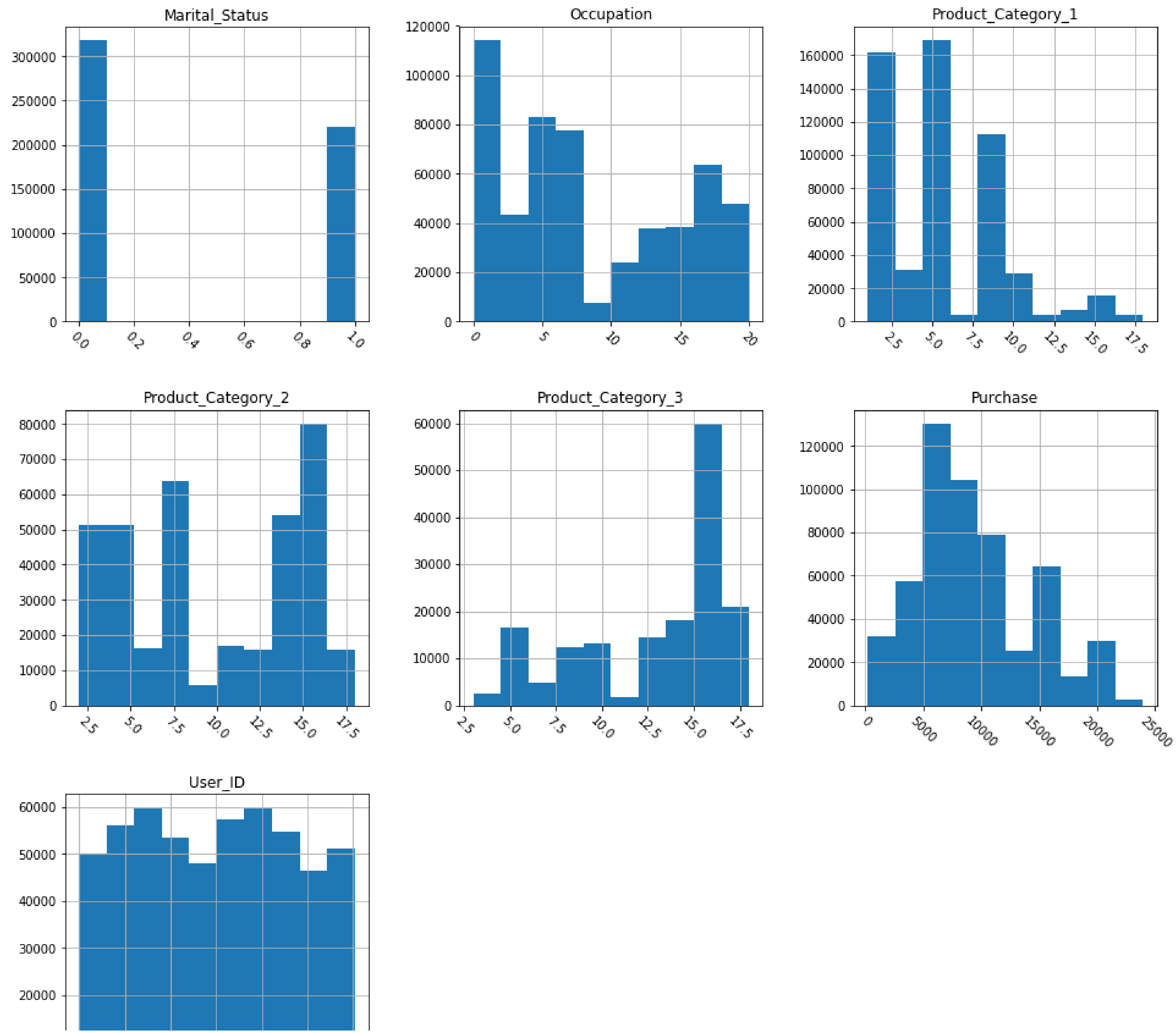
Filtering the categorical features:

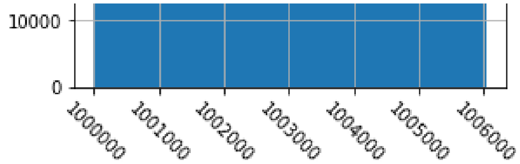
In [7]:
```
df.dtypes[df.dtypes=='object']
```

Out[7]:
```
Product_ID                    object
Gender                        object
Age                           object
City_Category                 object
Stay_In_Current_City_Years    object
dtype: object
```

Distributions of numeric features

In [8]:
```
# Plot histogram grid
df.hist(figsize=(16,16), xrot=-45) ## Display the labels rotated by 45 degress

# Clear the text "residue"
plt.show()
```


Marital_Status
0
50000
100000
150000
200000
250000
300000
0.0
0.2
0.4
0.6
0.8
1.0
Occupation
0
20000
40000
60000
80000
100000
120000
0
5
10
15
20
Product_Category_1
0
20000
40000
60000
80000
100000
120000
140000
160000
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Product_Category_2
0
10000
20000
30000
40000
50000
60000
70000
80000
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Product_Category_3
0
10000
20000
30000
40000
50000
60000
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Purchase
0
20000
40000
60000
80000
100000
120000
0
5000
10000
15000
20000
25000
User_ID
20000
30000
40000
50000
60000



Observations: We can make out quite a few observations:

For example, Marital status:

There are only two categories 0 and 1 - which can be assumed for married or not married, there are more observations for status 0.

Most observations in the Occupation column are for category 0.

Most purchases are for a sum between 5k-10k while least purchases are for less than 2,250.

Display summary statistics for the numerical features.

In [9]:
```
df.describe()
```

Out[9]:

	User_ID	Occupation	Marital_Status	Product_Category_1	Product_Category_2	Product_Category_3	Purchase
count	5.375770e+05	537577.00000	537577.000000	537577.000000	370591.000000	164278.000000	537577.000000
mean	1.002992e+06	8.08271	0.408797	5.295546	9.842144	12.669840	9333.859853
std	1.714393e+03	6.52412	0.491612	3.750701	5.087259	4.124341	4981.022133
min	1.000001e+06	0.00000	0.000000	1.000000	2.000000	3.000000	185.000000
25%	1.001495e+06	2.00000	0.000000	1.000000	5.000000	9.000000	5866.000000
50%	1.003031e+06	7.00000	0.000000	5.000000	9.000000	14.000000	8062.000000
75%	1.004417e+06	14.00000	1.000000	8.000000	15.000000	16.000000	12073.000000
max	1.006040e+06	20.00000	1.000000	18.000000	18.000000	18.000000	23961.000000

Obeservation:

i didn't find missing values excepet for category 2 and 3 -when these are sumed together I get closest the total sum of observations - I can infer that it can be possible that all observations were divided between the 2 by sum condition.

Distributions of categorical features

In [10]:
```
# Display summary of statistics for categorical features.
```

In [11]:
```
df.describe(include=['object'])
```

Out[11]:

	Product_ID	Gender	Age	City_Category	Stay_In_Current_City_Years
count	537577	537577	537577	537577	537577
unique	3623	2	7	3	5
top	P00265242	M	26-35	B	1
freq	1858	405380	214690	226493	189192

Observation:

I didn't find missing values in the object values columns.

There are 2 unique categories for gender from which M (Male) is the most common in 405k out of 537k.

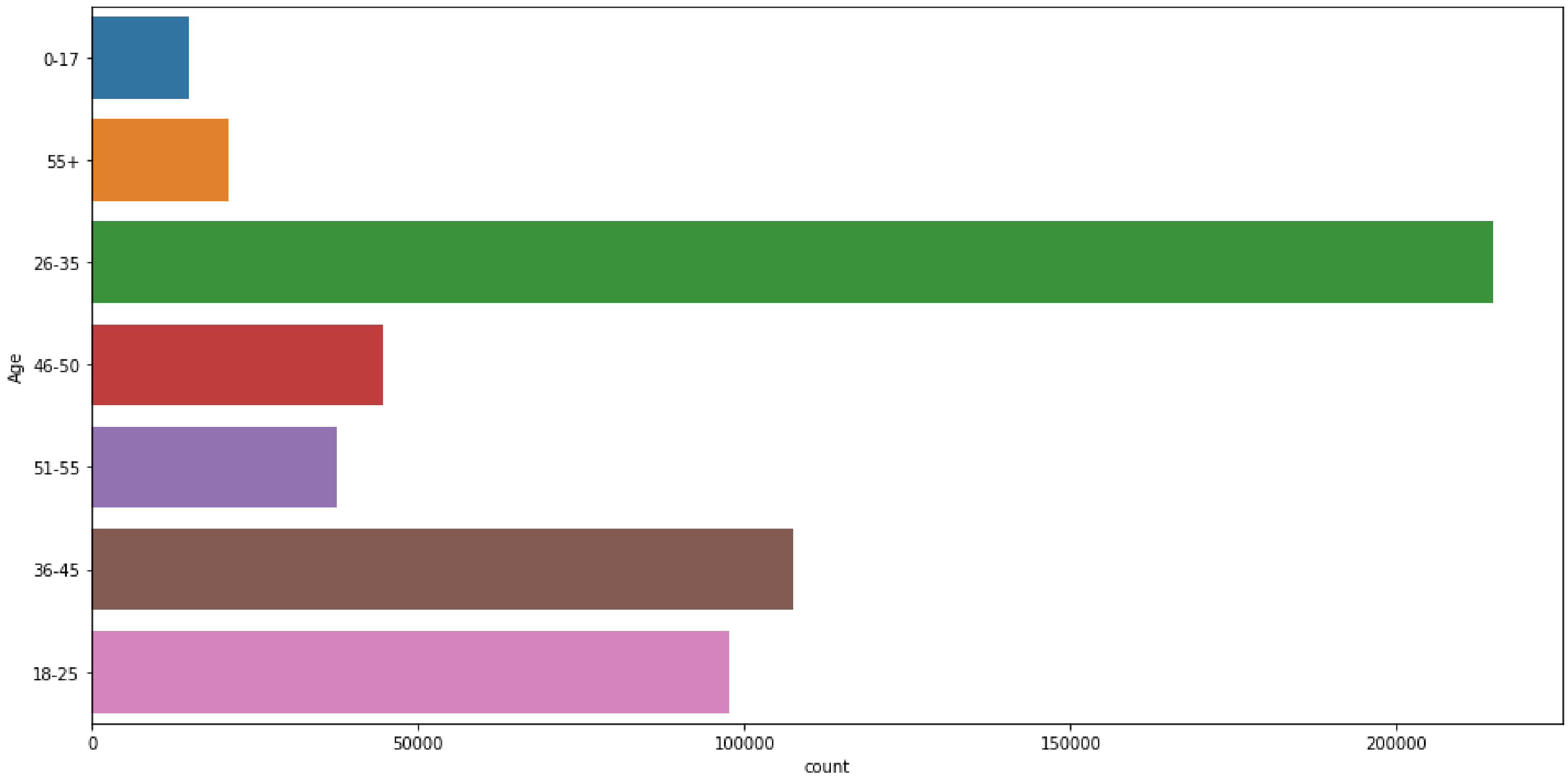
There are 7 unique categories for Age from which the most common one is peiople between 26-35.

Most observations are for people who stayed 1 year in the current city out of possible 5 categories.

Bar plots for categorical Features

In [12]:
```
plt.figure(figsize=(16,8))
sns.countplot(y='Age', data=df)
```

Out[12]: <matplotlib.axes._subplots.AxesSubplot at 0x16ac439e048>



Observations: Take a look at the frequencies of the classes.

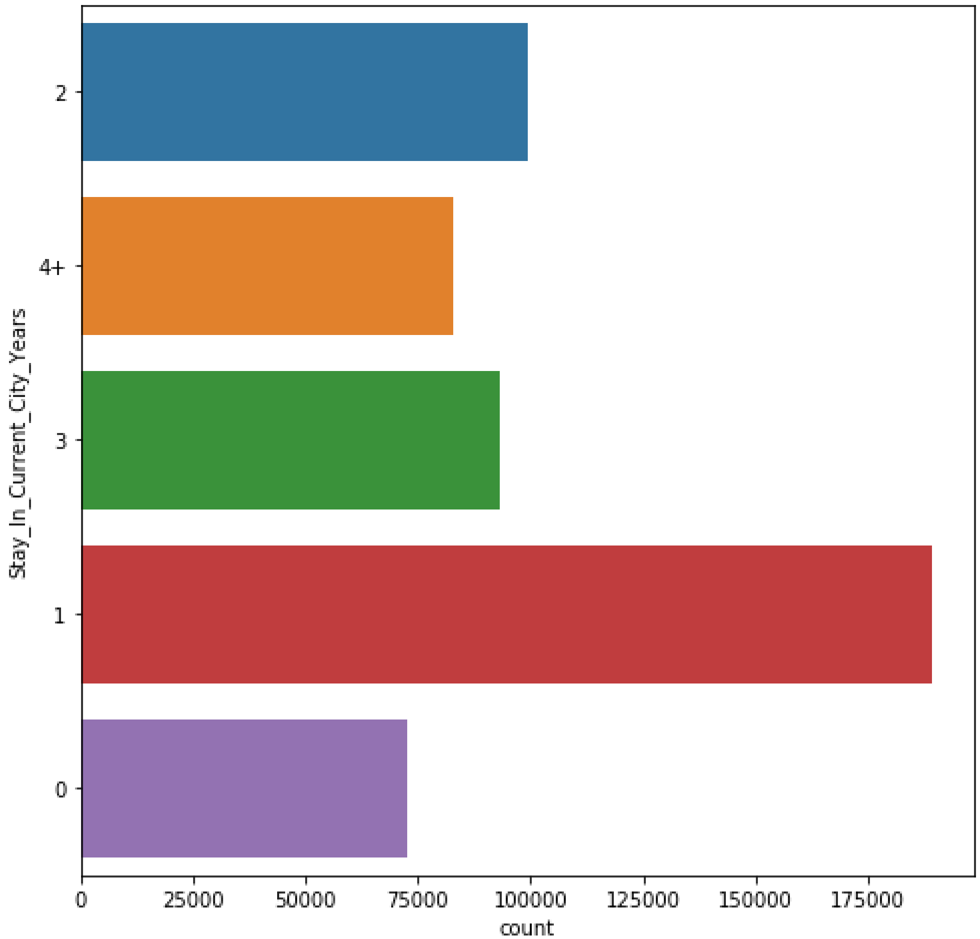
Some classes are quite prevalent in the dataset. It has longer bars. Those include:

'26-35' shows count of more than 200k which are most frequent, '36-45' is the 2nd with more than 100k, '18-25' is the 3rd with almost 100k

It has no sparse classes as well categories have a significant number of observations.

In [13]:
```
plt.figure(figsize=(8,8))
sns.countplot(y='Stay_In_Current_City_Years', data=df)
```

Out[13]: <matplotlib.axes._subplots.AxesSubplot at 0x16ac4418cf8>



Observations:

In this class which has a largest count is '1' Following are the rest of the classes with a larger amount of years. with similiar distribution between them.

'0', '2', '3' & '4+'.

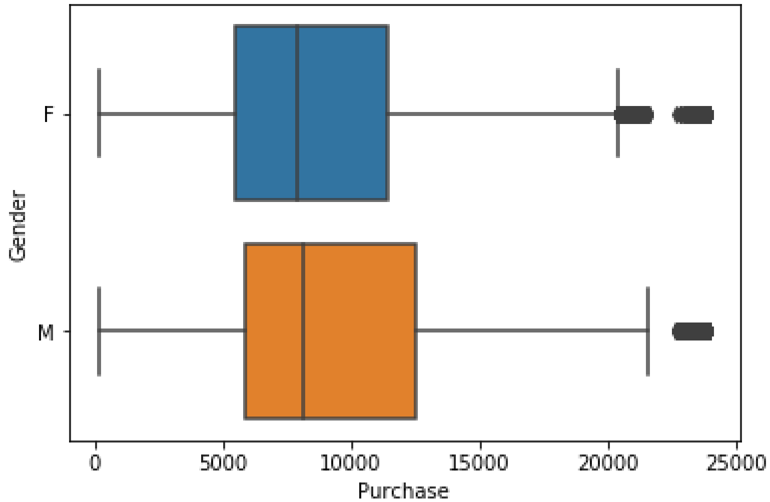
Segmentations

Segmentations are powerful ways to cut the data to observe the relationship between categorical features and numerical features.

Here segmenting the target variable by key categorical features.

In [14]:
```
sns.boxplot(y='Gender', x='Purchase', data=df)
```

Out[14]: <matplotlib.axes._subplots.AxesSubplot at 0x16ac4457e10>



Observation:

In general,base of Observations data,it looks like Males were making more purchases in this black friday.
Let's compare the two Gender categories across other features as well

In [16]:
```
df.groupby('Gender').mean()
```

Out[16]:

Gender	User_ID	Occupation	Marital_Status	Product_Category_1	Product_Category_2	Product_Category_3	Purchase
F	1.003088e+06	6.742672	0.417733	5.595445	10.007969	12.452318	8809.761349
M	1.002961e+06	8.519705	0.405883	5.197748	9.789072	12.732924	9504.771713

Observations :

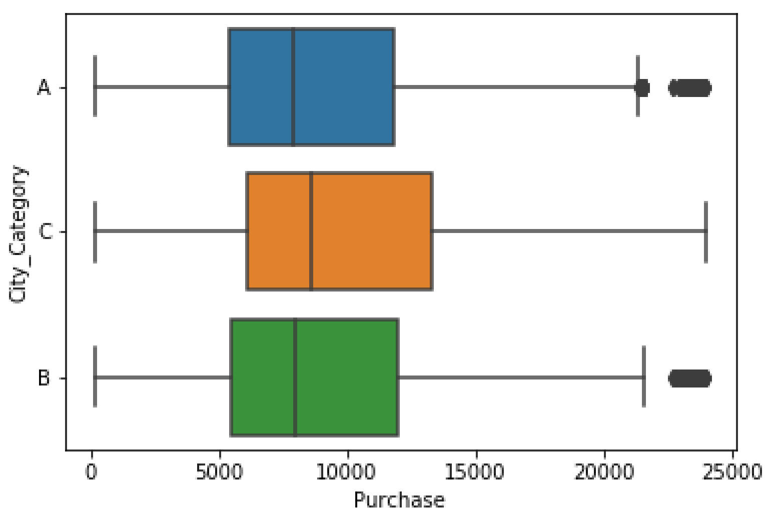
There has nearly the same 40% marital status of zero both for males and females.

The product of categories columns don't show significant change between the genders. excepet for a little higher numbers in category 1 and 3 towards female.

According to observation - there were more purchaes made by males The mean of male purchaes is 9504 while female's is 8809.

In [17]:
```
sns.boxplot(y='City_Category', x='Purchase', data=df)
```

Out[17]: <matplotlib.axes._subplots.AxesSubplot at 0x16ac463f160>



Observation:

For City_Category 1 and 3 there have less observations for people purchasing in more then 225k than there are from 0-225k.

Segment by property_type and display the means and standard deviations within each class

In [18]:
```
df.groupby('Age').agg([np.mean, np.std])
```

Out[18]:

	User_ID		Occupation		Marital_Status		...	Product_Category_2		Product_Category_3		Purchase	
	mean	std	mean	std	mean	std	...	mean	std	mean	std	mean	st
Age													
0-17	1.002676e+06	1755.525095	8.790236	4.491994	0.000000	0.000000	...	9.023027	5.176184	11.850282	4.383450	9020.126878	5
18-25	1.002766e+06	1716.270135	6.737141	5.949072	0.211412	0.408312	...	9.474317	5.140842	12.395286	4.243974	9235.197575	4
26-35	1.003075e+06	1719.986312	7.902343	6.698011	0.392035	0.488206	...	9.810403	5.075915	12.648689	4.123401	9314.588970	4
36-45	1.003030e+06	1677.032766	8.847152	6.588780	0.395418	0.488942	...	9.954321	5.082563	12.750717	4.078818	9401.478758	4
46-50	1.003152e+06	1768.300690	8.526367	6.682162	0.723038	0.447502	...	10.177195	5.016661	12.937952	3.993584	9284.872277	4
51-55	1.002950e+06	1667.161146	8.809506	6.664605	0.717183	0.450374	...	10.280446	5.028167	13.108187	3.941584	9620.616620	5
55+	1.002951e+06	1644.942652	9.537961	6.358962	0.634981	0.481447	...	10.462992	4.941885	13.154686	3.938299	9453.898579	4

7 rows × 14 columns

In [19]:
```
# Finally, let's take a look at the relationships between numeric features and other numeric features.
# Corelation is a value between -1 and 1 that represents how closely values for two separate features.
# Positive corelation means that as one feature increases, the other increases.
# Negative corelation means that as one feature increases, the other decreases.
# Corelations near -1 or 1 indicate a strong relationship.
# Those are closer to 0 indicate a weak relationship.
# 0 indicates no relationship.
```

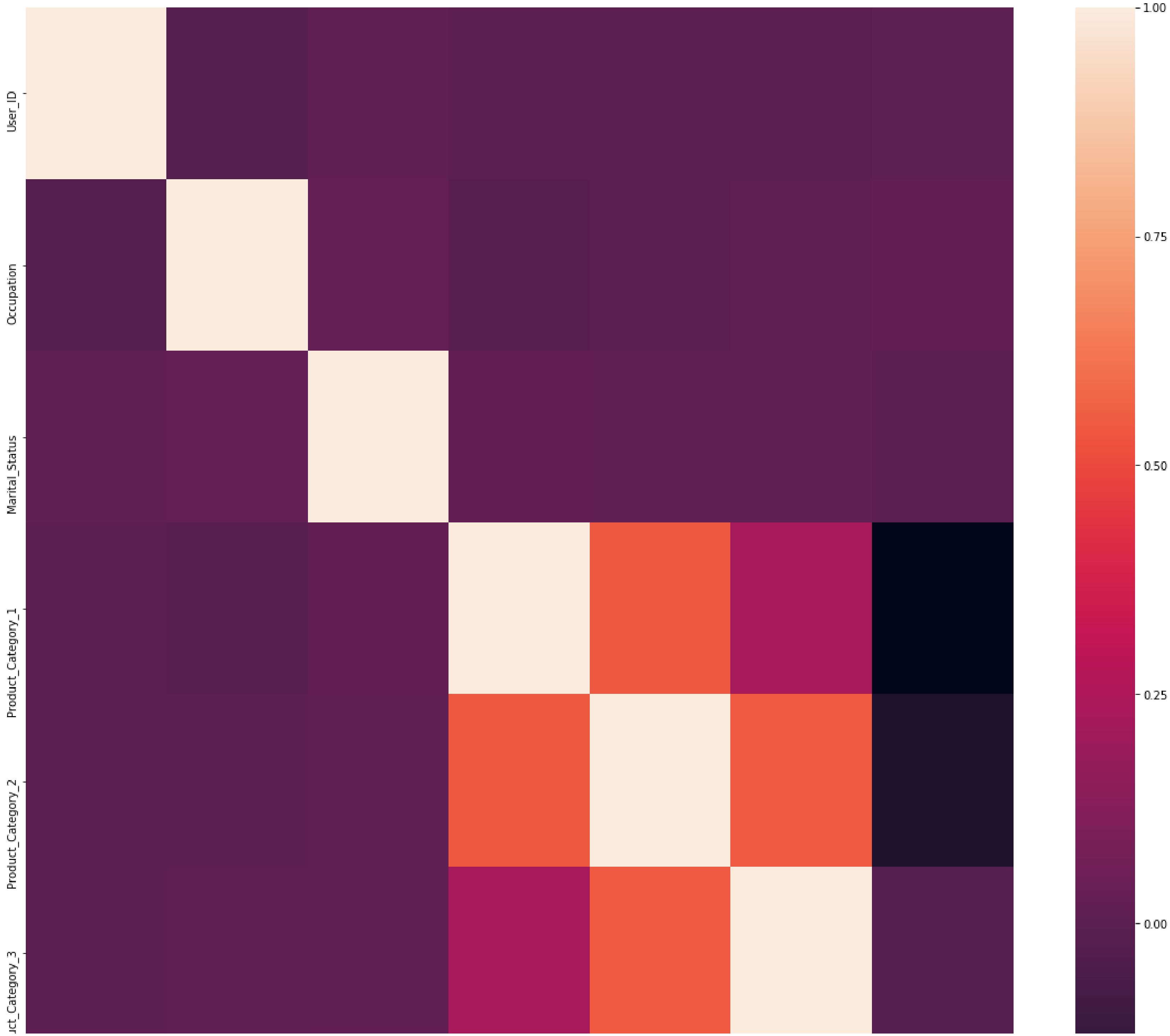
In [20]:
```
df.corr()
```

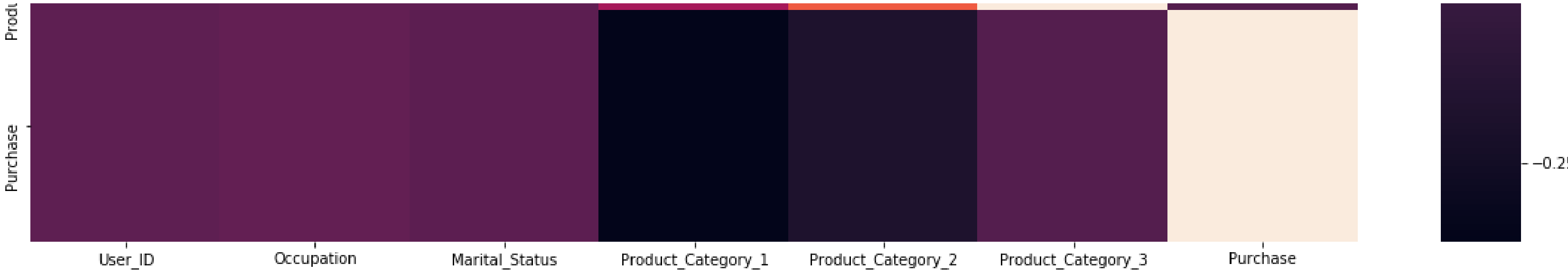
Out[20]:

	User_ID	Occupation	Marital_Status	Product_Category_1	Product_Category_2	Product_Category_3	Purchase
User_ID	1.000000	-0.023024	0.018732	0.003687	0.001471	0.004045	0.005389
Occupation	-0.023024	1.000000	0.024691	-0.008114	-0.000031	0.013452	0.021104
Marital_Status	0.018732	0.024691	1.000000	0.020546	0.015116	0.019452	0.000129
Product_Category_1	0.003687	-0.008114	0.020546	1.000000	0.540423	0.229490	-0.314125
Product_Category_2	0.001471	-0.000031	0.015116	0.540423	1.000000	0.543544	-0.209973
Product_Category_3	0.004045	0.013452	0.019452	0.229490	0.543544	1.000000	-0.022257
Purchase	0.005389	0.021104	0.000129	-0.314125	-0.209973	-0.022257	1.000000

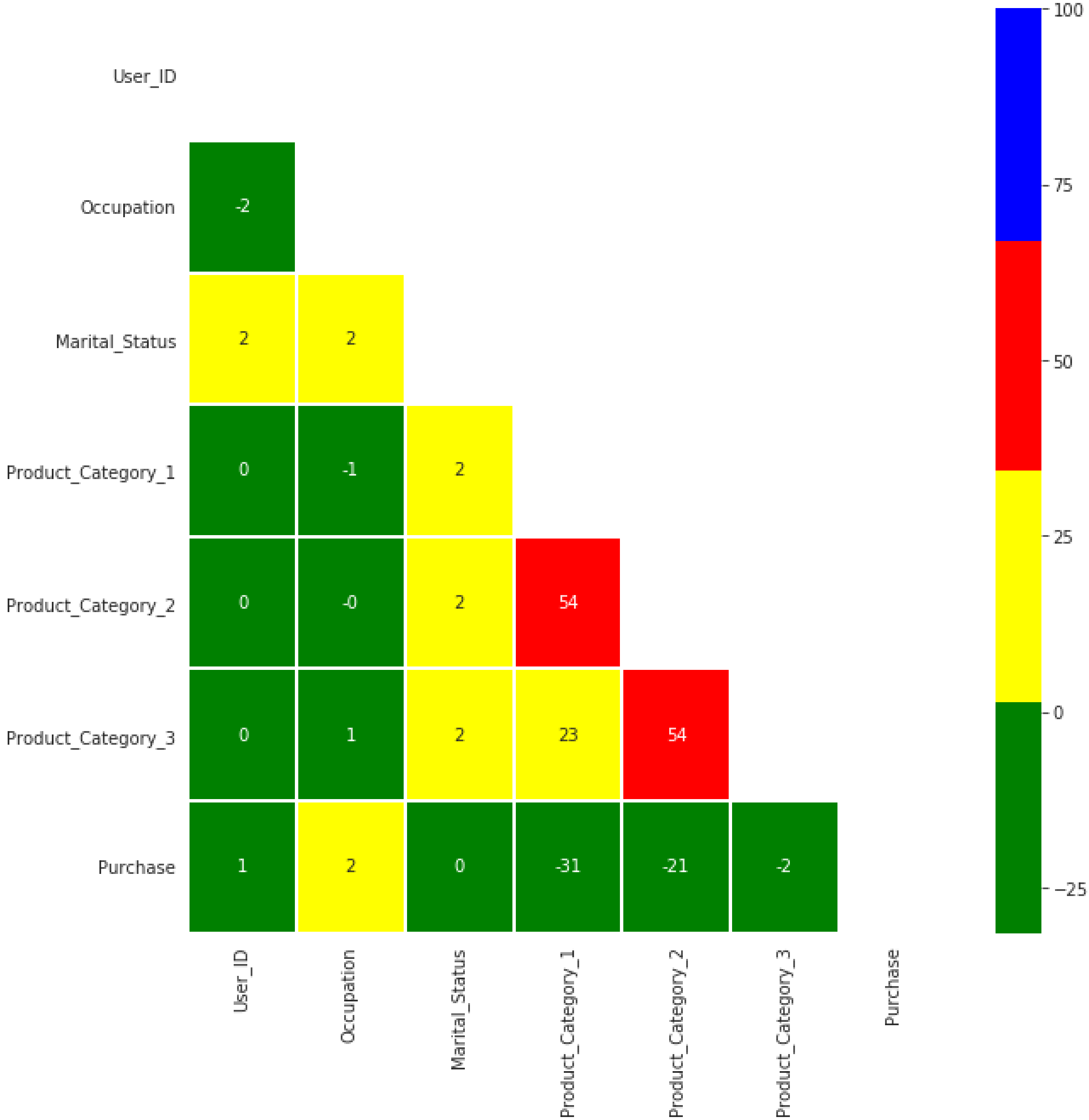
In [21]:
```
plt.figure(figsize=(20,20))
sns.heatmap(df.corr())
```

Out[21]: <matplotlib.axes._subplots.AxesSubplot at 0x16ac451a908>


User_ID
Occupation
Marital_Status
Product_Category_1
Product_Category_2
uct_Category_3
1.00
0.75
0.50
0.25
0.00


Produ
Purchase
User_ID
Occupation
Marital_Status
Product_Category_1
Product_Category_2
Product_Category_3
Purchase
−0.25

In [22]:
```
mask=np.zeros_like(df.corr())
mask[np.triu_indices_from(mask)] = True
plt.figure(figsize=(10,10))
with sns.axes_style("white"):
    ax = sns.heatmap(df.corr()*100, mask=mask, fmt='.0f', annot=True, lw=1, cmap=ListedColormap(['green', 'ye
llow', 'red','blue']))
```


User_ID
Occupation
Marital_Status
Product_Category_1
Product_Category_2
Product_Category_3
Purchase
-2
2
2
0
-1
2
0
-0
2
54
0
1
2
23
54
1
2
0
-31
-21
-2
100
75
50
25
0
−25

Data Cleaning

In [23]:
```
# Dropping the duplicates (De-duplication)
df = df.drop_duplicates()
print( df.shape )
```

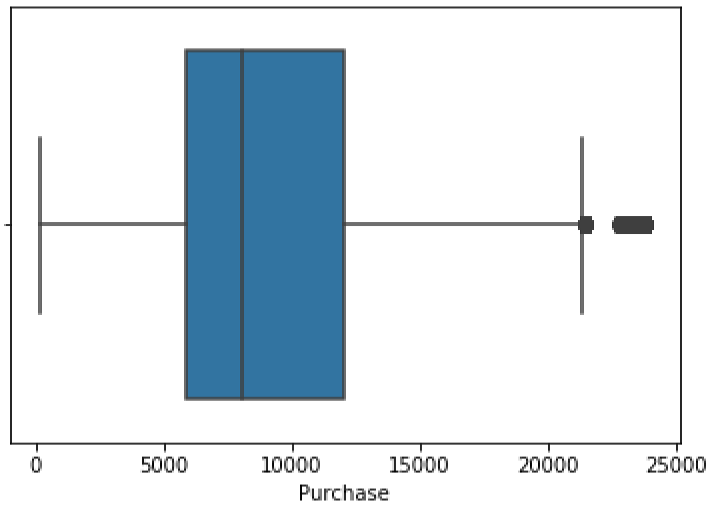
```
(537577, 12)
```

In [24]:
```
# It looks like we didn't have any duplicates in our original dataset. (Same case for black friday data)
# Even so, it's a good idea to check this as an easy first step for cleaning your dataset
```

In [78]:
```
# Fix structural errors
# I could not find similiar structural errors

# Typos and capitalization
# I could not find similiar types or capitalization errors

# Mislabeled classes
# I could not find the errors indicated

# Removing Outliers
# Outliers can cause problems with certain types of models.
# Boxplots are a nice way to detect outliers
# Let's start with a box plot of your target variable, since that's what you're actually trying to predict
```

In [25]:
```
sns.boxplot(df.Purchase)
```

Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0x16ac428d550>

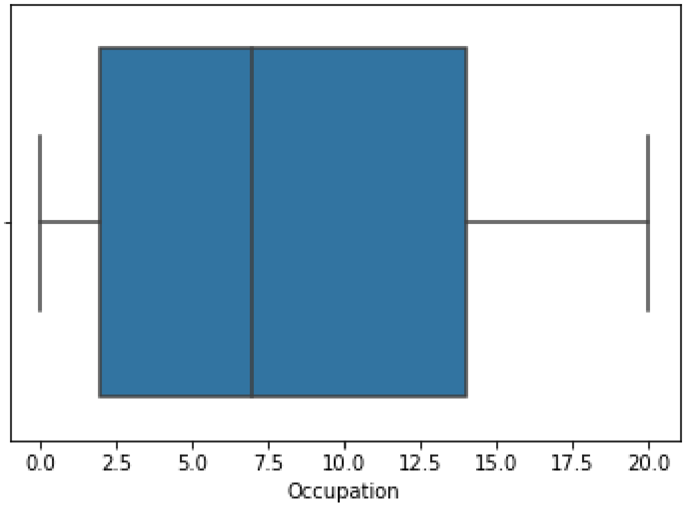


In [26]:
```
# Interpretation
```

In [79]:
```
# The two vertical bars on the ends are the min and max values.
# All purchases were between \$0 to \$25,000
# This box in the middle is the interquartile range (25th percentile to 75th percentile).
# Half of all observations fall in that box.
# Finally, the vertical bar in the middle of the box is the median.
```

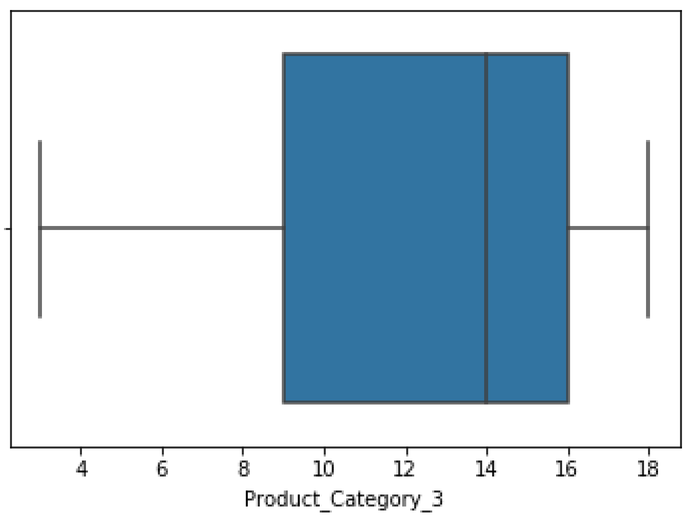
In [28]:
```
## Checking outliers in Occupation
sns.boxplot(df.Occupation)
```

Out[28]: <matplotlib.axes._subplots.AxesSubplot at 0x16ac415d198>



In [29]:
```
# Checking outliers in Product_Category_3
sns.boxplot(df.Product_Category_3)
```

Out[29]: <matplotlib.axes._subplots.AxesSubplot at 0x16ac41190f0>



Label missing categorical data

In [30]:
```
# We cannot simply ignore missing values in your dataset.
# We must handle them in some way for the very practical reason that Scikit-Learn algorithms
# I do not accept missing values.
```

In [80]:
```
# Displaying number of missing values by categorical feature
df.select_dtypes(include=['object']).isnull().sum()
```

Out[80]:
```
Series([], dtype: float64)
```

Observation: There are no missing values in the categorical columns

Flag and fill missing numeric data

In [32]:
```
# Display number of missing values by numeric feature
df.select_dtypes(exclude=['object']).isnull().sum()
```

Out[32]:
```
User_ID                    0
Occupation                 0
Marital_Status             0
Product_Category_1         0
Product_Category_2    166986
Product_Category_3    373299
Purchase                   0
dtype: int64
```

In [33]:
```
# I tried runnin this code to fill the nan values but it caused a problem
#instead Im running it afer loading the new analyticaldf and it works fine
```

Feature Engineering

Indicator variables

In [34]:
```
# Since there is no evident correlation that indicates a strong connection between some variables rather then
others
# there is no need to do this step.
```

Interaction features

In [35]:
```
# Since there is no evident correlation that indicates a strong connection between some variables rather then
others
# there is no need to do this step.
```

Handling Sparse Classes

In [36]:
```
# I did not identify Sparse classes in the data base.
```

Encode dummy variables (One Hot Encoding)

In [37]:
```
# Machine learning algorithms cannot directly handle categorical features. Specifically, they cannot handle t
ext values.
# Therefore, we need to create dummy variables for our categorical features.
# Dummy variables are a set of binary (0 or 1) features that each represent a single class from a categorical
feature.
```

In [38]:
```
df.head()
```

Out[38]:

	User_ID	Product_ID	Gender	Age	Occupation	City_Category	Stay_In_Current_City_Years	Marital_Status	Product_Category_1	Pro
0	1000001	P00069042	F	0-17	10	A	2	0	3	
1	1000001	P00248942	F	0-17	10	A	2	0	1	
2	1000001	P00087842	F	0-17	10	A	2	0	12	
3	1000001	P00085442	F	0-17	10	A	2	0	12	
4	1000002	P00285442	M	55+	16	C	4+	0	8	

In [39]:
```
# Create a new dataframe with dummy variables for for our categorical features.
df = pd.get_dummies(df, columns=['Gender', 'Age', 'City_Category', 'Stay_In_Current_City_Years'])
```

In [40]:
```
# Note: There are many ways to perform one-hot encoding,
# you can also use LabelEncoder and OneHotEncoder classes in SKLEARN or use the above pandas function.
```

In [41]:
```
df.head()
```

Out[41]:

	User_ID	Product_ID	Occupation	Marital_Status	Product_Category_1	Product_Category_2	...	City_Category_C	Stay_In_Current_Cit
0	1000001	P00069042	10	0	3	NaN	...	0	
1	1000001	P00248942	10	0	1	6.0	...	0	
2	1000001	P00087842	10	0	12	NaN	...	0	
3	1000001	P00085442	10	0	12	14.0	...	0	
4	1000002	P00285442	16	0	8	NaN	...	1	

5 rows × 25 columns

Remove unused or redundant features

In [42]:
```
df = df.drop(['User_ID'], axis=1)
df = df.drop(['Product_ID'], axis=1)
```

In [43]:
```
df.head(2)
```

Out[43]:

	Occupation	Marital_Status	Product_Category_1	Product_Category_2	Product_Category_3	Purchase	...	City_Category_C	Stay_In_C
0	10	0	3	NaN	NaN	8370	...	0	
1	10	0	1	6.0	14.0	15200	...	0	

2 rows × 23 columns

In [44]:
```
df.to_csv(r'C:\Users\OWNER\Desktop\Xman.csv', index=None)
```

In [45]:
```
df = pd.read_csv("Xman.csv")
```

In [46]:
```
df.shape
```

Out[46]:
```
(537577, 23)
```

In [47]:
```
df.head()
```

Out[47]:

	Occupation	Marital_Status	Product_Category_1	Product_Category_2	Product_Category_3	Purchase	...	City_Category_C	Stay_In_C
0	10	0	3	NaN	NaN	8370	...	0	
1	10	0	1	6.0	14.0	15200	...	0	
2	10	0	12	NaN	NaN	1422	...	0	
3	10	0	12	14.0	NaN	1057	...	0	
4	16	0	8	NaN	NaN	7969	...	1	

5 rows × 23 columns

In [48]:
```
df = df.fillna(0)
```

In [49]:
```
df.head()
```

Out[49]:

	Occupation	Marital_Status	Product_Category_1	Product_Category_2	Product_Category_3	Purchase	...	City_Category_C	Stay_In_C
0	10	0	3	0.0	0.0	8370	...	0	
1	10	0	1	6.0	14.0	15200	...	0	
2	10	0	12	0.0	0.0	1422	...	0	
3	10	0	12	14.0	0.0	1057	...	0	
4	16	0	8	0.0	0.0	7969	...	1	

5 rows × 23 columns

Train and Test Splits

In [50]:
```
# Create separate object for target variable
y = df.Purchase
# Create separate object for input features
X = df.drop('Purchase', axis=1)
```

In [54]:
```
# Split X and y into train and test sets: 80- and 20
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1234)
```

In [55]:
```
# Let's confirm we have the right number of observations in each subset
```

In [56]:
```
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
(430061, 22) (107516, 22) (430061,) (107516,)
```

Data standardization

In [57]:
```
# In Data Standardization we perform zero mean centring and unit scaling; i.e.
# we make the mean of all the features as zero and the standard deviation as 1.
# hus we use mean and standard deviation of each feature.
# It is very important to save the mean and standard deviation for each of the feature from the training set,
# because we use the same mean and standard deviation in the test set.
```

In [58]:
```
train_mean = X_train.mean(numeric_only=True)
# train_mean = X_train.mean()
```

In [59]:
```
train_std = X_train.std()
```

In [60]:
```
## Standardize the train data set
X_train = (X_train - train_mean) / (train_std)
```

In [61]:
```
## Checking for mean and std dev.
X_train.describe()
```

Out[61]:

	Occupation	Marital_Status	Product_Category_1	Product_Category_2	Product_Category_3	Gender_F	...	City_Category_C
count	4.300610e+05	4.300610e+05	4.300610e+05	4.300610e+05	4.300610e+05	4.300610e+05	...	4.300610e+05
mean	-1.303425e-15	1.139689e-14	-2.232773e-16	8.255428e-17	1.943340e-15	1.978527e-15	...	3.249014e-15
std	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	...	1.000000e+00
min	-1.239073e+00	-8.305494e-01	-1.146040e+00	-1.094008e+00	-6.181721e-01	-5.717209e-01	...	-6.698680e-01
25%	-9.323963e-01	-8.305494e-01	-1.146040e+00	-1.094008e+00	-6.181721e-01	-5.717209e-01	...	-6.698680e-01
50%	-1.657057e-01	-8.305494e-01	-7.924827e-02	-2.892124e-01	-6.181721e-01	-5.717209e-01	...	-6.698680e-01
75%	9.076610e-01	1.204020e+00	7.208454e-01	1.159419e+00	6.577321e-01	-5.717209e-01	...	1.492828e+00
max	1.827690e+00	1.204020e+00	3.387824e+00	1.803255e+00	2.252612e+00	1.749101e+00	...	1.492828e+00

8 rows × 22 columns

In [62]:
```
## We are using train_mean and train_std_dev to standerdize test data set
X_test = (X_test - train_mean) / train_std
```

In [63]:
```
## Checking for mean and std dev. - not exactly 0 and 1
X_test.describe()
```

Out[63]:

	Occupation	Marital_Status	Product_Category_1	Product_Category_2	Product_Category_3	Gender_F	...	City_Category_(
count	107516.000000	107516.000000	107516.000000	107516.000000	107516.000000	107516.000000	...	107516.00000(
mean	0.001575	0.005884	-0.002133	-0.009578	-0.003353	-0.005007	...	-0.00124:
std	1.001983	1.001084	1.001522	0.999048	0.996727	0.997039	...	0.99949:
min	-1.239073	-0.830549	-1.146040	-1.094008	-0.618172	-0.571721	...	-0.66986{
25%	-0.932396	-0.830549	-1.146040	-1.094008	-0.618172	-0.571721	...	-0.66986{
50%	-0.165706	-0.830549	-0.079248	-0.289212	-0.618172	-0.571721	...	-0.66986{
75%	0.907661	1.204020	0.720845	1.159419	0.657732	-0.571721	...	1.49282{
max	1.827690	1.204020	3.387824	1.803255	2.252612	1.749101	...	1.49282{

8 rows × 22 columns

Model 1 - Baseline Model

In [64]:
```
# In this model, for every test data point, we will simply predict the average of the train labels as the out
put.
# Using this simplest model to perform hypothesis testing for other complex models.
```

In [65]:
```
## Predict Train results
y_train_pred = np.ones(y_train.shape[0])*y_train.mean()
```

In [66]:
```
## Predict Test results
y_pred = np.ones(y_test.shape[0])*y_train.mean()
from sklearn.metrics import r2_score
```

In [67]:
```
print("Train Results for Baseline Model:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_train.values, y_train_pred)))
print("R-squared: ", r2_score(y_train.values, y_train_pred))
print("Mean Absolute Error: ", mae(y_train.values, y_train_pred))
```

```
Train Results for Baseline Model:
*******************************
Root mean squared error:  4981.515912062438
R-squared:  0.0
Mean Absolute Error:  4047.5660267444778
```

In [68]:
```
print("Results for Baseline Model:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_test, y_pred)))
print("R-squared: ", r2_score(y_test, y_pred))
print("Mean Absolute Error: ", mae(y_test, y_pred))
```

```
Results for Baseline Model:
*******************************
Root mean squared error:  4979.023398336429
R-squared:  -6.53743990053357e-08
Mean Absolute Error:  4047.0879520090007
```

Model-2 Ridge Regression

In [69]:
```
tuned_params = {'alpha': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000]}
model = GridSearchCV(Ridge(), tuned_params, scoring = 'neg_mean_absolute_error', cv=10, n_jobs=-1)
model.fit(X_train, y_train)
```

Out[69]:
```
GridSearchCV(cv=10, error_score='raise-deprecating',
       estimator=Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001),
       fit_params=None, iid='warn', n_jobs=-1,
       param_grid={'alpha': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='neg_mean_absolute_error', verbose=0)
```

In [70]:
```
model.best_estimator_
```

Out[70]:
```
Ridge(alpha=0.0001, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001)
```

In [71]:
```
## Predict Train results
y_train_pred = model.predict(X_train)
```

In [72]:
```
## Predict Test results
y_pred = model.predict(X_test)
```

In [73]:
```
print("Train Results for Ridge Regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_train.values, y_train_pred)))
print("R-squared: ", r2_score(y_train.values, y_train_pred))
print("Mean Absolute Error: ", mae(y_train.values, y_train_pred))
```

```
Train Results for Ridge Regression:
*******************************
Root mean squared error:  4631.276680504524
R-squared:  0.1356723412638342
Mean Absolute Error:  3546.422727543275
```

In [74]:
```
print("Test Results for Ridge Regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_test, y_pred)))
print("R-squared: ", r2_score(y_test, y_pred))
print("Mean Absolute Error: ", mae(y_test, y_pred))
```

```
Test Results for Ridge Regression:
*******************************
Root mean squared error:  4625.9122688787
R-squared:  0.13680984561801457
Mean Absolute Error:  3543.601394749331
```

Feature Importance

In [75]:
```
## Building the model with the best hyperparameters here
model = Ridge(alpha=100)
model.fit(X_train, y_train)
```

Out[75]:
```
Ridge(alpha=100, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001)
```

In [76]:
```
indices = np.argsort(-abs(model.coef_))
print("The features in order of importance are:")
print(50*'-')
for feature in X.columns[indices]:
    print(feature)
```

```
The features in order of importance are:
--------------------------------------------------
Product_Category_1
Product_Category_3
City_Category_C
City_Category_A
Gender_M
Gender_F
Age_51-55
Age_0-17
Age_18-25
City_Category_B
Product_Category_2
Occupation
Age_55+
Marital_Status
Age_36-45
Stay_In_Current_City_Years_0
Stay_In_Current_City_Years_2
Age_46-50
Age_26-35
Stay_In_Current_City_Years_4+
Stay_In_Current_City_Years_3
Stay_In_Current_City_Years_1
```

Model-3 Support Vector Regression

In [ ]:
```
tuned_params = {'C': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000], 'gamma': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000]}
```

In [ ]:
```
model = GridSearchCV(SVR(), tuned_params, scoring = 'neg_mean_absolute_error', cv=5, n_jobs=-1)
```

In [ ]:
```
model.fit(X_train, y_train)
## This takes around 20 minutes but doesn't run
```

In [ ]:
```
model.best_estimator_
```

In [ ]:
```
## Building the model again with the best hyperparameters
model = SVR(C=100000, gamma=0.01)
model.fit(X_train, y_train)
```

In [ ]:
```
## Predict Train results
y_train_pred = model.predict(X_train)
```

In [ ]:
```
## Predict Test results
y_pred = model.predict(X_test)
```

In [ ]:
```
print("Train Results for Support Vector Regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_train.values, y_train_pred)))
print("R-squared: ", r2_score(y_train.values, y_train_pred))
print("Mean Absolute Error: ", mae(y_train.values, y_train_pred))
```

Model-4 Random Forest Regression

In [ ]:
```
## Reference for random search on random forest
## https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa
77dd74
tuned_params = {'n_estimators': [100, 200, 300, 400, 500], 'min_samples_split': [2, 5, 10], 'min_samples_lea
f': [1, 2, 4]}
```

In [ ]:
```
model = RandomizedSearchCV(RandomForestRegressor(), tuned_params, n_iter=20, scoring = 'neg_mean_absolute_err
or', cv=5, n_jobs=-1)
```

In [ ]:
```
model.fit(X_train, y_train)
## This takes more than 15 minutes but doesn't run
```

In [ ]:
```
model.best_estimator_
```

In [ ]:
```
## Predict Train results
y_train_pred = model.predict(X_train)
```

In [ ]:
```
## Predict Test results
y_pred = model.predict(X_test)
```

In [ ]:
```
print("Train Results for Random Forest Regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_train.values, y_train_pred)))
print("R-squared: ", r2_score(y_train.values, y_train_pred))
print("Mean Absolute Error: ", mae(y_train.values, y_train_pred))
```

In [ ]:
```
print("Test Results for Random Forest Regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_test, y_pred)))
print("R-squared: ", r2_score(y_test, y_pred))
print("Mean Absolute Error: ", mae(y_test, y_pred))
```

Feature Importance

In [ ]:
```
## Building the model again with the best hyperparameters
model = RandomForestRegressor(n_estimators=200, min_samples_split=10, min_samples_leaf=2)
model.fit(X_train, y_train)
```

In [ ]:
```
indices = np.argsort(-model.feature_importances_)
print("The features in order of importance are:")
print(50*'-')
for feature in X.columns[indices]:
    print(feature)
```

Model-5 XGBoost Regression

In [ ]:
```
## Reference for random search on xgboost
## https://gist.github.com/wrwr/3f6b66bf4ee01bf48be965f60d14454d
tuned_params = {'max_depth': [1, 2, 3, 4, 5], 'learning_rate': [0.01, 0.05, 0.1], 'n_estimators': [100, 200,
300, 400, 500], 'reg_lambda': [0.001, 0.1, 1.0, 10.0, 100.0]}
model = RandomizedSearchCV(XGBRegressor(), tuned_params, n_iter=20, scoring = 'neg_mean_absolute_error', cv=5
, n_jobs=-1)
model.fit(X_train, y_train)
```

In [ ]:
```
model.best_estimator_
```

In [ ]:
```
## Predict Train results
y_train_pred = model.predict(X_train)
```

In [ ]:
```
## Predict Test results
y_pred = model.predict(X_test)
```

In [ ]:
```
print("Train Results for XGBoost Regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_train.values, y_train_pred)))
print("R-squared: ", rs(y_train.values, y_train_pred))
print("Mean Absolute Error: ", mae(y_train.values, y_train_pred))
```

In [ ]:
```
print("Test Results for XGBoost Regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_test, y_pred)))
print("R-squared: ", r2_score(y_test, y_pred))
print("Mean Absolute Error: ", mae(y_test, y_pred))
```

Feature Importance

In [ ]:
```
## Building the model again with the best hyperparameters
model = XGBRegressor(max_depth=2,learning_rate=0.05,n_estimators=400, reg_lambda=0.001)
model.fit(X_train, y_train)
```

In [ ]:
```
## Function to include figsize parameter
## Reference: https://stackoverflow.com/questions/40081888/xgboost-plot-importance-figure-size
def my_plot_importance(booster, figsize, **kwargs):
    from matplotlib import pyplot as plt
    from xgboost import plot_importance
    fig, ax = plt.subplots(1,1,figsize=figsize)
    return plot_importance(booster=booster, ax=ax, **kwargs)
```

In [ ]:
```
my_plot_importance(model, (10,10))
```

Model-6 Lasso Regression

In [ ]:
```
tuned_params = {'alpha': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000]}
```

In [ ]:
```
model = GridSearchCV(Lasso(), tuned_params, scoring = 'neg_mean_absolute_error', cv=20, n_jobs=-1)
```

In [ ]:
```
model.fit(X_train, y_train)
```

In [ ]:
```
model.best_estimator_
```

In [ ]:
```
## Predict Train results
y_train_pred = model.predict(X_train)
```

In [ ]:
```
## Predict Test results
y_pred = model.predict(X_test)
```

In [ ]:
```
print("Train Results for Lasso Regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_train.values, y_train_pred)))
print("R-squared: ", rs(y_train.values, y_train_pred))
print("Mean Absolute Error: ", mae(y_train.values, y_train_pred))
```

In [ ]:
```
print("Test Results for Lasso Regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_test, y_pred)))
print("R-squared: ", rs(y_test, y_pred))
print("Mean Absolute Error: ", mae(y_test, y_pred))
```

Feature Importance

In [ ]:
```
## Building the model again with the best hyperparameters
model = Lasso(alpha=1000)
model.fit(X_train, y_train)
```

In [ ]:
```
indices = np.argsort(-abs(model.coef_))
print("The features in order of importance are:")
print(50*'-')
for feature in X.columns[indices]:
    print(feature)
```

Model-7 Descision Tree Regression

In [ ]:
```
tuned_params = {'min_samples_split': [2, 3, 4, 5, 7], 'min_samples_leaf': [1, 2, 3, 4, 6], 'max_depth': [2, 3
, 4, 5, 6, 7]}
```

In [ ]:
```
model = RandomizedSearchCV(DecisionTreeRegressor(), tuned_params, n_iter=20, scoring = 'neg_mean_absolute_err
or', cv=10, n_jobs=-1)
```

In [ ]:
```
model.fit(X_train, y_train)
```

In [ ]:
```
model.best_estimator_
```

In [ ]:
```
## Predict Train results
y_train_pred = model.predict(X_train)
```

In [ ]:
```
## Predict Test results
y_pred = model.predict(X_test)
```

In [ ]:
```
print("Train Results for Decision Tree Regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_train.values, y_train_pred)))
print("R-squared: ", rs(y_train.values, y_train_pred))
print("Mean Absolute Error: ", mae(y_train.values, y_train_pred))
```

In [ ]:
```
print("Test Results for Decision Tree Regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_test, y_pred)))
print("R-squared: ", rs(y_test, y_pred))
print("Mean Absolute Error: ", mae(y_test, y_pred))
```

Model-8 KN Regression

In [ ]:
```
# creating odd list of K for KNN
neighbors = list(range(1,50,2))
# empty list that will hold cv scores
cv_scores = []
```

In [ ]:
```
# perform 10-fold cross validation
for k in neighbors:
    knn = KNeighborsRegressor(n_neighbors=k)
    scores = cross_val_score(knn, X_train, y_train, cv=10, scoring='neg_mean_absolute_error')
    cv_scores.append(scores.mean())
```

In [ ]:
```
# changing to misclassification error
MSE = [1 - x for x in cv_scores]
```

In [ ]:
```
# determining best k
optimal_k = neighbors[MSE.index(min(MSE))]
print('\nThe optimal number of neighbors is %d.' % optimal_k)
```

In [ ]:
```
model = KNeighborsRegressor(n_neighbors = optimal_k)
```

In [ ]:
```
model.fit(X_train, y_train)
```

In [ ]:
```
## Predict Train results
y_train_pred = model.predict(X_train)
```

In [ ]:
```
## Predict Test results
y_pred = model.predict(X_test)
```

In [ ]:
```
print("Train Results for KN Regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_train.values, y_train_pred)))
print("R-squared: ", rs(y_train.values, y_train_pred))
print("Mean Absolute Error: ", mae(y_train.values, y_train_pred))
```

In [ ]:
```
print("Test Results for KN Regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_test, y_pred)))
print("R-squared: ", rs(y_test, y_pred))
print("Mean Absolute Error: ", mae(y_test, y_pred))
```

In [ ]:
```
# Save XGBoost model to disk
```

In [ ]:
```
win_model = XGBRegressor(max_depth=2,learning_rate=0.05,n_estimators=400, reg_lambda=0.001)
win_model.fit(X_train, y_train)

win_model.save_model('0001.model')

win_model.dump_model('dump.raw.txt') # dump model
win_model.dump_model('dump.raw.txt','featmap.txt')# dump model with feature map
```

Compare these models

In [ ]:
```
#I saved the winning model to disk
```

In [77]:
```
# After model 2, my code dosenot run.It stuck before model 3 .After that models doesn't run.
# I just have two models result

# Model                 RMSE        RS          MAE
# 1-Baseline            4979        -6.53       4047
# 2-Ridge               4625.91     0.1368      3543.60
# 3-Support Vector
# 4-Random Forest
# 5-XGBoost
# 6-Lasso
# 7-Decision Tree
# 8-KN
```

Result: By Comparing the 1st two models

The best model to predict the purchase,

Based on the Lowest RMSE and MAE - with RS closest to 1 is:

Ridge with RS closest to 1 and lowest values for both RMSE and MAE

In [ ]:

```
```