PREDICTION OF BAD PURCHASE IN AUTOMOTIVE AUCTION

Post Graduate Program in Data Science Engineering

Location: Bangalore
Batch: PGP DSE-FT JAN 22

Submitted by

CHARAKANI SIVA PRASAD

ANOOP PRASAD

AKSHAY MP

S GAREEB BASHA

JEEVAN JACOB

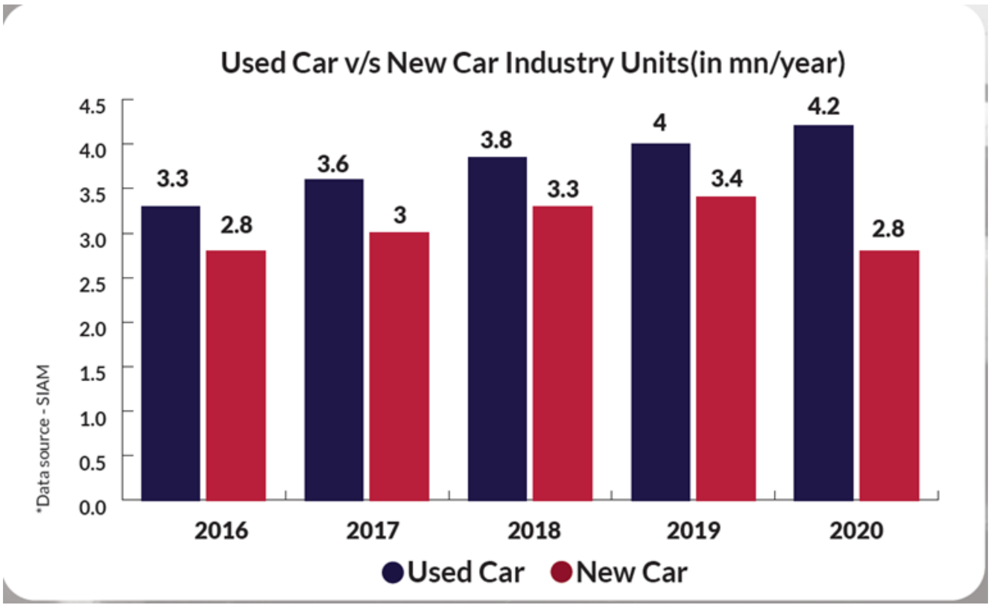
Mentored by

MR. ANIMESH TIWARI

Table of Contents

Introduction to the problem:

Buying and selling used cars is a common practice all around the world. The purchase of a used vehicle has its advantages like the cost being comparatively lower to that of a new car.



Auto dealerships purchase many of their used cars through auto auctions with the identical goals that you have: they want to buy as many cars as they can in the best condition possible. The problem that these dealerships often face is the risk of buying used cars that have serious issues, preventing them from being sold to customers. These bad purchases are called "kicks", and that they can be hard to spot for a variety of reasons. Many kicked cars are purchased due to tampered odometers or mechanical issues that could not be predicted ahead of time. For these reasons, car dealerships can benefit greatly from the predictive powers of machine learning. If there is a way to determine if a car would be kicked a priori, car dealerships can not only save themselves money, but also provide their customers with the best inventory selection possible.

Problem Statement:

Purchasing a second hand vehicle has advantages like lower price than a comparable new car, lower continuing ownership expenses like collision insurance and taxes, and mostly a used vehicle has already taken its biggest depreciation hit. Seemingly functional used cars that end up having no utility value, “lemons”, pose a big risk to auto dealerships because they may be very difficult to detect at an auction. Given the high stakes involved for auto dealerships, they have to ensure every car they purchase at an auction is not a lemon and will be sold to a customer. It would be extremely useful to find a better way to predict whether a car is a lemon or not at time of the auction.

The target variable is “IS-Bad-Buy”, expressed by a probability of being a lemon. This problem particularly needs to be wary of the high cost of a false negative, falsely predicting that a lemon has a higher probability of being a good buy.

Impact in business of your problem/Need for this study(Executive summary):

1. **High Inventory Costs**: Auto dealership buying the car thinking it would be sellable, incurring transportation/repair costs, and then realizing it is left with a defective car and unsellable inventory.
2. Greater predictability will reduce the likelihood of bad, costly purchases. Hence, the goal of this project is to predict if a car purchased at an Auction is a lemon (bad buy). So, this is a binary classification problem. We develop predictive models that can predict beforehand whether a given vehicle in an auction is good buy or not so that the buyers can avoid the bad ones. There is also an opportunity cost associated with a false positive, in this case falsely predicting that a good car has a higher likelihood of being a lemon. In this case the dealership would refrain from purchasing a car that would have otherwise generated profit for the company after being successfully sold to a customer.

Literature Review:

The used vehicle department is often viewed as a risky department by the average dealer. This is because the fundamentals of the used vehicle department are very different from the new vehicle department. Further, the dynamics have changed, with customers possessing higher bargaining power and knowledge than ever before. [1]

Pablo A. Muñoz Gallego Eva Lahuerta Otero [2]:
has revealed some very interesting data about the sector. Among buyers, 66.4% turn to the second-hand car market as a first option and consider price to be its main advantage (87.5%), followed by the guarantee (4.61%). 62.7% of buyers in this market acquired the vehicle they were initially looking for, although 14.7% of respondents still maintain that one cannot be sure of the condition the vehicle is in when buying it.

Andrews and Benzing [3] analyzed how auction, seller and product factors influence the price premium in an eBay used car auction market. For auctions that resulted in a sale, cars with clear title and dealers were able to secure significantly greater price premiums. Using a binary legit model, the study revealed that cars had a greater probability of selling if the seller had a better reputation.

Barris Mike [4] his article supports the prediction that cars produced by Japanese manufacturers will have higher perceived quality along with slower depreciation rates and thus higher resale value.

Dataset Information:

This data describes Auction of cars in USA. This particular dataset has 72983 records and 34 variables out of which 33 are dependent and 1 is an independent Target Variable of classification type. This dataset comprehends vehicles that are auctioned in the years 2009 and 2010 by, including the vehicles that are Good purchases along with those that are Bad Purchases.

Variable identification:

Independent Variables: *There are 33 independent variables are listed below.*

1. RefID	19. MMRAcquisitionAuctionCleanPrice
2. PurchDate	20. MMRAcquisitionRetailAveragePrice
3. Auction	21. MMRAcquisitonRetailCleanPrice
4. VehYear	22. MMRCurrentAuctionAveragePrice
5. VehicleAge	23. MMRCurrentAuctionCleanPrice
6. Make	24. MMRCurrentRetailAveragePrice
7. Model	25. MMRCurrentRetailCleanPrice
8. Trim	26. PRIMEUNIT
9. SubModel	27. AUCGUART
10. Color	28. BYRNO
11. Transmission	29. VNZIP
12. WheelTypeID	30. VNST
13. WheelType	31. VehBCost
14. VehOdo	32. IsOnlineSale
15. Nationality	33. WarrantyCost
16. Size	
17. TopThreeAmericanName	
18. MMRAcquisitionAuctionAverage Price	

Target Variable:

1. IsBadBuy

Variable Categorization with Description:

Numerical Variables from the Dataset:

Sr No.	Variable	DataType	Description
1	RefId	Int64	Unique (sequential) number assigned to vehicles
2	IsBadBuy	Int64	Identifies if the kicked vehicle was an avoidable purchase
3	VehYear	Int64	The manufacturer's year of the vehicle
4	VehicleAge	Int64	The Years elapsed since the manufacturer's year
5	WheelTypeID	Int64	The type id of the vehicle wheel
6	VehOdo	Int64	The vehicles odometer reading
7	MMRAcquisitionAuctionAveragePrice	Float64	Acquisition price for this vehicle in average condition at time of purchase
8	MMRAcquisitionAuctionCleanPrice	Float64	Acquisition price for this vehicle in the above Average condition at time of purchase
9	MMRAcquisitionRetailAveragePrice	Float64	Acquisition price for this vehicle in the retail market in average condition at time of purchase
10	MMRAcquisitonRetailCleanPrice	Float64	Acquisition price for this vehicle in the retail market in above average condition at time of purchase
11	MMRCurrentAuctionAveragePrice	Float64	Acquisition price for

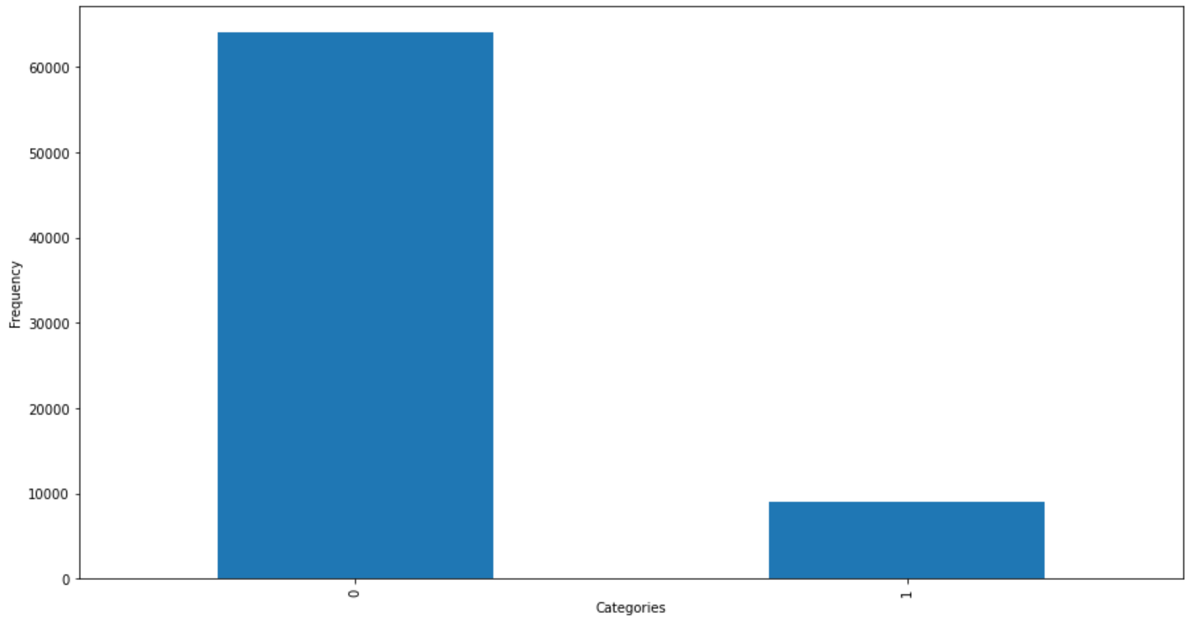
			this vehicle in average condition as of current day
12	MMRCurrentAuctionCleanPrice	Float64	Acquisition price for this vehicle in the above condition as of current day
13	MMRCurrentRetailAveragePrice	Float64	Acquisition price for this vehicle in the retail market in average condition as of current day
14	MMRCurrentRetailCleanPrice	Float64	Acquisition price for this vehicle in the retail market in above average condition as of current day
15	BYRNO	Int64	Unique number assigned to the buyer that purchased the vehicle
16	VINZIP1	Int64	Zip code where the car was purchased
17	VehBCost	Float64	Acquisition cost paid for the vehicle at time of purchase
18	IsOnlineSale	Int64	Identifies if the vehicle was originally purchased online
19	WarrantyCost	Int64	Warranty price (term=36month and millage=36K)

Categorical Variables from the Dataset:

Sr No.	Variable	DataType	Description
1	PurchDate	Object	The Date the vehicle was Purchased at Auction
2	Auction	Object	Auction provider at which the vehicle was purchased
3	Make	Object	Vehicle Manufacturer
4	Model	Object	Vehicle Model
5	Trim	Object	Vehicle Trim Level
6	SubModel	Object	Vehicle Submodel
7	color	Object	Vehicle Color
8	Transmission	Object	Vehicles transmission type (Automatic, Manual)
9	WheelType	Object	The vehicle wheel type description (Alloy, Covers)
10	Nationality	Object	The Manufacturer's country
11	Size	Object	The size category of the vehicle (Compact, SUV, etc.)
12	TopThreeAmericanName	Object	Identifies if the manufacturer is one of the top three American manufacturers
13	PRIMEUNIT	Object	Identifies if the vehicle would have a higher demand than a standard purchase
14	AUCGUART	Object	The level guarantee provided by auction for the vehicle (Green light - Guaranteed/arbitratable, Yellow Light - caution/issue, red light - sold as is)
15	VNST	Object	State where the car was purchased

Target Variable:

The target variable of the above dataset is IsBadBuy. We have to predict whether a vehicle purchased at auction is a good buy or not.



In the above dataset, 87.7% of the purchases are good purchases and 12.20% of the purchases are bad purchases. We observe that there is there is presence of high amount of class imbalance.

DATA PRE-PROCESSING:

Data pre-processing is a process of preparing the raw data and making it suitable for a machine learning model. It is the first and crucial step while creating a machine learning model. When creating a machine learning project, it is not always a case that we come across the clean and formatted data. And while doing any operation with data, it is mandatory to clean it and put in a formatted way. So for this, we use data pre-processing task.

A real-world data generally contains noises, missing values, and maybe in an unusable format which cannot be directly used for machine learning models. Data pre-processing is required tasks for cleaning the data and making it suitable for a machine learning model which also increases the accuracy and efficiency of a machine learning model.

The data consists of 72983 rows and 34 columns. Out of these we have 15 categorical columns and the rest as numerical.

Datatype Verification:

From the data description it is understood that few variables are labeled wrong, so we have to update the following datatypes.

Variable	Provided DataType	Actual DataType
IsBadBuy	Int64	Object
PurchDate	Object	DateTime
WheelTypeID	Float64	Object
BYRNO	Int64	Object
VNZIP1	Int64	Object
IsOnlineSale	Int64	Object

Missing Value Treatment:

The next step of data pre-processing is to handle missing data in the datasets. If our dataset contains some missing data, then it may create a huge problem for our machine learning model. Hence it is necessary to handle missing values present in the dataset.

	Variable	Percentage_error
0	AUCGUART	95.315347
1	PRIMEUNIT	95.315347
2	WheelType	4.348958
3	WheelTypeID	4.342107
4	Trim	3.233630
5	MMRCurrentAuctionAveragePrice	0.431607
6	MMRCurrentAuctionCleanPrice	0.431607
7	MMRCurrentRetailAveragePrice	0.431607
8	MMRCurrentRetailCleanPrice	0.431607
9	MMRAcquisitionAuctionCleanPrice	0.024663
10	MMRAcquisitionRetailAveragePrice	0.024663
11	MMRAcquisitonRetailCleanPrice	0.024663
12	MMRAcquisitionAuctionAveragePrice	0.024663
13	Transmission	0.012332
14	SubModel	0.010961
15	Color	0.010961
16	Nationality	0.006851
17	Size	0.006851
18	TopThreeAmericanName	0.006851

From the above analysis we can observe that AUCGUART and PRIMEUNIT has more than 95% of missing values so these can be dropped.
WheelType and Trim has roughly 4% of missing vales so they can be imputed with median if they belong to numerical data type or with mode if they belong to the object data type.

Feature Engineering:

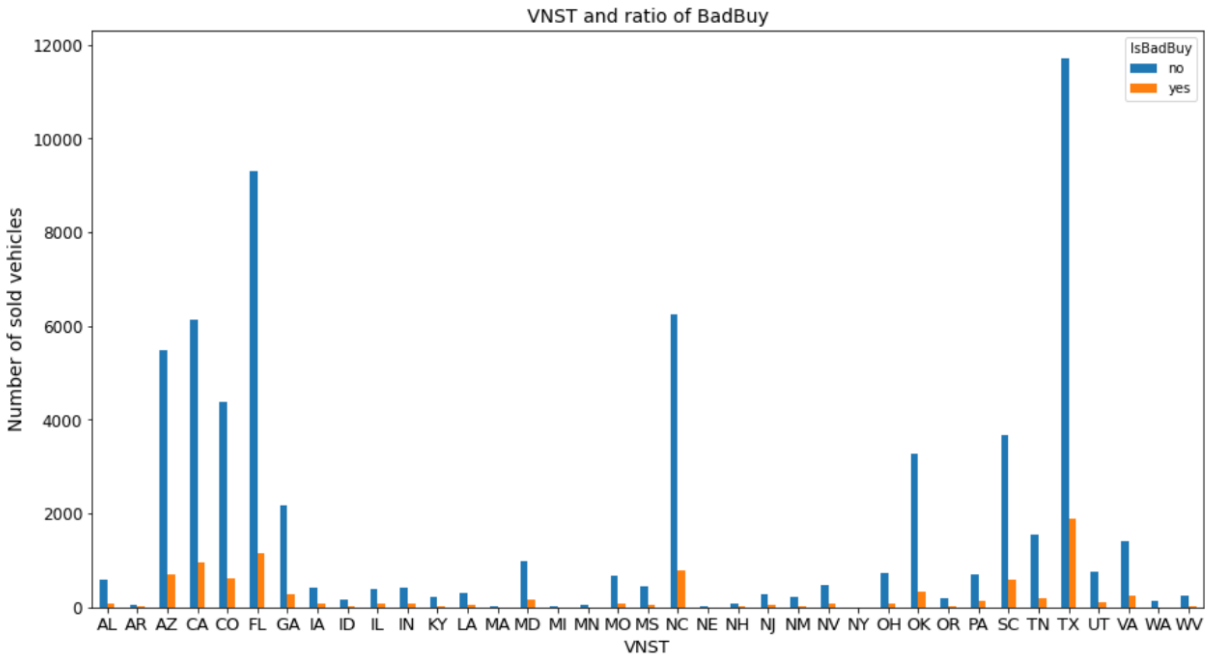
After finding out the unique values and count of unique values in each categorical variable it has been observed that few variables are divided into lot of categories. The detailed data is attached below.

Variable	**Unique Categorical Values**
Auction	3
Make	33
Model	1063
Trim	135
color	17
Transmission	4
WheelTypeID	5
WheelType	4
Nationality	5
Size	13
TopThreeAmericanName	5
PRIMEUNIT	3
AUCGUART	3
BYRNO	74
VNZIP1	153
VNST	37
IsOnlineSale	2
SubModel	1063

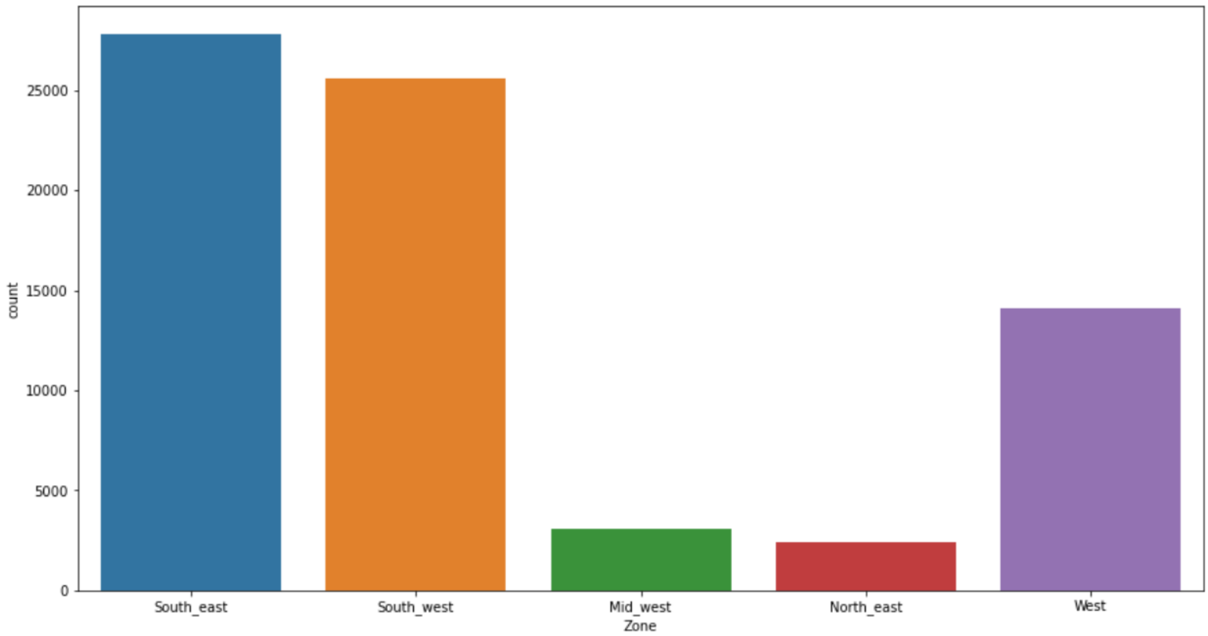
Inferences:

It can be observed from the above that the Variables Count Highlighted with Red Color has a Lot of unique values to work with by doing OneHotEncoding.

But VNST which has 37 different state codes can be modified to 5 different Zones in USA by merging all the states into 5different zones of USA.This is the data distribution before dividing to zones.



Midwest: ['IL','IN','IA','KS','MI','MN','MO','NE','ND','OH','SD','WI']
Northeast: ['CT','DE','ME','MD','MA','NH','NJ','NY','PA','RI','VT']
Southeast: ['AL','AR','FL','GA','KY','LA','MS','NC','SC','TN']
Southwest: ['AZ','NM','OK','TX','VA','WV']
West: ['AK','CA','CO','HI','ID','MT','NV','OR','UT','WA','WY']



This above picture represents Data after dividing to zones.

The above division has been made using the below attached diagram which has been obtained from US postal data.

Check for Outliers:

Data has outliers present in each of the numerical columns. For making the base model, we do not perform any outlier treatment and retain all the rows present in the data.

EXPLORATORY DATA ANALYSIS

Univariate Analysis:

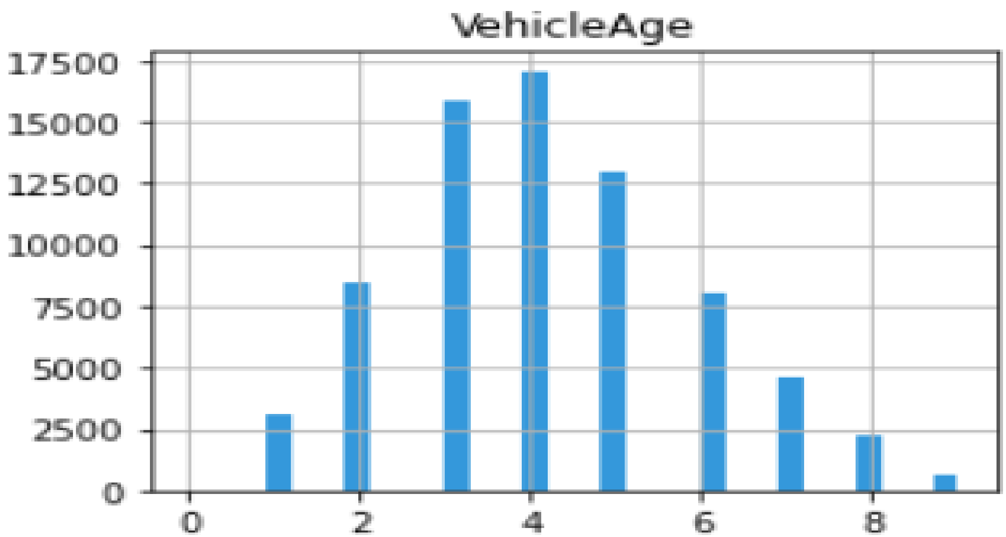
For Numerical Variables: - We plot the distribution curve to study the variation of the numerical data.

The skewness of the numerical variables is attached below for reference and it can be observed that VehBCost and WarrantyCost have high skewness rest all the data is almost similar to the normal distribution.

```
1 df_m.select_dtypes(np.number).skew()
```

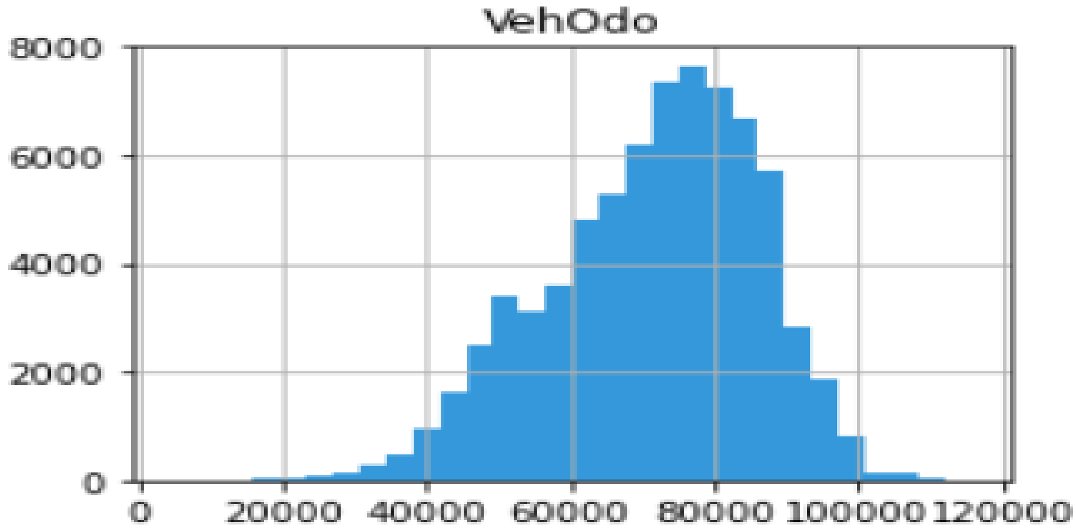
```
VehicleAge                              0.393616
VehOdo                                 -0.453145
MMRAcquisitionAuctionAveragePrice       0.463707
MMRAcquisitionAuctionCleanPrice         0.466577
MMRAcquisitionRetailAveragePrice        0.209252
MMRAcquisitonRetailCleanPrice           0.176335
MMRCurrentAuctionAveragePrice           0.524085
MMRCurrentAuctionCleanPrice             0.537056
MMRCurrentRetailAveragePrice            0.201988
MMRCurrentRetailCleanPrice              0.195368
VehBCost                                0.715931
WarrantyCost                            2.070831
dtype: float64
```

1. Vehicle Age:



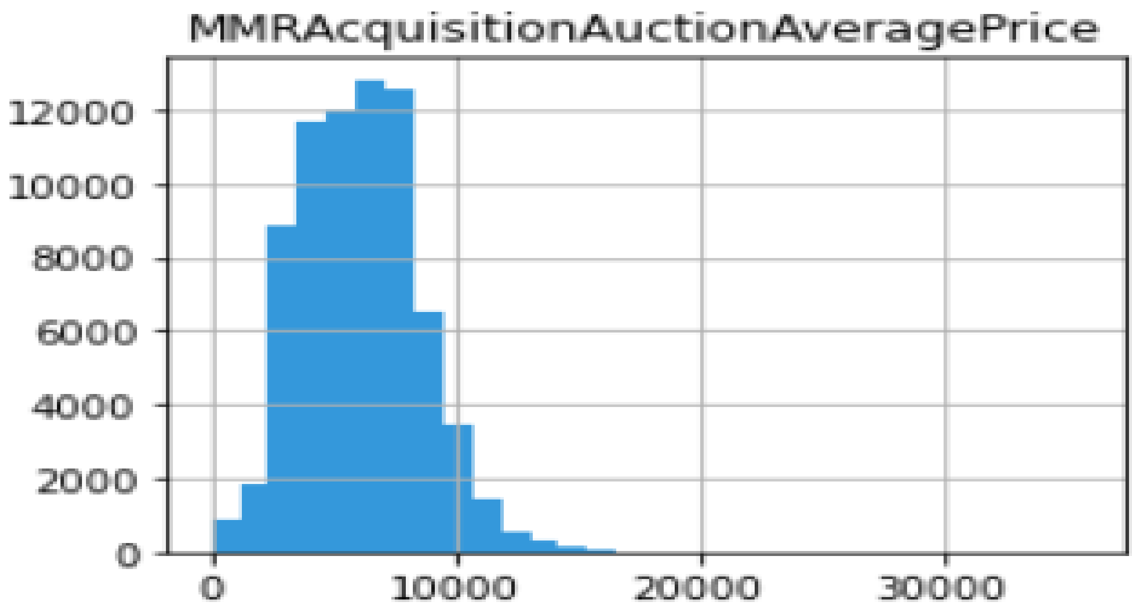
- It can be observed that VehicleAge here follows almost Normal distribution.
- We can also see that from 3 till 7 years' age of vehicle has higher no of Bad purchases

2. VehicleOdo



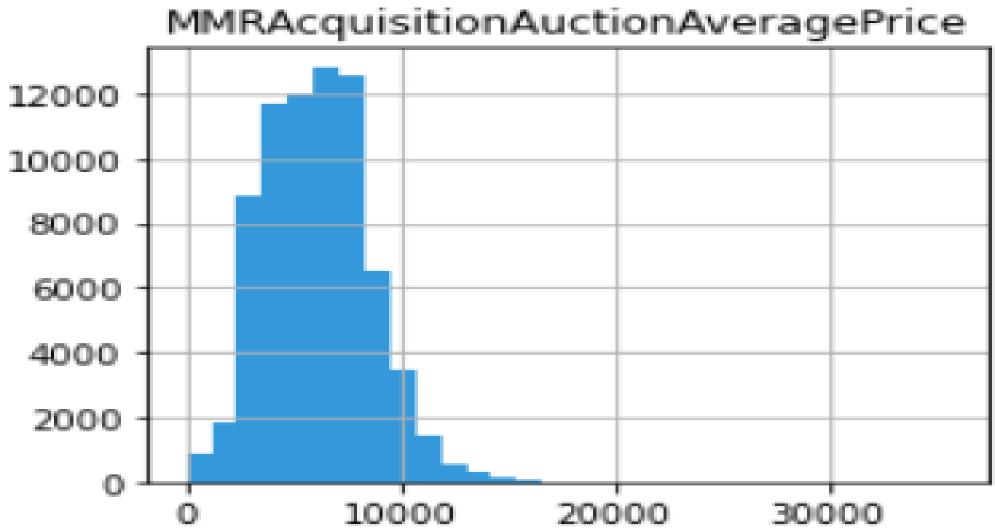
- It is observed that the VehicleOdo is a left skewed Data with much of the data falling under 40000 to 80000 range.

3. MMRAcquisitionAuctionAveragePrice



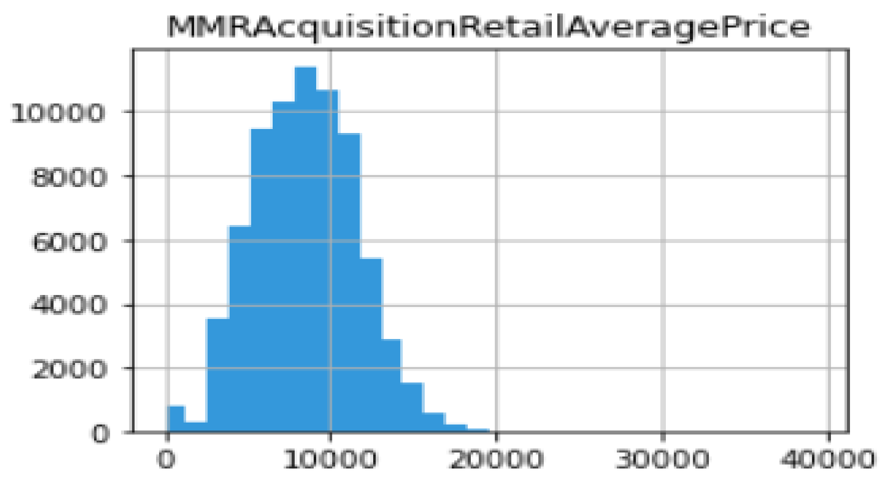
It is observed that the MMRAcquisitionAuctionAveragePrice is a little right skewed Data.

4. MMRAcquisitionAuctionCleanPrice



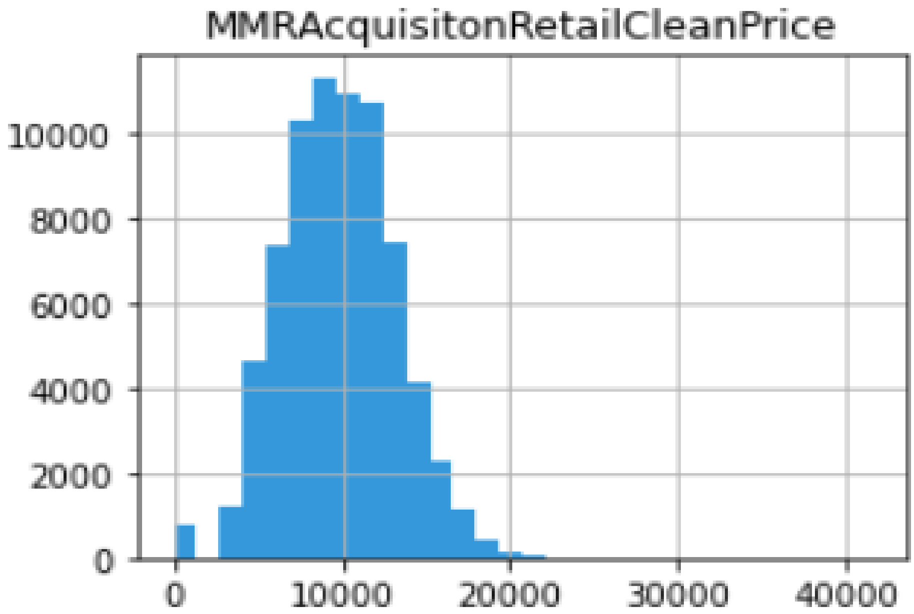
It is observed that the MMRAcquisitionAuctionAveragePrice is a little right skewed Data.

5. MMRAcquisitionRetailAveragePrice



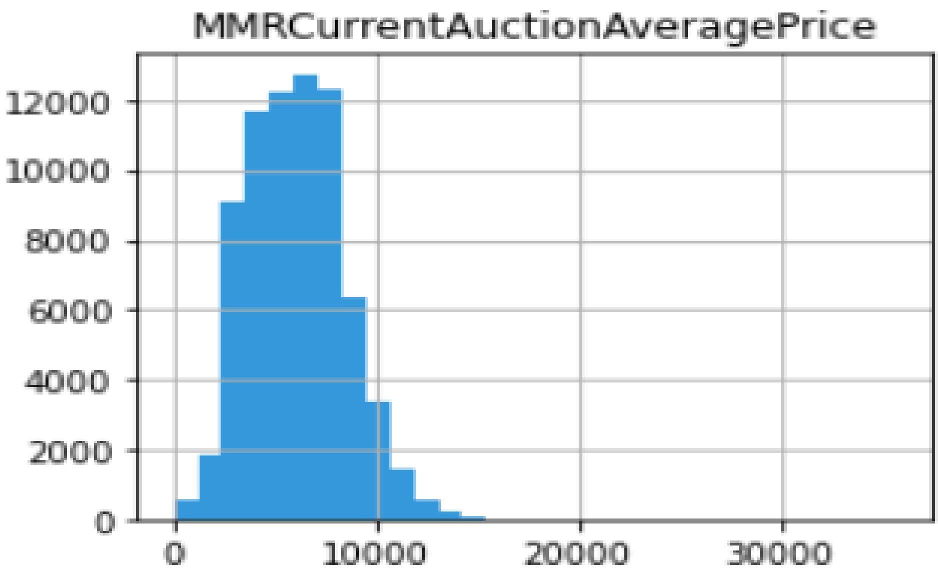
It is observed that the MMRAcquisitionrRetailAveragePrice is a little right skewed Data.

6. MMRAcquisitonRetailCleanPrice



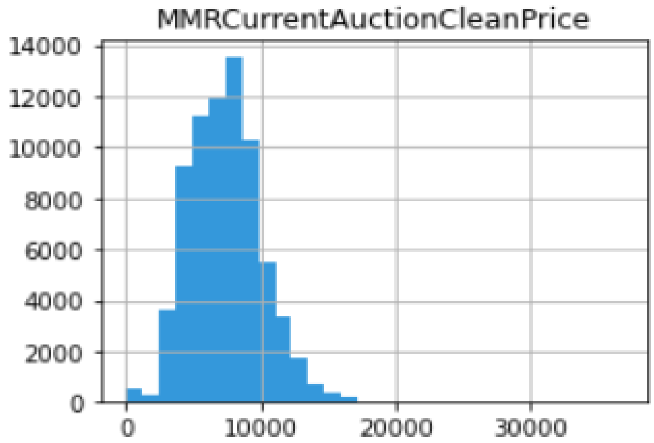
It is observed that the MMRAcquisitionRetailCleanPrice is a normally distributed pattern.

7. MMRCurrentAuctionAveragePrice

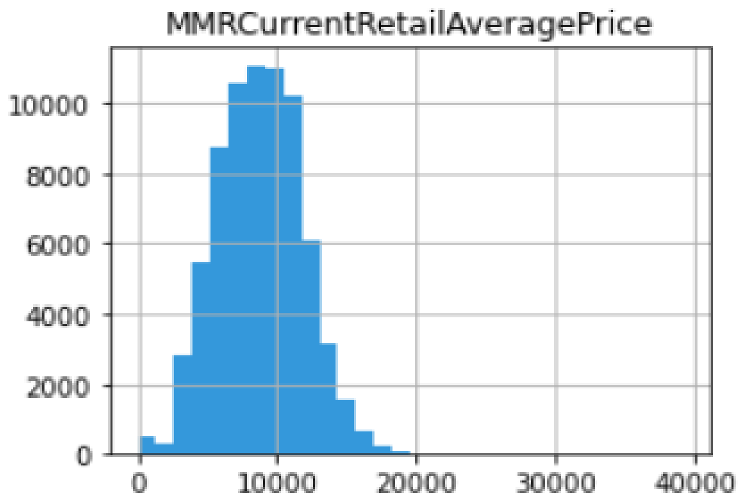


It is observed that the MMRAcquisitionAveragePrice is almost normally distributed pattern.

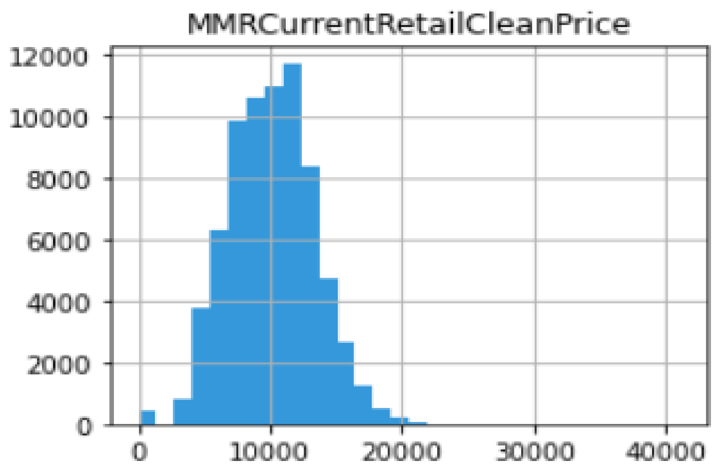
8. MMRCurrentAuctionCleanPrice



9. MMRCurrentRetailAveragePrice

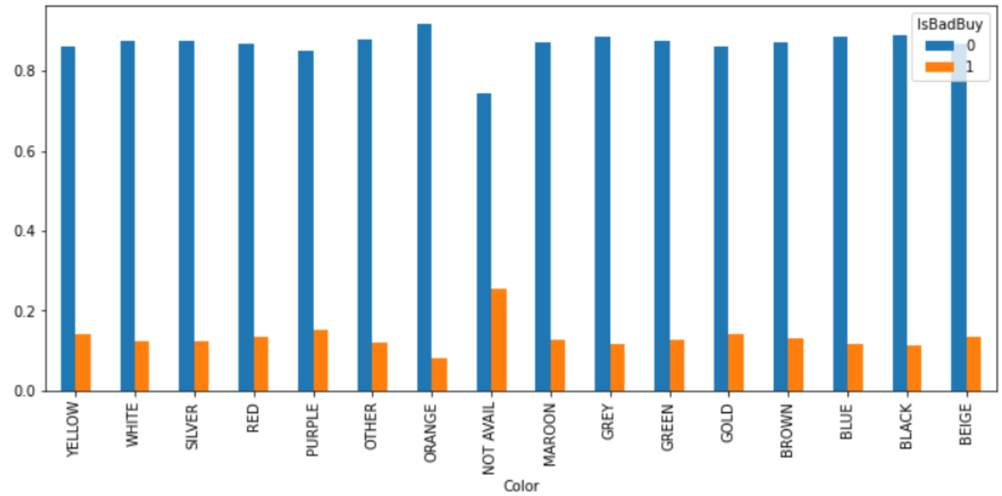


10. MMRCurrentRetailCleanPrice



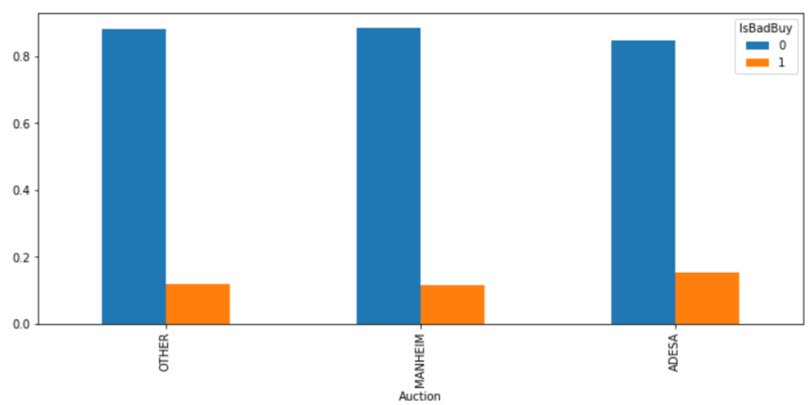
Bi-Variate Analysis:

1. IsBadBuy vs Color:



From the above bar chart it can be observed that almost all the colors have equal percentage of bad buys except for yellow and orange.

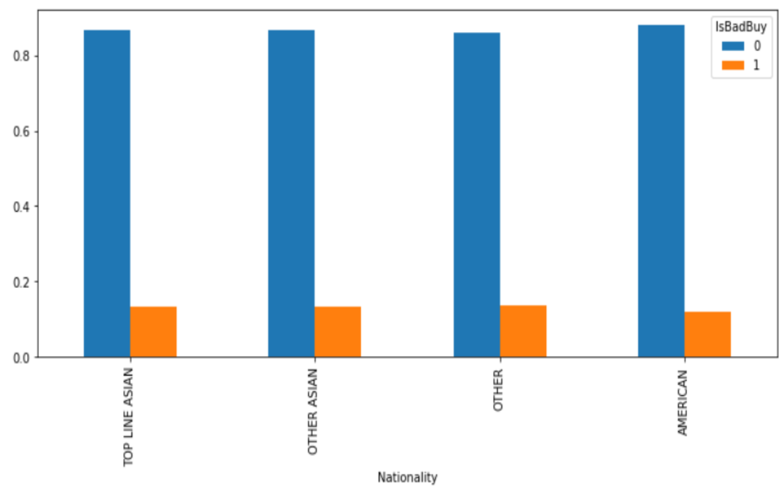
2. IsBadBuy vs Auction:



IsBadBuy	0	1
Auction		
OTHER	0.881835	0.118165
MANHEIM	0.885120	0.114880
ADESA	0.848120	0.151880

From the above bar chart it can be observed that ADESA have higher percentage of bad buys compared with the other two.

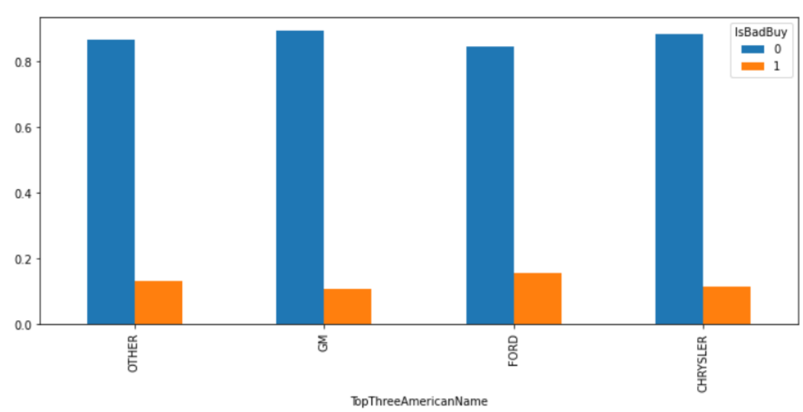
3. IsBadBuy vs Nationality:



IsBadBuy	0	1
Nationality		
TOP LINE ASIAN	0.865395	0.134605
OTHER ASIAN	0.867920	0.132080
OTHER	0.861538	0.138462
AMERICAN	0.878957	0.121043

From the above bar chart it can be observed that all the Nationality data have equal percentage of bad buys.

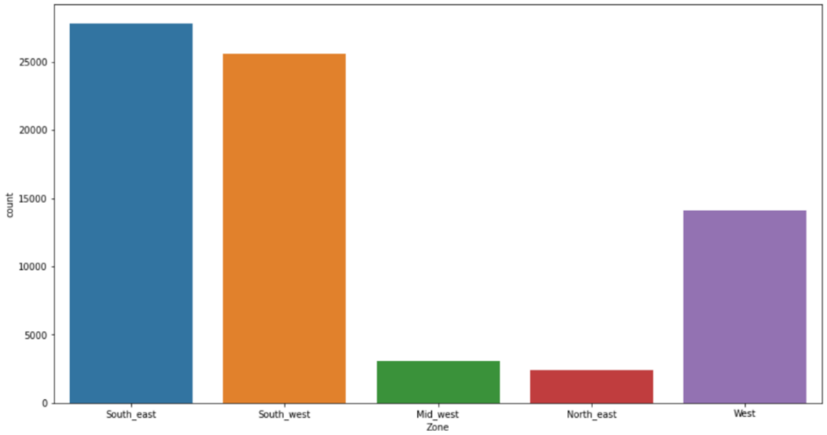
4. IsBadBuy vs TopThreeAmericanName:



IsBadBuy	0	1
TopThreeAmericanName		
OTHER	0.867029	0.132971
GM	0.891918	0.108082
FORD	0.843605	0.156395
CHRYSLER	0.883542	0.116458

It can be observed from the data that ford manufacturer has more badbuys compared to the other three manufacturers.

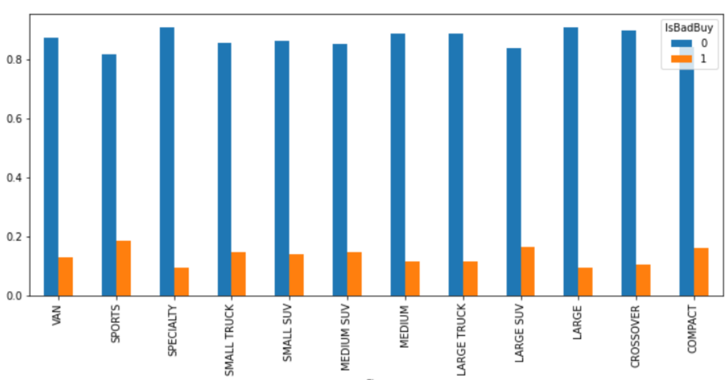
5. IsBadBuy vs Zone:



IsBadBuy	0	1
Zone		
West	0.872735	0.127265
South_west	0.873498	0.126502
South_east	0.883948	0.116052
North_east	0.851230	0.148770
Mid_west	0.883473	0.116527

From the above Bar Graph, it can be observed that the Zones Southwest and Southeast has more number of observations but percentage wise more defective parts are in Northeast zone.

6. IsBadBuy vs Size:



IsBadBuy	0	1
Size		
VAN	0.872566	0.127434
SPORTS	0.814672	0.185328
SPECIALTY	0.908094	0.091906
SMALL TRUCK	0.855324	0.144676
SMALL SUV	0.862478	0.137522
MEDIUM SUV	0.852534	0.147466
MEDIUM	0.884976	0.115024
LARGE TRUCK	0.886435	0.113565
LARGE SUV	0.838102	0.161898
LARGE	0.907571	0.092429
CROSSOVER	0.895964	0.104036
COMPACT	0.841083	0.158917

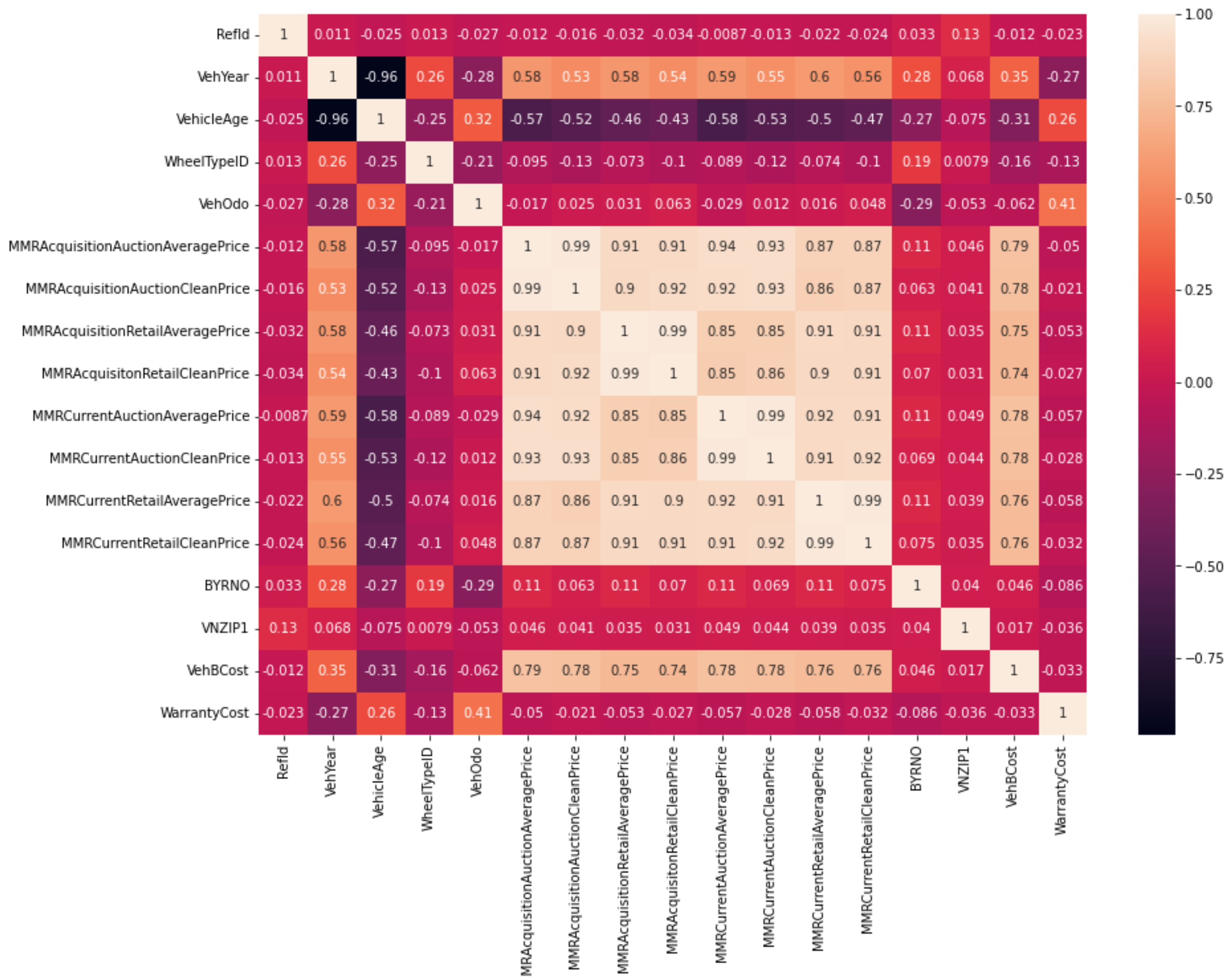
From the above data it can be observed that Sports, Large SUV and Compact have more percentage of defect vehicles compared to the other Size. Rest all the sizes come under range of 10-15%.

<u>**Correlation Matrix:**</u>

Heat-Map - Pearson Correlation Matrix

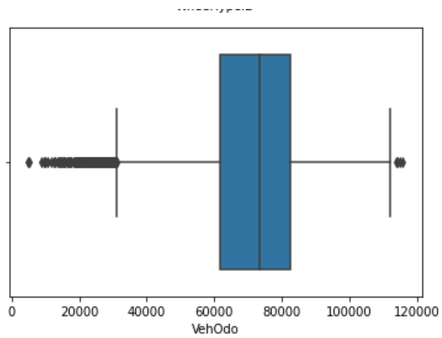
(Assumption: For the Pearson correlation, both variables should be normally distributed. Other assumptions include linearity and homoscedasticity)

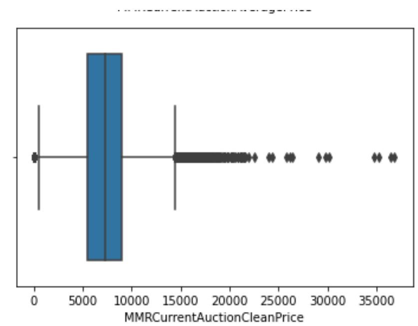
It gives a measure of how much two numeric variables are linearly correlated. It tries to obtain a best fit line between two numeric variables and how close the points are to a fitted line.

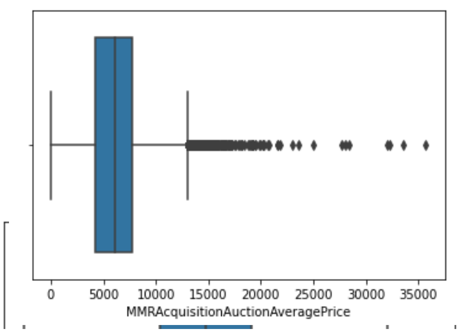


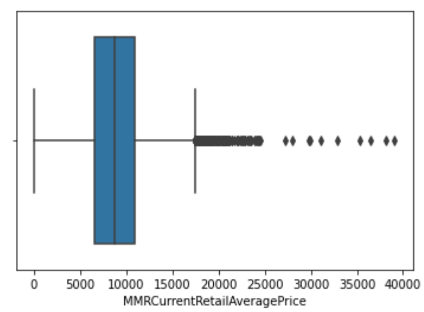
From the above Heat-Map it can be observed that there is very high correlation between MMRA – Values among themselves also they are highly correlated with VehBCost.

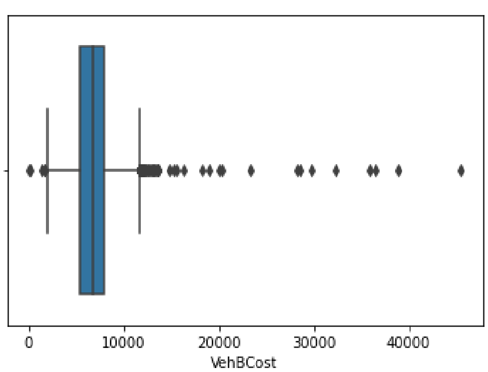
<u>Outliers:</u>

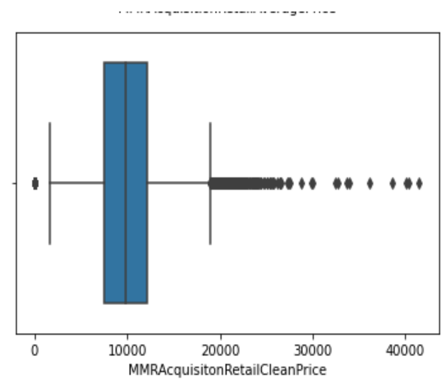


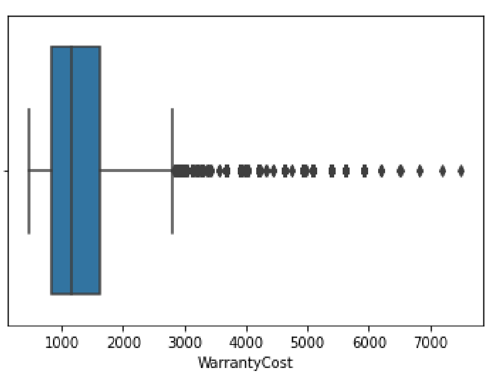


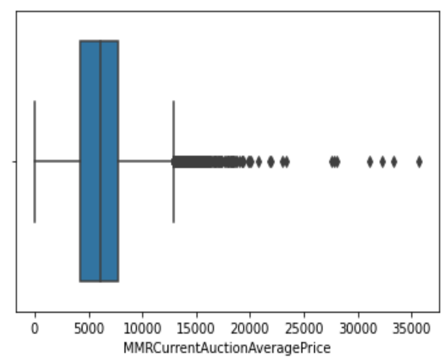


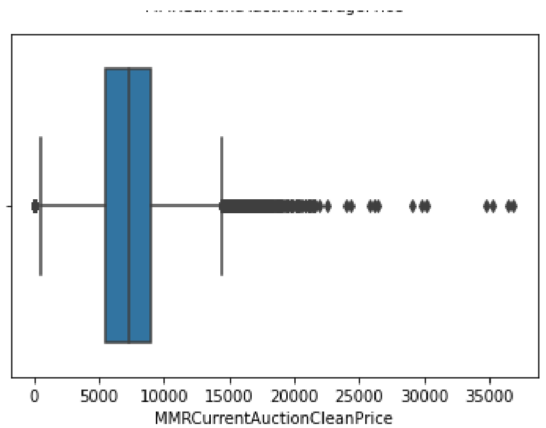


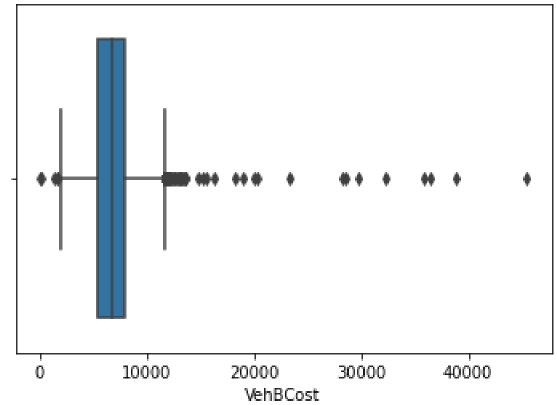


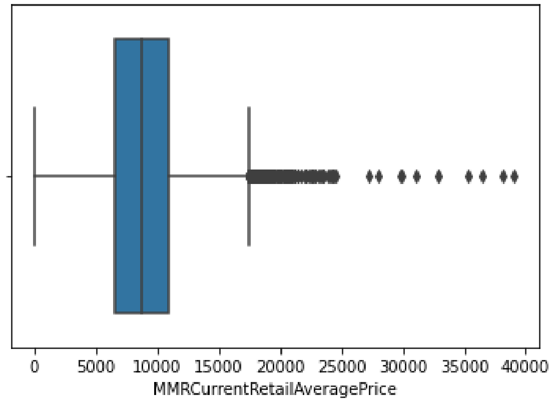


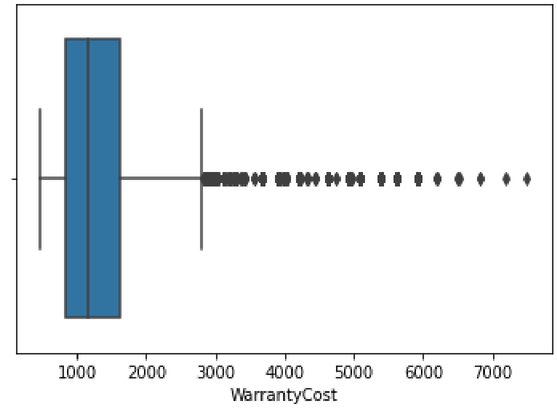












Outlier treatment has been done on the above columns but by doing so we are losing the data of “BadBuys” which is out target Category so for better model learning we choose to keep the outliers in the model.

STATISTICAL TESTS

Shapiro-Wilk test:

We perform Shapiro test to check if the numerical features are normally distributed or not.
Hypothesis for Shapiro Test
H0: Data is normally distributed
H1: Data is not normally distributed

```python
from scipy.stats import shapiro
for i in Num_feat.columns:
  print('Stat and p-value for',i,'is',shapiro(Num_feat[i]))
```

```
Stat and p-value for VehicleAge is ShapiroResult(statistic=0.9563164114952087, pvalue=0.0)
Stat and p-value for VehOdo is ShapiroResult(statistic=0.9820109605789185, pvalue=0.0)
Stat and p-value for MMRAcquisitionAuctionAveragePrice is ShapiroResult(statistic=0.9839088320732117, pvalue=0.0)
Stat and p-value for MMRAcquisitionAuctionCleanPrice is ShapiroResult(statistic=0.9833253622055054, pvalue=0.0)
Stat and p-value for MMRAcquisitionRetailAveragePrice is ShapiroResult(statistic=0.9932205080986023, pvalue=0.0)
Stat and p-value for MMRAcquisitonRetailCleanPrice is ShapiroResult(statistic=0.9920430779457092, pvalue=0.0)
Stat and p-value for MMRCurrentAuctionAveragePrice is ShapiroResult(statistic=0.982910692214966, pvalue=0.0)
Stat and p-value for MMRCurrentAuctionCleanPrice is ShapiroResult(statistic=0.981952190391699, pvalue=0.0)
Stat and p-value for MMRCurrentRetailAveragePrice is ShapiroResult(statistic=0.9939961433410645, pvalue=2.382207389352189e-44)
Stat and p-value for MMRCurrentRetailCleanPrice is ShapiroResult(statistic=0.9931825399398804, pvalue=0.0)
Stat and p-value for VehBCost is ShapiroResult(statistic=0.9751319289207458, pvalue=0.0)
Stat and p-value for WarrantyCost is ShapiroResult(statistic=0.8638250231742859, pvalue=0.0)
```

Since p-value is less than 0.05 for all the independent numerical variables, we reject the null hypothesis. Hence the data is not normally distributed and we perform non parametric tests.

As the data is not Normally distributed Anova cannot be performed so a Non – Parametric test Mann Whitney U test is being performed to get the Significant variables for further model building.

Mann Whitney U test:

This is a nonparametric test of the null hypothesis that, for randomly selected values X and Y from two populations, the probability of X being greater than Y is equal to the probability of Y being greater than X.

Hypothesis of Mann-Whitney U Test
H0: Two samples have the same mean (insignificant)
H1: Two samples have different mean (significant)

```
import scipy.stats as stats
Utest_results = []
for i in Num_col.columns:
  u_value,p = stats.mannwhitneyu(x=Cat_col['IsBadBuy_1'], y=Num_col[i], alternative = 'two-sided')
  Utest_results.append([i, u_value, p])

columns = ['feature', 'Utest-statistic', 'p-value']
Utest_df = pd.DataFrame(Utest_results, columns=columns)
Utest_df = Utest_df.sort_values('p-value').set_index('feature')
Utest_df
```

feature	Utest-statistic	p-value
VehicleAge	13967831.0	0.0
VehOdo	0.0	0.0
MMRAcquisitionAuctionAveragePrice	33931026.0	0.0
MMRAcquisitionAuctionCleanPrice	29150639.5	0.0
MMRAcquisitionRetailAveragePrice	33931026.0	0.0
MMRAcquisitonRetailCleanPrice	33931026.0	0.0
MMRCurrentAuctionAveragePrice	20653668.0	0.0
MMRCurrentAuctionCleanPrice	16055739.0	0.0
MMRCurrentRetailAveragePrice	20653668.0	0.0
MMRCurrentRetailCleanPrice	20653668.0	0.0
VehBCost	4488.0	0.0
WarrantyCost	0.0	0.0

```
[81] threshold = 0.05
     signi_Utest = Utest_df[Utest_df['p-value'] < threshold]

     print("Features with significant MannWhitneyTest p-value: {}".format(signi_Utest.shape[0]))
     print("Features with insignificant MannWhitneyTest p-value: {}".format(Utest_df.shape[0] - signi_Utest.shape[0]))
```

```
Features with significant MannWhitneyTest p-value: 12
Features with insignificant MannWhitneyTest p-value: 0
```

From the above table we will only consider the variables having p-value less than 0.05. But it can be seen that all the variables in the given data come out to be significant variables.

Chi Square Test:

Categorical columns – For categorical columns we perform chi-square test to check for the significance of the categorical column with respect to 'IsBadBuy' Column.

Hypothesis of Chi-square test

H0: Attributes are independent
H1: Attributes are dependent

```
from scipy.stats import chi2,chi2_contingency
chi_sq = pd.DataFrame(columns = ['Variable','P-Value'])
for i in df_m.select_dtypes(np.object):
    dataset_table = pd.crosstab(df_m[i],df_m['IsBadBuy'])
    observed = dataset_table.values
    val2 = stats.chi2_contingency(dataset_table)
    expected = val2[3]
    chi_square = sum([(o-e)**2./e for o,e in zip(observed,expected)])
    chi_square_statistic = chi_square[0]+chi_square[1]
    p_value = 1-chi2.cdf(x=chi_square_statistic,df=3)
    chi_sq = chi_sq.append({'Variable':i,'P-Value':p_value},ignore_index = True)
chi_sq
```

	Variable	P-Value
0	IsBadBuy	0.000000e+00
1	Auction	0.000000e+00
2	Make	0.000000e+00
3	Color	3.619327e-14
4	Transmission	7.269705e-01
5	WheelType	0.000000e+00
6	Nationality	3.829918e-03
7	Size	0.000000e+00
8	TopThreeAmericanName	0.000000e+00
9	Zone	2.027562e-07
10	IsOnlineSale	8.017878e-01

```
[83] threshold = 0.05
     signi_chi = chi_sq[chi_sq['P-Value'] < threshold]

     print("Features with significant Chi_sq p-value: {}".format(signi_chi.shape[0]))
     print("Features with insignificant Chi_sq p-value: {}".format(chi_sq.shape[0] - signi_chi.shape[0]))
```

```
Features with significant Chi_sq p-value: 9
Features with insignificant Chi_sq p-value: 2
```

```
[84] # The insignificant variables are IsSaleOnline and transmission
```

It is observed from chi square test that the variables 'IsSaleOnline' and 'Transmission' are insignificant as their p-value is greater than 0.05.

BASE MODEL:

Logistic Regression Model & KNN model:

We have select Logistic Regression and K Nearest Neighbors as our base model. For this we have encoded all the categorical variables using n-1 Dummy Encoding and have scaled the data using MinMaxScaler.

Encoding:

In [106]:
```
Cat_col = pd.get_dummies(df_m[cat],drop_first = True)
```

In [107]:
```
Cat_col
```

Out[107]:

	IsBadBuy_1	Auction_MANHEIM	Auction_OTHER	Make_BUICK	Make_CADILLAC	Make_CHEVROLET	Make_CHRYSLER	Make_DODGE	Make_FORD	M
0	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	1	0	
2	0	0	0	0	0	0	0	1	0	
3	0	0	0	0	0	0	0	1	0	
4	0	0	0	0	0	0	0	0	1	
...	...	...	...	...	...	...	...	...	...	
72978	1	0	0	0	0	0	0	0	0	
72979	0	0	0	0	0	1	0	0	0	
72980	0	0	0	0	0	0	0	0	0	
72981	0	0	0	0	0	1	0	0	0	
72982	0	0	0	0	0	0	0	0	0	

The above is the representation of Encoded categorical variables. After completing n-1 dummy encoding we are left with 76 categorical columns, which has to be decreased using the feature selection process.

MixMaxScaling:

In [109]:
```
from sklearn.preprocessing import MinMaxScaler
scaler=MinMaxScaler()
scaler.fit(Num_col)
Num_scaled=scaler.transform(Num_col)
```

MinMaxScaler has been used to scale the data.

In [110]:
```
Num_feat = pd.DataFrame(Num_scaled,columns = Num_col.columns)
```

In [111]:
```
Num_feat
```

Out[111]:

	VehicleAge	VehOdo	MMRAcquisitionAuctionAveragePrice	MMRAcquisitionAuctionCleanPrice	MMRAcquisitionRetailAveragePrice	MMRAcquisitonRetailClea
0	0.333333	0.759487	0.228291	0.266665	0.297748	0.
1	0.555556	0.800491	0.191871	0.227434	0.278838	0.
2	0.444444	0.622065	0.089637	0.129141	0.177661	0.
3	0.555556	0.548209	0.052993	0.072574	0.119191	0.
4	0.444444	0.582026	0.109540	0.137117	0.197620	0.
...	...	...	...	...	...	
72978	0.888889	0.364400	0.055876	0.081201	0.067963	0.
72979	0.222222	0.603596	0.179665	0.198730	0.190148	0.
72980	0.444444	0.754563	0.239208	0.270192	0.248951	0.
72981	0.333333	0.673890	0.179721	0.206300	0.190225	0.
72982	0.333333	0.559373	0.210934	0.237961	0.221034	0.

The above is a representation of Minimax Scaled data of the numerical columns in the dataset.

Logistic Regression Model:

In [177]:
```
logreg = LogisticRegression()
clf = logreg.fit(Xtrain, ytrain)
ypred_train = clf.predict(Xtrain)
ypred_test = clf.predict(Xtest)
acc_train_log = round(logreg.score(Xtrain, ytrain), 3)
acc_test_log = round(logreg.score(Xtest, ytest), 3)
roc_test_log = round(roc_auc_score(ytest, clf.predict_proba(Xtest)[:, 1]),3)
print('logistic regression train accurary: ',acc_train_log)
print('logistic regression test accurary: ',acc_test_log)
print('logistic regression test ROC: ',roc_test_log)
```

```
logistic regression train accurary:  0.877
logistic regression test accurary:  0.876
logistic regression test ROC:  0.685
```

Confusion Matrix:

```
In [178]:
1 print('confusion matrix for train ROC: ','\n',confusion_matrix(ytrain,ypred_train))
2 print('confusion matrix for test ROC: ','\n',confusion_matrix(ytest,ypred_test))
```

```
confusion matrix for train ROC:
 [[44804    20]
 [ 6251    13]]
confusion matrix for test ROC:
 [[19177     6]
 [ 2708     4]]
```

It can be observed that the True positives in the above model are really low.

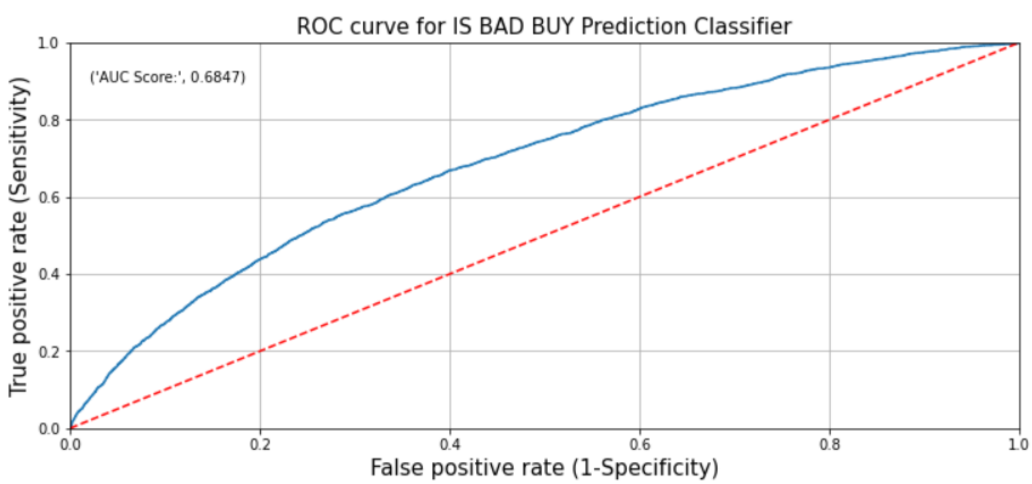
Classification Report:

```
In [179]:
1 print('classification report for train ROC: ','\n',classification_report(ytrain,ypred_train))
2 print('classification report for test ROC: ','\n',classification_report(ytest,ypred_test))
```

```
classification report for train ROC:
               precision    recall  f1-score   support

           0       0.88      1.00      0.93     44824
           1       0.39      0.00      0.00      6264

    accuracy                           0.88     51088
   macro avg       0.64      0.50      0.47     51088
weighted avg       0.82      0.88      0.82     51088

classification report for test ROC:
               precision    recall  f1-score   support

           0       0.88      1.00      0.93     19183
           1       0.40      0.00      0.00      2712

    accuracy                           0.88     21895
   macro avg       0.64      0.50      0.47     21895
weighted avg       0.82      0.88      0.82     21895
```

ROC_AUC curve:

```
In [180]:
1 plot_roc(clf,Xtest)
```



K Nearest Neighbors Method:

In [166]:
```
knn = KNeighborsClassifier(n_neighbors = 3)
knn_m = knn.fit(Xtrain, ytrain)
ypred_train = knn_m.predict(Xtrain)
ypred_test = knn_m.predict(Xtest)
acc_train_log = round(knn.score(Xtrain, ytrain), 3)
acc_test_log = round(knn.score(Xtest, ytest), 3)
roc_test_log = round(roc_auc_score(ytest, knn_m.predict_proba(Xtest)[:, 1]),3)
print('Decision Tree train accurary: ',acc_train_log)
print('Decision Tree test accurary: ',acc_test_log)
print('Decision Tree test ROC: ',roc_test_log)
```

```
Decision Tree train accurary:  0.901
Decision Tree test accurary:  0.848
Decision Tree test ROC:  0.564
```

Confusion Matrix:

In [167]:
```
print('confusion matrix for train ROC: ','\n',confusion_matrix(ytrain,ypred_train))
print('confusion matrix for test ROC: ','\n',confusion_matrix(ytest,ypred_test))
```

```
confusion matrix for train ROC:
 [[44014   810]
 [ 4235  2029]]
confusion matrix for test ROC:
 [[18310   873]
 [ 2463   249]]
```

It can be observed that the true positive count has increased by good number compared to the previous model.

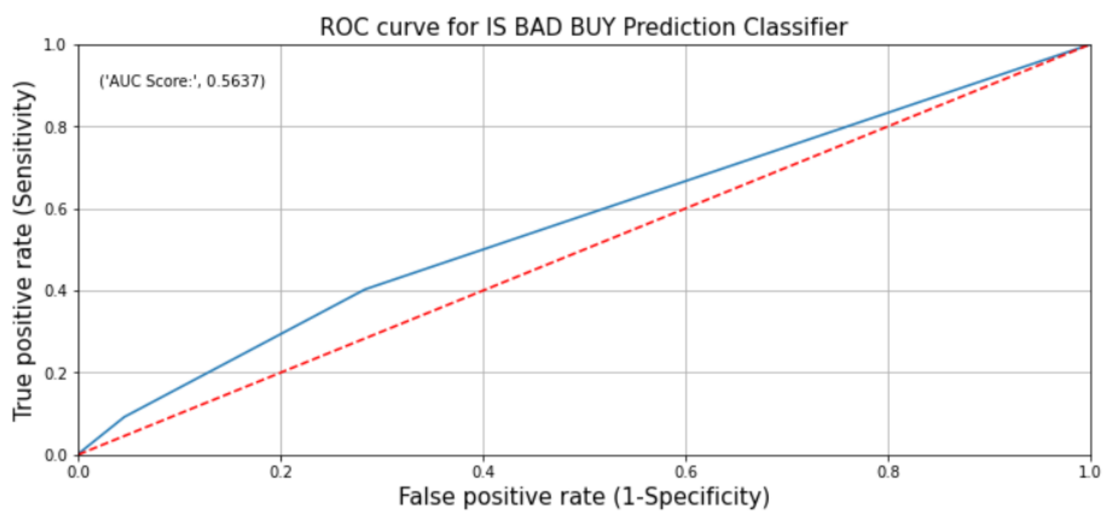
Classification Report:

```
In [168]:
print('classification report for train ROC: ','\n',classification_report(ytrain,ypred_train))
print('classification report for test ROC: ','\n',classification_report(ytest,ypred_test))
```

```
classification report for train ROC: 
               precision    recall  f1-score   support

           0       0.91      0.98      0.95     44824
           1       0.71      0.32      0.45      6264

    accuracy                           0.90     51088
   macro avg       0.81      0.65      0.70     51088
weighted avg       0.89      0.90      0.88     51088

classification report for test ROC: 
               precision    recall  f1-score   support

           0       0.88      0.95      0.92     19183
           1       0.22      0.09      0.13      2712

    accuracy                           0.85     21895
   macro avg       0.55      0.52      0.52     21895
weighted avg       0.80      0.85      0.82     21895
```

ROC_AUC_CURVE:

```
In [176]:
plot_roc(knn_m,Xtest)
```



The roc_auc_curve of KNN model is very poor as it is near the line of 0.5

Model Building:

Step by step approach for model building: -

1. After performing encoding for the categorical features and transforming the numerical variables, we split the data into train data and test data. Model data uses train data to learn whereas test data is used to evaluate or validate the trained model.
2. For the categorical variables we used dummy encoder and our initial models which we built were Logistic Regression and KNN.
3. From these models, we did not achieve desired amount of accuracy, precision and recall Even though we achieve moderate level of accuracy for the model, we get low precision and recall value. since there is presence of high amount of class imbalance.
4. We built non-linear models such as Decision Tree, Random Forest, KNN and Ada Boost Classifier and LGBM. For these models, we performed hyper parameter tuning.

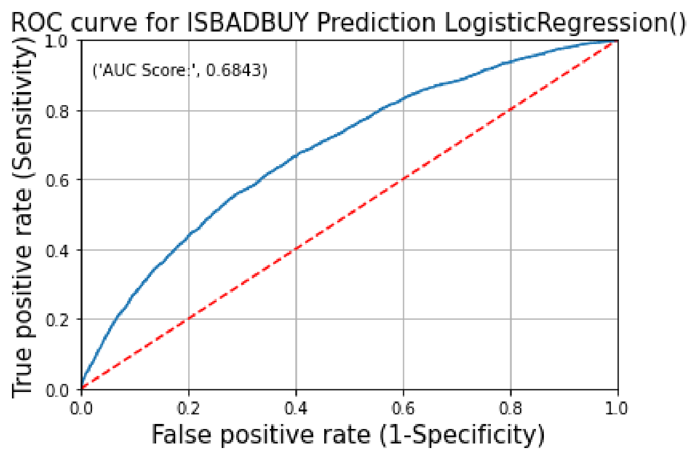
Model building:

1. Logistic Regression tuned:

```
logreg = LogisticRegression()

[95] print('confusion matrix for base model Logistic Regression train: ','\n',confusion_matrix(ytrain,ypred_train))
     print('confusion matrix for base model Logistic Regression  test: ','\n',confusion_matrix(ytest,ypred_test))

confusion matrix for base model Logistic Regression train:
 [[44806    18]
 [ 6251    13]]
confusion matrix for base model Logistic Regression  test:
 [[19176     7]
 [ 2708     4]]

print('classification report for base model Logistic Regression train: ','\n',classification_report(ytrain,ypred_train))
print('classification report for base model Logistic Regression test: ','\n',classification_report(ytest,ypred_test))

classification report for base model Logistic Regression train:
               precision    recall  f1-score   support

           0       0.88      1.00      0.93     44824
           1       0.42      0.00      0.00      6264

    accuracy                           0.88     51088
   macro avg       0.65      0.50      0.47     51088
weighted avg       0.82      0.88      0.82     51088

classification report for base model Logistic Regression test:
               precision    recall  f1-score   support

           0       0.88      1.00      0.93     19183
           1       0.36      0.00      0.00      2712

    accuracy                           0.88     21895
   macro avg       0.62      0.50      0.47     21895
weighted avg       0.81      0.88      0.82     21895
```

A base model has been created and the scores are observed in the below attached score card.

```
plot_roc(log_model,Xtest)
```



	recall	accuracy	cohen_kappa	roc_score
Logistic Regression without Balancing the Weights	0.001475	0.875999	0.001939	0.684

2. <u>Logistic Regression balanced:</u>

```
logreg1 = LogisticRegression(class_weight = 'balanced') # threshold value improvement
log_model = logreg1.fit(Xtrain, ytrain)
ypred_train = log_model.predict(Xtrain)
ypred_test = log_model.predict(Xtest)
acc_train_log = round(logreg1.score(Xtrain, ytrain), 3)
acc_test_log = round(logreg1.score(Xtest, ytest), 3)
roc_test_log = round(roc_auc_score(ytest, log_model.predict_proba(Xtest)[:, 1]),3)
print('logistic regression balanced train  accurary: ',acc_train_log)
print('logistic regression balanced test accurary: ',acc_test_log)
```

```
logistic regression balanced train  accurary:  0.642
logistic regression balanced test accurary:  0.637
```

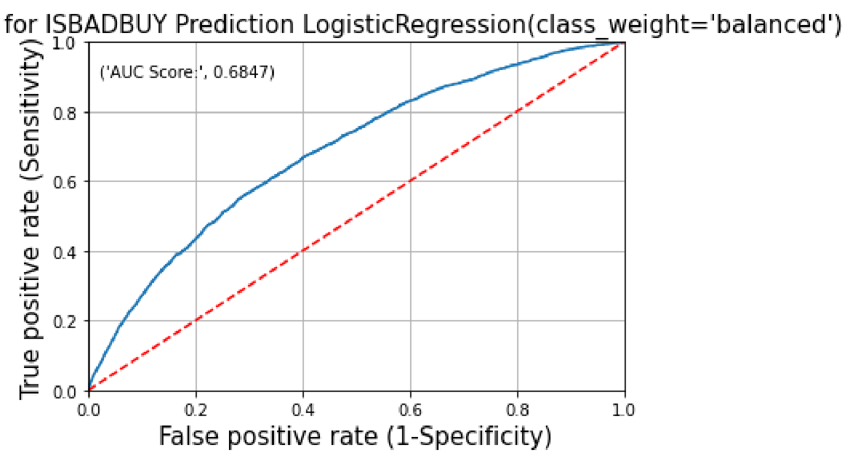
In this model the Logistic model is modified by using the ‘Class_weight = ‘balanced” parameter which in turn balances the categories present in the given Target Variable to provide more weightage to the variable with less frequency.
This decreases the Accuracy of the model but helps greatly in increasing the recall and other parameters of the model.

```
[100] print('confusion matrix for Logistic Regression balanced train: ','\n',confusion_matrix(ytrain,ypred_train))
      print('confusion matrix for Logistic Regression  balanced test: ','\n',confusion_matrix(ytest,ypred_test))

confusion matrix for Logistic Regression balanced train: 
 [[28798 16026]
 [ 2264  4000]]
confusion matrix for Logistic Regression  balanced test: 
 [[12251  6932]
 [ 1018  1694]]
```

```
print('classification report for Logistic Regression balanced train: ','\n',classification_report(ytrain,ypred_train))
print('classification report for Logistic Regression balanced test: ','\n',classification_report(ytest,ypred_test))

classification report for Logistic Regression balanced train: 
               precision    recall  f1-score   support

           0       0.93      0.64      0.76     44824
           1       0.20      0.64      0.30      6264

    accuracy                           0.64     51088
   macro avg       0.56      0.64      0.53     51088
weighted avg       0.84      0.64      0.70     51088

classification report for Logistic Regression balanced test: 
               precision    recall  f1-score   support

           0       0.92      0.64      0.76     19183
           1       0.20      0.62      0.30      2712

    accuracy                           0.64     21895
   macro avg       0.56      0.63      0.53     21895
weighted avg       0.83      0.64      0.70     21895
```

We can observe that the True Positives in the above confusion matrix has increased significantly.

```
plot_roc(log_model,Xtest)
```



	recall	accuracy	cohen_kappa	roc_score
Logistic Regression without Balancing the Weights	0.001475	0.875999	0.001939	0.684
Logistic Regression by Balancing the Weights	0.624631	0.636903	0.135973	0.685

We can observe the change in recall value.

3. KNN Classifier base model:

```
[110] knn = KNeighborsClassifier()
      knn_b = knn.fit(Xtrain, ytrain)
      ypred_train = knn_b.predict(Xtrain)
      ypred_test = knn_b.predict(Xtest)
      acc_train_knn = round(knn.score(Xtrain, ytrain), 3)
      acc_test_knn = round(knn.score(Xtest, ytest), 3)
      roc_test_knn = round(roc_auc_score(ytest, knn_b.predict_proba(Xtest)[:, 1]),3)
      print('KNN train accurary: ',acc_train_knn)
      print('KNN test accurary: ',acc_test_knn)
      print('KNN test ROC: ',roc_test_knn)
```

```
KNN train accurary:  0.887
KNN test accurary:  0.863
KNN test ROC:  0.583
```

```
print('confusion matrix for KNN train : ','\n',confusion_matrix(ytrain,ypred_train))
print('confusion matrix for KNN test : ','\n',confusion_matrix(ytest,ypred_test))
```

```
confusion matrix for KNN train :
 [[44316   508]
 [ 5280   984]]
confusion matrix for KNN test :
 [[18732   451]
 [ 2553   159]]
```

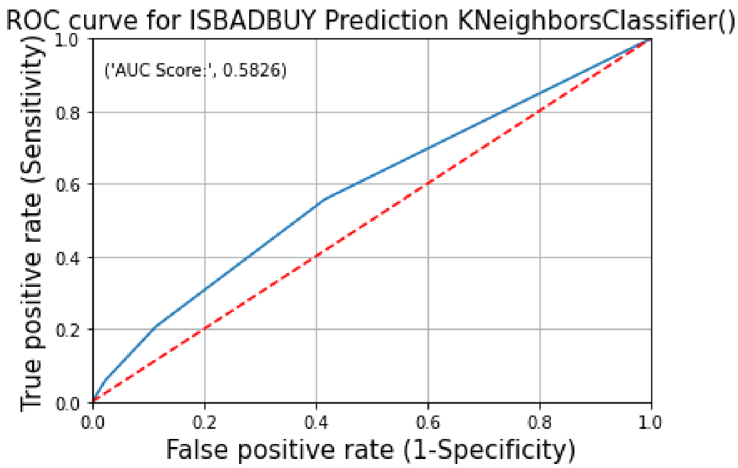
KNN base model is fit with default no of k neighbors and an accuracy of 88.7% is reported even though the True positives in the data are significantly less.

```
[112] print('classification report for KNN train: ','\n',classification_report(ytrain,ypred_train))
      print('classification report for KNN test: ','\n',classification_report(ytest,ypred_test))
```

```
classification report for KNN train:
               precision    recall  f1-score   support

           0       0.89      0.99      0.94     44824
           1       0.66      0.16      0.25      6264

    accuracy                           0.89     51088
   macro avg       0.78      0.57      0.60     51088
weighted avg       0.86      0.89      0.85     51088

classification report for KNN test:
               precision    recall  f1-score   support

           0       0.88      0.98      0.93     19183
           1       0.26      0.06      0.10      2712

    accuracy                           0.86     21895
   macro avg       0.57      0.52      0.51     21895
weighted avg       0.80      0.86      0.82     21895
```

From the above classification report it can be seen that the model is having somewhat better recall compared to the base logistic regression model, which is having almost negligible recall.

```
plot_roc(knn_b,Xtest)
```



	recall	accuracy	cohen_kappa	roc_score
Logistic Regression without Balancing the Weights	0.001475	0.875999	0.001939	0.684
Logistic Regression by Balancing the Weights	0.624631	0.636903	0.135973	0.685
Logistic Regression, C=5 and class weight balanced	0.623525	0.636447	0.135169	0.685
knn Classifier base Model	0.058628	0.862800	0.052631	0.583

We can observe an ROC score of 0.583 which is less compared to other models.

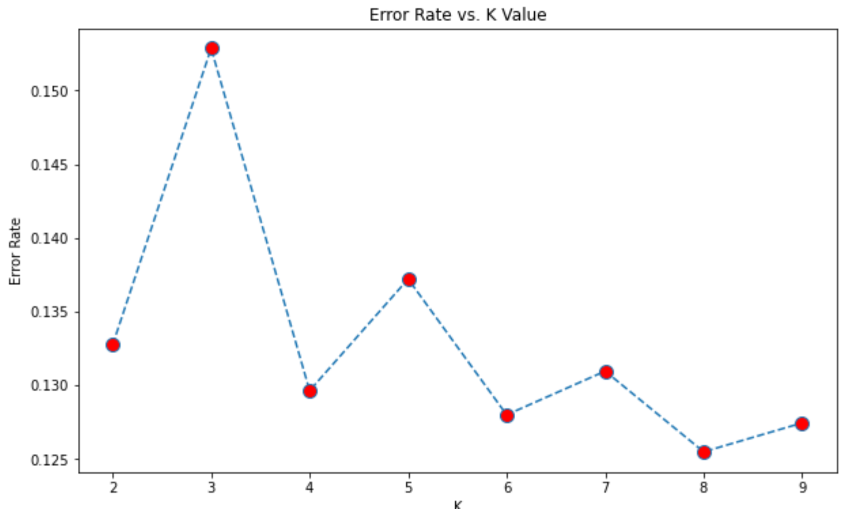
We can further improve this model by trying to get the optimal number of K values and run a model with those number of K .

4. KNN Classifier tuned :

```
[115] error_rate = []
      for i in range(2,10):

       knn = KNeighborsClassifier(n_neighbors=i)
       knn.fit(Xtrain,ytrain)
       pred_i = knn.predict(Xtest)
       error_rate.append(np.mean(pred_i != ytest))
```

```
plt.figure(figsize=(10,6))
plt.plot(range(2,10),error_rate, linestyle='dashed', marker='o',
 markerfacecolor='red', markersize=10)
plt.title('Error Rate vs. K Value')
plt.xlabel('K')
plt.ylabel('Error Rate')
```

```
Text(0, 0.5, 'Error Rate')
```



We have plotted a graph between Error –rate and K value to find out the value of k at which we have the least error. But because of computational limitations range of (2,10) is used. From the above we can observe that k = 8 we have the least error rate.

So a Tuned Model is build using k = 8 and the following observations are made from the below attached code snippets.

```
[117] knn = KNeighborsClassifier(weights = 'distance',n_neighbors = 8)
      knn_m = knn.fit(Xtrain, ytrain)
      ypred_train = knn_m.predict(Xtrain)
      ypred_test = knn_m.predict(Xtest)
      acc_train_knn = round(knn.score(Xtrain, ytrain), 3)
      acc_test_knn = round(knn.score(Xtest, ytest), 3)
      roc_test_knn = round(roc_auc_score(ytest, knn_m.predict_proba(Xtest)[:, 1]),3)
      print('KNN tuned model train accurary: ',acc_train_knn)
      print('KNN tuned model test accurary: ',acc_test_knn)
      print('KNN tuned model test ROC: ',roc_test_knn)
```

```
KNN tuned model train accurary:  1.0
KNN tuned model test accurary:  0.864
KNN tuned model test ROC:  0.599
```

```
print('confusion matrix for KNN tuned model train : ','\n',confusion_matrix(ytrain,ypred_train))
print('confusion matrix for KNN tuned model test : ','\n',confusion_matrix(ytest,ypred_test))
```

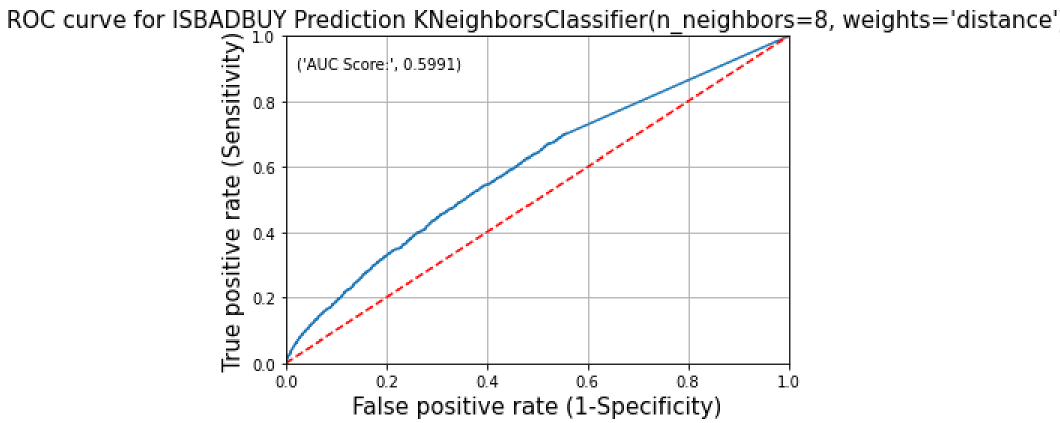
```
confusion matrix for KNN tuned model train :
 [[44824     0]
 [    0  6264]]
confusion matrix for KNN tuned model test :
 [[18728   455]
 [ 2524   188]]
```

It can be observed that the model is an over fit model as there is a significant variation between train and test accuracies.

```
[119] print('classification report for KNN tuned model train: ','\n',classification_report(ytrain,ypred_train))
      print('classification report for KNN tuned model test: ','\n',classification_report(ytest,ypred_test))
```

```
classification report for KNN tuned model train:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     44824
           1       1.00      1.00      1.00      6264

    accuracy                           1.00     51088
   macro avg       1.00      1.00      1.00     51088
weighted avg       1.00      1.00      1.00     51088

classification report for KNN tuned model test:
              precision    recall  f1-score   support

           0       0.88      0.98      0.93     19183
           1       0.29      0.07      0.11      2712

    accuracy                           0.86     21895
   macro avg       0.59      0.52      0.52     21895
weighted avg       0.81      0.86      0.83     21895
```

```
[120] plot_roc(knn_m,Xtest)
```



	recall	accuracy	cohen_kappa	roc_score
Logistic Regression without Balancing the Weights	0.001475	0.875999	0.001939	0.684
Logistic Regression by Balancing the Weights	0.624631	0.636903	0.135973	0.685
Logistic Regression, C=5 and class weight balanced	0.623525	0.636447	0.135169	0.685
knn Classifier base Model	0.058628	0.862800	0.052631	0.583
knn Classifier Tuned (n=8)	0.069322	0.863942	0.067813	0.599

We can observe that there is very minimal change between the base and Tuned models in KNN.

5. Decision Tree classifier base:

```
from sklearn.tree import DecisionTreeClassifier
dt  = DecisionTreeClassifier(random_state = 10)
dt_m = dt.fit(Xtrain,ytrain)
ypred_train = dt_m.predict(Xtrain)
ypred_test = dt_m.predict(Xtest)
acc_train_dt = round(dt.score(Xtrain, ytrain), 3)
acc_test_dt = round(dt.score(Xtest, ytest), 3)
roc_test_dt = round(roc_auc_score(ytest, dt_m.predict_proba(Xtest)[:, 1]),3)
print('Decision Tree base train accurary: ',acc_train_dt)
print('Decision Tree base test accurary: ',acc_test_dt)
```

```
Decision Tree base train accurary:  1.0
Decision Tree base test accurary:  0.793
```

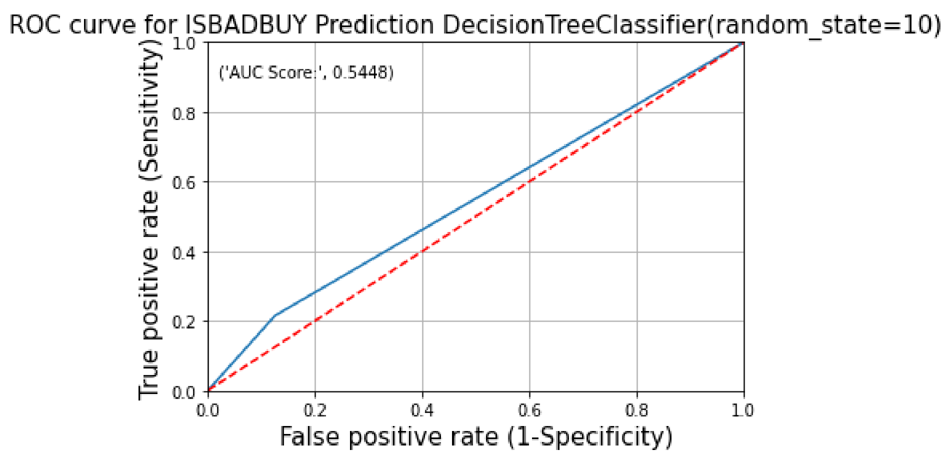
```
[135] print('confusion matrix for Decision Tree base train : ','\n',confusion_matrix(ytrain,ypred_train))
print('confusion matrix for Decision Tree base test : ','\n',confusion_matrix(ytest,ypred_test))
```

```
confusion matrix for Decision Tree base train :
 [[44824     0]
 [    0  6264]]
confusion matrix for Decision Tree base test :
 [[16777  2406]
 [ 2129   583]]
```

```
print('classification report for Decision Tree base train : ','\n',classification_report(ytrain,ypred_train))
print('classification report for Decision Tree base test : ','\n',classification_report(ytest,ypred_test))
```

```
classification report for Decision Tree base train :
               precision    recall  f1-score   support

           0       1.00      1.00      1.00     44824
           1       1.00      1.00      1.00      6264

    accuracy                           1.00     51088
   macro avg       1.00      1.00      1.00     51088
weighted avg       1.00      1.00      1.00     51088

classification report for Decision Tree base test :
               precision    recall  f1-score   support

           0       0.89      0.87      0.88     19183
           1       0.20      0.21      0.20      2712

    accuracy                           0.79     21895
   macro avg       0.54      0.54      0.54     21895
weighted avg       0.80      0.79      0.80     21895
```

```
plot_roc(dt,Xtest)
```



	recall	accuracy	cohen_kappa	roc_score
Logistic Regression without Balancing the Weights	0.001475	0.875999	0.001939	0.684
Logistic Regression by Balancing the Weights	0.624631	0.636903	0.135973	0.685
Logistic Regression, C=5 and class weight balanced	0.623525	0.636447	0.135169	0.685
knn Classifier base Model	0.058628	0.862800	0.052631	0.583
knn Classifier Tuned (n=8)	0.069322	0.863942	0.067813	0.599
Random Forest Base Model	0.019543	0.876730	0.029821	0.691
Random Forest Tuned Model (max_depth=50,max_features=40,n_estimators=1	0.205015	0.790774	0.075390	0.539
Decision Tree base model	0.214971	0.792875	0.085785	0.545

A base model is constructed with decision tree and we can observe that the Recall value is somewhat better that other models except for logistic regression.

6. Decision Tree Tuned model:

```
kfold = KFold(n_splits=5, shuffle=True, random_state=42)

params = {'criterion': ['gini', 'entropy'],
          'max_depth':[None, 15, 25],
          'min_samples_split': [2, 5],
          'min_samples_leaf': [1, 5],
          'max_features' : [None, 10, 20]}

GS_dt = GridSearchCV(estimator=DecisionTreeClassifier(),
                     param_grid=params,
                     scoring='recall',
                     cv=kfold,
                     n_jobs=-1,
                     verbose=2)

GS_dt.fit(Xtrain, ytrain)
```

```
Fitting 5 folds for each of 72 candidates, totalling 360 fits
GridSearchCV(cv=KFold(n_splits=5, random_state=42, shuffle=True),
             estimator=DecisionTreeClassifier(), n_jobs=-1,
             param_grid={'criterion': ['gini', 'entropy'],
                         'max_depth': [None, 15, 25],
                         'max_features': [None, 10, 20],
                         'min_samples_leaf': [1, 5],
                         'min_samples_split': [2, 5]},
             scoring='recall', verbose=2)
```

```
[140] GS_dt.best_params_
```

```
{'criterion': 'gini',
 'max_depth': None,
 'max_features': None,
 'min_samples_leaf': 1,
 'min_samples_split': 2}
```

DT tuned model is used to further refine the performance and we got the above mentioned values as the best parameters which are used to create a tuned model.

```
dt_tuned = DecisionTreeClassifier(criterion = 'gini',max_depth=None,max_features=20,class_weight = 'balanced',min_samples_leaf=1 ,min_samples_split = 2)
dt_t = dt_tuned.fit(Xtrain,ytrain)
ypred_train = dt_t.predict(Xtrain)
ypred_test = dt_t.predict(Xtest)
acc_train_dt = round(dt_tuned.score(Xtrain, ytrain), 3)
acc_test_dt = round(dt_tuned.score(Xtest, ytest), 3)
roc_test_dt = round(roc_auc_score(ytest, dt_t.predict_proba(Xtest)[:, 1]),3)
print('Decision Tree tuned model train accurary: ',acc_train_dt)
print('Decision Tree tuned model test accurary: ',acc_test_dt)
```

```
Decision Tree tuned model train accurary:  1.0
Decision Tree tuned model test accurary:  0.802
```

```
[142] print('confusion matrix for Decision Tree Tuned model train : ','\n',confusion_matrix(ytrain,ypred_train))
      print('confusion matrix for Decision Tree Tuned model test : ','\n',confusion_matrix(ytest,ypred_test))
```

```
confusion matrix for Decision Tree Tuned model train :
 [[44824     0]
 [    0  6264]]
confusion matrix for Decision Tree Tuned model test :
 [[17059  2124]
 [ 2204   508]]
```

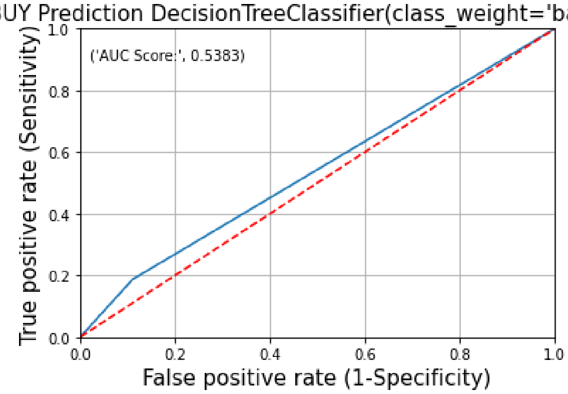
The model is a overfit model but we can observe that there is not much variation in the recall values between the Tuned and untuned Decision tree models.

```
print('classification report for Decision Tree Tuned model train : ','\n',classification_report(ytrain,ypred_train))
print('classification report for Decision Tree Tuned model test : ','\n',classification_report(ytest,ypred_test))
```

```
classification report for Decision Tree Tuned model train :
               precision    recall  f1-score   support

           0       1.00      1.00      1.00     44824
           1       1.00      1.00      1.00      6264

    accuracy                           1.00     51088
   macro avg       1.00      1.00      1.00     51088
weighted avg       1.00      1.00      1.00     51088

classification report for Decision Tree Tuned model test :
               precision    recall  f1-score   support

           0       0.89      0.89      0.89     19183
           1       0.19      0.19      0.19      2712

    accuracy                           0.80     21895
   macro avg       0.54      0.54      0.54     21895
weighted avg       0.80      0.80      0.80     21895
```

```
plot_roc(dt_tuned,Xtest)
```

ROC curve for ISBADBUY Prediction DecisionTreeClassifier(class_weight='balanced', max_features=20)



	recall	accuracy	cohen_kappa	roc_score
Logistic Regression without Balancing the Weights	0.001475	0.875999	0.001939	0.684
Logistic Regression by Balancing the Weights	0.624631	0.636903	0.135973	0.685
Logistic Regression, C=5 and class weight balanced	0.623525	0.636447	0.135169	0.685
knn Classifier base Model	0.058628	0.862800	0.052631	0.583
knn Classifier Tuned (n=8)	0.069322	0.863942	0.067813	0.599
Random Forest Base Model	0.019543	0.876730	0.029821	0.691
Random Forest Tuned Model (max_depth=50,max_features=40,n_estimators=1	0.205015	0.790774	0.075390	0.539
Decision Tree base model	0.214971	0.792875	0.085785	0.545
Decision Tree Tuned Model ((criterion = gini,max_features=20,min_samples_leaf=1 ,min_samples_split = 2)	0.187316	0.802329	0.077575	0.538

We can observe the score comparisons between DT tuned and other models.

7. Random Forest Base Model:

```
random_forest = RandomForestClassifier()
rf_b = random_forest.fit(Xtrain, ytrain)
ypred_train = rf_b.predict(Xtrain)
ypred_test = rf_b.predict(Xtest)
acc_train_rf = round(random_forest.score(Xtrain, ytrain), 3)
acc_test_rf = round(random_forest.score(Xtest, ytest), 3)
roc_test_rf = round(roc_auc_score(ytest, rf_b.predict_proba(Xtest)[:, 1]),3)
print('Random Forest base model train accurary: ',acc_train_rf)
print('Random Forest base model test accurary: ',acc_test_rf)
```

```
Random Forest base model train accurary:  1.0
Random Forest base model test accurary:  0.877
```

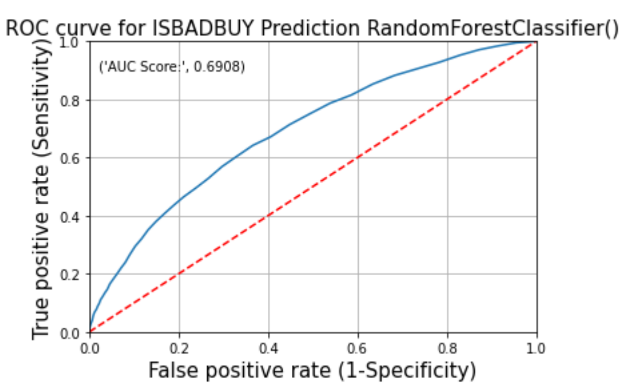
```
[123] print('confusion matrix for Random_forest base model train : ','\n',confusion_matrix(ytrain,ypred_train))
      print('confusion matrix for Random_forest base model test : ','\n',confusion_matrix(ytest,ypred_test))
```

```
confusion matrix for Random_forest base model train :
 [[44824     0]
 [    5  6259]]
confusion matrix for Random_forest base model test :
 [[19143    40]
 [ 2659    53]]
```

```
print('classification report for Random_forest base model train : ','\n',classification_report(ytrain,ypred_train))
print('classification report for Random_forest base model test : ','\n',classification_report(ytest,ypred_test))
```

```
classification report for Random_forest base model train :
               precision    recall  f1-score   support

           0       1.00      1.00      1.00     44824
           1       1.00      1.00      1.00      6264

    accuracy                           1.00     51088
   macro avg       1.00      1.00      1.00     51088
weighted avg       1.00      1.00      1.00     51088

classification report for Random_forest base model test :
               precision    recall  f1-score   support

           0       0.88      1.00      0.93     19183
           1       0.57      0.02      0.04      2712

    accuracy                           0.88     21895
   macro avg       0.72      0.51      0.49     21895
weighted avg       0.84      0.88      0.82     21895
```

```
plot_roc(random_forest,Xtest)
```



	recall	accuracy	cohen_kappa	roc_score
Logistic Regression without Balancing the Weights	0.001475	0.875999	0.001939	0.684
Logistic Regression by Balancing the Weights	0.624631	0.636903	0.135973	0.685
Logistic Regression, C=5 and class weight balanced	0.623525	0.636447	0.135169	0.685
knn Classifier base Model	0.058628	0.862800	0.052631	0.583
knn Classifier Tuned (n=8)	0.069322	0.863942	0.067813	0.599
Random Forest Base Model	0.019543	0.876730	0.029821	0.691

8. Random Forest Tuned Model:

```
kfold = KFold(n_splits=5, shuffle=True, random_state=42)
params = {'n_estimators': [1, 2, 5], 'max_depth': [None, 10, 25, 50],
         'max_features': [ 10, 20, 30, 40]}

GS_rf = GridSearchCV(estimator=RandomForestClassifier(),
                     param_grid=params,
                     scoring='recall',
                     cv=kfold,
                     n_jobs=-1,
                     verbose=2)

GS_rf.fit(Xtrain, ytrain)
```

```
Fitting 5 folds for each of 48 candidates, totalling 240 fits
GridSearchCV(cv=KFold(n_splits=5, random_state=42, shuffle=True),
             estimator=RandomForestClassifier(), n_jobs=-1,
             param_grid={'max_depth': [None, 10, 25, 50],
                         'max_features': [10, 20, 30, 40],
                         'n_estimators': [1, 2, 5]},
             scoring='recall', verbose=2)
```

```
[128] GS_rf.best_params_
```

```
{'max_depth': 50, 'max_features': 40, 'n_estimators': 1}
```

RF tuned model is used to further refine the performance and we got the above mentioned values as the best parameters which are used to create a tuned model.

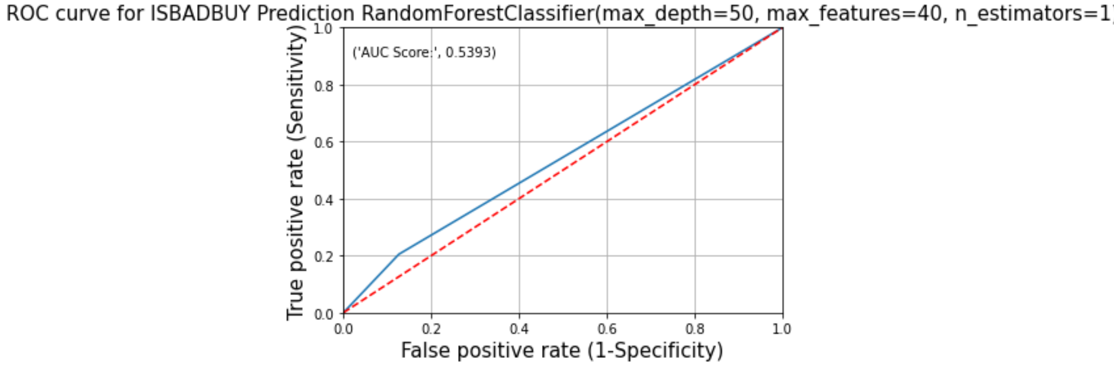
```
[129] rf_tuned = RandomForestClassifier(n_estimators=1,
                                        max_depth=50,
                                        max_features=40)

      rf_t = rf_tuned.fit(Xtrain, ytrain)
      ypred_train = rf_t.predict(Xtrain)
      ypred_test = rf_t.predict(Xtest)
      acc_train_rf = round(rf_tuned.score(Xtrain, ytrain), 3)
      acc_test_rf = round(rf_tuned.score(Xtest, ytest), 3)
      roc_test_rf = round(roc_auc_score(ytest, rf_t.predict_proba(Xtest)[:, 1]),3)
      print('Random Forest tuned model train accurary: ',acc_train_rf)
      print('Random Forest tuned model test accurary: ',acc_test_rf)

      Random Forest tuned model train accurary:  0.922
      Random Forest tuned model test accurary:  0.791
```

```
print('confusion matrix for Random_forest Tuned model train : ','\n',confusion_matrix(ytrain,ypred_train))
print('confusion matrix for Random_forest Tuned model test : ','\n',confusion_matrix(ytest,ypred_test))

confusion matrix for Random_forest Tuned model train :
 [[42701  2123]
 [ 1842  4422]]
confusion matrix for Random_forest Tuned model test :
 [[16758  2425]
 [ 2156   556]]
```

It can be seen from the above that there is good improvement over the base RF model, we can also observe that the True Positives of the model has increased significantly from the base model.

```
print('classification report for Random_forest Tuned model train : ','\n',classification_report(ytrain,ypred_train))
print('classification report for Random_forest Tuned model test : ','\n',classification_report(ytest,ypred_test))

classification report for Random_forest Tuned model train :
               precision    recall  f1-score   support

           0       0.96      0.95      0.96     44824
           1       0.68      0.71      0.69      6264

    accuracy                           0.92     51088
   macro avg       0.82      0.83      0.82     51088
weighted avg       0.92      0.92      0.92     51088

classification report for Random_forest Tuned model test :
               precision    recall  f1-score   support

           0       0.89      0.87      0.88     19183
           1       0.19      0.21      0.20      2712

    accuracy                           0.79     21895
   macro avg       0.54      0.54      0.54     21895
weighted avg       0.80      0.79      0.79     21895
```

```
[132] plot_roc(rf_tuned,Xtest)
```



	recall	accuracy	cohen_kappa	roc_score
Logistic Regression without Balancing the Weights	0.001475	0.875999	0.001939	0.684
Logistic Regression by Balancing the Weights	0.624631	0.636903	0.135973	0.685
Logistic Regression, C=5 and class weight balanced	0.623525	0.636447	0.135169	0.685
knn Classifier base Model	0.058628	0.862800	0.052631	0.583
knn Classifier Tuned (n=8)	0.069322	0.863942	0.067813	0.599
Random Forest Base Model	0.019543	0.876730	0.029821	0.691
Random Forest Tuned Model (max_depth=50,max_features=40,n_estimators=1	0.205015	0.790774	0.075390	0.539

It is observed that the tuned RF model is having better recall and other values compared to the base model.

9. Ada Boost Base model:

Here Adaboosting base model is performed on a DecisionTreeclassifier to improve the model accuracy.

```
[146] ada_dt = AdaBoostClassifier(base_estimator=DecisionTreeClassifier())
      ada_b = ada_dt.fit(Xtrain, ytrain)
      ypred_train = ada_b.predict(Xtrain)
      ypred_test = ada_b.predict(Xtest)
      acc_train_ada = round(ada_dt.score(Xtrain, ytrain), 3)
      acc_test_ada = round(ada_dt.score(Xtest, ytest), 3)
      roc_test_ada = round(roc_auc_score(ytest, ada_b.predict_proba(Xtest)[:, 1]),3)
      print('Ada boost with Decision Tree model train accurary: ',acc_train_ada)
      print('Ada boost with Decision Tree model test accurary: ',acc_test_ada)

Ada boost with Decision Tree model train accurary:  1.0
Ada boost with Decision Tree model test accurary:  0.792
```

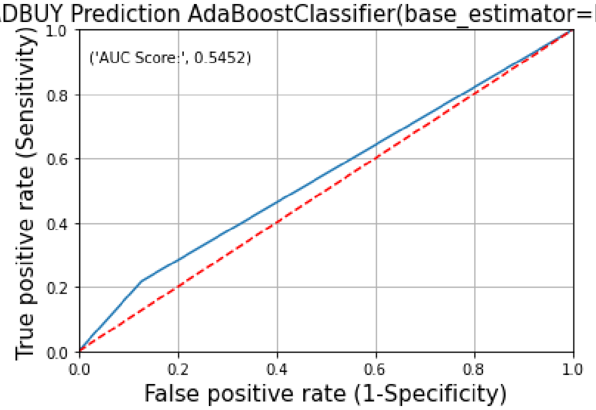
```
print('confusion matrix for Ada boost w Loading... on Tree model train : ','\n',confusion_matrix(ytrain,ypred_train))
print('confusion matrix for Ada boost with Decision Tree model test : ','\n',confusion_matrix(ytest,ypred_test))

confusion matrix for Ada boost with Decision Tree model train :
 [[44824     0]
 [    0  6264]]
confusion matrix for Ada boost with Decision Tree model test :
 [[16742  2441]
 [ 2122   590]]
```

```
print('classification report for Ada boost with Decision Tree model train : ','\n',classification_report(ytrain,ypred_train))
print('classification report for Ada boost with Decision Tree model test : ','\n',classification_report(ytest,ypred_test))
```

```
classification report for Ada boost with Decision Tree model train :
               precision    recall  f1-score   support

           0       1.00      1.00      1.00     44824
           1       1.00      1.00      1.00      6264

    accuracy                           1.00     51088
   macro avg       1.00      1.00      1.00     51088
weighted avg       1.00      1.00      1.00     51088

classification report for Ada boost with Decision Tree model test :
               precision    recall  f1-score   support

           0       0.89      0.87      0.88     19183
           1       0.19      0.22      0.21      2712

    accuracy                           0.79     21895
   macro avg       0.54      0.55      0.54     21895
weighted avg       0.80      0.79      0.80     21895
```

```
plot_roc(ada_dt,Xtest)
```

ROC curve for ISBADBUY Prediction AdaBoostClassifier(base_estimator=DecisionTreeClassifier())



	recall	accuracy	cohen_kappa	roc_score
Logistic Regression without Balancing the Weights	0.001475	0.875999	0.001939	0.684
Logistic Regression by Balancing the Weights	0.624631	0.636903	0.135973	0.685
Logistic Regression, C=5 and class weight balanced	0.623525	0.636447	0.135169	0.685
knn Classifier base Model	0.058628	0.862800	0.052631	0.583
knn Classifier Tuned (n=8)	0.069322	0.863942	0.067813	0.599
Random Forest Base Model	0.019543	0.876730	0.029821	0.691
Random Forest Tuned Model (max_depth=50,max_features=40,n_estimators=1	0.205015	0.790774	0.075390	0.539
Decision Tree base model	0.214971	0.792875	0.085785	0.545
Decision Tree Tuned Model ((criterion = gini,max_features=20,min_samples_leaf=1 ,min_samples_split = 2)	0.187316	0.802329	0.077575	0.538
Ada boost with Decision Tree model	0.217552	0.791596	0.085963	0.545

The scored for the model has been observed above and we can see that there
Is not much comparative

10. Ada boost Tuned model:

```
params = {'n_estimators': [1, 10, 50, 100],
          'learning_rate': [0.1, 0.5, 1, 5]}

GS_ada_dt = GridSearchCV(estimator=AdaBoostClassifier(),
                         param_grid=params,
                         scoring='recall',
                         cv=3,
                         n_jobs=-1,
                         verbose=2)

GS_ada_dt.fit(Xtrain, ytrain)
```

```
Fitting 3 folds for each of 16 candidates, totalling 48 fits
GridSearchCV(cv=3, estimator=AdaBoostClassifier(), n_jobs=-1,
             param_grid={'learning_rate': [0.1, 0.5, 1, 5],
                         'n_estimators': [1, 10, 50, 100]},
             scoring='recall', verbose=2)
```

```
[152] GS_ada_dt.best_params_
```

```
{'learning_rate': 5, 'n_estimators': 10}
```

Ada boost tuned model has the above mentioned tuned parameters but due to computational restrictions it is not the best tuned model that can be achieved.

```
[153] ada_tuned = AdaBoostClassifier(learning_rate= 5,n_estimators = 10)
      ada_dt_tuned = ada_tuned.fit(Xtrain, ytrain)
      ypred_train = ada_dt_tuned.predict(Xtrain)
      ypred_test = ada_dt_tuned.predict(Xtest)
      acc_train_ada = round(ada_dt.score(Xtrain, ytrain), 3)
      acc_test_ada = round(ada_dt.score(Xtest, ytest), 3)
      roc_test_ada = round(roc_auc_score(ytest, ada_dt_tuned.predict_proba(Xtest)[:, 1]),3)
      print('Ada boost with Decision Tree model train accurary: ',acc_train_ada)
      print('Ada boost with Decision Tree model test accurary: ',acc_test_ada)
```

```
Ada boost with Decision Tree model train accurary:  1.0
Ada boost with Decision Tree model test accurary:  0.792
```

```
[154] print('confusion matrix for Ada boost with Decision Tree Tuned model train : ','\n',confusion_matrix(ytrain,ypred_train))
      print('confusion matrix for Ada boost with Decision Tree Tuned model test : ','\n',confusion_matrix(ytest,ypred_test))
```

```
confusion matrix for Ada boost with Decision Tree Tuned model train :
 [[    0 44824]
 [    0  6264]]
confusion matrix for Ada boost with Decision Tree Tuned model test :
 [[    0 19183]
 [    0  2712]]
```

It can be seen from the above that the model is very poor at performing so need better computational capabilities to get a better result.

```
print('classification report for Ada boost with Decision Tree Tuned model train : ','\n',classification_report(ytrain,ypred_train))
print('classification report for Ada boost with Decision Tree Tuned model test : ','\n',classification_report(ytest,ypred_test))
```

```
classification report for Ada boost with Decision Tree Tuned model train :
               precision    recall  f1-score   support

           0       0.00      0.00      0.00     44824
           1       0.12      1.00      0.22      6264

    accuracy                           0.12     51088
   macro avg       0.06      0.50      0.11     51088
weighted avg       0.02      0.12      0.03     51088

classification report for Ada boost with Decision Tree Tuned model test :
               precision    recall  f1-score   support

           0       0.00      0.00      0.00     19183
           1       0.12      1.00      0.22      2712

    accuracy                           0.12     21895
   macro avg       0.06      0.50      0.11     21895
weighted avg       0.02      0.12      0.03     21895
```

	recall	accuracy	cohen_kappa	roc_score
Logistic Regression without Balancing the Weights	0.001475	0.875999	0.001939	0.684
Logistic Regression by Balancing the Weights	0.624631	0.636903	0.135973	0.685
Logistic Regression, C=5 and class weight balanced	0.623525	0.636447	0.135169	0.685
knn Classifier base Model	0.058628	0.862800	0.052631	0.583
knn Classifier Tuned (n=8)	0.069322	0.863942	0.067813	0.599
Random Forest Base Model	0.019543	0.876730	0.029821	0.691
Random Forest Tuned Model (max_depth=50,max_features=40,n_estimators=1	0.205015	0.790774	0.075390	0.539
Decision Tree base model	0.214971	0.792875	0.085785	0.545
Decision Tree Tuned Model ((criterion = gini,max_features=20,min_samples_leaf=1 ,min_samples_split = 2)	0.187316	0.802329	0.077575	0.538
Ada boost with Decision Tree model	0.217552	0.791596	0.085963	0.545
Ada boost with Decision Tree Tuned model(learning_rate= 5,n_estimators = 10)	1.000000	0.123864	0.000000	0.395

11. LGBM base Model:

```
[158] lgbm = LGBMClassifier()
      lgbm_m = lgbm.fit(Xtrain, ytrain)
      ypred_train = lgbm_m.predict(Xtrain)
      ypred_test = lgbm_m.predict(Xtest)
      acc_train_lgbm = round(lgbm.score(Xtrain, ytrain), 3)
      acc_test_lgbm = round(lgbm.score(Xtest, ytest), 3)
      roc_test_lgbm = round(roc_auc_score(ytest, lgbm_m.predict_proba(Xtest)[:, 1]),3)
      print('LGBM train accurary: ',acc_train_lgbm)
      print('LGBM test accurary: ',acc_test_lgbm)
```

```
LGBM train accurary:  0.881
LGBM test accurary:  0.877
```

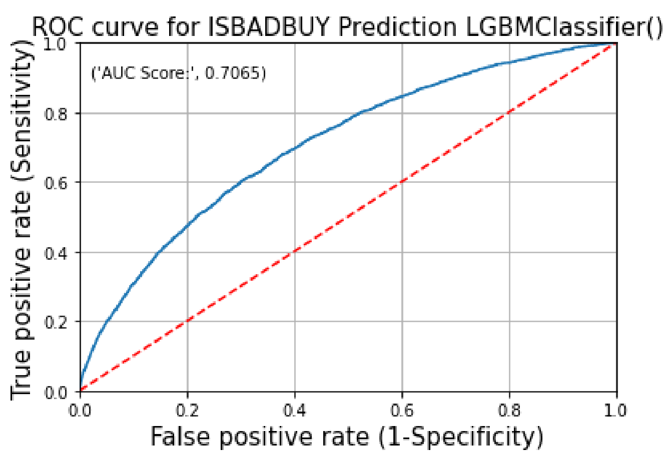
```
print('confusion matrix for LGBM train : ','\n',confusion_matrix(ytrain,ypred_train))
print('confusion matrix for LGBM test : ','\n',confusion_matrix(ytest,ypred_test))
```

```
confusion matrix for LGBM train :
 [[44798    26]
 [ 6028   236]]
confusion matrix for LGBM test :
 [[19155    28]
 [ 2673    39]]
```

```
[160] print('classification report for LGBM train : ','\n',classification_report(ytrain,ypred_train))
      print('classification report for LGBM test: ','\n',classification_report(ytest,ypred_test))
```

```
classification report for LGBM train :
               precision    recall  f1-score   support

           0       0.88      1.00      0.94     44824
           1       0.90      0.04      0.07      6264

    accuracy                           0.88     51088
   macro avg       0.89      0.52      0.50     51088
weighted avg       0.88      0.88      0.83     51088

classification report for LGBM test:
               precision    recall  f1-score   support

           0       0.88      1.00      0.93     19183
           1       0.58      0.01      0.03      2712

    accuracy                           0.88     21895
   macro avg       0.73      0.51      0.48     21895
weighted avg       0.84      0.88      0.82     21895
```

```
plot_roc(lgbm_m,Xtest)
```



	recall	accuracy	cohen_kappa	roc_score
Logistic Regression without Balancing the Weights	0.001475	0.875999	0.001939	0.684
Logistic Regression by Balancing the Weights	0.624631	0.636903	0.135973	0.685
Logistic Regression, C=5 and class weight balanced	0.623525	0.636447	0.135169	0.685
knn Classifier base Model	0.058628	0.862800	0.052631	0.583
knn Classifier Tuned (n=8)	0.069322	0.863942	0.067813	0.599
Random Forest Base Model	0.019543	0.876730	0.029821	0.691
Random Forest Tuned Model (max_depth=50,max_features=40,n_estimators=1	0.205015	0.790774	0.075390	0.539
Decision Tree base model	0.214971	0.792875	0.085785	0.545
Decision Tree Tuned Model ((criterion = gini,max_features=20,min_samples_leaf=1 ,min_samples_split = 2)	0.187316	0.802329	0.077575	0.538
Ada boost with Decision Tree model	0.217552	0.791596	0.085963	0.545
Ada boost with Decision Tree Tuned model(learning_rate= 5,n_estimators = 10)	1.000000	0.123864	0.000000	0.395
LGBM Base Classifier	0.014381	0.876639	0.022228	0.706

12. LGBM tuned Model:

```
[163] lgbm_tuned = LGBMClassifier(class_weight='balanced',learning_rate=0.09,max_depth=-5)
      lgbm_t = lgbm_tuned.fit(Xtrain, ytrain)
      ypred_train = lgbm_t.predict(Xtrain)
      ypred_test = lgbm_t.predict(Xtest)
      acc_train_lgbm = round(lgbm_tuned.score(Xtrain, ytrain), 3)
      acc_test_lgbm = round(lgbm_tuned.score(Xtest, ytest), 3)
      roc_test_lgbm = round(roc_auc_score(ytest, lgbm_t.predict_proba(Xtest)[:, 1]),3)
      print('LGBM Tuned train accurary: ',acc_train_lgbm)
      print('LGBM Tuned test accurary: ',acc_test_lgbm)
```

```
LGBM Tuned train accurary:  0.711
LGBM Tuned test accurary:  0.676
```

```
print('confusion matrix for LGBM Tuned train : ','\n',confusion_matrix(ytrain,ypred_train))
print('confusion matrix for LGBM Tuned test : ','\n',confusion_matrix(ytest,ypred_test))
```

```
confusion matrix for LGBM Tuned train :
 [[31562 13262]
 [ 1515  4749]]
confusion matrix for LGBM Tuned test :
 [[13148  6035]
 [ 1049  1663]]
```

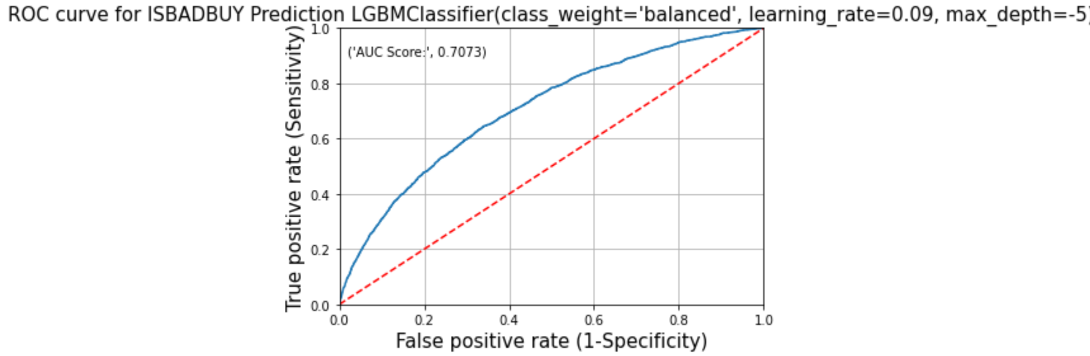
It can be observed that the True positives in the Tuned LGBM are much better compared to the other models, so far.

```
[165] print('classification report for LGBM Tuned train : ','\n',classification_report(ytrain,ypred_train))
      print('classification report for LGBM Tuned test: ','\n',classification_report(ytest,ypred_test))
```

```
classification report for LGBM Tuned train :
              precision    recall  f1-score   support

           0       0.95      0.70      0.81     44824
           1       0.26      0.76      0.39      6264

    accuracy                           0.71     51088
   macro avg       0.61      0.73      0.60     51088
weighted avg       0.87      0.71      0.76     51088

classification report for LGBM Tuned test:
              precision    recall  f1-score   support

           0       0.93      0.69      0.79     19183
           1       0.22      0.61      0.32      2712

    accuracy                           0.68     21895
   macro avg       0.57      0.65      0.55     21895
weighted avg       0.84      0.68      0.73     21895
```

Also it can be observed that is has the highest recall value among all other models.

```
[166] plot_roc(lgbm_t,Xtest)
```



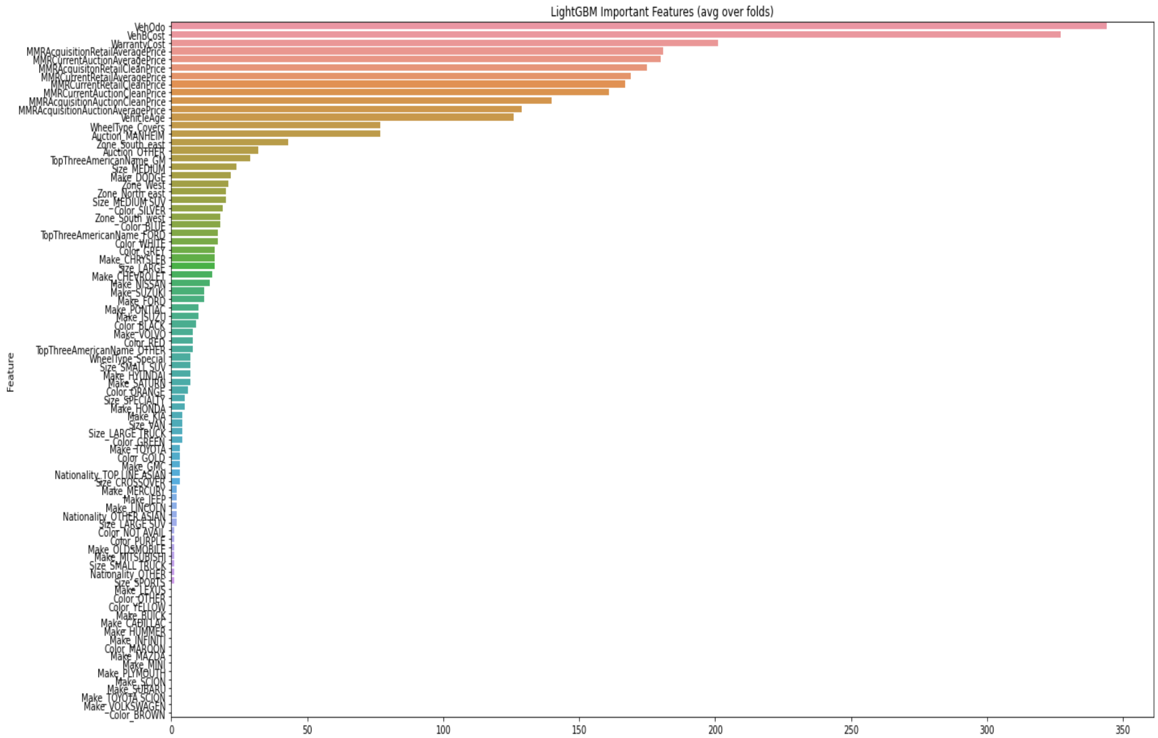
	recall	accuracy	cohen_kappa	roc_score
Logistic Regression without Balancing the Weights	0.001475	0.875999	0.001939	0.684
Logistic Regression by Balancing the Weights	0.624631	0.636903	0.135973	0.685
Logistic Regression, C=5 and class weight balanced	0.623525	0.636447	0.135169	0.685
knn Classifier base Model	0.058628	0.862800	0.052631	0.583
knn Classifier Tuned (n=8)	0.069322	0.863942	0.067813	0.599
Random Forest Base Model	0.019543	0.876730	0.029821	0.691
Random Forest Tuned Model (max_depth=50,max_features=40,n_estimators=1	0.205015	0.790774	0.075390	0.539
Decision Tree base model	0.214971	0.792875	0.085785	0.545
Decision Tree Tuned Model ((criterion = gini,max_features=20,min_samples_leaf=1 ,min_samples_split = 2)	0.187316	0.802329	0.077575	0.538
Ada boost with Decision Tree model	0.217552	0.791596	0.085963	0.545
Ada boost with Decision Tree Tuned model(learning_rate= 5,n_estimators = 10)	1.000000	0.123864	0.000000	0.395
LGBM Base Classifier	0.014381	0.876639	0.022228	0.706
LGBM tuned Classifier(learning_rate=0.09,max_depth=-5)	0.613201	0.676456	0.166881	0.707

It can be observed that the so LGBM tuned model has the best performance compared to other models. The recall value, Roc_score, cohen_kappa is highest for the LGBM tuned model

Feature selection:

LGBM tuned model features has been found from the model. Which shows us clearly that VehOdo reading plays a major role in the model performance also we can observe that VehBcost also plays a major role in the model. Then comes the warranty cost followed by the MMR columns with the same importance for almost all of the MMR columns.

The below chart Explains the above



Conclusion:

1. We observed that the important features for the model performance are of technical features which are good predictors of Purchase of the car.
2. Higher the odometer reading of the car, definitely reduces the performance of the car. More the distance travelled would cause the car to have some kind of wear and tear. So, mileage is an import factor in predicting the car price
3. VehBcost is also an important factor in predicting the purchase of the car.
4. Many factors other than the make and model of the car, are good predictors of the car.
5. In the future, more data will be collected using different web-scraping techniques, and deep learning classifiers will be tested.

Limitations of Data:

1. The dataset belongs to United States and consists of data of only 72000 used car details. The model will be more robust if the data would have belonged from different regions of the world.
2. Also, the duration of data collected is from Jan 2011 and to September 2011. A larger time frame would have been better.

Challenges:

1. High cardinality results in huge training effort in model tuning due to increase in model complexity (i.e. more number of features)
2. We also faced challenges on robust model tuning on all the models. Due to computational limitations, we are limited to using Randomized Search, and Grid Search as hyper parameter tuning techniques instead of using Hyper Opt etc. Scope

Scope for some future work is:

1. Perform more hyper parameter tuning techniques for the XGB model since due to lower processing power of our laptops, we couldn't do that.
2. Exploring some robust data sampling technique as part of choosing smaller sample (a true representation of population data) from the population data.
3. Train the model again once more data comes in.
4. Try to work on more balanced data and in order to achieve better recall and precision.