Importing Necessary Libraries

In [1]:

```
import pandas as pd
import numpy as np
import sklearn
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, accuracy_score
from sklearn.metrics import confusion_matrix
import numpy as np
from collections import defaultdict
import pydot
from io import StringIO
from sklearn.tree import export_graphviz
from sklearn.model_selection import GridSearchCV
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.feature_selection import SelectFromModel
from sklearn.metrics import roc_auc_score
from sklearn.ensemble import VotingClassifier
from sklearn.feature_selection import RFECV
from sklearn.metrics import roc_curve
from itertools import compress
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import RandomOverSampler
import warnings
warnings.filterwarnings('ignore')

'''
TODO:

1. Try to improve

2. Desing the replace_val for each column

3. Creat preprocess procedure for every class.
'''


%matplotlib inline

rs = 101
```

Task 1. Data Selection and Distribution.

In [2]:

```
## Read Data
df = pd.read_csv("CaseStudyData.csv")
```

1. What is the proportion of cars who can be classified as a "kick"?

In [3]:

```
## Exploring the features in this dataset
print("Number of Columns: ", len(df.columns))
print("Columns: ",list(df.columns))
```

```
Number of Columns:  31
Columns:  ['PurchaseID', 'PurchaseTimestamp', 'PurchaseDate', 'Aucti
on', 'VehYear', 'Make', 'Color', 'Transmission', 'WheelTypeID', 'Whe
elType', 'VehOdo', 'Nationality', 'Size', 'TopThreeAmericanName', 'M
MRAcquisitionAuctionAveragePrice', 'MMRAcquisitionAuctionCleanPric
e', 'MMRAcquisitionRetailAveragePrice', 'MMRAcquisitonRetailCleanPri
ce', 'MMRCurrentAuctionAveragePrice', 'MMRCurrentAuctionCleanPrice',
'MMRCurrentRetailAveragePrice', 'MMRCurrentRetailCleanPrice', 'MMRCu
rrentRetailRatio', 'PRIMEUNIT', 'AUCGUART', 'VNST', 'VehBCost', 'IsO
nlineSale', 'WarrantyCost', 'ForSale', 'IsBadBuy']
```

In [4]:

```
print("Number of Observations: ", len(df))
```

```
Number of Observations:  41476
```

In [5]:

```
proportionOfKicks = len(df[df['IsBadBuy'] == 1]) / len(list(df['IsBadBuy']))
print("The proportion of kicks: ", proportionOfKicks)
```

```
The proportion of kicks:  0.12949657633333012
```

2. Did you have to fix any data quality problems? Detail them.

In [6]:

```
#### PREPROCESSING STATEGY
NEW_STATEGY = True
ResamplingMethod = 'rus' #['ros', 'rus']
if NEW_STATEGY:
    print("Using New Preprocessing Strategy")
    using_cat = False
    categorial_cols = ['Auction', 'VehYear', 'Make', 'Color', 'Transmission','Wh
eelTypeID', 'WheelType', 'Nationality', 'Size', 'TopThreeAmericanName','PRIMEUNI
T','AUCGUART','VNST','IsOnlineSale','ForSale' ] # Replaced by the most common
    interval_cols = ['VehOdo','MMRAcquisitionAuctionAveragePrice','MMRAcquisitio
nAuctionCleanPrice','MMRAcquisitionRetailAveragePrice','MMRAcquisitonRetailClean
Price','VehBCost','WarrantyCost' ]
    drop_cols = ['PurchaseID','PurchaseDate','PurchaseTimestamp']
    questionMark_data = ['MMRCurrentAuctionAveragePrice','MMRCurrentAuctionClean
Price','MMRCurrentRetailAveragePrice','MMRCurrentRetailCleanPrice','MMRCurrentRe
tailRatio']
    replaced_vals = ['?', '#VALUE!']
    if using_cat:
        categorial_cols += questionMark_data
        print("See [MMRCurrentAuctionAveragePrice" +
              "MMRCurrentAuctionCleanPrice, MMRCurrentRetailAveragePrice," +
              " MMRCurrentRetailCleanPrice, MMRCurrentRetailRatio] as Categorial
Data")
    else:
        interval_cols += questionMark_data
        print("See [MMRCurrentAuctionAveragePrice" +
              "MMRCurrentAuctionCleanPrice, MMRCurrentRetailAveragePrice," +
              " MMRCurrentRetailCleanPrice, MMRCurrentRetailRatio] as Interval D
ata")
else:
    print("Using Old Preprocessing Strategy")
    drop_cols = ['PurchaseID','PurchaseDate']
    categorial_cols = ['Auction', 'VehYear', 'Make', 'Color', 'Transmission','Wh
eelTypeID', 'WheelType', 'Nationality', 'Size', 'TopThreeAmericanName','PRIMEUNI
T','AUCGUART','VNST','IsOnlineSale','ForSale' ] # Replaced by the most common
    interval_cols = ['PurchaseTimestamp', 'VehOdo','MMRAcquisitionAuctionAverage
Price','MMRAcquisitionAuctionCleanPrice','MMRAcquisitionRetailAveragePrice','MMR
AcquisitonRetailCleanPrice','MMRCurrentAuctionAveragePrice','MMRCurrentAuctionCl
eanPrice','MMRCurrentRetailAveragePrice','MMRCurrentRetailCleanPrice','MMRCurren
tRetailRatio','VehBCost','WarrantyCost' ] # Replaced by the mean
    replaced_vals = ['?', '#VALUE!']

print("Total null before Replacing: ", df.isnull().sum().sum())
```

```
Using New Preprocessing Strategy
See [MMRCurrentAuctionAveragePriceMMRCurrentAuctionCleanPrice, MMRCu
rrentRetailAveragePrice, MMRCurrentRetailCleanPrice, MMRCurrentRetai
lRatio] as Interval Data
Total null before Replacing:  1691
```

In [7]:

```
def printColumnInfo():

    '''
    Display the information of this Dataframe
    '''

    for colName in df.columns:
        print("====================== " + str(colName) + " ===================
===")
        print("---------------------- FIRST FIVE ----------------------")
        print(df[colName][:5])
        print("---------------------- DESCIRBE ----------------------")
        print(df[colName].describe())
        print("---------------------- COUNTS ----------------------")
        commonList = list(df[colName].value_counts().keys())
        if len(commonList) > 100:
            print("Five Most Common: ", commonList[:5])
        else:
            print("Count List: \n", df[colName].value_counts())
        print("Num of NULL: ", df[colName].isnull().sum())
        for rep in replaced_vals:
            print("Number of "+str(rep)+" : " + str(len(df[df[colName] == rep
])))
printColumnInfo()
```

```
======================= PurchaseID =======================
----------------------- FIRST FIVE -----------------------
0    0
1    1
2    2
3    3
4    4
Name: PurchaseID, dtype: int64
----------------------- DESCIRBE -----------------------
count    41476.000000
mean     20737.500000
std      11973.234219
min          0.000000
25%      10368.750000
50%      20737.500000
75%      31106.250000
max      41475.000000
Name: PurchaseID, dtype: float64
----------------------- COUNTS -----------------------
Five Most Common:  [2047, 11567, 15693, 13644, 3403]
Num of NULL:  0
Number of ? : 0
Number of #VALUE! : 0
======================= PurchaseTimestamp =======================
----------------------- FIRST FIVE -----------------------
0    1253232000
1    1253232000
2    1253232000
3    1253232000
4    1253232000
Name: PurchaseTimestamp, dtype: int64
----------------------- DESCIRBE -----------------------
count    4.147600e+04
mean     1.262260e+09
std      1.796895e+07
min      1.231114e+09
25%      1.247530e+09
50%      1.262045e+09
75%      1.277770e+09
max      1.293667e+09
Name: PurchaseTimestamp, dtype: float64
----------------------- COUNTS -----------------------
Five Most Common:  [1235520000, 1259020800, 1234396800, 1264032000,
1287014400]
Num of NULL:  0
Number of ? : 0
Number of #VALUE! : 0
======================= PurchaseDate =======================
----------------------- FIRST FIVE -----------------------
0    18/09/2009 10:00
1    18/09/2009 10:00
2    18/09/2009 10:00
3    18/09/2009 10:00
4    18/09/2009 10:00
Name: PurchaseDate, dtype: object
----------------------- DESCIRBE -----------------------
count                 41476
unique                  497
top       25/02/2009 10:00
freq                    242
Name: PurchaseDate, dtype: object
```

```
----------------------- COUNTS -----------------------
Five Most Common:  ['25/02/2009 10:00', '12/02/2009 10:00', '24/11/2
009 10:00', '21/01/2010 10:00', '14/10/2010 10:00']
Num of NULL:  0
Number of ? : 0
Number of #VALUE! : 0
======================= Auction =======================
----------------------- FIRST FIVE -----------------------
0    OTHER
1    OTHER
2    OTHER
3    OTHER
4    OTHER
Name: Auction, dtype: object
----------------------- DESCIRBE -----------------------
count       41432
unique          3
top       MANHEIM
freq        22168
Name: Auction, dtype: object
----------------------- COUNTS -----------------------
Count List:
 MANHEIM    22168
ADESA      11086
OTHER       8178
Name: Auction, dtype: int64
Num of NULL:  44
Number of ? : 0
Number of #VALUE! : 0
======================= VehYear =======================
----------------------- FIRST FIVE -----------------------
0    2008.0
1    2008.0
2    2008.0
3    2008.0
4    2008.0
Name: VehYear, dtype: float64
----------------------- DESCIRBE -----------------------
count    41432.000000
mean      2005.360615
std          1.730587
min       2001.000000
25%       2004.000000
50%       2005.000000
75%       2007.000000
max       2010.000000
Name: VehYear, dtype: float64
----------------------- COUNTS -----------------------
Count List:
 2006.0    9630
2005.0    8682
2007.0    6514
2004.0    5792
2008.0    4177
2003.0    3554
2002.0    1879
2001.0     816
2009.0     387
2010.0       1
Name: VehYear, dtype: int64
Num of NULL:  44
```

```
Number of ? : 0
Number of #VALUE! : 0
====================== Make ======================
---------------------- FIRST FIVE ----------------------
0        DODGE
1        DODGE
2     CHRYSLER
3    CHEVROLET
4        DODGE
Name: Make, dtype: object
---------------------- DESCIRBE ----------------------
count         41432
unique           30
top       CHEVROLET
freq           9548
Name: Make, dtype: object
---------------------- COUNTS ----------------------
Count List:
 CHEVROLET     9548
DODGE         7385
FORD          6458
CHRYSLER      5259
PONTIAC       2355
KIA           1337
SATURN        1245
NISSAN        1186
JEEP           985
HYUNDAI        957
SUZUKI         842
TOYOTA         664
MITSUBISHI     569
MAZDA          532
MERCURY        527
BUICK          413
GMC            351
HONDA          263
OLDSMOBILE     146
ISUZU           82
SCION           77
VOLKSWAGEN      73
LINCOLN         54
INFINITI        27
MINI            19
ACURA           19
CADILLAC        17
SUBARU          17
LEXUS           13
VOLVO           12
Name: Make, dtype: int64
Num of NULL:  44
Number of ? : 0
Number of #VALUE! : 0
====================== Color ======================
---------------------- FIRST FIVE ----------------------
0       RED
1       RED
2    SILVER
3       RED
4    SILVER
Name: Color, dtype: object
---------------------- DESCIRBE ----------------------
```

```
count      41432
unique        17
top       SILVER
freq        8541
Name: Color, dtype: object
----------------------- COUNTS -----------------------
Count List:
 SILVER       8541
WHITE        6890
BLUE         5855
BLACK        4392
GREY         4248
RED          3661
GOLD         3059
GREEN        1796
MAROON       1039
BEIGE         894
ORANGE        255
BROWN         249
PURPLE        205
YELLOW        141
OTHER         136
NOT AVAIL      65
?               6
Name: Color, dtype: int64
Num of NULL:  44
Number of ? : 6
Number of #VALUE! : 0
======================= Transmission =======================
----------------------- FIRST FIVE -----------------------
0    AUTO
1    AUTO
2    AUTO
3    AUTO
4    AUTO
Name: Transmission, dtype: object
----------------------- DESCIRBE -----------------------
count     41432
unique        4
top        AUTO
freq      39930
Name: Transmission, dtype: object
----------------------- COUNTS -----------------------
Count List:
 AUTO      39930
MANUAL     1495
?             6
Manual        1
Name: Transmission, dtype: int64
Num of NULL:  44
Number of ? : 6
Number of #VALUE! : 0
======================= WheelTypeID =======================
----------------------- FIRST FIVE -----------------------
0    2
1    2
2    2
3    2
4    2
Name: WheelTypeID, dtype: object
----------------------- DESCIRBE -----------------------
```

```
count     41432
unique        5
top           1
freq      20426
Name: WheelTypeID, dtype: object
----------------------- COUNTS -----------------------
Count List:
 1    20426
2    18791
?     1775
3      437
0        3
Name: WheelTypeID, dtype: int64
Num of NULL:  44
Number of ? : 1775
Number of #VALUE! : 0
======================= WheelType =======================
----------------------- FIRST FIVE -----------------------
0    Covers
1    Covers
2    Covers
3    Covers
4    Covers
Name: WheelType, dtype: object
----------------------- DESCIRBE -----------------------
count     41380
unique        4
top       Alloy
freq      20406
Name: WheelType, dtype: object
----------------------- COUNTS -----------------------
Count List:
 Alloy      20406
Covers     18761
?           1777
Special      436
Name: WheelType, dtype: int64
Num of NULL:  96
Number of ? : 1777
Number of #VALUE! : 0
======================= VehOdo =======================
----------------------- FIRST FIVE -----------------------
0    51099.0
1    48542.0
2    46318.0
3    50413.0
4    50199.0
Name: VehOdo, dtype: float64
----------------------- DESCIRBE -----------------------
count     41432.000000
mean      71300.010427
std       14724.041171
min         577.000000
25%       61578.000000
50%       73128.500000
75%       82259.250000
max      480444.000000
Name: VehOdo, dtype: float64
----------------------- COUNTS -----------------------
Five Most Common:  [84675.0, 85884.0, 67464.0, 72101.0, 79600.0]
Num of NULL:  44
```

```
Number of ? : 0
Number of #VALUE! : 0
======================= Nationality =======================
----------------------- FIRST FIVE -----------------------
0    AMERICAN
1    AMERICAN
2    AMERICAN
3    AMERICAN
4    AMERICAN
Name: Nationality, dtype: object
----------------------- DESCIRBE -----------------------
count        41432
unique           6
top       AMERICAN
freq         34616
Name: Nationality, dtype: object
----------------------- COUNTS -----------------------
Count List:
 AMERICAN          34616
OTHER ASIAN        4474
TOP LINE ASIAN     2110
USA                 125
OTHER               104
?                     3
Name: Nationality, dtype: int64
Num of NULL:  44
Number of ? : 3
Number of #VALUE! : 0
======================= Size =======================
----------------------- FIRST FIVE -----------------------
0     MEDIUM
1     MEDIUM
2     MEDIUM
3    COMPACT
4     MEDIUM
Name: Size, dtype: object
----------------------- DESCIRBE -----------------------
count      41432
unique        13
top       MEDIUM
freq       17540
Name: Size, dtype: object
----------------------- COUNTS -----------------------
Count List:
 MEDIUM         17540
LARGE           4968
MEDIUM SUV      4569
COMPACT         4035
VAN             3367
LARGE TRUCK     1897
SMALL SUV       1332
SPECIALTY        998
CROSSOVER        974
LARGE SUV        830
SMALL TRUCK      494
SPORTS           425
?                  3
Name: Size, dtype: int64
Num of NULL:  44
Number of ? : 3
Number of #VALUE! : 0
```

```
====================== TopThreeAmericanName ======================
---------------------- FIRST FIVE ----------------------
0    CHRYSLER
1    CHRYSLER
2    CHRYSLER
3          GM
4    CHRYSLER
Name: TopThreeAmericanName, dtype: object
---------------------- DESCIRBE ----------------------
count     41432
unique        5
top          GM
freq      14075
Name: TopThreeAmericanName, dtype: object
---------------------- COUNTS ----------------------
Count List:
 GM          14075
CHRYSLER    13627
FORD         7039
OTHER        6688
?               3
Name: TopThreeAmericanName, dtype: int64
Num of NULL:  44
Number of ? : 3
Number of #VALUE! : 0
====================== MMRAcquisitionAuctionAveragePrice ==========
=============
---------------------- FIRST FIVE ----------------------
0    8566
1    8566
2    8835
3    7165
4    8566
Name: MMRAcquisitionAuctionAveragePrice, dtype: object
---------------------- DESCIRBE ----------------------
count     41416
unique     9271
top           0
freq        502
Name: MMRAcquisitionAuctionAveragePrice, dtype: object
---------------------- COUNTS ----------------------
Five Most Common:  ['0', '5480', '6311', '7811', '7644']
Num of NULL:  60
Number of ? : 7
Number of #VALUE! : 0
====================== MMRAcquisitionAuctionCleanPrice ============
===========
---------------------- FIRST FIVE ----------------------
0    9325
1    9325
2    9428
3    7770
4    9325
Name: MMRAcquisitionAuctionCleanPrice, dtype: object
---------------------- DESCIRBE ----------------------
count     41429
unique    10010
top           0
freq        415
Name: MMRAcquisitionAuctionCleanPrice, dtype: object
---------------------- COUNTS ----------------------
```

```
Five Most Common:  ['0', '6461', '7450', '1', '8258']
Num of NULL:  47
Number of ? : 7
Number of #VALUE! : 0
======================= MMRAcquisitionRetailAveragePrice ===========
============
----------------------- FIRST FIVE -----------------------
0     9751
1     9751
2    10042
3     8238
4     9751
Name: MMRAcquisitionRetailAveragePrice, dtype: object
----------------------- DESCIRBE -----------------------
count     41429
unique    11070
top           0
freq        502
Name: MMRAcquisitionRetailAveragePrice, dtype: object
----------------------- COUNTS -----------------------
Five Most Common:  ['0', '6418', '7316', '11114', '8756']
Num of NULL:  47
Number of ? : 7
Number of #VALUE! : 0
======================= MMRAcquisitonRetailCleanPrice ==============
=========
----------------------- FIRST FIVE -----------------------
0    10571
1    10571
2    10682
3     8892
4    10571
Name: MMRAcquisitonRetailCleanPrice, dtype: object
----------------------- DESCIRBE -----------------------
count     41327
unique    11583
top           0
freq        501
Name: MMRAcquisitonRetailCleanPrice, dtype: object
----------------------- COUNTS -----------------------
Five Most Common:  ['0', '7478', '8546', '11562', '10103']
Num of NULL:  149
Number of ? : 7
Number of #VALUE! : 0
======================= MMRCurrentAuctionAveragePrice ==============
=========
----------------------- FIRST FIVE -----------------------
0    7781
1    8568
2    8137
3    7074
4    7857
Name: MMRCurrentAuctionAveragePrice, dtype: object
----------------------- DESCIRBE -----------------------
count     41429
unique     9183
top           0
freq        287
Name: MMRCurrentAuctionAveragePrice, dtype: object
----------------------- COUNTS -----------------------
Five Most Common:  ['0', '?', '5480', '6311', '7269']
```

```
Num of NULL:  47
Number of ? : 184
Number of #VALUE! : 0
======================= MMRCurrentAuctionCleanPrice ================
=======
----------------------- FIRST FIVE -----------------------
0    8545
1    9325
2    8733
3    7629
4    8711
Name: MMRCurrentAuctionCleanPrice, dtype: object
----------------------- DESCIRBE -----------------------
count     41429
unique     9890
top           0
freq        206
Name: MMRCurrentAuctionCleanPrice, dtype: object
----------------------- COUNTS -----------------------
Five Most Common:  ['0', '?', '6461', '1', '7450']
Num of NULL:  47
Number of ? : 184
Number of #VALUE! : 0
======================= MMRCurrentRetailAveragePrice ===============
========
----------------------- FIRST FIVE -----------------------
0    11777
1     9753
2     9288
3     8140
4     8986
Name: MMRCurrentRetailAveragePrice, dtype: object
----------------------- DESCIRBE -----------------------
count     41409
unique    10935
top           0
freq        287
Name: MMRCurrentRetailAveragePrice, dtype: object
----------------------- COUNTS -----------------------
Five Most Common:  ['0', '?', '6418', '7316', '8756']
Num of NULL:  67
Number of ? : 184
Number of #VALUE! : 0
======================= MMRCurrentRetailCleanPrice =================
======
----------------------- FIRST FIVE -----------------------
0    12505
1    10571
2     9932
3     8739
4     9908
Name: MMRCurrentRetailCleanPrice, dtype: object
----------------------- DESCIRBE -----------------------
count     41409
unique    11363
top           0
freq        287
Name: MMRCurrentRetailCleanPrice, dtype: object
----------------------- COUNTS -----------------------
Five Most Common:  ['0', '?', '7478', '8546', '10103']
Num of NULL:  67
```

```
Number of ? : 184
Number of #VALUE! : 0
====================== MMRCurrentRetailRatio ======================
=
----------------------- FIRST FIVE -----------------------
0    0.941783287
1    0.922618485
2    0.935159082
3    0.931456688
4    0.906943884
Name: MMRCurrentRetailRatio, dtype: object
----------------------- DESCIRBE -----------------------
count       41116
unique      25870
top       #VALUE!
freq          178
Name: MMRCurrentRetailRatio, dtype: object
----------------------- COUNTS -----------------------
Five Most Common:  ['#VALUE!', '0.858250869', '0.856073017', '0.8666
73265', '0.949268378']
Num of NULL:  360
Number of ? : 0
Number of #VALUE! : 178
====================== PRIMEUNIT ======================
----------------------- FIRST FIVE -----------------------
0    ?
1    ?
2    ?
3    ?
4    ?
Name: PRIMEUNIT, dtype: object
----------------------- DESCIRBE -----------------------
count     41432
unique        3
top           ?
freq      39634
Name: PRIMEUNIT, dtype: object
----------------------- COUNTS -----------------------
Count List:
 ?      39634
NO      1764
YES       34
Name: PRIMEUNIT, dtype: int64
Num of NULL:  44
Number of ? : 39634
Number of #VALUE! : 0
====================== AUCGUART ======================
----------------------- FIRST FIVE -----------------------
0    ?
1    ?
2    ?
3    ?
4    ?
Name: AUCGUART, dtype: object
----------------------- DESCIRBE -----------------------
count     41432
unique        3
top           ?
freq      39634
Name: AUCGUART, dtype: object
----------------------- COUNTS -----------------------
```

```
Count List:
 ?        39634
GREEN     1754
RED         44
Name: AUCGUART, dtype: int64
Num of NULL:  44
Number of ? : 39634
Number of #VALUE! : 0
====================== VNST ======================
---------------------- FIRST FIVE ----------------------
0    NC
1    NC
2    NC
3    NC
4    NC
Name: VNST, dtype: object
---------------------- DESCIRBE ----------------------
count     41432
unique       31
top          TX
freq       9076
Name: VNST, dtype: object
---------------------- COUNTS ----------------------
Count List:
 TX    9076
FL    5250
CO    3623
NC    3594
AZ    3383
CA    3268
OK    2595
SC    1662
TN    1471
GA    1287
VA    1093
MO     758
PA     700
NV     553
IN     486
MS     412
LA     349
NJ     317
NM     239
KY     230
AL     179
IL     165
UT     165
WV     137
WA     136
OR     136
NH      97
NE      26
OH      25
ID      14
NY       6
Name: VNST, dtype: int64
Num of NULL:  44
Number of ? : 0
Number of #VALUE! : 0
====================== VehBCost ======================
---------------------- FIRST FIVE ----------------------
```

```
0    7800
1    7800
2    7800
3    6000
4    7800
Name: VehBCost, dtype: object
----------------------- DESCIRBE -----------------------
count     41432
unique     1869
top        7500
freq        459
Name: VehBCost, dtype: object
----------------------- COUNTS ----------------------
Five Most Common:  ['7500', '6500', '7800', '7200', '7000']
Num of NULL:  44
Number of ? : 29
Number of #VALUE! : 0
======================= IsOnlineSale =======================
----------------------- FIRST FIVE ----------------------
0    0
1    0
2    0
3    0
4    0
Name: IsOnlineSale, dtype: object
----------------------- DESCIRBE -----------------------
count     41432.0
unique        8.0
top           0.0
freq      31368.0
Name: IsOnlineSale, dtype: float64
----------------------- COUNTS ----------------------
Count List:
 0.0     31368
0        8572
1.0       753
-1.0      601
1         134
?           2
4.0         1
2.0         1
Name: IsOnlineSale, dtype: int64
Num of NULL:  44
Number of ? : 2
Number of #VALUE! : 0
======================= WarrantyCost =======================
----------------------- FIRST FIVE ----------------------
0    920.0
1    834.0
2    834.0
3    671.0
4    920.0
Name: WarrantyCost, dtype: float64
----------------------- DESCIRBE -----------------------
count    41432.000000
mean      1273.050758
std        599.188662
min        462.000000
25%        834.000000
50%       1155.000000
75%       1623.000000
```

```
max       7498.000000
Name: WarrantyCost, dtype: float64
----------------------- COUNTS -----------------------
Five Most Common:  [920.0, 1974.0, 2152.0, 1215.0, 1389.0]
Num of NULL:  44
Number of ? : 0
Number of #VALUE! : 0
======================= ForSale =======================
----------------------- FIRST FIVE -----------------------
0    Yes
1    Yes
2    Yes
3    Yes
4    Yes
Name: ForSale, dtype: object
----------------------- DESCIRBE -----------------------
count     41476
unique        6
top         Yes
freq      27402
Name: ForSale, dtype: object
----------------------- COUNTS -----------------------
Count List:
 Yes    27402
YES     8544
yes     5524
?          3
No         2
0          1
Name: ForSale, dtype: int64
Num of NULL:  0
Number of ? : 3
Number of #VALUE! : 0
======================= IsBadBuy =======================
----------------------- FIRST FIVE -----------------------
0    0
1    0
2    0
3    0
4    0
Name: IsBadBuy, dtype: int64
----------------------- DESCIRBE -----------------------
count    41476.000000
mean         0.129497
std          0.335753
min          0.000000
25%          0.000000
50%          0.000000
75%          0.000000
max          1.000000
Name: IsBadBuy, dtype: float64
----------------------- COUNTS -----------------------
Count List:
 0    36105
1     5371
Name: IsBadBuy, dtype: int64
Num of NULL:  0
Number of ? : 0
Number of #VALUE! : 0
```

In [8]:

```
if NEW_STATEGY:

    class filling_method():
        MOST_COMMON = "MOST_COMMON"
        MEAN = "MEAN"
        CERTAIN_VALUE = "CERTAIN_VALUE"

    def replaceFunc(colName):
        for replaced, target in preprocessStrategy[colName]['replace_pairs']:
            df[colName].replace(replaced, target, inplace=True)

    def removeOutlier(colName):  # FOR THE INTERVAL ONLY
        global df
        df = df[df[colName] < df[colName].quantile(0.999)]

    def replacingValueCol(colName):
        for replaced in preprocessStrategy[colName]['replaced_vals']:
            print("In the Column: " + str(colName) + " : " + str(len(
                df[df[colName] == replaced])) + ", " + str(replaced) + "have bee
n replaced by null")
            # Replacing the null in this process #Inplacing for saving the memor
y
            df[colName].replace(replaced, float('nan'), inplace=True)

    def loweringCol(colName):
        df[colName] = df[colName].str.lower()

    def fillingTheNullValue(colName):  # method can be ["MEAN", "MOST_COMMON"]
        if preprocessStrategy[colName]['filling_method'] == filling_method.MEAN:
            df[colName] = df[colName].astype('float')
            df[colName].fillna(df[colName].astype(
                'float').mean(), inplace=True)
        elif preprocessStrategy[colName]['filling_method'] == filling_method.MOS
T_COMMON:
            df[colName] = df[colName].astype('category')
            df[colName].fillna(df[colName].astype(
                'category').describe()['top'], inplace=True)
        elif preprocessStrategy[colName]['filling_method'] == filling_method.CER
TAIN_VALUE:
            df[colName] = df[colName].astype('category')
            df[colName] = df[colName].cat.add_categories(
                [preprocessStrategy[colName]['filling_value']])
            df[colName].fillna(preprocessStrategy[colName]
                               ['filling_value'], inplace=True)

    def filterOutRareValue(colName):

        def checkingKeepValue(v, savingValues):
            if v in savingValues:
                return v
            return "LESS_FREQ"

        k = [v for v in df[colName].value_counts().values if v >
             preprocessStrategy[colName]['min_freq']]
        savingValues = df[colName].value_counts().keys()[:len(k)]

        df[colName] = [checkingKeepValue(v, savingValues) for v in df[colName]]
```

```
    def changeToType(colName):
        df[colName] = df[colName].astype(
            preprocessStrategy[colName]['changeToType'])

    def newData_prep(df):
        '''
        For Preprocessing through the whole dictionary
        '''
        df.drop(drop_cols, axis=1, inplace=True)

        for colName in df.columns:  # df.columns:

            print("Preprocess the col: " + colName)

            for stra in preprocessStrategy[colName]['strategies']:
                if not stra:
                    continue
                stra(colName)

        if not using_cat:
            df['MMRCurrentRetailRatio'] = df['MMRCurrentRetailAveragePrice'] / \
                (df['MMRCurrentRetailCleanPrice']+1e-8)  # Prvent divided by 0

        return df

    preprocessStrategy = defaultdict(dict)

    preprocessStrategy['Auction'] = {
        "strategies":
            [
                replacingValueCol,
                loweringCol,
                fillingTheNullValue,
            ],
        "replaced_vals": ['?'],
        "filling_method": filling_method.MOST_COMMON
    }

    preprocessStrategy['VehYear'] = {
        "strategies":
            [
                fillingTheNullValue,
            ],
        "filling_method": filling_method.CERTAIN_VALUE,
        "filling_value": "UNKNOWN_VALUE"
    }

    preprocessStrategy['Make'] = {
        "strategies":
            [
                loweringCol,
                fillingTheNullValue,
            ],
        "filling_method": filling_method.MOST_COMMON
    }

    preprocessStrategy['Color'] = {
        "strategies":
            [
                loweringCol,
                replacingValueCol,
```

```
                fillingTheNullValue,
            ],
        "replaced_vals": ['?'],
        "filling_method": filling_method.MOST_COMMON
    }

    preprocessStrategy['Transmission'] = {
        "strategies":
            [
                loweringCol,
                replacingValueCol,
                fillingTheNullValue,
            ],
        "replaced_vals": ['?'],
        "filling_method": filling_method.MOST_COMMON
    }

    preprocessStrategy['WheelTypeID'] = {
        "strategies":
            [
                fillingTheNullValue,
            ],
        "filling_method": filling_method.MOST_COMMON
    }

    preprocessStrategy['WheelType'] = {
        "strategies":
            [
                loweringCol,
                fillingTheNullValue,
            ],
        "filling_method": filling_method.MOST_COMMON
    }

    preprocessStrategy['VehOdo'] = {
        "strategies":
            [
                fillingTheNullValue,
            ],
        "filling_method": filling_method.MEAN
    }

    preprocessStrategy['Nationality'] = {  # Should I merge USA with AMERICAN?
        "strategies":
            [
                replaceFunc,
                loweringCol,
                replacingValueCol,
                fillingTheNullValue,
            ],
        "replaced_vals": ['?'],
        "filling_method": filling_method.MOST_COMMON,
        "replace_pairs": [("USA", "AMERICAN")]

    }

    preprocessStrategy['Size'] = {
        "strategies":
            [
                loweringCol,
                replacingValueCol,
```

```
                fillingTheNullValue,
            ],
        "replaced_vals": ['?'],
        "filling_method": filling_method.MOST_COMMON

    }

    preprocessStrategy['TopThreeAmericanName'] = {
        "strategies":
            [
                loweringCol,
                replacingValueCol,
                fillingTheNullValue,
            ],
        "replaced_vals": ['?'],
        "filling_method": filling_method.MOST_COMMON
    }

    preprocessStrategy['TopThreeAmericanName'] = {
        "strategies":
            [
                loweringCol,
                replacingValueCol,
                fillingTheNullValue,
            ],
        "replaced_vals": ['?'],
        "filling_method": filling_method.MOST_COMMON
    }

    preprocessStrategy['MMRAcquisitionAuctionAveragePrice'] = {
        "strategies":
            [
                replacingValueCol,
                fillingTheNullValue,
            ],
        "replaced_vals": ['?'],
        "filling_method": filling_method.MEAN
    }

    preprocessStrategy['MMRAcquisitionAuctionCleanPrice'] = {
        "strategies":
            [
                replacingValueCol,
                fillingTheNullValue,
            ],
        "replaced_vals": ['?'],
        "filling_method": filling_method.MEAN
    }

    preprocessStrategy['MMRAcquisitionRetailAveragePrice'] = {
        "strategies":
            [
                replacingValueCol,
                fillingTheNullValue,
            ],
        "replaced_vals": ['?'],
        "filling_method": filling_method.MEAN
    }

    preprocessStrategy['MMRAcquisitonRetailCleanPrice'] = {
        "strategies":
```

```
            [
                replacingValueCol,
                fillingTheNullValue,
            ],
        "replaced_vals": ['?'],
        "filling_method": filling_method.MEAN
    }

    ################################################################

    int_stra = {
        "strategies":
            [
                replacingValueCol,
                fillingTheNullValue,
            ],
        "replaced_vals": ['?', '#VALUE!'],  # GOT 184 '?'
        "filling_method": filling_method.MEAN,
    }

    cat_stra = {  # HOW DO WE DEAL WITH ? in this column
        "strategies":
            [
                filterOutRareValue,
                fillingTheNullValue,
            ],
        #     "replaced_vals": ['?'], # GOT 184 '?'
        "filling_method": filling_method.CERTAIN_VALUE,
        "filling_value": 'NULL',
        "min_freq": 50
    }

    preprocessStrategy['MMRCurrentAuctionAveragePrice'] \
        = preprocessStrategy['MMRCurrentAuctionCleanPrice'] \
        = preprocessStrategy['MMRCurrentRetailAveragePrice'] \
        = preprocessStrategy['MMRCurrentRetailCleanPrice'] \
        = preprocessStrategy['MMRCurrentRetailRatio'] \
        = cat_stra if using_cat else int_stra

    ################################################################

    preprocessStrategy['PRIMEUNIT'] = {  # HOW DO WE DEAL WITH ? in this column
        "strategies":
            [
                loweringCol,
                fillingTheNullValue,
            ],
        #     "replaced_vals": ['?'], # GOT 184 '?'
        "filling_method": filling_method.CERTAIN_VALUE,
        "filling_value": 'NULL',
    }

    preprocessStrategy['AUCGUART'] = {  # HOW DO WE DEAL WITH ? in this column
        "strategies":
            [
                loweringCol,
                fillingTheNullValue,
            ],
        #     "replaced_vals": ['?'], # GOT 184 '?'
        "filling_method": filling_method.CERTAIN_VALUE,
        "filling_value": 'NULL',
```

```
    }

    preprocessStrategy['VNST'] = {  # HOW DO WE DEAL WITH ? in this column
        "strategies":
            [
                loweringCol,
                fillingTheNullValue,
            ],
        #     "replaced_vals": ['?'], # GOT 184 '?'
        "filling_method": filling_method.CERTAIN_VALUE,
        "filling_value": 'NULL',
    }

    preprocessStrategy['VehBCost'] = {  # HOW DO WE DEAL WITH ? in this column
        "strategies":
            [
                replacingValueCol,

                fillingTheNullValue,
            ],
        "replaced_vals": ['?'],  # GOT 184 '?'
        "filling_method": filling_method.MEAN
    }

    preprocessStrategy['IsOnlineSale'] = {  # HOW DO WE DEAL WITH ? in this colu
mn
        "strategies":
            [
                replacingValueCol,
                changeToType,
                fillingTheNullValue,
            ],
        "replaced_vals": ['?', 2.0, 4.0],  # GOT 184 '?'
        "filling_method": filling_method.MOST_COMMON,
        "changeToType": 'float'
    }

    preprocessStrategy['WarrantyCost'] = {  # HOW DO WE DEAL WITH ? in this colu
mn
        "strategies":
            [
                fillingTheNullValue,
            ],
        "replaced_vals": ['?'],  # GOT 184 '?'
        "filling_method": filling_method.MEAN,
    }

    preprocessStrategy['ForSale'] = {  # HOW DO WE DEAL WITH ? in this column
        "strategies":
            [
                loweringCol,
                replacingValueCol,
                fillingTheNullValue,
            ],
        "replaced_vals": ['?', 0],  # GOT 184 '?'
        "filling_method": filling_method.MOST_COMMON,
    }

    # HOW DO WE DEAL WITH ? in this column
    preprocessStrategy['IsBadBuy'] = {"strategies": [None]}
```

```
    newData_prep(df)


else:

    def data_prep(df):
        '''
        For Preprocessing the Data (OLD_METHOD)
        '''

        # Check the replaced values are not in the dataset

        for colName in df.columns:

            if colName in categorial_cols:

                if colName == "IsOnlineSale":
                    df[colName] = df[colName].astype(
                        'float').astype('category')
                    df[colName].fillna(df[colName].astype(
                        'category').describe()['top'], inplace=True)

                # Try to lower the data if the data type is string
                try:
                    df[colName] = df[colName].str.lower()
                except:
                    print(colName, " can't be lowered")

                for replaced in replaced_vals:
                    print("In the Column: " + str(colName) + ": " +
                          str(len(df[df[colName] == replaced])) + " -> " + str(r
eplaced))
                    df[colName].replace(replaced, float('nan'), inplace=True)

                df[colName] = df[colName].astype('category')

                # Replacing the null by the most common category
                df[colName].fillna(df[colName].astype(
                    'category').describe()['top'], inplace=True)

            if colName in interval_cols:

                if colName == "MMRCurrentRetailRatio":  # Dealing with this calc
ulated value at the last
                    continue

                for replaced in replaced_vals:
                    print("In the Column: " + str(colName) + ": " +
                          str(len(df[df[colName] == replaced])) + " -> " + str(r
eplaced))
                    df[colName].replace(replaced, float('nan'), inplace=True)

                df[colName] = df[colName].astype('float')

                # Removing outlier
                df = df[df[colName] < df[colName].quantile(0.999)]

                # Replacing the null by the mean
                df[colName].fillna(df[colName].astype(
                    'float').mean(), inplace=True)
```

```
    df['MMRCurrentRetailRatio'] = df['MMRCurrentRetailAveragePrice'] / \
        (df['MMRCurrentRetailCleanPrice']+1e-8)  # Prvent divided by 0

    df.drop(drop_cols, axis=1, inplace=True)

    return df

df = data_prep(df)
```

```
Preprocess the col: Auction
In the Column: Auction : 0, ?have been replaced by null
Preprocess the col: VehYear
Preprocess the col: Make
Preprocess the col: Color
In the Column: Color : 6, ?have been replaced by null
Preprocess the col: Transmission
In the Column: Transmission : 6, ?have been replaced by null
Preprocess the col: WheelTypeID
Preprocess the col: WheelType
Preprocess the col: VehOdo
Preprocess the col: Nationality
In the Column: Nationality : 3, ?have been replaced by null
Preprocess the col: Size
In the Column: Size : 3, ?have been replaced by null
Preprocess the col: TopThreeAmericanName
In the Column: TopThreeAmericanName : 3, ?have been replaced by null
Preprocess the col: MMRAcquisitionAuctionAveragePrice
In the Column: MMRAcquisitionAuctionAveragePrice : 7, ?have been rep
laced by null
Preprocess the col: MMRAcquisitionAuctionCleanPrice
In the Column: MMRAcquisitionAuctionCleanPrice : 7, ?have been repla
ced by null
Preprocess the col: MMRAcquisitionRetailAveragePrice
In the Column: MMRAcquisitionRetailAveragePrice : 7, ?have been repl
aced by null
Preprocess the col: MMRAcquisitonRetailCleanPrice
In the Column: MMRAcquisitonRetailCleanPrice : 7, ?have been replace
d by null
Preprocess the col: MMRCurrentAuctionAveragePrice
In the Column: MMRCurrentAuctionAveragePrice : 184, ?have been repla
ced by null
In the Column: MMRCurrentAuctionAveragePrice : 0, #VALUE!have been r
eplaced by null
Preprocess the col: MMRCurrentAuctionCleanPrice
In the Column: MMRCurrentAuctionCleanPrice : 184, ?have been replace
d by null
In the Column: MMRCurrentAuctionCleanPrice : 0, #VALUE!have been rep
laced by null
Preprocess the col: MMRCurrentRetailAveragePrice
In the Column: MMRCurrentRetailAveragePrice : 184, ?have been replac
ed by null
In the Column: MMRCurrentRetailAveragePrice : 0, #VALUE!have been re
placed by null
Preprocess the col: MMRCurrentRetailCleanPrice
In the Column: MMRCurrentRetailCleanPrice : 184, ?have been replaced
by null
In the Column: MMRCurrentRetailCleanPrice : 0, #VALUE!have been repl
aced by null
Preprocess the col: MMRCurrentRetailRatio
In the Column: MMRCurrentRetailRatio : 0, ?have been replaced by nul
l
In the Column: MMRCurrentRetailRatio : 178, #VALUE!have been replace
d by null
Preprocess the col: PRIMEUNIT
Preprocess the col: AUCGUART
Preprocess the col: VNST
Preprocess the col: VehBCost
In the Column: VehBCost : 29, ?have been replaced by null
Preprocess the col: IsOnlineSale
In the Column: IsOnlineSale : 2, ?have been replaced by null
```

```
In the Column: IsOnlineSale : 1, 2.0have been replaced by null
In the Column: IsOnlineSale : 1, 4.0have been replaced by null
Preprocess the col: WarrantyCost
Preprocess the col: ForSale
In the Column: ForSale : 3, ?have been replaced by null
In the Column: ForSale : 0, 0have been replaced by null
Preprocess the col: IsBadBuy
```

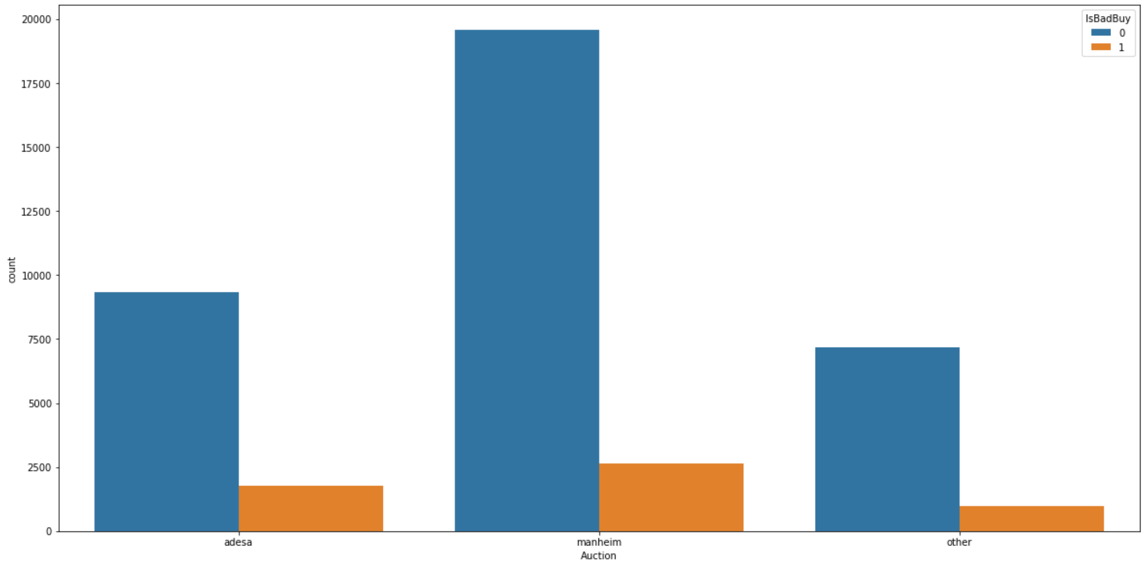
3. Can you identify any clear patterns by initial exploration of the data using histogram or box plot?

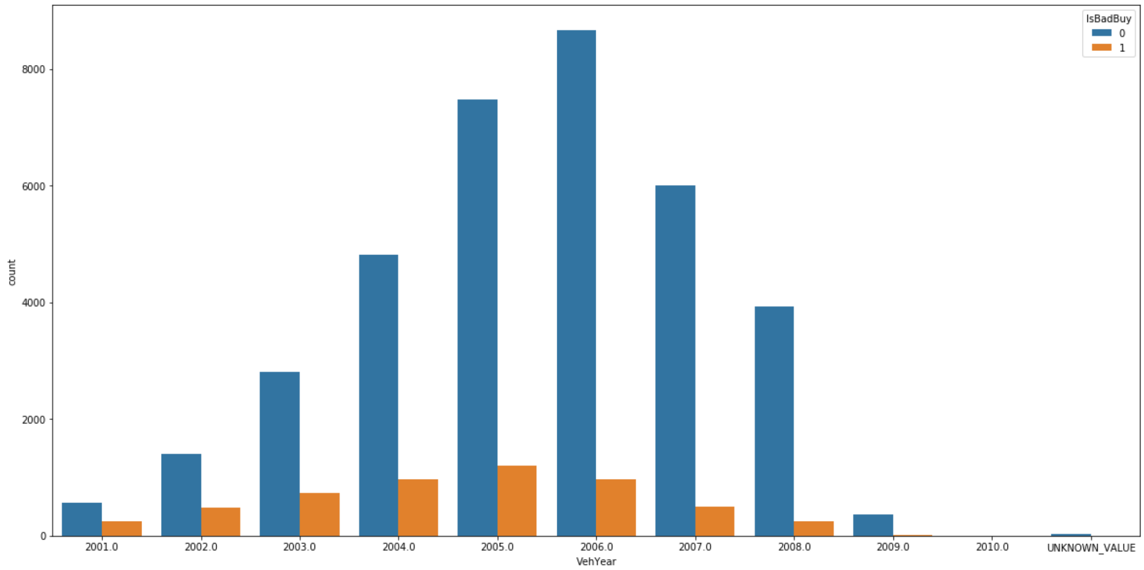
In [9]:

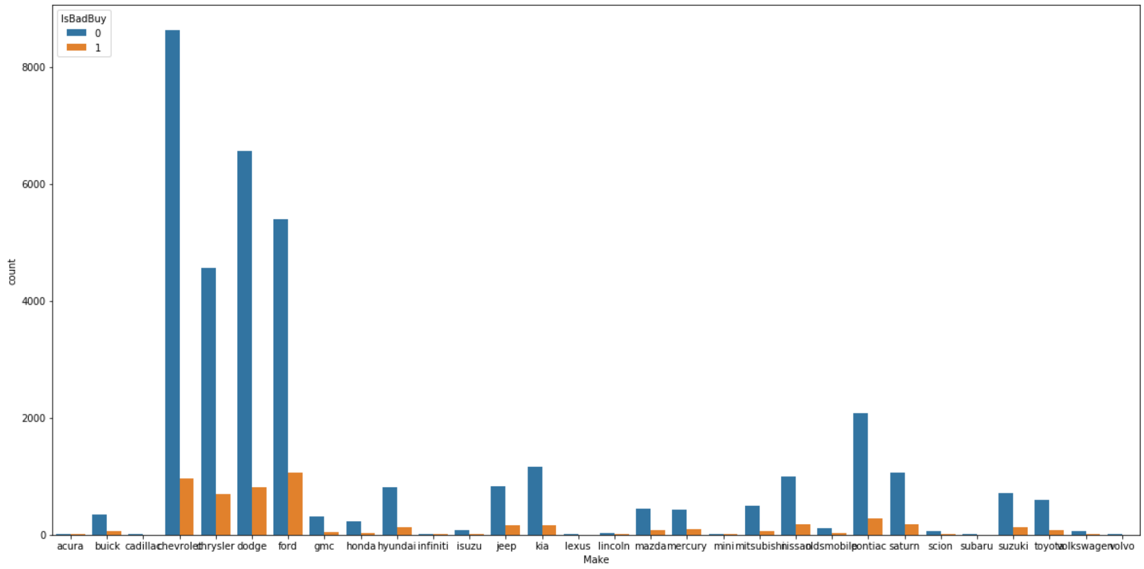
```
def plotAllCols (df):
    for colName in df.columns:
        plt.figure(figsize=(20,10))
        if colName in categorial_cols:
            ### if it's categorial column, plot hist diagram
            sns.countplot(x=colName, data = df, hue="IsBadBuy")
        elif colName in interval_cols:
            ### if it's interval column, plot box diagram
            sns.boxplot(x="IsBadBuy", y=colName, data = df )
```

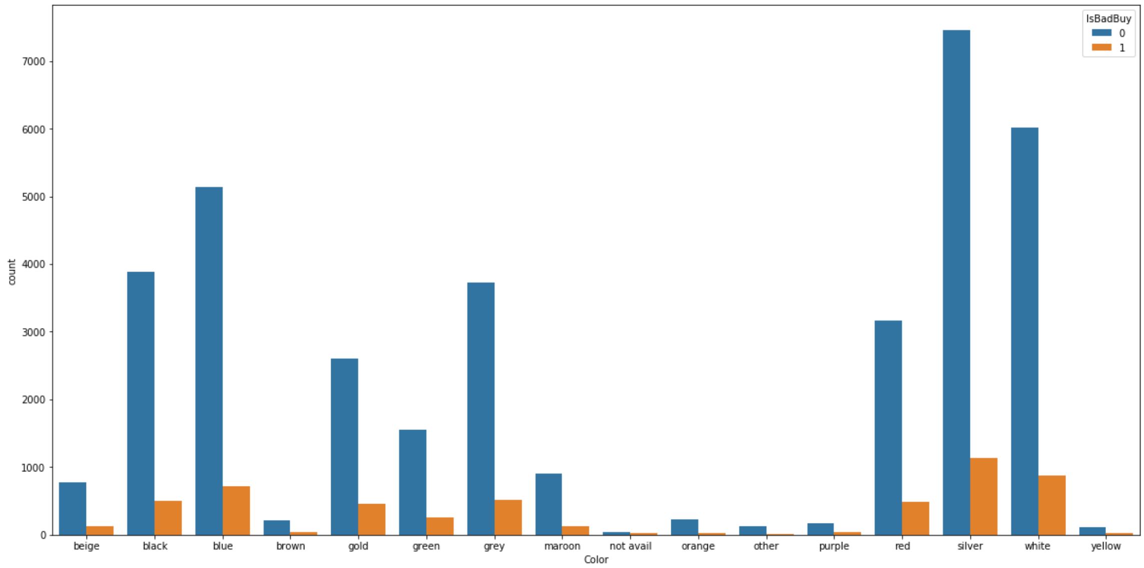
In [10]:

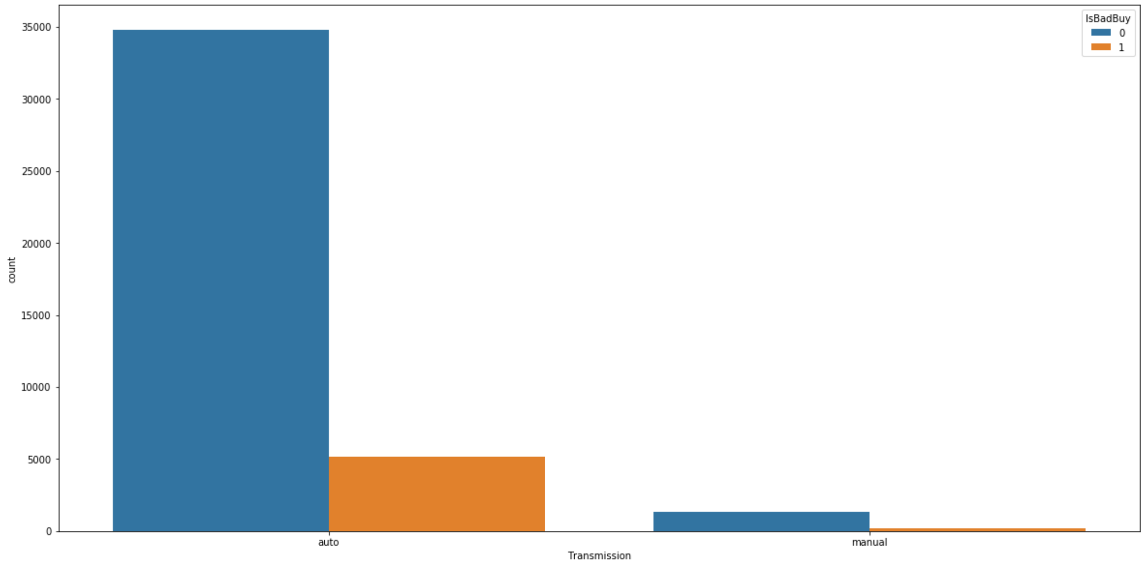
```
plotAllCols(df)
```

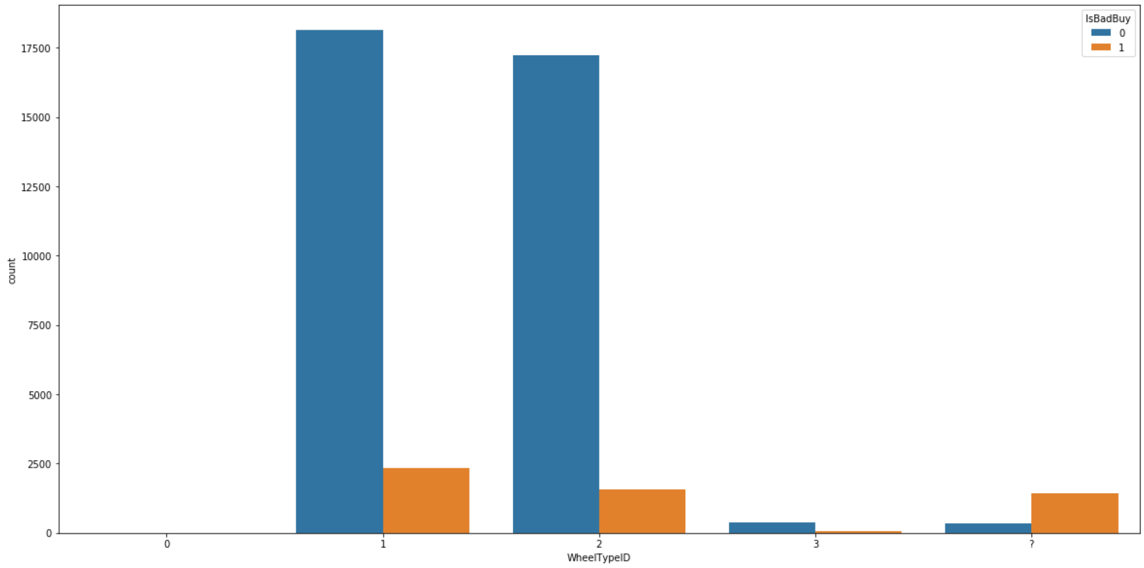

IsBadBuy
0
1
count
20000
17500
15000
12500
10000
7500
5000
2500
0
adesa
manheim
other
Auction

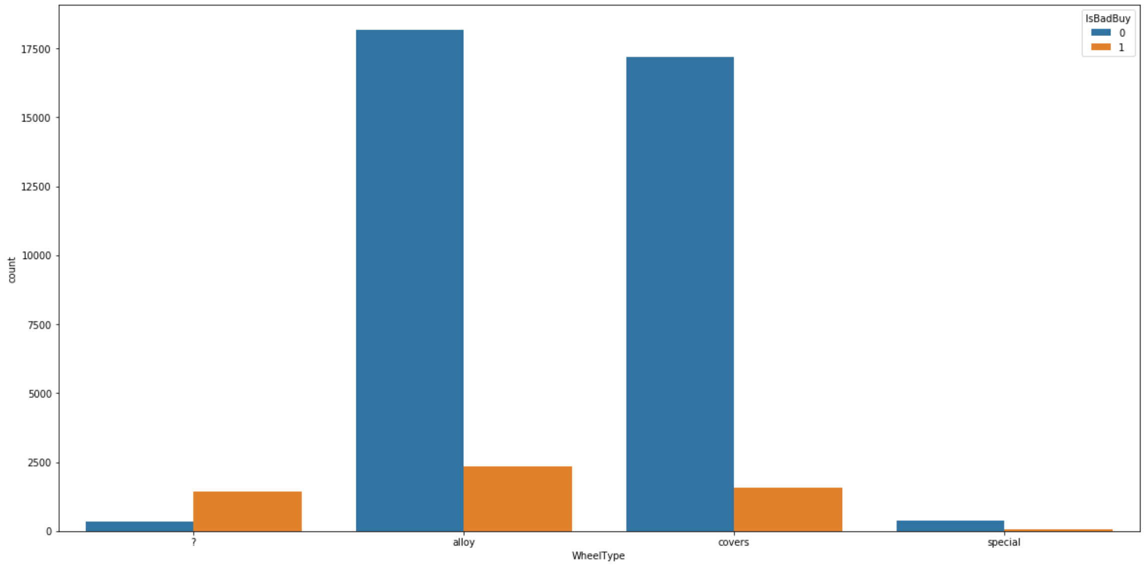

IsBadBuy
0
1
count
8000
6000
4000
2000
0
2001.0
2002.0
2003.0
2004.0
2005.0
2006.0
2007.0
2008.0
2009.0
2010.0
UNKNOWN_VALUE
VehYear

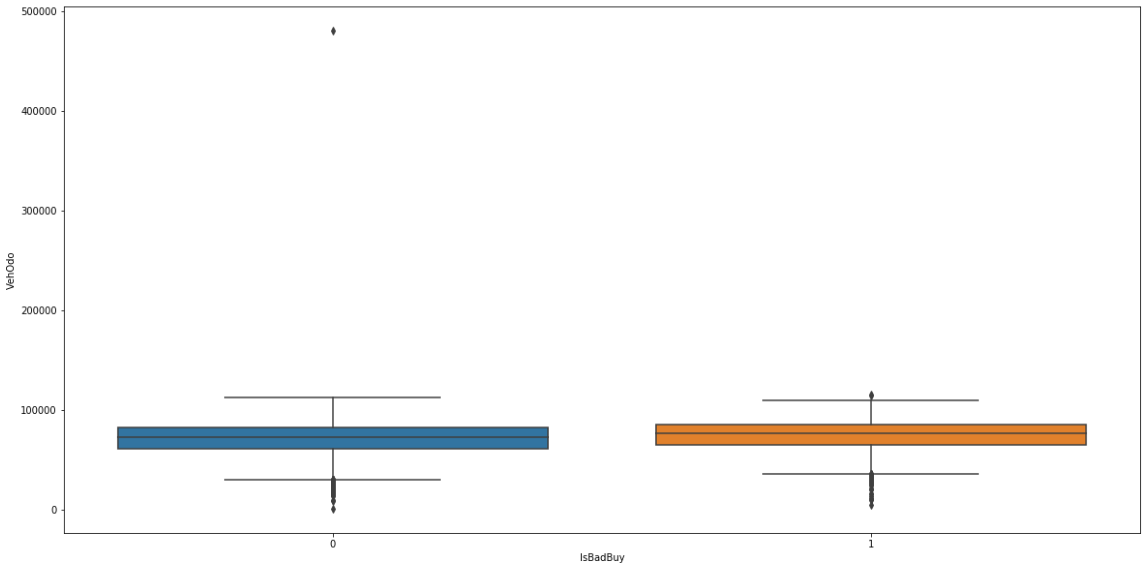

IsBadBuy
0
1
count
8000
6000
4000
2000
0
acura
buick
cadillac
chevrolet
chrysler
dodge
ford
gmc
honda
hyundai
infiniti
isuzu
jeep
kia
lexus
lincoln
mazda
mercury
mini
mitsubishi
nissan
oldsmobile
pontiac
saturn
scion
subaru
suzuki
toyota
volkswagen
volvo
Make

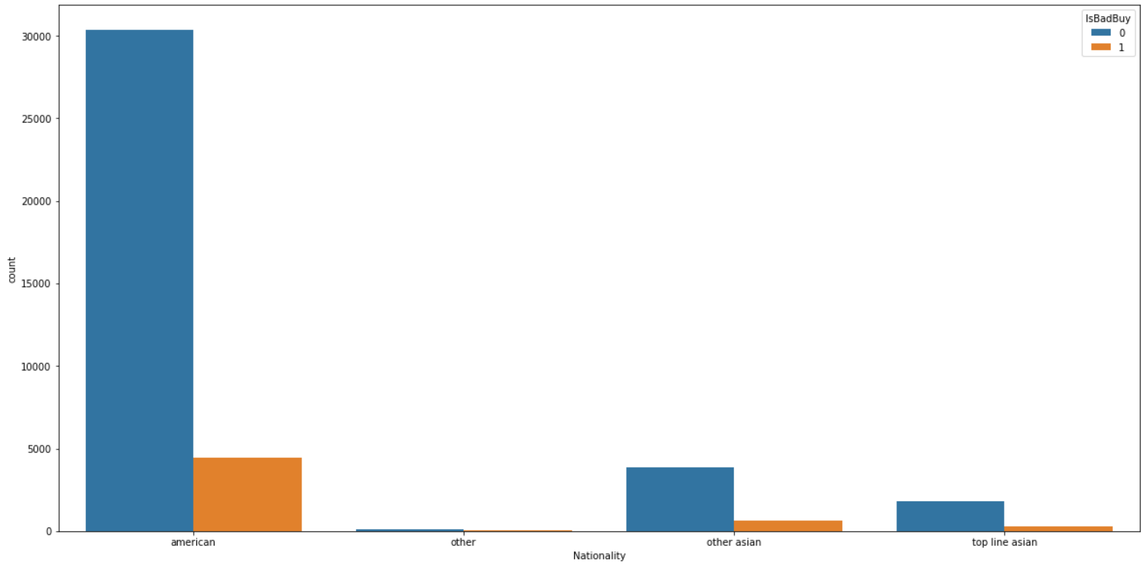

IsBadBuy
0
1
count
7000
6000
5000
4000
3000
2000
1000
0
beige
black
blue
brown
gold
green
grey
maroon
not avail
orange
other
purple
red
silver
white
yellow
Color

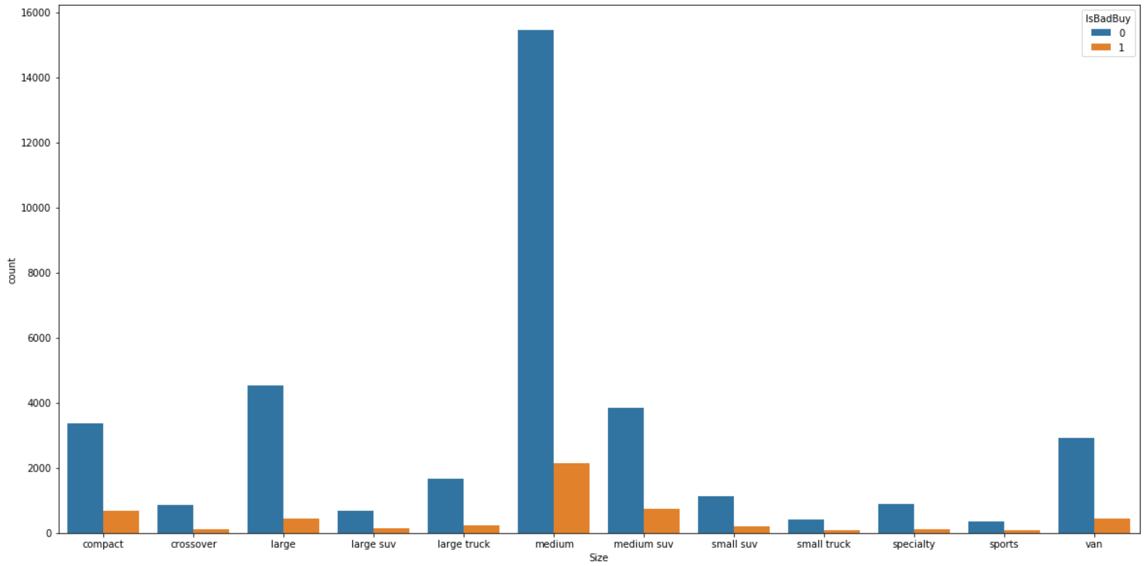

IsBadBuy
0
1
35000
30000
25000
20000
15000
10000
5000
0
count
auto
manual
Transmission

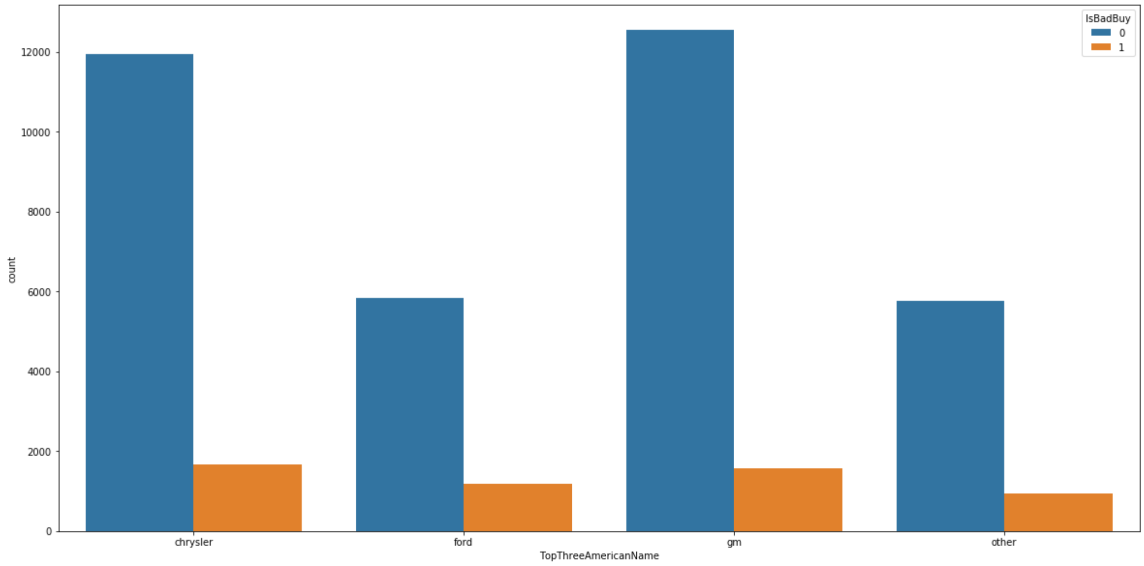

IsBadBuy
0
1
17500
15000
12500
10000
7500
5000
2500
0
count
0
1
2
3
?
WheelTypeID

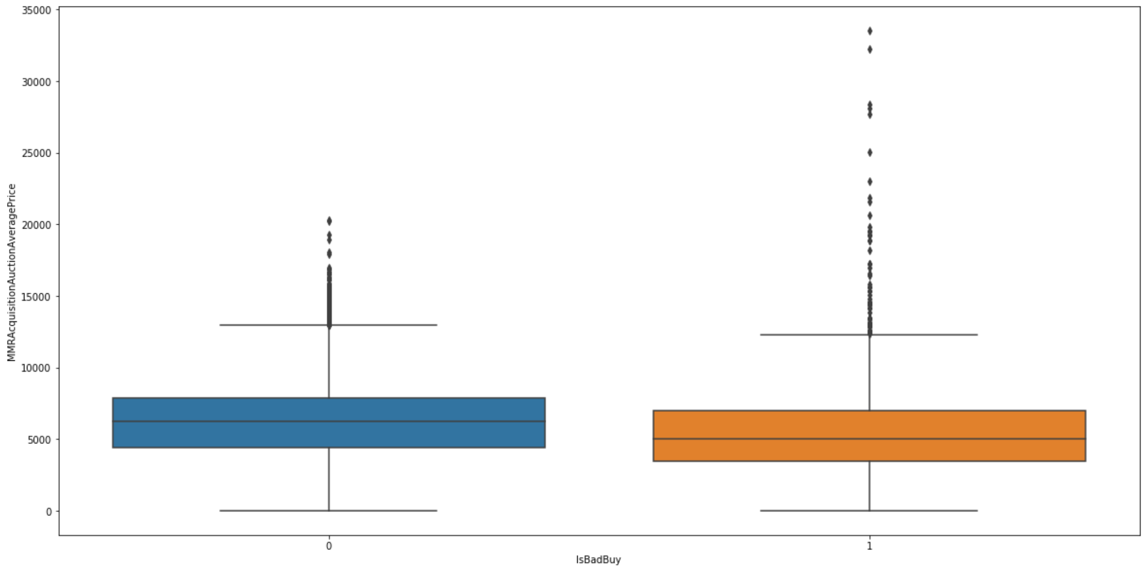

IsBadBuy
0
1
17500
15000
12500
10000
7500
5000
2500
0
count
?
alloy
covers
special
WheelType

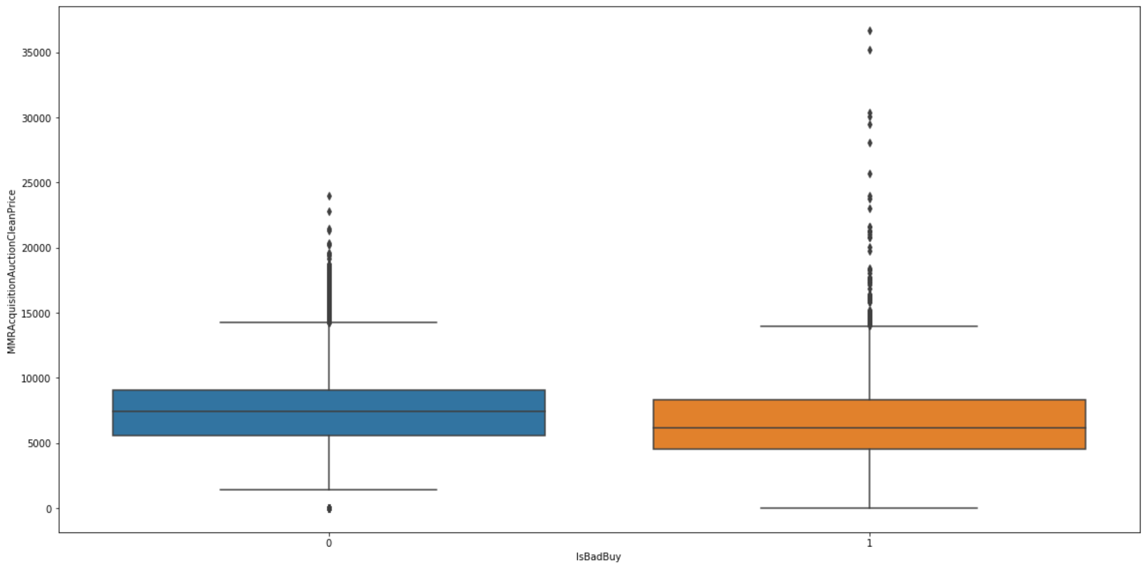

500000
400000
300000
200000
100000
0
VehOdo
0
1
IsBadBuy

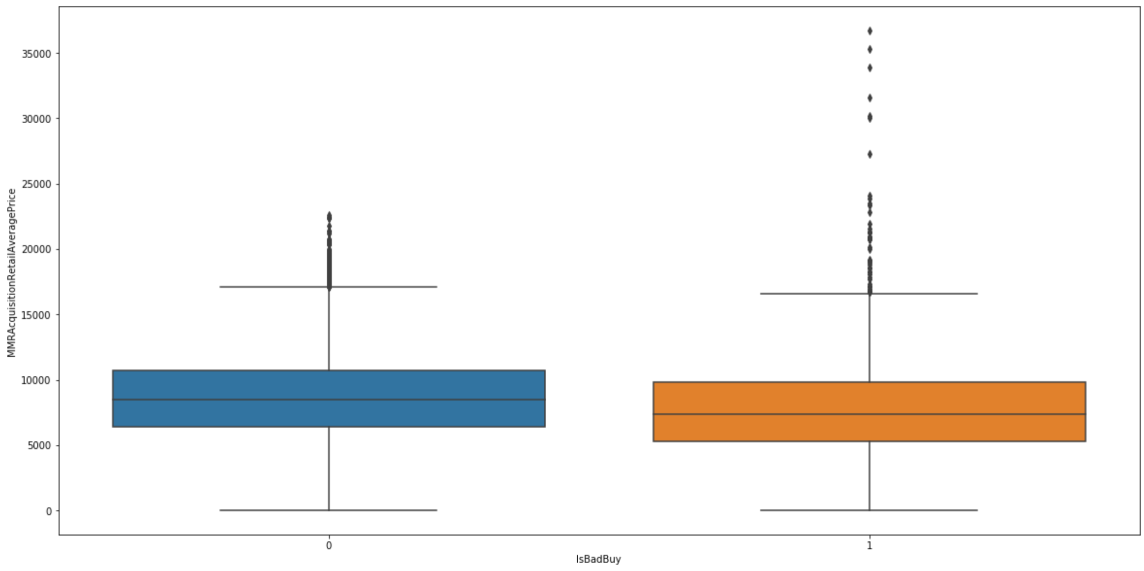

IsBadBuy
0
1
30000
25000
20000
15000
10000
5000
0
count
american
other
other asian
top line asian
Nationality

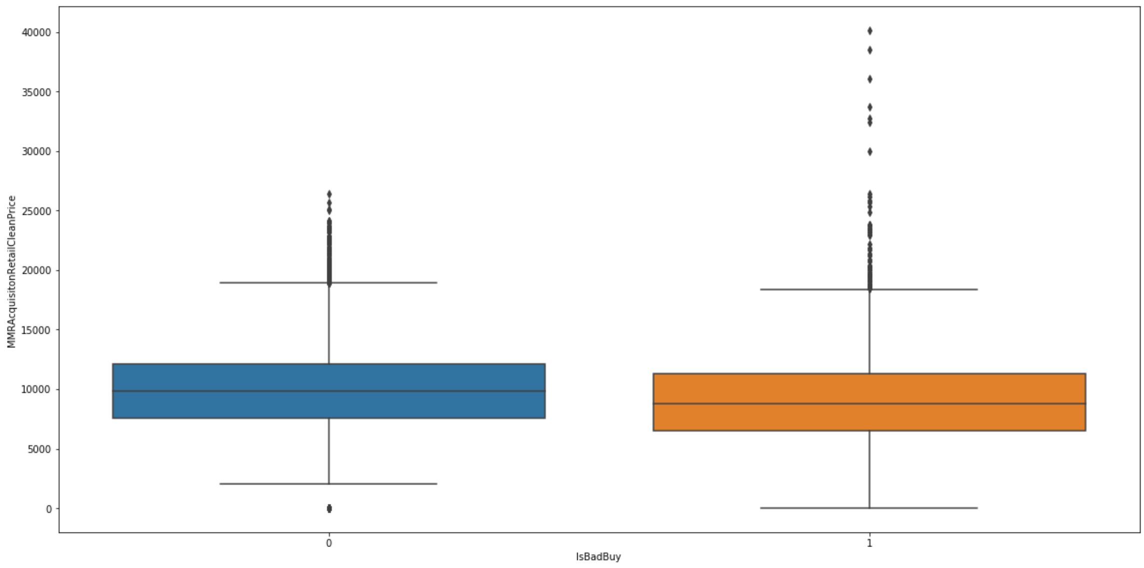

IsBadBuy
0
1
16000
14000
12000
10000
8000
6000
4000
2000
0
count
compact
crossover
large
large suv
large truck
medium
medium suv
small suv
small truck
specialty
sports
van
Size

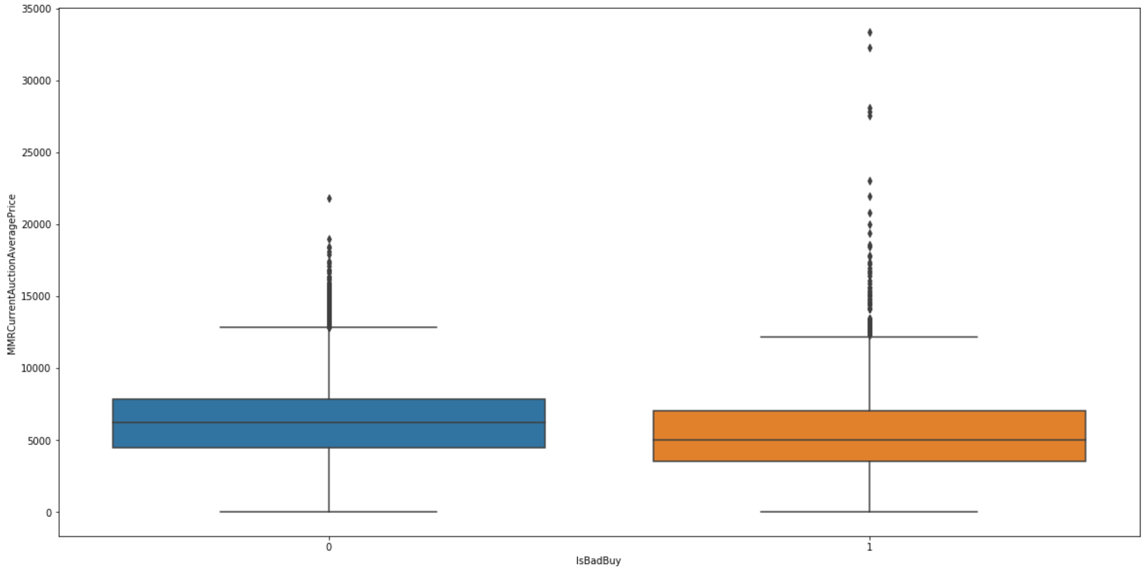

IsBadBuy
0
1
count
0
2000
4000
6000
8000
10000
12000
chrysler
ford
gm
other
TopThreeAmericanName

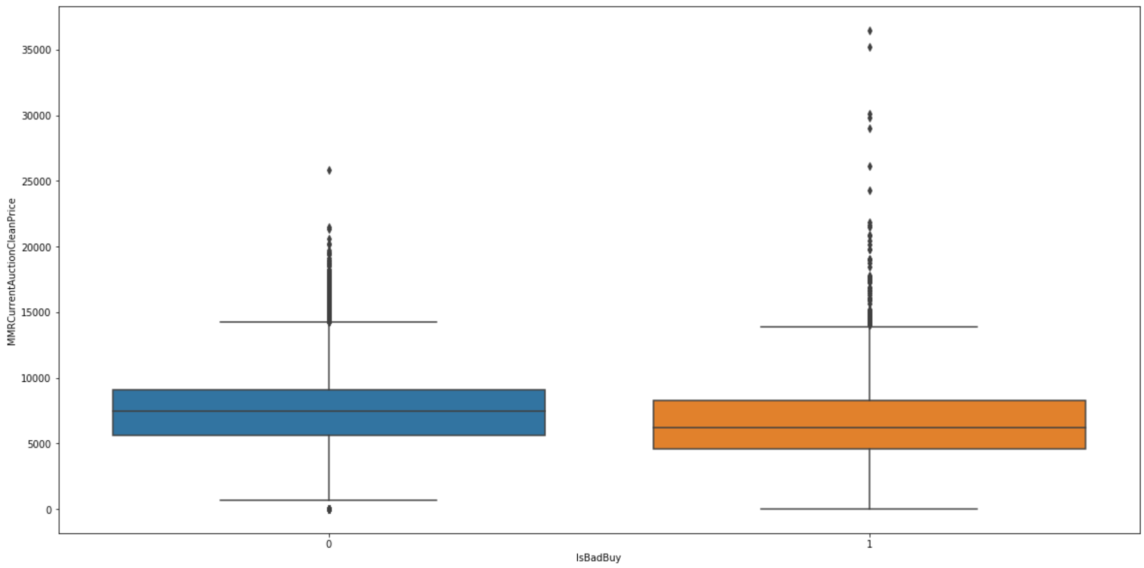

MMRAcquisitionAuctionAveragePrice
0
5000
10000
15000
20000
25000
30000
35000
0
1
IsBadBuy


MMRAcquisitionAuctionCleanPrice
35000
30000
25000
20000
15000
10000
5000
0
0
1
IsBadBuy

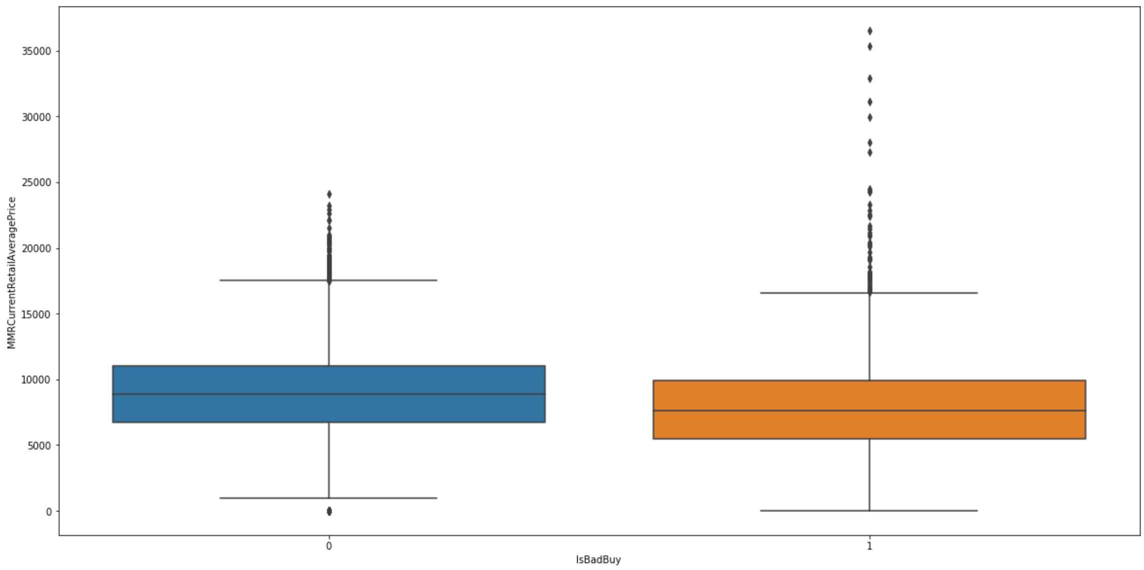

MMRAcquisitionRetailAveragePrice
35000
30000
25000
20000
15000
10000
5000
0
0
1
IsBadBuy


MMRAcquisitonRetailCleanPrice
IsBadBuy

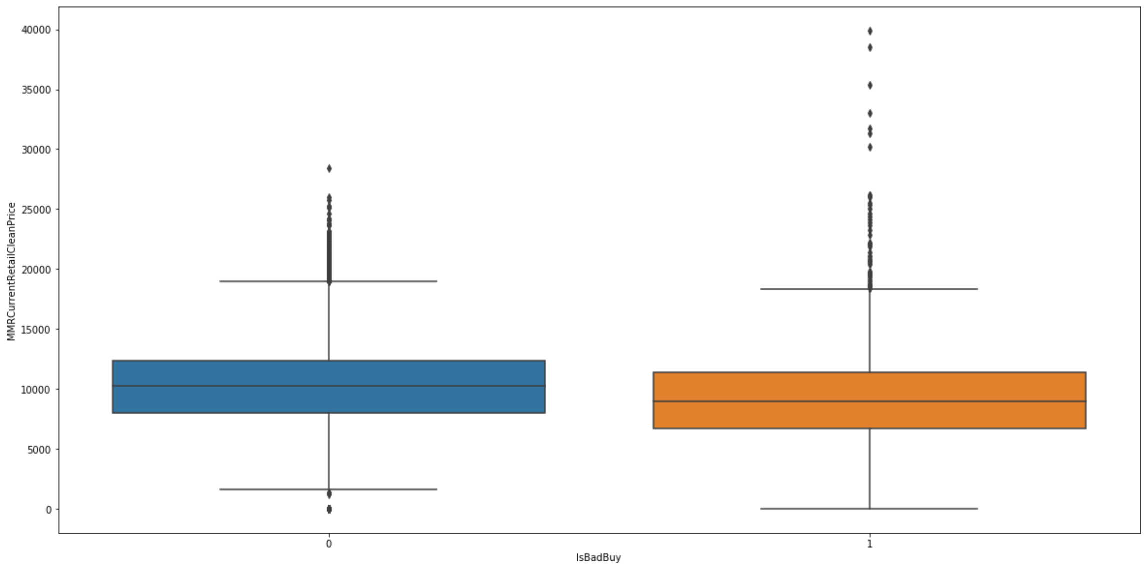

MMRCurrentAuctionAveragePrice
IsBadBuy

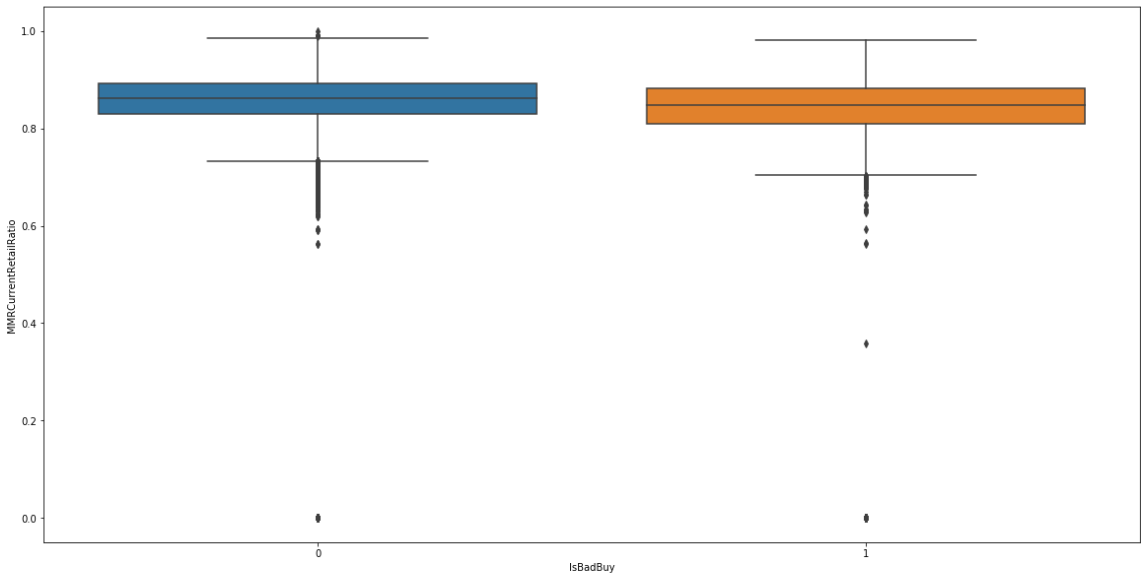


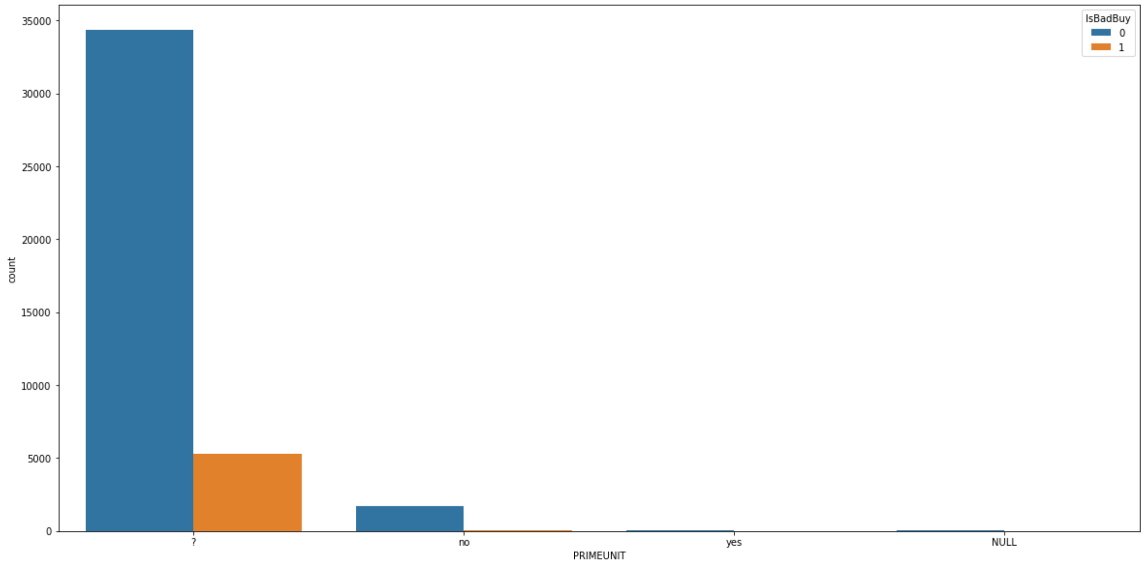
35000
30000
25000
20000
15000
10000
5000
0
MMRCurrentAuctionCleanPrice
0
1
IsBadBuy

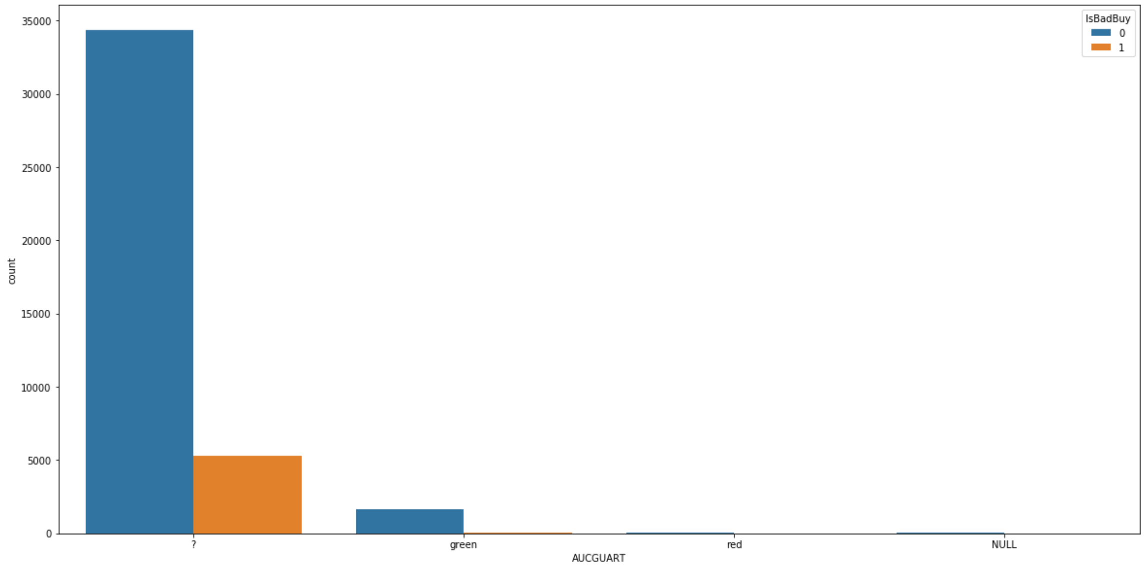


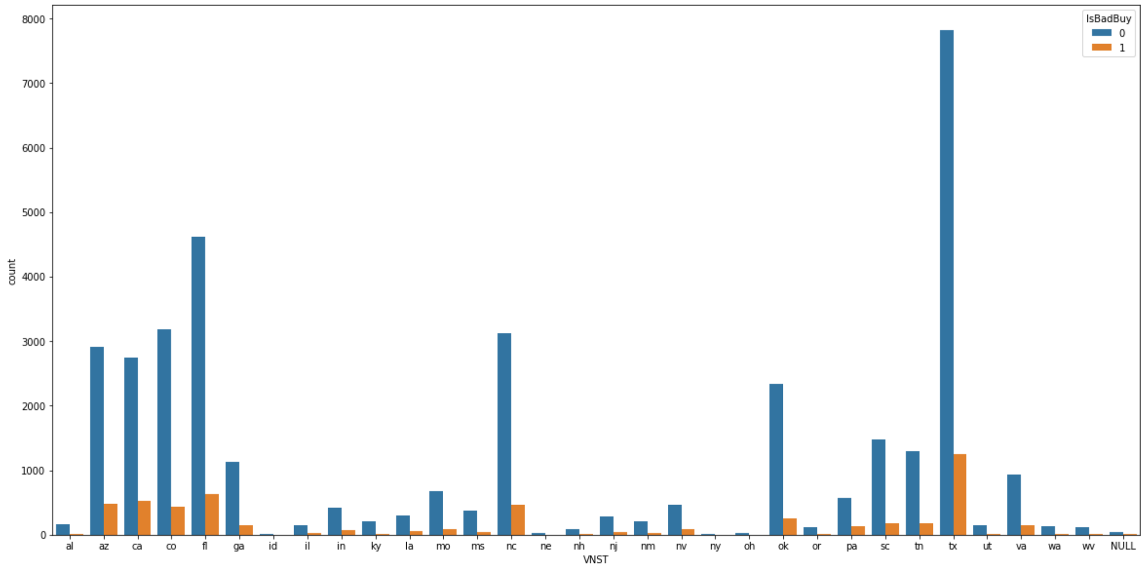
35000
30000
25000
20000
15000
10000
5000
0
MMRCurrentRetailAveragePrice
0
1
IsBadBuy

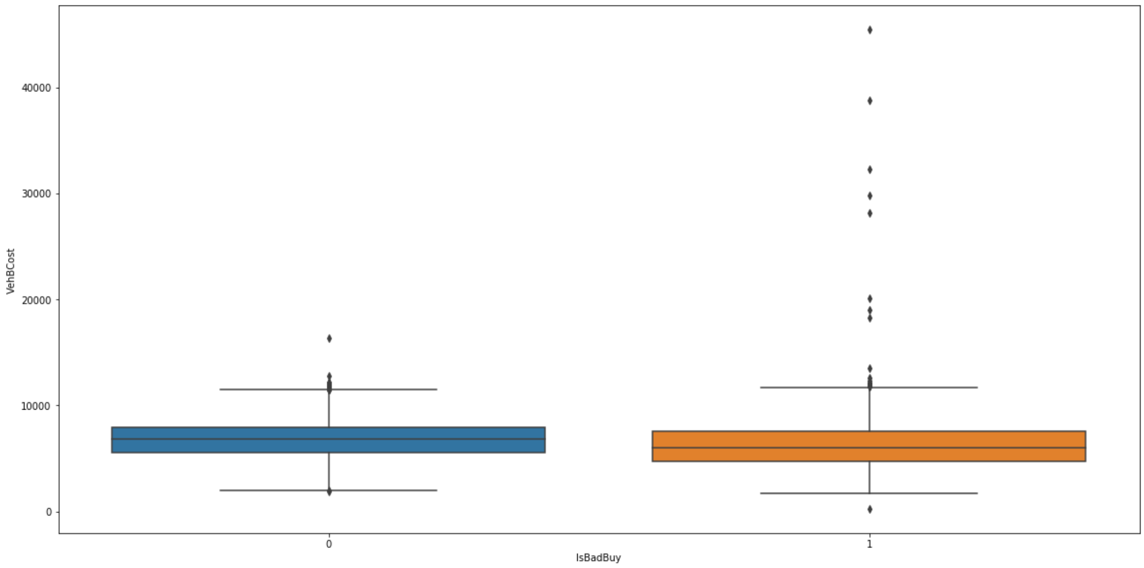

MMRCurrentRetailCleanPrice
40000
35000
30000
25000
20000
15000
10000
5000
0
0
1
IsBadBuy

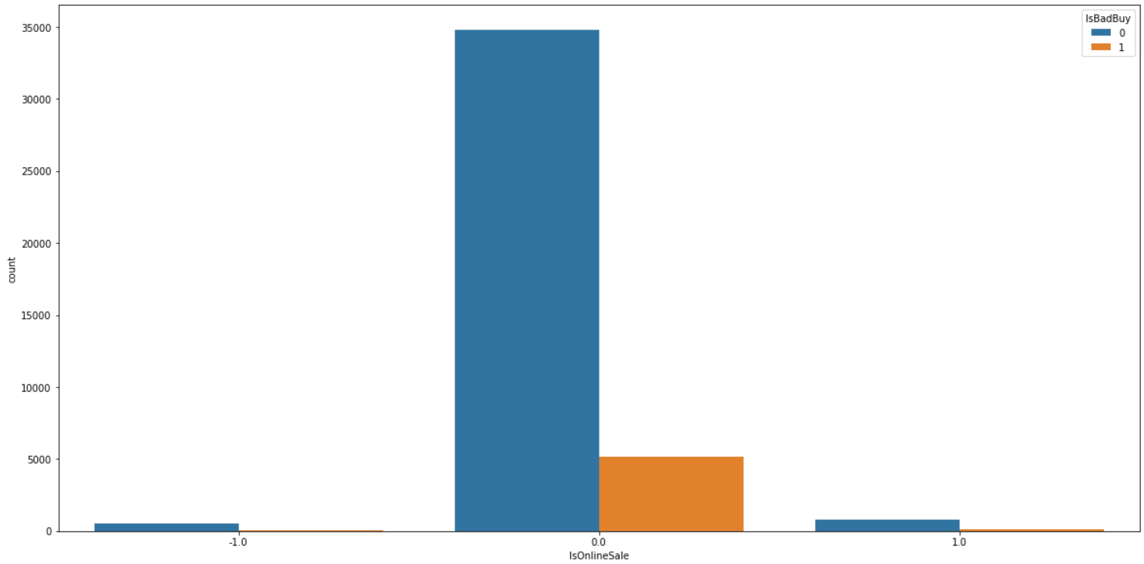

MMRCurrentRetailRatio
1.0
0.8
0.6
0.4
0.2
0.0
0
1
IsBadBuy

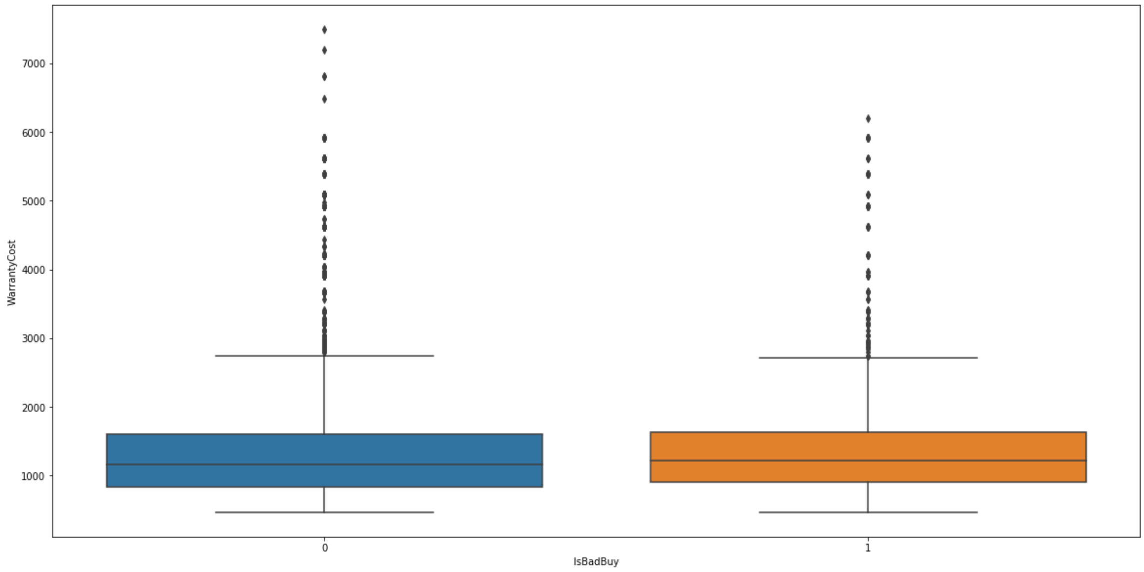

IsBadBuy
0
1
count
35000
30000
25000
20000
15000
10000
5000
0
?
no
yes
NULL
PRIMEUNIT

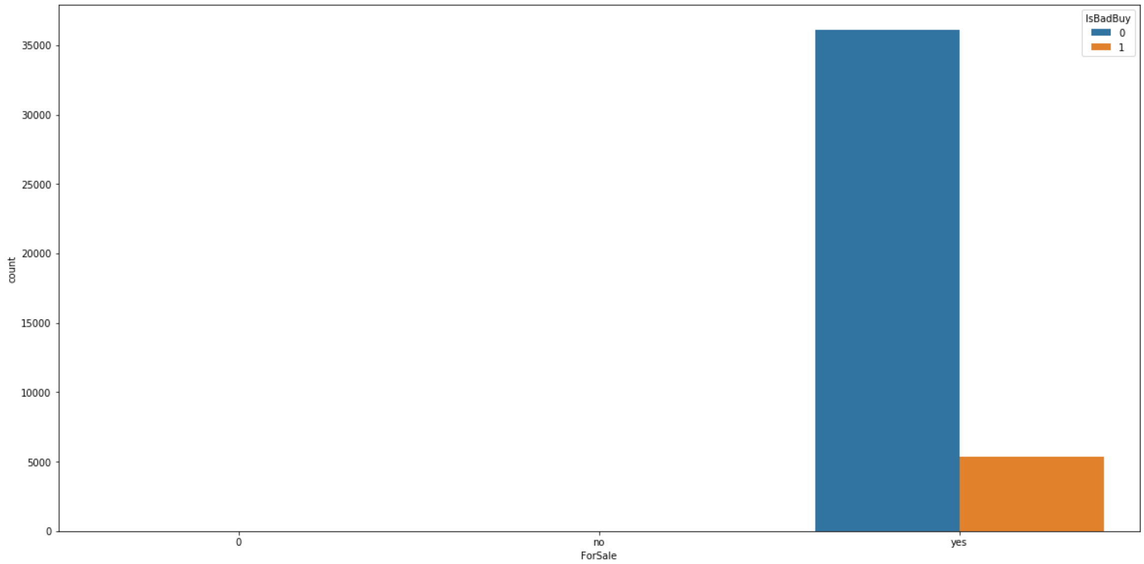

IsBadBuy
0
1
count
35000
30000
25000
20000
15000
10000
5000
0
?
green
red
NULL
AUCGUART


IsBadBuy
0
1
count
VNST
al
az
ca
co
fl
ga
id
il
in
ky
la
mo
ms
nc
ne
nh
nj
nm
nv
ny
oh
ok
or
pa
sc
tn
tx
ut
va
wa
wv
NULL


VehBCost
IsBadBuy


IsBadBuy
0
1
count
35000
30000
25000
20000
15000
10000
5000
0
-1.0
0.0
1.0
IsOnlineSale


WarrantyCost
7000
6000
5000
4000
3000
2000
1000
0
1
IsBadBuy



<Figure size 1440x720 with 0 Axes>

4. What variables did you include in the analysis and what were their roles and measurement level set? Justify your choice

In [ ]:

```

```

5. What distribution scheme did you use? What data partitioning allocation did you set? Explain your selection.

In [11]:

```
# Change to the dummy

df = pd.get_dummies(df)

feature_names = df.drop("IsBadBuy", axis=1).columns
print("Num of Features:")

### Split to the training and test set.
#  The test size is 3%

# y = df['IsBadBuy']
# X = df.drop(['IsBadBuy'], axis=1)
# X_mat = X.as_matrix()

# X_train, X_test, y_train, y_test = train_test_split(X_mat, y, test_size=0.3, s
tratify=y, random_state=rs)

X_train, X_test, y_train, y_test = train_test_split(df.drop("IsBadBuy", axis=1),
df['IsBadBuy'], test_size=0.3, stratify=df['IsBadBuy'], random_state=rs)

if ResamplingMethod == 'ros':
    print("Using ROS Resmapling")
    ros = RandomOverSampler(random_state=rs)
    X_train, y_train = ros.fit_resample(X_train, y_train)
elif  ResamplingMethod == 'rus':
    print("Using RUS Resmapling")
    rus = RandomUnderSampler(random_state=rs)
    X_train, y_train = rus.fit_resample(X_train, y_train)
else:
    print("No Resampling Method Used")
```

```
Num of Features:
Using RUS Resmapling
```

In [12]:

```
print("Number of Training: ", len(X_train))
print("Number of Test: ", len(X_test) )
```

```
Number of Training:  7520
Number of Test:  12443
```

Task 2. Predictive Modeling Using Decision Trees

1. Python: Build a decision tree using the default setting.

In [13]:

```
def printLRTopImportant(model, top = 5):

    coef = model.coef_[0]
    indices = np.argsort(np.absolute(coef))
    indices = np.flip(indices, axis=0)
    indices = indices[:top]
    for i in indices:
        print(feature_names[i], ':', coef[i])

def analyse_feature_importance(dm_model, feature_names, n_to_display=20):
    # grab feature importances from the model
    importances = dm_model.feature_importances_

    # sort them out in descending order
    indices = np.argsort(importances)
    indices = np.flip(indices, axis=0)

    # limit to 20 features, you can leave this out to print out everything
    indices = indices[:n_to_display]

    for i in indices:
        print(feature_names[i], ':', importances[i])

def visualize_decision_tree(dm_model, feature_names, save_name):
    dotfile = StringIO()
    export_graphviz(dm_model, out_file=dotfile, feature_names=feature_names)
    graph = pydot.graph_from_dot_data(dotfile.getvalue())
    graph[0].write_png(save_name) # saved in the following file
```

In [14]:

```
# simple decision tree training
model = DecisionTreeClassifier(random_state=rs)
model.fit(X_train, y_train)
```

Out[14]:

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_dept
h=None,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_stat
e=101,
            splitter='best')
```

a. What is the classification accuracy on training and test datasets?

In [15]:

```
print("Train accuracy:", model.score(X_train, y_train))
print("Test accuracy:", model.score(X_test, y_test))
y_pred = model.predict(X_test)
print(classification_report(y_test, y_pred))
confusion_matrix(y_test, y_pred) ## Confusion Matrix on the TestSet
```

```
Train accuracy: 0.9998670212765958
Test accuracy: 0.6226794181467492
              precision    recall  f1-score   support

           0       0.92      0.62      0.74     10832
           1       0.20      0.62      0.30      1611

   micro avg       0.62      0.62      0.62     12443
   macro avg       0.56      0.62      0.52     12443
weighted avg       0.82      0.62      0.68     12443
```

Out[15]:

```
array([[6743, 4089],
       [ 606, 1005]])
```

b. What is the size of tree (i.e. number of nodes)?

In [16]:

```
print("Number of nodes: ", model.tree_.node_count)
```

```
Number of nodes:  2745
```

c. How many leaves are in the tree that is selected based on the validation dataset?

In [ ]:

```

```

d. Which variable is used for the first split? What are the competing splits for this first split?

In [17]:

```
visualize_decision_tree(model, df.drop("IsBadBuy", axis=1).columns, "Tree_Struct.png")
```

e. What are the 5 important variables in building the tree?

In [18]:

```
analyse_feature_importance(model, df.drop("IsBadBuy", axis=1).columns, 5)
```

```
WheelTypeID_? : 0.13480585505228698
VehBCost : 0.0697126923506575
VehOdo : 0.06920195012906506
MMRCurrentRetailRatio : 0.06552441838480952
MMRCurrentAuctionAveragePrice : 0.06249076040165891
```

f. Report if you see any evidence of model overfitting.

In [ ]:

```

```

g. Did changing the default setting (i.e., only focus on changing the setting of the number of splits to create a node) help improving the model? Answer the above questions on the best performing tree.

In [ ]:

```

```

2. Python: Build another decision tree tuned with GridSearchCV

In [ ]:

```

```

In [19]:

```
# grid search CV
params = {'criterion': ['gini', 'entropy'],
          'max_depth': list(range(2,7)) +[200, 500] + list(range(1, 6000, 1000))
+ [None],
          'splitter': ['best', 'random'],
          'min_samples_leaf': range(1,  4),
          'min_samples_split': [2, 0.5, 0.3],
          'max_features':['auto','sqrt','log2', None],
          'class_weight':['balanced', None]
         }

cv = GridSearchCV(param_grid=params, estimator=DecisionTreeClassifier(random_sta
te=rs), cv=3)
cv.fit(X_train, y_train)
```

Out[19]:

```
GridSearchCV(cv=3, error_score='raise-deprecating',
       estimator=DecisionTreeClassifier(class_weight=None, criterion
='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_stat
e=101,
            splitter='best'),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid={'criterion': ['gini', 'entropy'], 'max_depth':
[2, 3, 4, 5, 6, 200, 500, 1, 1001, 2001, 3001, 4001, 5001, None], 's
plitter': ['best', 'random'], 'min_samples_leaf': range(1, 4), 'min_
samples_split': [2, 0.5, 0.3], 'max_features': ['auto', 'sqrt', 'log
2', None], 'class_weight': ['balanced', None]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='war
n',
       scoring=None, verbose=0)
```

a. What is the classification accuracy on training and test datasets?

In [20]:

```
print("Train accuracy:", cv.score(X_train, y_train))
print("Test accuracy:", cv.score(X_test, y_test))

# test the best model
y_pred = cv.predict(X_test)
print(classification_report(y_test, y_pred))
# print parameters of the best model
print(cv.best_params_)

dt_model = cv.best_estimator_
```

```
Train accuracy: 0.6804521276595744
Test accuracy: 0.8203005706019448
              precision    recall  f1-score   support

           0       0.92      0.87      0.89     10832
           1       0.35      0.47      0.40      1611

   micro avg       0.82      0.82      0.82     12443
   macro avg       0.64      0.67      0.65     12443
weighted avg       0.84      0.82      0.83     12443

{'class_weight': 'balanced', 'criterion': 'entropy', 'max_depth': 6,
'max_features': None, 'min_samples_leaf': 3, 'min_samples_split': 2,
'splitter': 'random'}
```

b. What is the size of tree (i.e. number of nodes)? Is the size different from the maximal tree or the tree in the previous step? Why?

In [21]:

```
print("Number of nodes: ", cv.best_estimator_.tree_.node_count)
```

```
Number of nodes:  87
```

c. How many leaves are in the tree that is selected based on the validation dataset?

In [ ]:

```

```

d. Which variable is used for the first split? What are the competing splits for this first split?

In [22]:

```
visualize_decision_tree(cv.best_estimator_, df.drop("IsBadBuy", axis=1).columns,
"Tree_Struct_CV.png")
```

e. What are the 5 important variables in building the tree?

In [23]:

```
analyse_feature_importance(cv.best_estimator_, df.drop("IsBadBuy", axis=1).colum
ns, 5)
```

```
WheelType_? : 0.6085239761677225
MMRCurrentRetailAveragePrice : 0.07627438205924118
VehYear_2008.0 : 0.044790614328421435
MMRAcquisitionAuctionCleanPrice : 0.03514978515957266
Auction_manheim : 0.022389295471556957
```

f. Report if you see any evidence of model overfitting.

In [ ]:

```

```

g. What are the parameters used? Explain your choices.

In [ ]:

```

```

3. What is the significant difference do you see between these two decision tree models (steps 2.1 & 2.2)? How do they compare performance-wise? Explain why those changes may have happened.

In [ ]:

```

```

4. From the better model, can you identify which cars could potential be “kicks”? Can you provide some descriptive summary of those cars?

In [ ]:

```

```

In [ ]:

```

```

Task 3. Predictive Modeling Using Regression

1. In preparation for regression, is any imputation of missing values needed for this data set? List the variables that needed this.

In [24]:

```
# We've already done this in the prep_data function
```

2. Apply transformation method(s) to the variable(s) that need it. List the variables that needed it

In [25]:

```
## Doing the log transformation


### Q: It's enoguh?
columns_to_transform = interval_cols

def logTransformation(df):

    df_log = df.copy()


    for col in columns_to_transform:
        df_log[col] = df_log[col].apply(lambda x: x+1)
        df_log[col] = df_log[col].apply(np.log)


    return df_log

df_log = logTransformation(df)
X_train_log, X_test_log, y_train_log, y_test_log = train_test_split(df_log.drop
(['IsBadBuy'], axis=1), df_log['IsBadBuy'], test_size=0.3, stratify=df_log['IsBa
dBuy']
,random_state=rs)

# Standardise
scaler_log = StandardScaler()
X_train_log = scaler_log.fit_transform(X_train_log, y_train_log)
X_test_log = scaler_log.transform(X_test_log)
```

3. Build a regression model using the default regression method with all inputs. Once you done it, build another one and tune it using GridSearchCV. Answer the followings:

In [26]:

```
### Traing Logistic Regression
model = LogisticRegression(random_state=rs)
model.fit(X_train_log, y_train_log)
```

Out[26]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_interce
pt=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=101, solver='war
n',
          tol=0.0001, verbose=0, warm_start=False)
```

In [27]:

```
## GridSearch for Logistic Regression
params = {
    'C': [pow(10, x) for x in range(-4, 1)],
    'solver' : ['newton-cg',"lbfgs", "liblinear", "sag", "saga"],
    'max_iter': [30, 50, 100],
    'warm_start': [True, False],
    'class_weight':['balanced', None]
}

cv = GridSearchCV(param_grid=params, estimator=LogisticRegression(random_state=r
s), cv=3, n_jobs=-1)
cv.fit(X_train_log, y_train_log)
```

Out[27]:

```
GridSearchCV(cv=3, error_score='raise-deprecating',
       estimator=LogisticRegression(C=1.0, class_weight=None, dual=F
alse, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=101, solver='war
n',
          tol=0.0001, verbose=0, warm_start=False),
       fit_params=None, iid='warn', n_jobs=-1,
       param_grid={'C': [0.0001, 0.001, 0.01, 0.1, 1], 'solver': ['n
ewton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'], 'max_iter': [30, 5
0, 100], 'warm_start': [True, False], 'class_weight': ['balanced', N
one]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='war
n',
       scoring=None, verbose=0)
```

h. Name the regression function used.

In [ ]:

```

```

i. How much was the difference in performance of two models build, default and optimal?

In [28]:

```
print("Train accuracy:", model.score(X_train_log, y_train_log))
print("Test accuracy:", model.score(X_test_log, y_test_log))
print("GridSearch Train accuracy:", cv.score(X_train_log, y_train_log))
print("GridSearch Test accuracy:", cv.score(X_test_log, y_test_log))
```

```
Train accuracy: 0.8966348637757036
Test accuracy: 0.8982560475769509
GridSearch Train accuracy: 0.8961526538766231
GridSearch Test accuracy: 0.8984167805191674
```

j. Show the set parameters for the best model. What are the parameters used? Explain your decision. What are the optimal parameters?

In [29]:

```
print("The best model parameters: ", cv.best_params_)
```

```
The best model parameters:  {'C': 0.001, 'class_weight': None, 'max_
iter': 30, 'solver': 'newton-cg', 'warm_start': True}
```

k. Report which variables are included in the regression model.

In [ ]:

```

```

l. Report the top-5 important variables (in the order) in the model.

In [30]:

```
def printLRTopImportant(model, top = 5):

    coef = model.coef_[0]
    indices = np.argsort(np.absolute(coef))
    indices = np.flip(indices, axis=0)
    indices = indices[:top]
    for i in indices:
        print(feature_names[i], ':', coef[i])
```

In [31]:

```
printLRTopImportant(model, 5)
```

```
MMRAcquisitionAuctionAveragePrice : -1.3421704081048444
MMRAcquisitionRetailAveragePrice : 1.1753374313929883
MMRCurrentAuctionAveragePrice : 0.7514553467571049
MMRCurrentRetailCleanPrice : -0.6579437881110104
MMRAcquisitonRetailCleanPrice : 0.6566173157712023
```

m. What is classification accuracy on training and test datasets?

In [32]:

```
y_pred = model.predict(X_test_log)
print("Classification Report: \n\n",classification_report(y_test_log, y_pred))

y_pred = cv.predict(X_test_log)
print("GridSearch Classification Report: \n\n",classification_report(y_test_log,
y_pred))
log_reg_model = cv.best_estimator_
```

```
Classification Report: 

              precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.84      0.26      0.40      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.87      0.63      0.67     12443
weighted avg       0.89      0.90      0.87     12443

GridSearch Classification Report: 

              precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.84      0.27      0.40      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.87      0.63      0.67     12443
weighted avg       0.89      0.90      0.87     12443
```

n. Report any sign of overfitting.

In [33]:

```
## The GridSearch Precision and Recall is weird
```

4. Build another regression model using the subset of inputs selected by RFE and selection by model method. Answer the followings:

In [34]:

```
rfe = RFECV(estimator = LogisticRegression(random_state=rs), cv=3)
rfe.fit(X_train_log, y_train_log)
X_train_rfe = rfe.transform(X_train_log)
X_test_rfe = rfe.transform(X_test_log)

selectmodel = SelectFromModel(dt_model, prefit=True)
X_train_sel_model = selectmodel.transform(X_train_log)
X_test_sel_model = selectmodel.transform(X_test_log)
```

a. Report which variables are included in the regression model.

In [35]:

```
print("Original feature set", X_train.shape[1])
print("Number of RFE-selected features: ", rfe.n_features_)
print("Number of selectFromModel features: ",  X_train_sel_model.shape[1])
```

```
Original feature set 149
Number of RFE-selected features:  80
Number of selectFromModel features:  15
```

In [36]:

```
print("The RFE-selected features: \n\n", list(compress(feature_names, rfe.suppor
t_)))
print("\n\n")
print("The SelectFromModel features: \n\n",list(compress(feature_names, selectmo
del.get_support())))
```

```
The RFE-selected features: 

 ['VehOdo', 'MMRAcquisitionAuctionAveragePrice', 'MMRAcquisitionAuct
ionCleanPrice', 'MMRAcquisitionRetailAveragePrice', 'MMRAcquisitonRe
tailCleanPrice', 'MMRCurrentAuctionAveragePrice', 'MMRCurrentAuction
CleanPrice', 'MMRCurrentRetailCleanPrice', 'MMRCurrentRetailRatio',
'VehBCost', 'WarrantyCost', 'Auction_adesa', 'Auction_manheim', 'Veh
Year_2001.0', 'VehYear_2002.0', 'VehYear_2003.0', 'VehYear_2004.0',
'VehYear_2005.0', 'VehYear_2006.0', 'VehYear_2007.0', 'VehYear_2008.
0', 'VehYear_2009.0', 'VehYear_UNKNOWN_VALUE', 'Make_acura', 'Make_d
odge', 'Make_honda', 'Make_infiniti', 'Make_isuzu', 'Make_lincoln',
'Make_mini', 'Make_nissan', 'Make_pontiac', 'Make_subaru', 'Make_suz
uki', 'Make_toyota', 'Make_volvo', 'Color_green', 'Color_other', 'Co
lor_white', 'WheelTypeID_0', 'WheelTypeID_1', 'WheelTypeID_2', 'Whee
lTypeID_3', 'WheelTypeID_?', 'WheelType_?', 'WheelType_alloy', 'Whee
lType_covers', 'WheelType_special', 'Nationality_other asian', 'Nati
onality_top line asian', 'Size_large', 'Size_large suv', 'Size_mediu
m', 'Size_medium suv', 'Size_van', 'TopThreeAmericanName_chrysler',
'TopThreeAmericanName_gm', 'PRIMEUNIT_?', 'PRIMEUNIT_no', 'PRIMEUNIT
_yes', 'PRIMEUNIT_NULL', 'AUCGUART_?', 'VNST_co', 'VNST_fl', 'VNST_g
a', 'VNST_id', 'VNST_ky', 'VNST_la', 'VNST_nc', 'VNST_ne', 'VNST_n
h', 'VNST_ny', 'VNST_or', 'VNST_pa', 'VNST_sc', 'VNST_tn', 'VNST_u
t', 'VNST_wa', 'IsOnlineSale_1.0', 'ForSale_yes']



The SelectFromModel features: 

 ['VehOdo', 'MMRAcquisitionAuctionCleanPrice', 'MMRAcquisitonRetailC
leanPrice', 'MMRCurrentRetailAveragePrice', 'Auction_manheim', 'VehY
ear_2001.0', 'VehYear_2006.0', 'VehYear_2007.0', 'VehYear_2008.0',
'WheelType_?', 'Size_large', 'Size_medium suv', 'TopThreeAmericanNam
e_ford', 'PRIMEUNIT_no', 'AUCGUART_?']
```

b. Report the top-5 important variables (in the order) in the model.

In [37]:

```
params = {
    'C': [pow(10, x) for x in range(-4, 1)],
    'solver' : ['newton-cg',"lbfgs", "liblinear", "sag", "saga"],
    'max_iter': [30, 50, 100],
    'warm_start': [True, False],
    'class_weight':['balanced', None]

}
rfe_cv = GridSearchCV(param_grid=params, estimator=LogisticRegression(random_sta
te=rs, verbose=True), cv=3, n_jobs=-1)
rfe_cv.fit(X_train_rfe, y_train_log)

selectModel_cv = GridSearchCV(param_grid=params, estimator=LogisticRegression(ra
ndom_state=rs, verbose=True), cv=3, n_jobs=-1)
selectModel_cv.fit(X_train_sel_model, y_train_log)
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurr
ent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.4s finishe
d
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurr
ent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.3s finishe
d
```

Out[37]:

```
GridSearchCV(cv=3, error_score='raise-deprecating',
       estimator=LogisticRegression(C=1.0, class_weight=None, dual=F
alse, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=101, solver='war
n',
          tol=0.0001, verbose=True, warm_start=False),
       fit_params=None, iid='warn', n_jobs=-1,
       param_grid={'C': [0.0001, 0.001, 0.01, 0.1, 1], 'solver': ['n
ewton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'], 'max_iter': [30, 5
0, 100], 'warm_start': [True, False], 'class_weight': ['balanced', N
one]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='war
n',
       scoring=None, verbose=0)
```

In [38]:

```
print("Top-5 important variables for RFE: \n")
printLRTopImportant(rfe_cv.best_estimator_, 5)
print("\n\n")
print("Top-5 important variables for selectModel \n")
printLRTopImportant(selectModel_cv.best_estimator_, 5)
```

```
Top-5 important variables for RFE: 

Make_mercury : 0.35873351247562235
Make_mini : 0.337853228316747
MMRCurrentRetailRatio : -0.2582420901617004
MMRAcquisitionRetailAveragePrice : 0.2432460273505872
MMRAcquisitionAuctionAveragePrice : -0.24047249906816362



Top-5 important variables for selectModel 

MMRCurrentRetailRatio : 0.7197082753601514
MMRAcquisitionAuctionAveragePrice : -0.3512683893922687
MMRAcquisitionAuctionCleanPrice : 0.33380598413193824
MMRCurrentRetailCleanPrice : -0.27923125209284044
MMRCurrentRetailAveragePrice : -0.24945177578546104
```

c. What are the parameters used? Explain your choices. What are the optimal parameters? Which regression function is being used?

In [39]:

```
print("Optimal Parameters for RFE", rfe_cv.best_params_)
print("Optimal Parameters for selectModel", selectModel_cv.best_params_)
```

```
Optimal Parameters for RFE {'C': 0.01, 'class_weight': None, 'max_it
er': 50, 'solver': 'lbfgs', 'warm_start': True}
Optimal Parameters for selectModel {'C': 0.01, 'class_weight': None,
'max_iter': 30, 'solver': 'newton-cg', 'warm_start': True}
```

d. Report any sign of overfitting

In [ ]:

```

```

e. What is classification accuracy on training and test datasets?

In [40]:

```
print("GridSearch Train accuracy:", cv.score(X_train_log, y_train_log))
print("GridSearch Test accuracy:", cv.score(X_test_log, y_test_log))
print("\n\nRFE:\n")
print("Train accuracy:", rfe_cv.score(X_train_rfe, y_train_log))
print("Test accuracy:", rfe_cv.score(X_test_rfe, y_test_log))
print("\n\nselectModel:\n")
print("Train accuracy:", selectModel_cv.score(X_train_sel_model, y_train_log))
print("Test accuracy:", selectModel_cv.score(X_test_sel_model, y_test_log))
```

```
GridSearch Train accuracy: 0.8961526538766231
GridSearch Test accuracy: 0.8984167805191674


RFE:

Train accuracy: 0.8965659766472635
Test accuracy: 0.8984971469902756


selectModel:

Train accuracy: 0.8957393311059828
Test accuracy: 0.8981756811058427
```

f. Did it improve/worsen the performance? Explain why those changes may have happened

In [41]:

```
y_pred = rfe_cv.predict(X_test_rfe)
print("REF classification report: \n",classification_report(y_test, y_pred))
print("\n\n")
y_pred = selectModel_cv.predict(X_test_sel_model)
print("selectModel classification report: \n",classification_report(y_test, y_pr
ed))
```

```
REF classification report:
               precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.85      0.26      0.40      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.87      0.63      0.67     12443
weighted avg       0.89      0.90      0.87     12443




selectModel classification report:
               precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.83      0.27      0.40      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.87      0.63      0.67     12443
weighted avg       0.89      0.90      0.87     12443
```

Task4 - Predicting using neural network

1. Build a Neural Network model using the default setting. Answer the following:

In [42]:

```
model = MLPClassifier(random_state=rs)
model.fit(X_train_log, y_train_log)
```

Out[42]:

```
MLPClassifier(activation='relu', alpha=0.0001, batch_size='auto', be
ta_1=0.9,
       beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=(100,), learning_rate='constant',
       learning_rate_init=0.001, max_iter=200, momentum=0.9,
       n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
       random_state=101, shuffle=True, solver='adam', tol=0.0001,
       validation_fraction=0.1, verbose=False, warm_start=False)
```

a. What is the network architecture?

In [43]:

```
def printMLPArchitecture(model):

    print("Number of Layers: ",model.n_layers_ )
    print("The First layer is Input Layer, and the last layer is the output laye
r")
    for i, w in enumerate(model.coefs_):
        print("{} Layer with hidden size {}".format(i+1, w.shape[0]))
        if (i+1) == len(model.coefs_):
            print("{} Layer with hidden size {}".format(i+2, w.shape[1]))

    print("The activation function: ", model.activation)

printMLPArchitecture(model)
```

```
Number of Layers:  3
The First layer is Input Layer, and the last layer is the output lay
er
1 Layer with hidden size 149
2 Layer with hidden size 100
3 Layer with hidden size 1
The activation function:  relu
```

b. How many iterations are needed to train this network?

In [44]:

```
print("Number of iterations it ran: ", model.n_iter_)
```

```
Number of iterations it ran:  200
```

c. Do you see any sign of over-fitting?

In [45]:

```
# fig = plt.figure(figsize=(10, 5))
# plt.ylabel('Accuracy',fontsize=15)
# plt.xlabel('Number of iterations',fontsize=15)
# plt.title('Validation Accuracy',fontsize=20,fontweight ="bold")
# plt.plot(model.validation_scores_, label="Validation Accuracy")
```

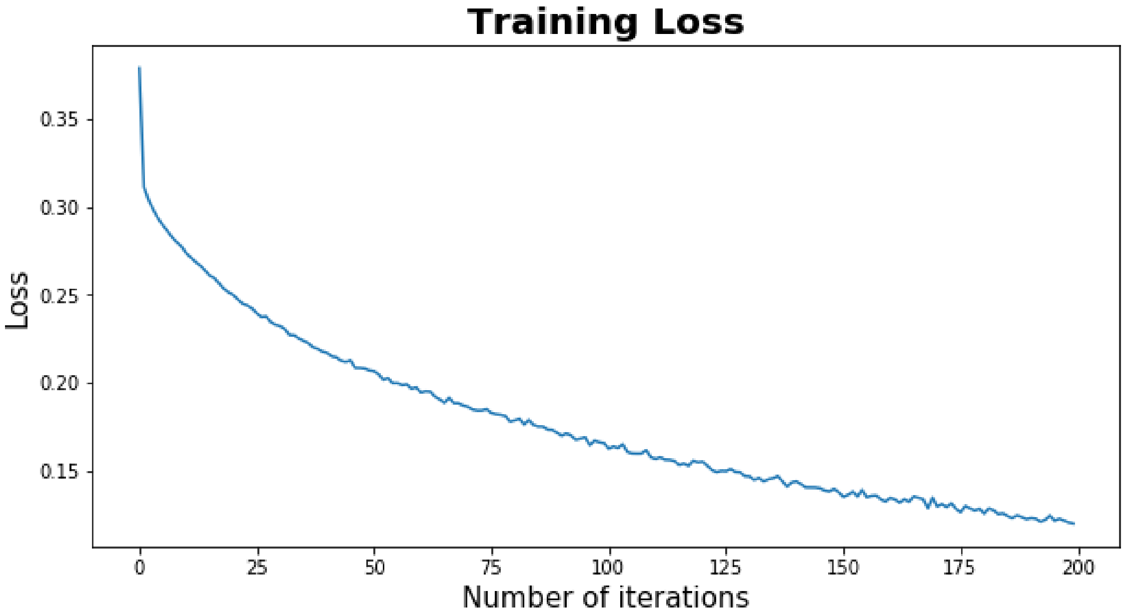
d. Did the training process converge and resulted in the best model?

In [46]:

```
fig = plt.figure(figsize=(10, 5))
plt.ylabel('Loss',fontsize=15)
plt.xlabel('Number of iterations',fontsize=15)
plt.title('Training Loss',fontsize=20,fontweight ="bold")
plt.plot(model.loss_curve_, label="Training Loss")

### The Loss curve is still decreasing
```

Out[46]:

```
[<matplotlib.lines.Line2D at 0x7ff8d63b9390>]
```



e. What is classification accuracy on training and test datasets?

In [47]:

```
print("MLP Train accuracy:", model.score(X_train, y_train))
print("MLP Test accuracy:", model.score(X_test, y_test))
print("\n\n")
y_pred = model.predict(X_test)
print("MLP classification report: \n",classification_report(y_test, y_pred))
```

```
MLP Train accuracy: 0.5001329787234042
MLP Test accuracy: 0.8705296150446034



MLP classification report: 
               precision    recall  f1-score   support

           0       0.87      1.00      0.93     10832
           1       0.00      0.00      0.00      1611

   micro avg       0.87      0.87      0.87     12443
   macro avg       0.44      0.50      0.47     12443
weighted avg       0.76      0.87      0.81     12443
```

2. Refine this network by tuning it with GridSearchCV.

In [48]:

```
# Default
# params = {'hidden_layer_sizes': [(3,), (5,), (7,), (9,)], 'alpha': [0.01,0.00
1, 0.0001, 0.00001]}


params = [
    {
        'hidden_layer_sizes': [(32,),(128,)],
        'activation': ['logistic', 'relu','identity'],
        'solver' : ['adam',],
        'batch_size': [ 64],
        'shuffle': [True],
        'learning_rate_init': [pow(10, x) for x in range(-4, -2)],
        'n_iter_no_change': [10],
        'max_iter':[200, 500],
        'warm_start': [True, False],
    },
    {
        'hidden_layer_sizes': [(32,),(128,)],
        'learning_rate' : ['constant', 'invscaling', 'adaptive'],
        'activation': ['logistic', 'relu','identity'],
        'solver' : ['sgd'],
        'shuffle': [True],
        'batch_size': [64],
        'max_iter':[200, 500],
        'learning_rate_init': [pow(10, x) for x in range(-4, -2)],
        'n_iter_no_change': [10],
        'warm_start': [True, False],
    },
    {
        'hidden_layer_sizes': [(32,),(128,)],
        'activation': ['logistic', 'relu','identity'],
        'solver' : ['lbfgs'],
        'max_iter':[200, 500],
        'batch_size': [64],
        'learning_rate_init': [pow(10, x) for x in range(-4, -2)],
        'n_iter_no_change': [10],
        'warm_start': [True, False],
    }

]


cv = GridSearchCV(param_grid=params, estimator=MLPClassifier(random_state=rs, ea
rly_stopping = True, verbose=True), cv=3, n_jobs=-1)
# cv = GridSearchCV(param_grid=params, estimator=MLPClassifier(random_state=rs,
 early_stopping=True, max_iter = max_iter, n_iter_no_change = max_iter ), cv=3,
 n_jobs=-1)
cv.fit(X_train_log, y_train_log)
```

```
Iteration 1, loss = 0.32857511
Validation score: 0.890496
Iteration 2, loss = 0.31191295
Validation score: 0.890496
Iteration 3, loss = 0.31137095
Validation score: 0.891185
Iteration 4, loss = 0.31018978
Validation score: 0.890840
Iteration 5, loss = 0.30922495
Validation score: 0.891529
Iteration 6, loss = 0.30831522
Validation score: 0.891529
Iteration 7, loss = 0.30758839
Validation score: 0.891185
Iteration 8, loss = 0.30660339
Validation score: 0.891185
Iteration 9, loss = 0.30571344
Validation score: 0.891185
Iteration 10, loss = 0.30481675
Validation score: 0.891185
Iteration 11, loss = 0.30362134
Validation score: 0.891185
Iteration 12, loss = 0.30283829
Validation score: 0.892218
Iteration 13, loss = 0.30120792
Validation score: 0.892218
Iteration 14, loss = 0.29949388
Validation score: 0.892218
Iteration 15, loss = 0.29829281
Validation score: 0.892906
Iteration 16, loss = 0.29680944
Validation score: 0.893251
Iteration 17, loss = 0.29516226
Validation score: 0.892906
Iteration 18, loss = 0.29294147
Validation score: 0.892562
Iteration 19, loss = 0.29150188
Validation score: 0.892562
Iteration 20, loss = 0.28919079
Validation score: 0.892218
Iteration 21, loss = 0.28730894
Validation score: 0.892906
Iteration 22, loss = 0.28529079
Validation score: 0.891873
Iteration 23, loss = 0.28228859
Validation score: 0.892906
Iteration 24, loss = 0.27981098
Validation score: 0.892218
Iteration 25, loss = 0.27729154
Validation score: 0.892218
Iteration 26, loss = 0.27511337
Validation score: 0.892562
Iteration 27, loss = 0.27187226
Validation score: 0.892218
Validation score did not improve more than tol=0.000100 for 10 conse
cutive epochs. Stopping.
```

Out[48]:

```
GridSearchCV(cv=3, error_score='raise-deprecating',
       estimator=MLPClassifier(activation='relu', alpha=0.0001, batc
h_size='auto', beta_1=0.9,
       beta_2=0.999, early_stopping=True, epsilon=1e-08,
       hidden_layer_sizes=(100,), learning_rate='constant',
       learning_rate_init=0.001, max_iter=200, momentum=0.9,
       n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
       random_state=101, shuffle=True, solver='adam', tol=0.0001,
       validation_fraction=0.1, verbose=True, warm_start=False),
       fit_params=None, iid='warn', n_jobs=-1,
       param_grid=[{'hidden_layer_sizes': [(128,)], 'activation':
['logistic', 'relu', 'identity'], 'solver': ['adam'], 'batch_size':
[64], 'shuffle': [True], 'learning_rate_init': [0.0001, 0.001], 'n_i
ter_no_change': [10], 'max_iter': [200, 500], 'warm_start': [True, F
alse]}, {'hidden_layer_sizes': [(128,...[64], 'learning_rate_init':
[0.0001, 0.001], 'n_iter_no_change': [10], 'warm_start': [True, Fals
e]}],
       pre_dispatch='2*n_jobs', refit=True, return_train_score='war
n',
       scoring=None, verbose=0)
```

a. What is the network architecture?

In [66]:

```
print("Best Parameters of NN: ", cv.best_params_)
```

```
Best Parameters of NN:  {'activation': 'logistic', 'batch_size': 64,
'hidden_layer_sizes': (128,), 'learning_rate_init': 0.001, 'max_ite
r': 200, 'n_iter_no_change': 10, 'shuffle': True, 'solver': 'adam',
'warm_start': True}
```

In [50]:

```
printMLPArchitecture(cv.best_estimator_)
```

```
Number of Layers:  3
The First layer is Input Layer, and the last layer is the output lay
er
1 Layer with hidden size 149
2 Layer with hidden size 128
3 Layer with hidden size 1
The activation function:  logistic
```

b. How many iterations are needed to train this network?

In [51]:

```
print("Number of iterations it ran: ",cv.best_estimator_.n_iter_)
```

```
Number of iterations it ran:  27
```

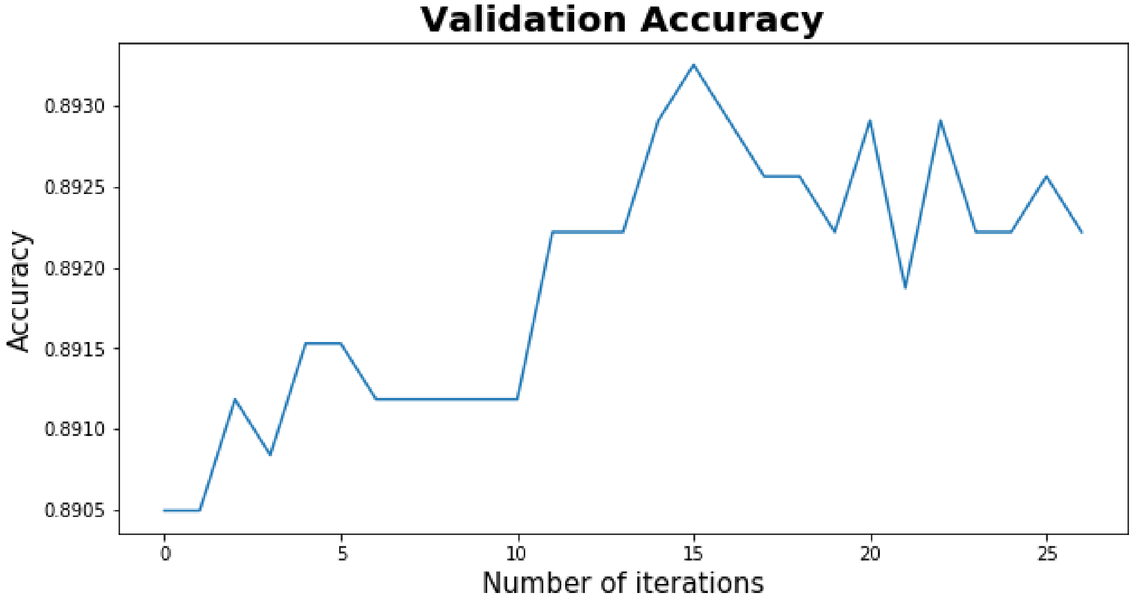
c. Sign of overfitting?

In [52]:

```
fig = plt.figure(figsize=(10, 5))
plt.ylabel('Accuracy',fontsize=15)
plt.xlabel('Number of iterations',fontsize=15)
plt.title('Validation Accuracy',fontsize=20,fontweight ="bold")
plt.plot(cv.best_estimator_.validation_scores_, label="Validation Accuracy")
```

Out[52]:

```
[<matplotlib.lines.Line2D at 0x7ff8d6046a90>]
```



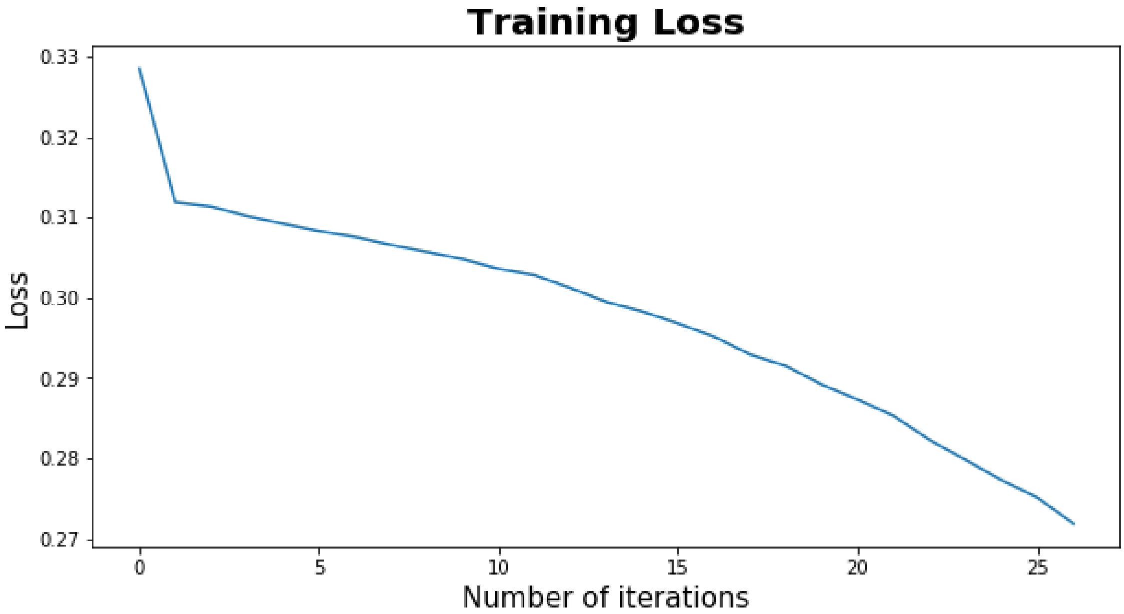
d. Did the training process converge and resulted in the best model?

In [53]:

```
fig = plt.figure(figsize=(10, 5))
plt.ylabel('Loss',fontsize=15)
plt.xlabel('Number of iterations',fontsize=15)
plt.title('Training Loss',fontsize=20,fontweight ="bold")
plt.plot(cv.best_estimator_.loss_curve_, label="Training Loss")
```

Out[53]:

```
[<matplotlib.lines.Line2D at 0x7ff8b2b88278>]
```



e. What is classification accuracy on training and test datasets? Is there any improvement in the outcome?

In [54]:

```
print("GridSearch NN Train accuracy:", cv.score(X_train_log, y_train_log))
print("GridSearch NN Test accuracy:", cv.score(X_test_log, y_test_log))

print("\n\n")
y_pred = cv.predict(X_test_log)
print("GridSearch NN Classification Report: \n",classification_report(y_test_log
, y_pred))

print("Best Parameters of NN: ", cv.best_params_)
nn_model = cv.best_estimator_
```

```
GridSearch NN Train accuracy: 0.8991836875279854
GridSearch NN Test accuracy: 0.8985775134613839



GridSearch NN Classification Report: 
               precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.86      0.26      0.40      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.88      0.63      0.67     12443
weighted avg       0.90      0.90      0.87     12443

Best Parameters of NN:  {'activation': 'logistic', 'batch_size': 64,
'hidden_layer_sizes': (128,), 'learning_rate_init': 0.001, 'max_ite
r': 200, 'n_iter_no_change': 10, 'shuffle': True, 'solver': 'adam',
'warm_start': True}
```

3. Would feature selection help here? Build another Neural Network model with inputs selected from RFE with regression (use the best model generated in Task 3) and selection with decision tree (use the best model from Task 2).

In [55]:

```
params = [
    {
        'hidden_layer_sizes': [(32,),(128,)],
        'activation': ['logistic', 'relu','identity'],
        'solver' : ['adam',],
        'batch_size': [ 64],
        'shuffle': [True],
        'learning_rate_init': [pow(10, x) for x in range(-4, -2)],
        'n_iter_no_change': [10],
        'max_iter':[200, 500],
        'warm_start': [True, False],
    },
    {
        'hidden_layer_sizes': [(32,), (128,)],
        'learning_rate' : ['constant', 'invscaling', 'adaptive'],
        'activation': ['logistic', 'relu','identity'],
        'solver' : ['sgd'],
        'shuffle': [True],
        'batch_size': [64],
        'max_iter':[200, 500],
        'learning_rate_init': [pow(10, x) for x in range(-4, -2)],
        'n_iter_no_change': [10],
        'warm_start': [True, False],
    },
    {
        'hidden_layer_sizes': [(32,),(128,)],
        'activation': ['logistic', 'relu','identity'],
        'solver' : ['lbfgs'],
        'max_iter':[200, 500],
        'batch_size': [64],
        'learning_rate_init': [pow(10, x) for x in range(-4, -2)],
        'n_iter_no_change': [10],
        'warm_start': [True, False],
    }

]


rfe_cv = GridSearchCV(param_grid=params, estimator=MLPClassifier(random_state=rs
, early_stopping=True, verbose=True), cv=3, n_jobs=-1)
rfe_cv.fit(X_train_rfe, y_train_log)
modelSelect_cv = GridSearchCV(param_grid=params, estimator=MLPClassifier(random_
state=rs, early_stopping=True, verbose=True), cv=3, n_jobs=-1)
modelSelect_cv.fit(X_train_sel_model, y_train_log)
```

```
Iteration 1, loss = 0.32866056
Validation score: 0.899105
Iteration 2, loss = 0.30725139
Validation score: 0.899449
Iteration 3, loss = 0.30192863
Validation score: 0.898760
Iteration 4, loss = 0.29865307
Validation score: 0.895317
Iteration 5, loss = 0.29619341
Validation score: 0.899105
Iteration 6, loss = 0.29378594
Validation score: 0.896694
Iteration 7, loss = 0.29167638
Validation score: 0.895661
Iteration 8, loss = 0.29023446
Validation score: 0.895661
Iteration 9, loss = 0.28901530
Validation score: 0.895661
Iteration 10, loss = 0.28718890
Validation score: 0.896694
Iteration 11, loss = 0.28605909
Validation score: 0.895317
Iteration 12, loss = 0.28466641
Validation score: 0.897383
Iteration 13, loss = 0.28332199
Validation score: 0.898416
Validation score did not improve more than tol=0.000100 for 10 conse
cutive epochs. Stopping.
Iteration 1, loss = 0.34135647
Validation score: 0.890152
Iteration 2, loss = 0.31283224
Validation score: 0.889807
Iteration 3, loss = 0.31074299
Validation score: 0.890152
Iteration 4, loss = 0.30942286
Validation score: 0.891873
Iteration 5, loss = 0.30859754
Validation score: 0.891529
Iteration 6, loss = 0.30773727
Validation score: 0.890152
Iteration 7, loss = 0.30757123
Validation score: 0.892218
Iteration 8, loss = 0.30685670
Validation score: 0.891873
Iteration 9, loss = 0.30692636
Validation score: 0.891873
Iteration 10, loss = 0.30627212
Validation score: 0.892218
Iteration 11, loss = 0.30614969
Validation score: 0.891873
Iteration 12, loss = 0.30592975
Validation score: 0.891529
Iteration 13, loss = 0.30583908
Validation score: 0.892218
Iteration 14, loss = 0.30602981
Validation score: 0.891873
Iteration 15, loss = 0.30539011
Validation score: 0.892562
Iteration 16, loss = 0.30552810
Validation score: 0.891873
Iteration 17, loss = 0.30488838
```

```
Validation score: 0.890840
Iteration 18, loss = 0.30500280
Validation score: 0.892218
Iteration 19, loss = 0.30460776
Validation score: 0.892562
Iteration 20, loss = 0.30463070
Validation score: 0.891873
Iteration 21, loss = 0.30454815
Validation score: 0.891873
Iteration 22, loss = 0.30458575
Validation score: 0.890840
Iteration 23, loss = 0.30451712
Validation score: 0.891529
Iteration 24, loss = 0.30437507
Validation score: 0.892562
Iteration 25, loss = 0.30435229
Validation score: 0.892562
Iteration 26, loss = 0.30445594
Validation score: 0.892562
Validation score did not improve more than tol=0.000100 for 10 conse
cutive epochs. Stopping.
```

Out[55]:

```
GridSearchCV(cv=3, error_score='raise-deprecating',
       estimator=MLPClassifier(activation='relu', alpha=0.0001, batc
h_size='auto', beta_1=0.9,
       beta_2=0.999, early_stopping=True, epsilon=1e-08,
       hidden_layer_sizes=(100,), learning_rate='constant',
       learning_rate_init=0.001, max_iter=200, momentum=0.9,
       n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
       random_state=101, shuffle=True, solver='adam', tol=0.0001,
       validation_fraction=0.1, verbose=True, warm_start=False),
       fit_params=None, iid='warn', n_jobs=-1,
       param_grid=[{'hidden_layer_sizes': [(3,), (128,)], 'activatio
n': ['logistic', 'relu', 'identity'], 'solver': ['adam'], 'batch_siz
e': [64], 'shuffle': [True], 'learning_rate_init': [0.0001, 0.001],
'n_iter_no_change': [10], 'max_iter': [200, 500], 'warm_start': [Tru
e, False]}, {'hidden_layer_sizes': ...[64], 'learning_rate_init':
[0.0001, 0.001], 'n_iter_no_change': [10], 'warm_start': [True, Fals
e]}],
       pre_dispatch='2*n_jobs', refit=True, return_train_score='war
n',
       scoring=None, verbose=0)
```

a. Did feature selection help here? Any change in the network architecture? What inputs are being used as the network input?

In [56]:

```
print("Best Parameters of NN: ", cv.best_params_)
print("Best Parameters of RFE NN: ", rfe_cv.best_params_)
print("Best Parameters of modelSelect NN: ", modelSelect_cv.best_params_)
print("\n\n")

print("GridSearch:")
printMLPArchitecture(cv.best_estimator_)
print("\n")
print("RFE:")
printMLPArchitecture(rfe_cv.best_estimator_)
print("\n")
print("modelSelect:")
printMLPArchitecture(modelSelect_cv.best_estimator_)
print("\n")
```

```
Best Parameters of NN:  {'activation': 'logistic', 'batch_size': 64,
'hidden_layer_sizes': (128,), 'learning_rate_init': 0.001, 'max_ite
r': 200, 'n_iter_no_change': 10, 'shuffle': True, 'solver': 'adam',
'warm_start': True}
Best Parameters of RFE NN:  {'activation': 'relu', 'batch_size': 64,
'hidden_layer_sizes': (128,), 'learning_rate_init': 0.001, 'max_ite
r': 200, 'n_iter_no_change': 10, 'shuffle': True, 'solver': 'adam',
'warm_start': True}
Best Parameters of modelSelect NN:  {'activation': 'relu', 'batch_si
ze': 64, 'hidden_layer_sizes': (128,), 'learning_rate_init': 0.001,
'max_iter': 200, 'n_iter_no_change': 10, 'shuffle': True, 'solver':
'adam', 'warm_start': True}



GridSearch:
Number of Layers:  3
The First layer is Input Layer, and the last layer is the output lay
er
1 Layer with hidden size 149
2 Layer with hidden size 128
3 Layer with hidden size 1
The activation function:  logistic


RFE:
Number of Layers:  3
The First layer is Input Layer, and the last layer is the output lay
er
1 Layer with hidden size 80
2 Layer with hidden size 128
3 Layer with hidden size 1
The activation function:  relu


modelSelect:
Number of Layers:  3
The First layer is Input Layer, and the last layer is the output lay
er
1 Layer with hidden size 15
2 Layer with hidden size 128
3 Layer with hidden size 1
The activation function:  relu
```

b. What is classification accuracy on training and test datasets? Is there any improvement in the outcome?

In [57]:

```
print("GridSearch NN Train accuracy:", cv.score(X_train_log, y_train_log))
print("GridSearch NN Test accuracy:", cv.score(X_test_log, y_test_log))
print("RFE NN Train accuracy:", rfe_cv.score(X_train_rfe, y_train_log))
print("RFE NNTest accuracy:", rfe_cv.score(X_test_rfe, y_test_log))
print("modelSelect NN Train accuracy:", modelSelect_cv.score(X_train_sel_model,
y_train_log))
print("modelSelect NN Test accuracmodelSelect_cvy:", modelSelect_cv.score(X_test
_sel_model, y_test_log))
```

```
GridSearch NN Train accuracy: 0.8991836875279854
GridSearch NN Test accuracy: 0.8985775134613839
RFE NN Train accuracy: 0.8983570419867047
RFE NNTest accuracy: 0.8982560475769509
modelSelect NN Train accuracy: 0.8982192677298246
modelSelect NN Test accuracmodelSelect_cvy: 0.8978542152214096
```

c. How many iterations are now needed to train this network?

In [58]:

```
print("Number of iterations GS ran: ",cv.best_estimator_.n_iter_)
print("Number of iterations rfe ran: ",rfe_cv.best_estimator_.n_iter_)
print("Number of iterations modelSelect ran: ",modelSelect_cv.best_estimator_.n_
iter_)
```

```
Number of iterations GS ran:  27
Number of iterations rfe ran:  13
Number of iterations modelSelect ran:  26
```

d. Do you see any sign of over-fitting?

In [ ]:

```

```

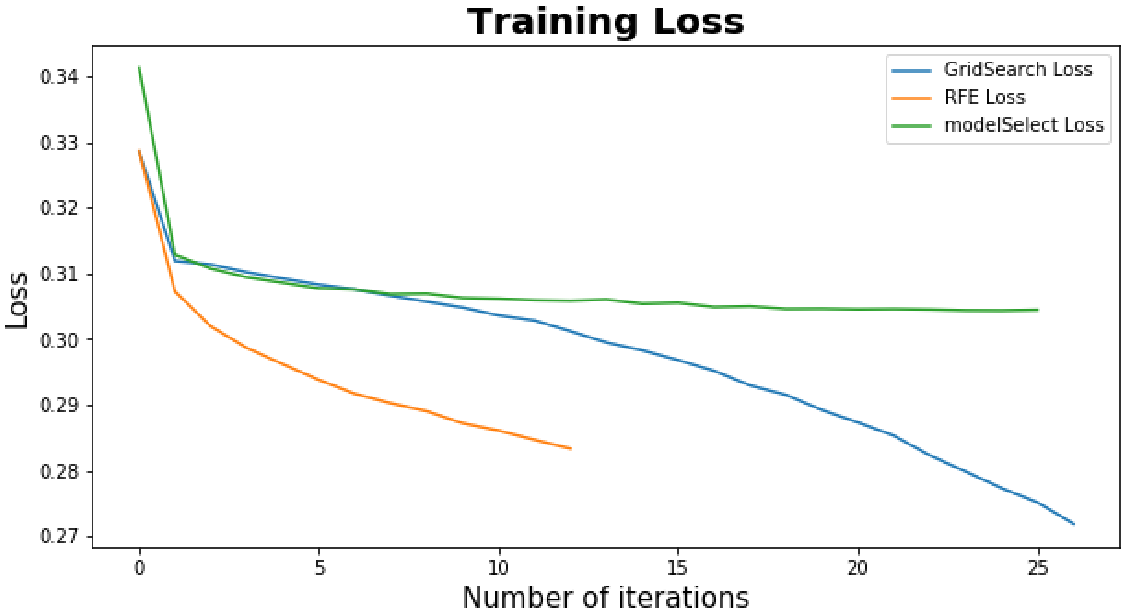
e. Did the training process converge and resulted in the best model?

In [59]:

```
fig = plt.figure(figsize=(10, 5))
plt.ylabel('Loss',fontsize=15)
plt.xlabel('Number of iterations',fontsize=15)
plt.title('Training Loss',fontsize=20,fontweight ="bold")
plt.plot(cv.best_estimator_.loss_curve_, label="GridSearch Loss")
plt.plot(rfe_cv.best_estimator_.loss_curve_, label="RFE Loss")
plt.plot(modelSelect_cv.best_estimator_.loss_curve_, label="modelSelect Loss")
plt.legend(loc='upper right')
```

Out[59]:

```
<matplotlib.legend.Legend at 0x7ff8d1e7b278>
```



4. Using the comparison methods, which of the models (i.e one with selected variables and another with all variables) appears to be better? From the better model, can you identify cars those could potential be “kicks”? Can you provide some descriptive summary of those cars? Is it easy to comprehend the performance of the best neural network model for decision making?

In [60]:

```
print("GridSearch Classification Report: ")
y_pred = cv.predict(X_test_log)
print(classification_report(y_test_log, y_pred))
print("\n\nRFE Classification Report: ")
y_pred = rfe_cv.predict(X_test_rfe)
print(classification_report(y_test_log, y_pred))
print("\n\nmodelSelect Classification Report: ")
y_pred = modelSelect_cv.predict(X_test_sel_model)
print(classification_report(y_test_log, y_pred))
```

```
GridSearch Classification Report: 
              precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.86      0.26      0.40      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.88      0.63      0.67     12443
weighted avg       0.90      0.90      0.87     12443



RFE Classification Report: 
              precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.87      0.25      0.39      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.88      0.62      0.67     12443
weighted avg       0.90      0.90      0.87     12443



modelSelect Classification Report: 
              precision    recall  f1-score   support

           0       0.90      1.00      0.94     10832
           1       0.88      0.24      0.38      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.89      0.62      0.66     12443
weighted avg       0.90      0.90      0.87     12443
```

Task 5. Generating an Ensemble Model and Comparing Models

1. Generate an ensemble model to include the best regression model, best decision tree model, and best neural network model.

In [61]:

```
voting = VotingClassifier(estimators=[('dt', dt_model), ('lr', log_reg_model), (
'nn', nn_model)], voting='soft')
voting.fit(X_train_log, y_train_log)

y_pred_dt = dt_model.predict(X_test_log)
y_pred_log_reg = log_reg_model.predict(X_test_log)
y_pred_nn = nn_model.predict(X_test_log)
y_pred_ensemble = voting.predict(X_test_log)
```

```
Iteration 1, loss = 0.32857511
Validation score: 0.890496
Iteration 2, loss = 0.31191295
Validation score: 0.890496
Iteration 3, loss = 0.31137095
Validation score: 0.891185
Iteration 4, loss = 0.31018978
Validation score: 0.890840
Iteration 5, loss = 0.30922495
Validation score: 0.891529
Iteration 6, loss = 0.30831522
Validation score: 0.891529
Iteration 7, loss = 0.30758839
Validation score: 0.891185
Iteration 8, loss = 0.30660339
Validation score: 0.891185
Iteration 9, loss = 0.30571344
Validation score: 0.891185
Iteration 10, loss = 0.30481675
Validation score: 0.891185
Iteration 11, loss = 0.30362134
Validation score: 0.891185
Iteration 12, loss = 0.30283829
Validation score: 0.892218
Iteration 13, loss = 0.30120792
Validation score: 0.892218
Iteration 14, loss = 0.29949388
Validation score: 0.892218
Iteration 15, loss = 0.29829281
Validation score: 0.892906
Iteration 16, loss = 0.29680944
Validation score: 0.893251
Iteration 17, loss = 0.29516226
Validation score: 0.892906
Iteration 18, loss = 0.29294147
Validation score: 0.892562
Iteration 19, loss = 0.29150188
Validation score: 0.892562
Iteration 20, loss = 0.28919079
Validation score: 0.892218
Iteration 21, loss = 0.28730894
Validation score: 0.892906
Iteration 22, loss = 0.28529079
Validation score: 0.891873
Iteration 23, loss = 0.28228859
Validation score: 0.892906
Iteration 24, loss = 0.27981098
Validation score: 0.892218
Iteration 25, loss = 0.27729154
Validation score: 0.892218
Iteration 26, loss = 0.27511337
Validation score: 0.892562
Iteration 27, loss = 0.27187226
Validation score: 0.892218
Validation score did not improve more than tol=0.000100 for 10 conse
cutive epochs. Stopping.
```

a. Does the Ensemble model outperform the underlying models? Resonate your answer.

In [62]:

```
print("Report for DT: \n",classification_report(y_test_log, y_pred_dt))
print("\nReport for Logistic Regression: \n",classification_report(y_test_log, y
_pred_log_reg))
print("\nReport for NN: \n",classification_report(y_test_log, y_pred_nn))
print("\nReport for Ensemble: \n",classification_report(y_test_log, y_pred_ensem
ble))
```

```
Report for DT: 
               precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.83      0.27      0.40      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.87      0.63      0.67     12443
weighted avg       0.89      0.90      0.87     12443


Report for Logistic Regression: 
               precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.84      0.27      0.40      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.87      0.63      0.67     12443
weighted avg       0.89      0.90      0.87     12443


Report for NN: 
               precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.86      0.26      0.40      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.88      0.63      0.67     12443
weighted avg       0.90      0.90      0.87     12443


Report for Ensemble: 
               precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.84      0.26      0.40      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.87      0.63      0.67     12443
weighted avg       0.89      0.90      0.87     12443
```

2. Use the comparison methods (or the comparison node) to compare the best decision tree model, the best regression model, the best neural network model and the ensemble model.

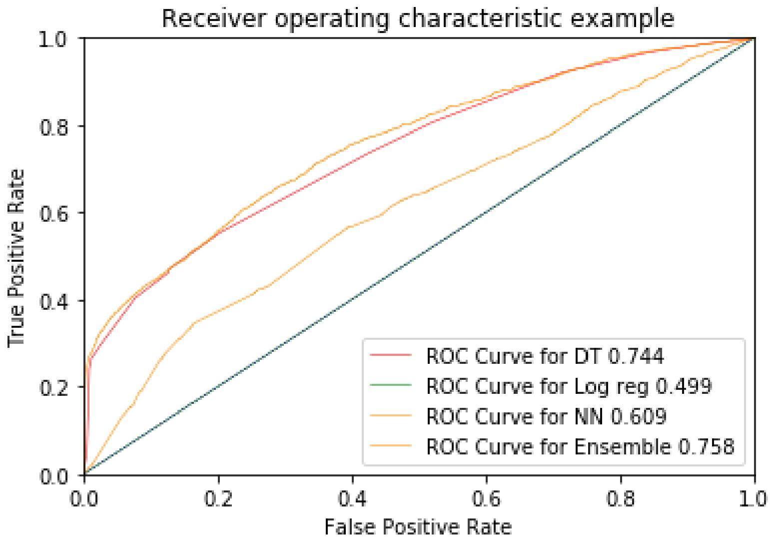
a. Discuss the findings led by (a) ROC Chart (and Index); (b) Score Ranking (or Accuracy Score); (c) Fit Statistics; (or Classification report) and (4) Output.

(a) ROC Chart (and Index)

In [63]:

```
#### ROC

y_pred_proba_dt = dt_model.predict_proba(X_test)
y_pred_proba_log_reg = log_reg_model.predict_proba(X_test)
y_pred_proba_nn = nn_model.predict_proba(X_test)
y_pred_proba_ensemble = voting.predict_proba(X_test_log)

roc_index_dt = roc_auc_score(y_test, y_pred_proba_dt[:, 1])
roc_index_log_reg = roc_auc_score(y_test, y_pred_proba_log_reg[:, 1])
roc_index_nn = roc_auc_score(y_test, y_pred_proba_nn[:, 1])
roc_index_ensemble = roc_auc_score(y_test_log, y_pred_proba_ensemble[:, 1])


print("ROC index on test for DT:", roc_index_dt)
print("ROC index on test for logistic regression:", roc_index_log_reg)
print("ROC index on test for NN:", roc_index_nn)
print("ROC index on voting classifier:", roc_index_ensemble)


fpr_dt, tpr_dt, thresholds_dt = roc_curve(y_test, y_pred_proba_dt[:,1])
fpr_log_reg, tpr_log_reg, thresholds_log_reg = roc_curve(y_test, y_pred_proba_lo
g_reg[:,1])
fpr_nn, tpr_nn, thresholds_nn = roc_curve(y_test, y_pred_proba_nn[:,1])
fpr_ensemble, tpr_ensemble, thresholds_ensemble = roc_curve(y_test, y_pred_proba
_ensemble[:,1])


plt.plot(fpr_dt, tpr_dt, label='ROC Curve for DT {:.3f}'.format(roc_index_dt), c
olor='red', lw=0.5)
plt.plot(fpr_log_reg, tpr_log_reg, label='ROC Curve for Log reg {:.3f}'.format(r
oc_index_log_reg), color='green', lw=0.5)
plt.plot(fpr_nn, tpr_nn, label='ROC Curve for NN {:.3f}'.format(roc_index_nn), c
olor='darkorange', lw=0.5)
plt.plot(fpr_ensemble, tpr_ensemble, label='ROC Curve for Ensemble {:.3f}'.forma
t(roc_index_ensemble), color='darkorange', lw=0.5)

plt.plot([0, 1], [0, 1], color='navy', lw=0.5, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic example')
plt.legend(loc="lower right")
plt.show()
```

```
ROC index on test for DT: 0.7437187226939606
ROC index on test for logistic regression: 0.49947161524306216
ROC index on test for NN: 0.6089292640056774
ROC index on voting classifier: 0.7583617224454842
```



(b) Score Ranking (or Accuracy Score)

In [64]:

```
print("Accuracy score on test for DT:", accuracy_score(y_test_log, y_pred_dt))
print("Accuracy score on test for Logistic Regression:", accuracy_score(y_test_l
og, y_pred_log_reg))
print("Accuracy score on test for NN:", accuracy_score(y_test_log, y_pred_nn))
print("Accuracy score on test for Ensemble:", accuracy_score(y_test_log, y_pred_
ensemble))
```

```
Accuracy score on test for DT: 0.8980149481636261
Accuracy score on test for Logistic Regression: 0.8984167805191674
Accuracy score on test for NN: 0.8985775134613839
Accuracy score on test for Ensemble: 0.8981756811058427
```

(c) Classification report

In [65]:

```
print("Report for DT: \n",classification_report(y_test_log, y_pred_dt))
print("\nReport for Logistic Regression: \n",classification_report(y_test_log, y
_pred_log_reg))
print("\nReport for NN: \n",classification_report(y_test_log, y_pred_nn))
print("\nReport for Ensemble: \n",classification_report(y_test_log, y_pred_ensem
ble))
```

```
Report for DT:
               precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.83      0.27      0.40      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.87      0.63      0.67     12443
weighted avg       0.89      0.90      0.87     12443


Report for Logistic Regression:
               precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.84      0.27      0.40      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.87      0.63      0.67     12443
weighted avg       0.89      0.90      0.87     12443


Report for NN:
               precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.86      0.26      0.40      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.88      0.63      0.67     12443
weighted avg       0.90      0.90      0.87     12443


Report for Ensemble:
               precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.84      0.26      0.40      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.87      0.63      0.67     12443
weighted avg       0.89      0.90      0.87     12443
```

(d) Output

In [ ]:

```

```

b. Do all the models agree on the cars characteristics? How do they vary?

In [ ]:

```

```

Task 6. Final Remarks: Decision Making

1. Finally, based on all models and analysis, is there

2. Can you summarise positives and negatives of each predictive modelling method based on this analysis?

3. How the outcome of this study can be used by decision makers?

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```