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.

4. Put confusion matrix after all training
'''


%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:  0.1294965763333012
```

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

In [6]:

```
#### PREPROCESSING STATEGY
NEW_STATEGY = True
ResamplingMethod = 'ros' #['ros', 'rus']
if NEW_STATEGY:
    print("Using New Preprocessing Strategy")
    using_cat = False
    categorial_cols = ['Auction', 'VehYear', 'Make', 'Color', 'Transmission','WheelTypeID', 'WheelType', 'Nationality', 'Size', 'TopThreeAmericanName','PRIMEUNIT','AUCGUART','VNST','IsOnlineSale','ForSale' ] # Replaced by the most common
    interval_cols = ['VehOdo','MMRAcquisitionAuctionAveragePrice','MMRAcquisitionAuctionCleanPrice','MMRAcquisitionRetailAveragePrice','MMRAcquisitonRetailCleanPrice','VehBCost','WarrantyCost' ]
    drop_cols = ['PurchaseID','PurchaseDate','PurchaseTimestamp']
    questionMark_data = ['MMRCurrentAuctionAveragePrice','MMRCurrentAuctionCleanPrice','MMRCurrentRetailAveragePrice','MMRCurrentRetailCleanPrice','MMRCurrentRetailRatio']
    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 Data")
else:
    print("Using Old Preprocessing Strategy")
    drop_cols = ['PurchaseID','PurchaseDate']
    categorial_cols = ['Auction', 'VehYear', 'Make', 'Color', 'Transmission','WheelTypeID', 'WheelType', 'Nationality', 'Size', 'TopThreeAmericanName','PRIMEUNIT','AUCGUART','VNST','IsOnlineSale','ForSale' ] # Replaced by the most common
    interval_cols = ['PurchaseTimestamp', 'VehOdo','MMRAcquisitionAuctionAveragePrice','MMRAcquisitionAuctionCleanPrice','MMRAcquisitionRetailAveragePrice','MMRAcquisitonRetailCleanPrice','MMRCurrentAuctionAveragePrice','MMRCurrentAuctionCleanPrice','MMRCurrentRetailAveragePrice','MMRCurrentRetailCleanPrice','MMRCurrentRetailRatio','VehBCost','WarrantyCost' ] # Replaced by the mean
    replaced_vals = ['?', '#VALUE!']

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

```
Using New Preprocessing Strategy
See [MMRCurrentAuctionAveragePriceMMRCurrentAuctionCleanPrice, MMRCurrentRetailAveragePrice, MMRCurrentRetailCleanPrice, MMRCurrentRetailRatio] as Interval Data
Total null before Replacing:  1691
```

In [7]:

```python
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       12/02/2009 10:00
freq                   242
Name: PurchaseDate, dtype: object
```

```
----------------------- COUNTS -----------------------
Five Most Common:  ['12/02/2009 10:00', '25/02/2009 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
ACURA           19
MINI            19
SUBARU          17
CADILLAC        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
UT     165
IL     165
WV     137
OR     136
WA     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
```

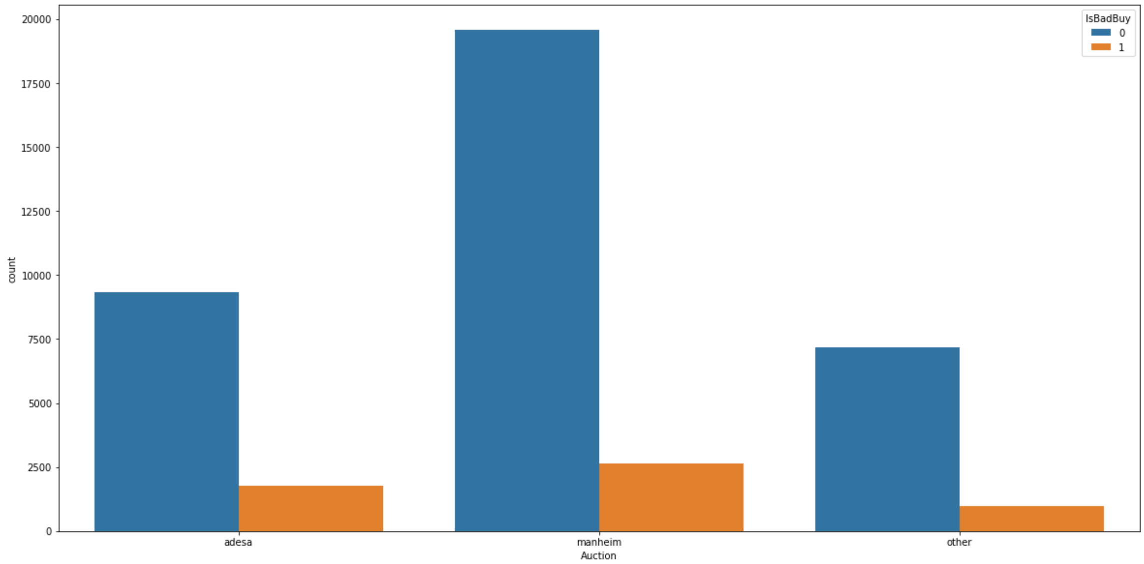
3. Can you identify any clear patterns by initial exploration of the data using histogram or box plot?

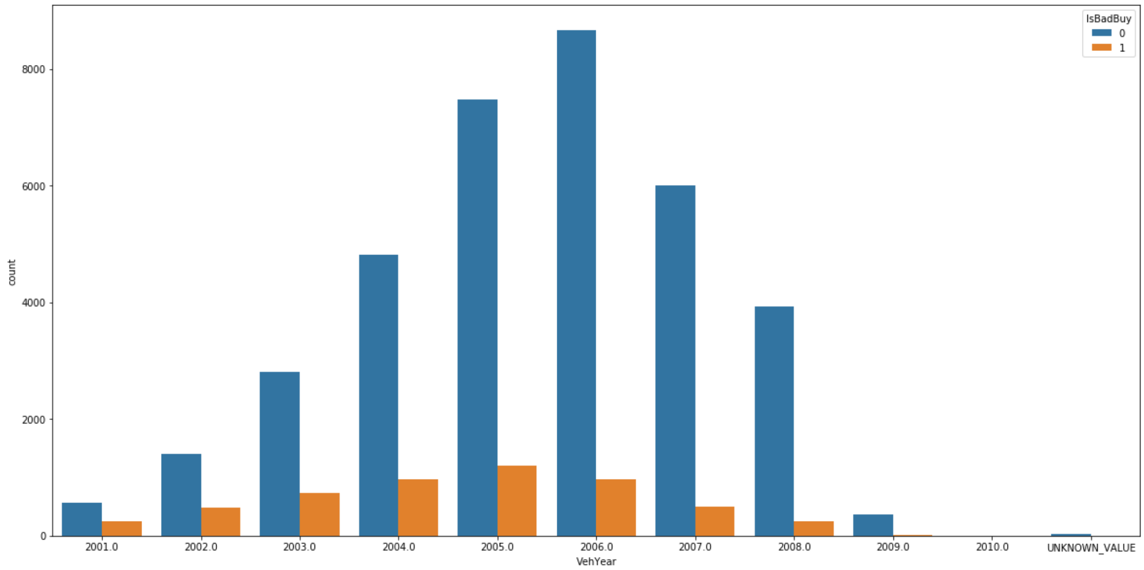
In [9]:

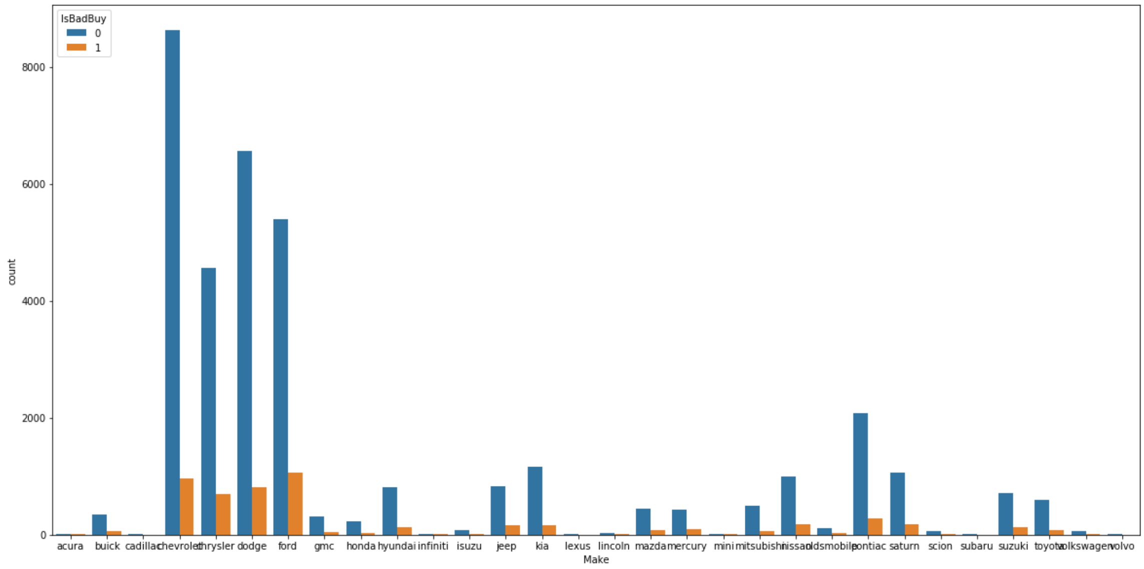
```
def plotAllCols (df):
    for colName in df.columns:
        plt.figure(figsize=(20,10))
        if colName in categorial_cols:
            ### if it's categorial column, plot hist diagram
            sns.countplot(x=colName, data = df, hue="IsBadBuy")
        elif colName in interval_cols:
    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

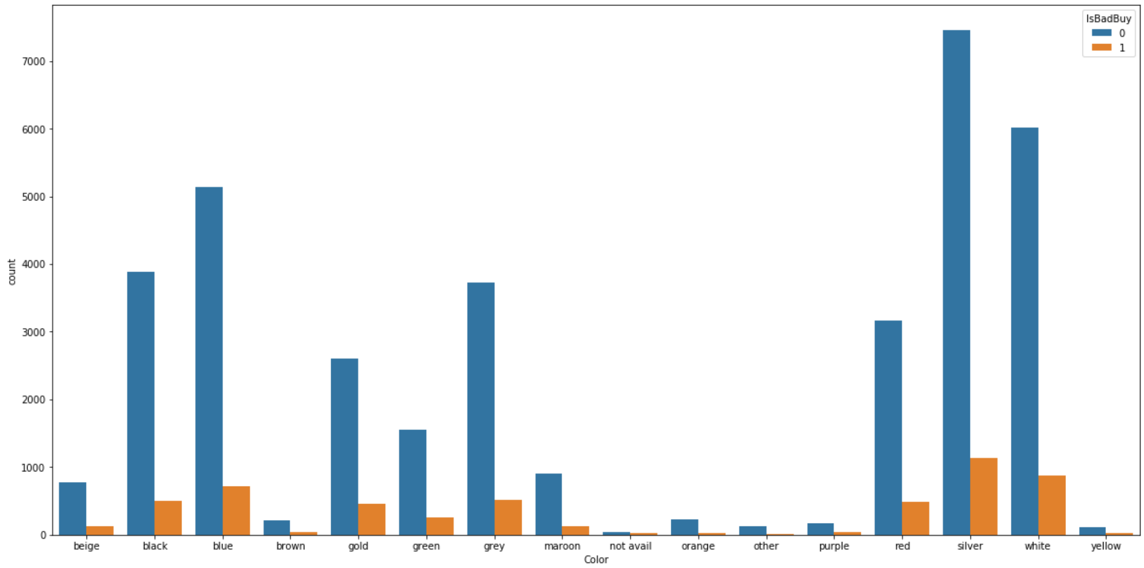
In [10]:

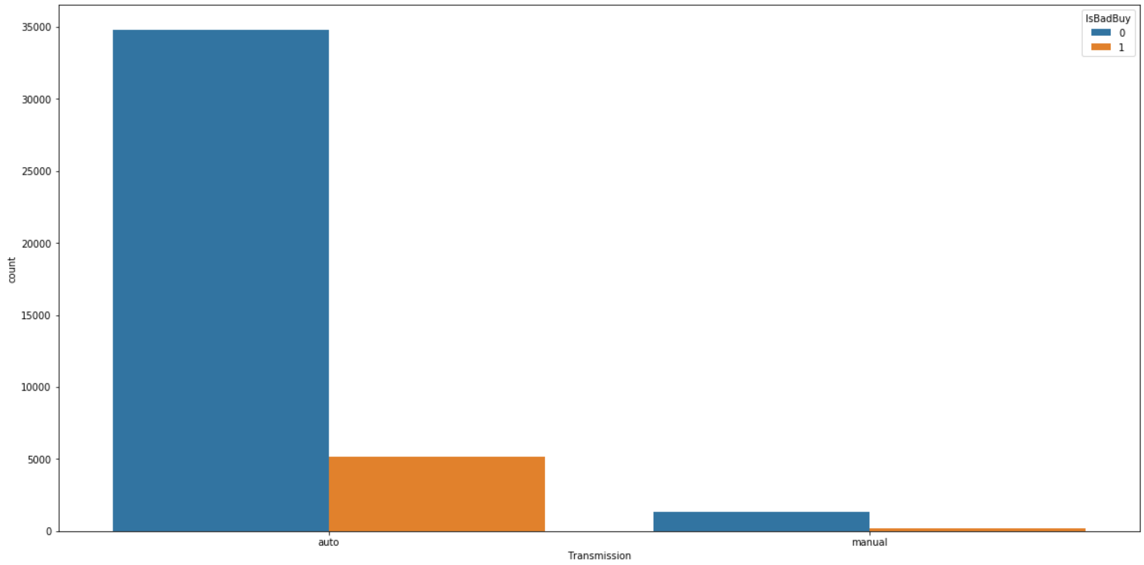
```
plotAllCols(df)
```

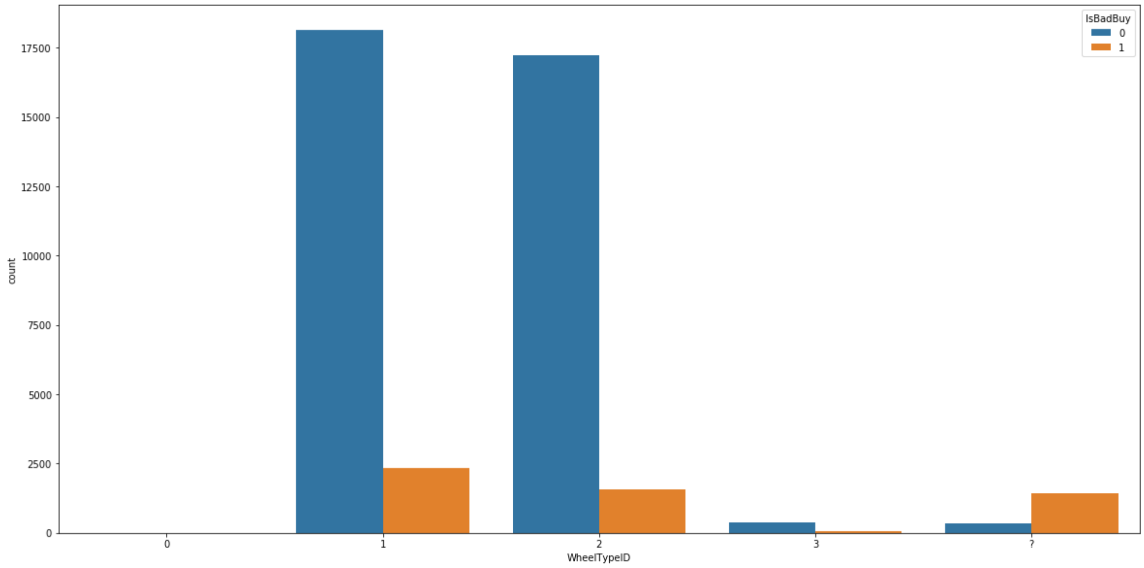

IsBadBuy
0
1
count
0
2500
5000
7500
10000
12500
15000
17500
20000
adesa
manheim
other
Auction

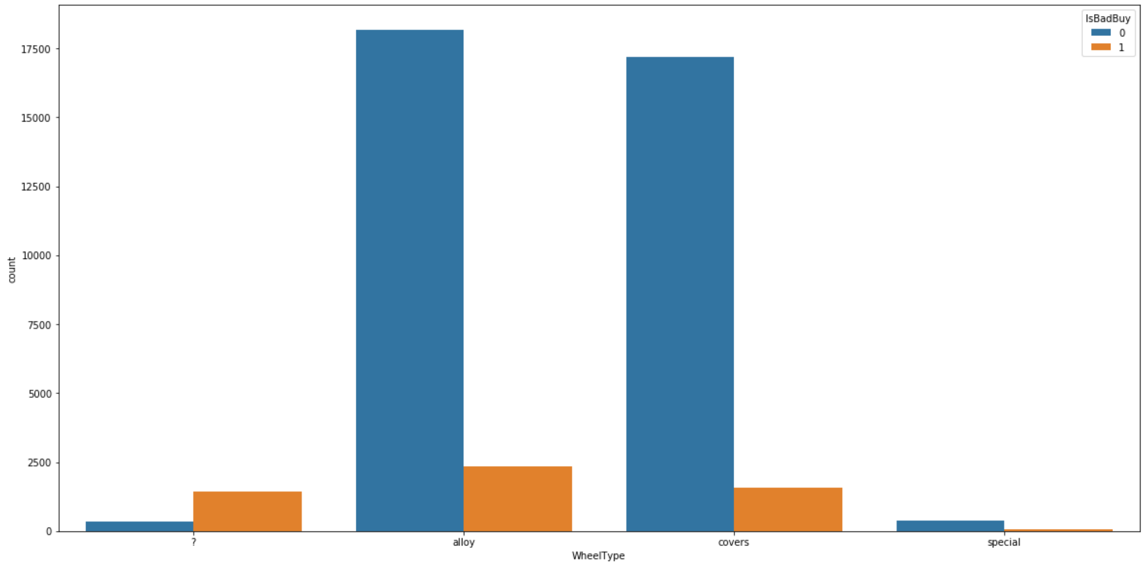

IsBadBuy
0
1
count
0
2000
4000
6000
8000
2001.0
2002.0
2003.0
2004.0
2005.0
2006.0
2007.0
2008.0
2009.0
2010.0
UNKNOWN_VALUE
VehYear

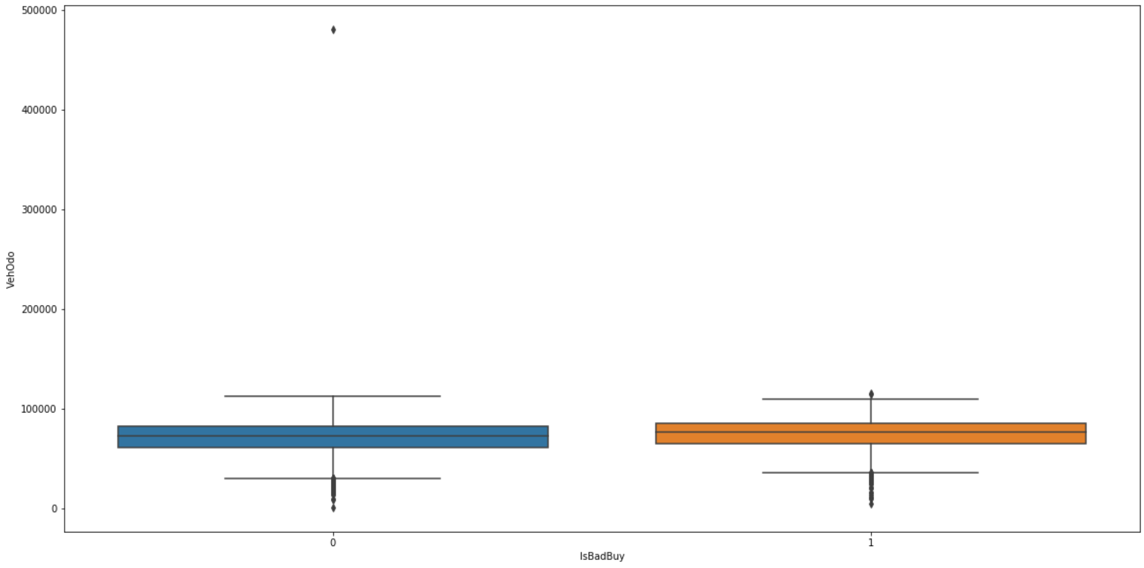

IsBadBuy
0
1
8000
6000
4000
2000
0
count
acura buick cadillac chevrolet chrysler dodge ford gmc honda hyundai infiniti isuzu jeep kia lexus lincoln mazda mercury mini mitsubishi nissan oldsmobile pontiac saturn scion subaru suzuki toyota volkswagen volvo
Make

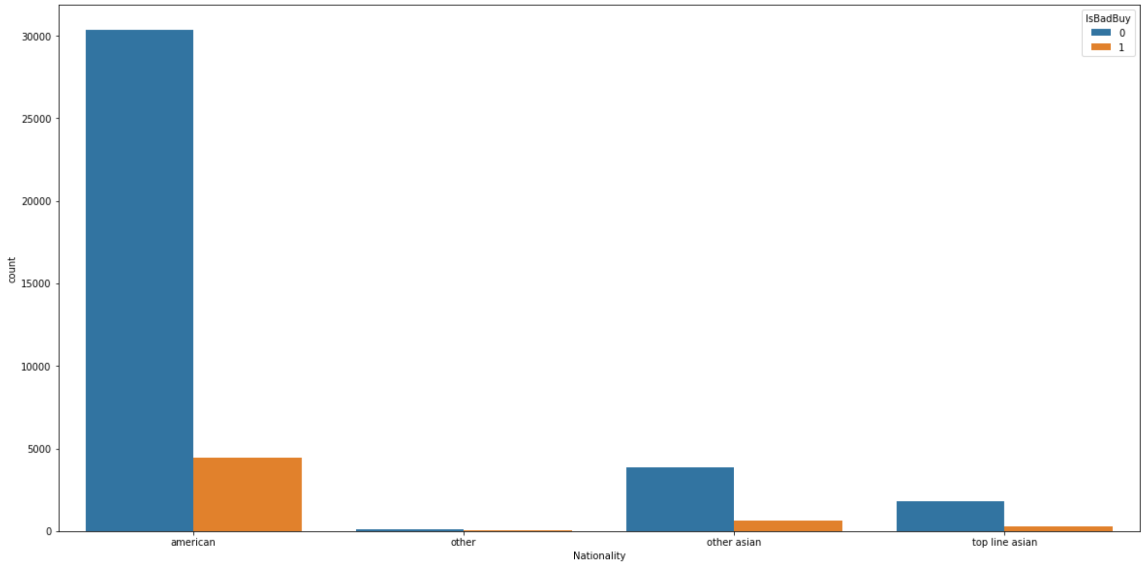

IsBadBuy
0
1
7000
6000
5000
4000
3000
2000
1000
0
count
beige black blue brown gold green grey maroon not avail orange other purple red silver white yellow
Color


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

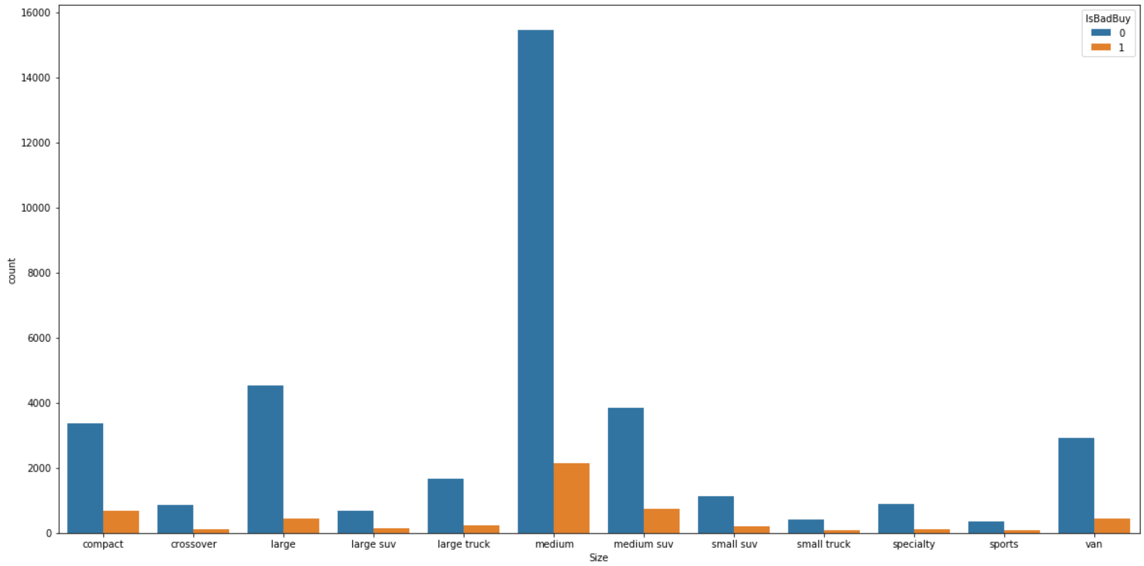

IsBadBuy
0
1
17500
15000
12500
10000
7500
5000
2500
0
count
0
1
2
3
?
WheelTypeID


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

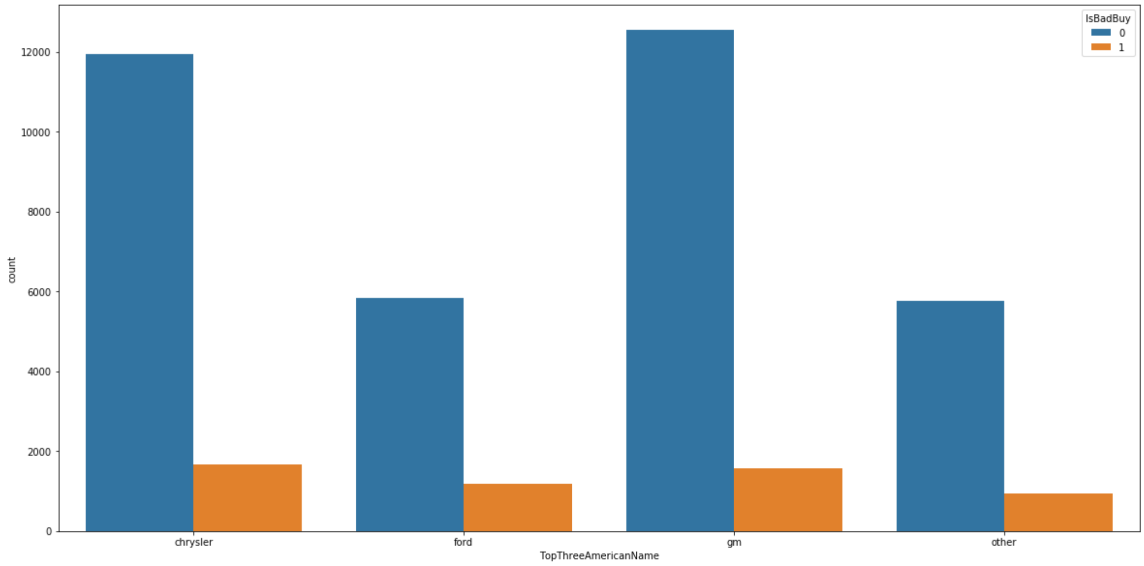

500000
400000
300000
200000
100000
0
VehOdo
0
1
IsBadBuy

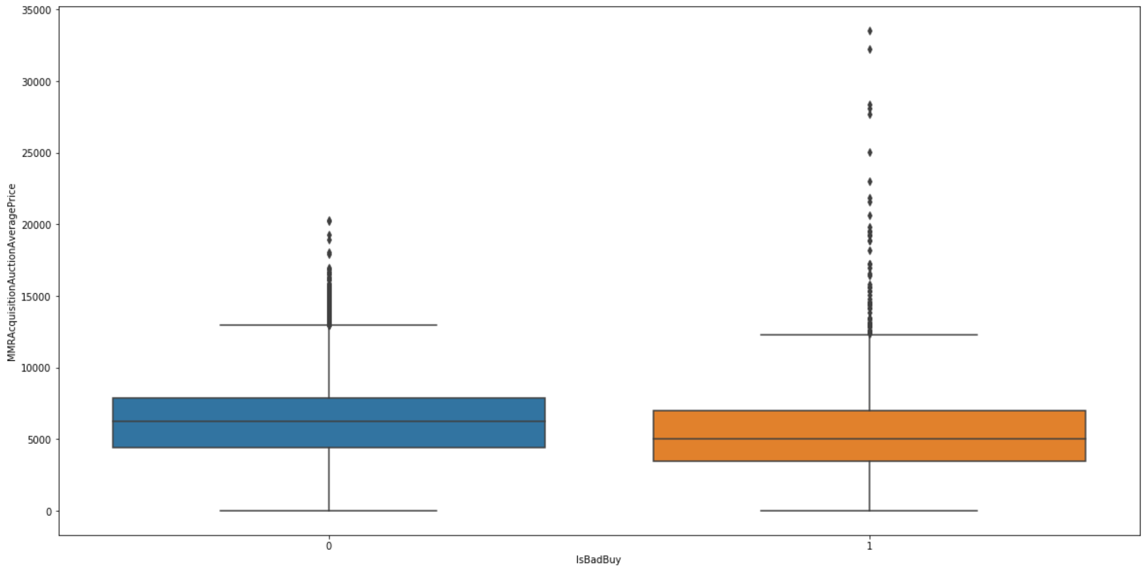


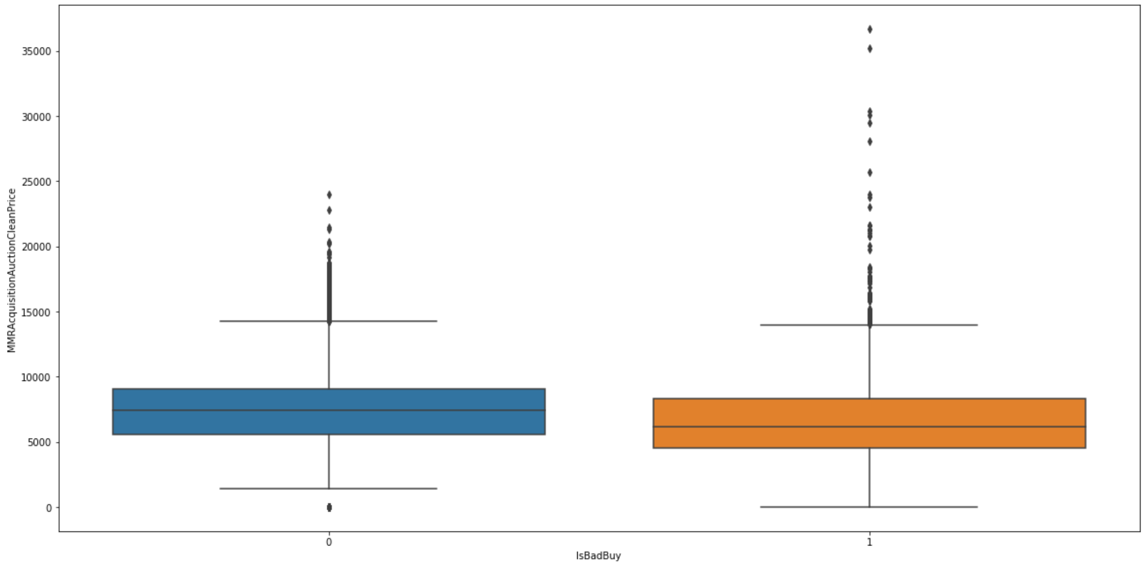
IsBadBuy
0
1
count
0
5000
10000
15000
20000
25000
30000
american
other
other asian
top line asian
Nationality

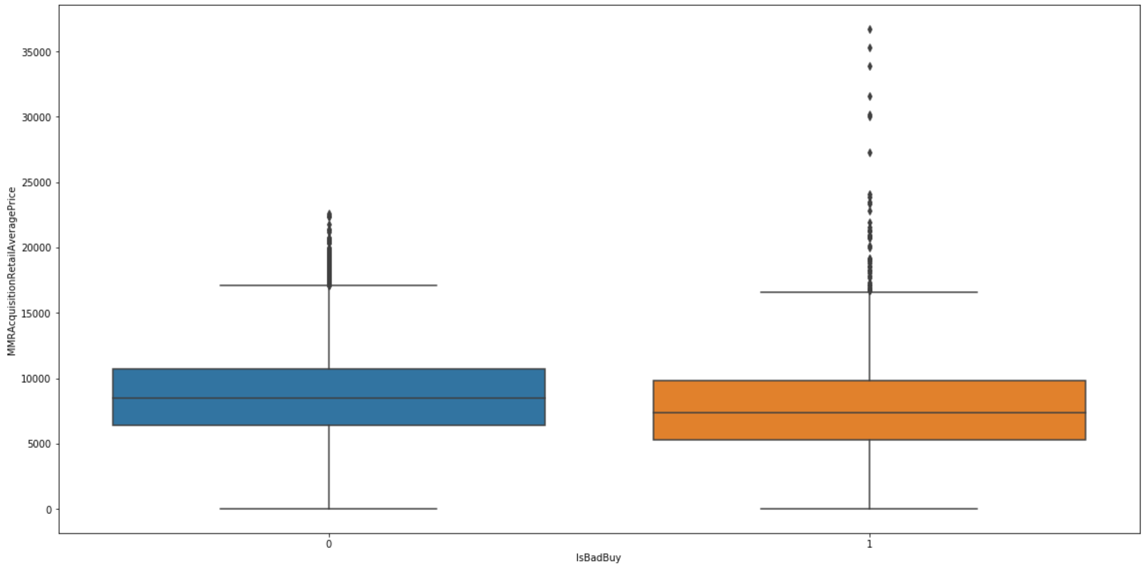


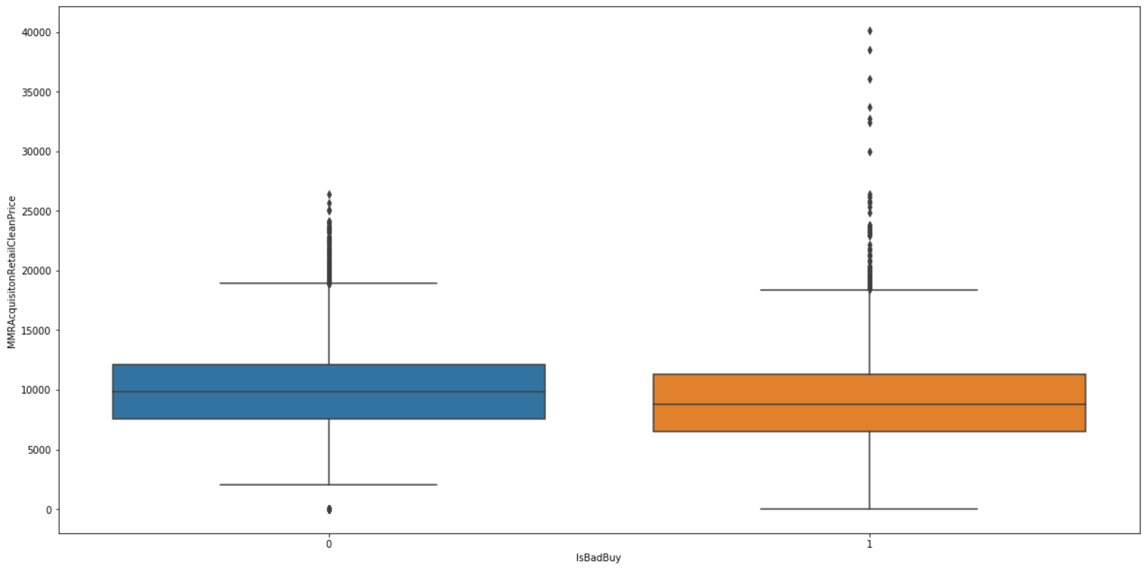
IsBadBuy
0
1
count
0
2000
4000
6000
8000
10000
12000
14000
16000
compact
crossover
large
large suv
large truck
medium
medium suv
small suv
small truck
specialty
sports
van
Size

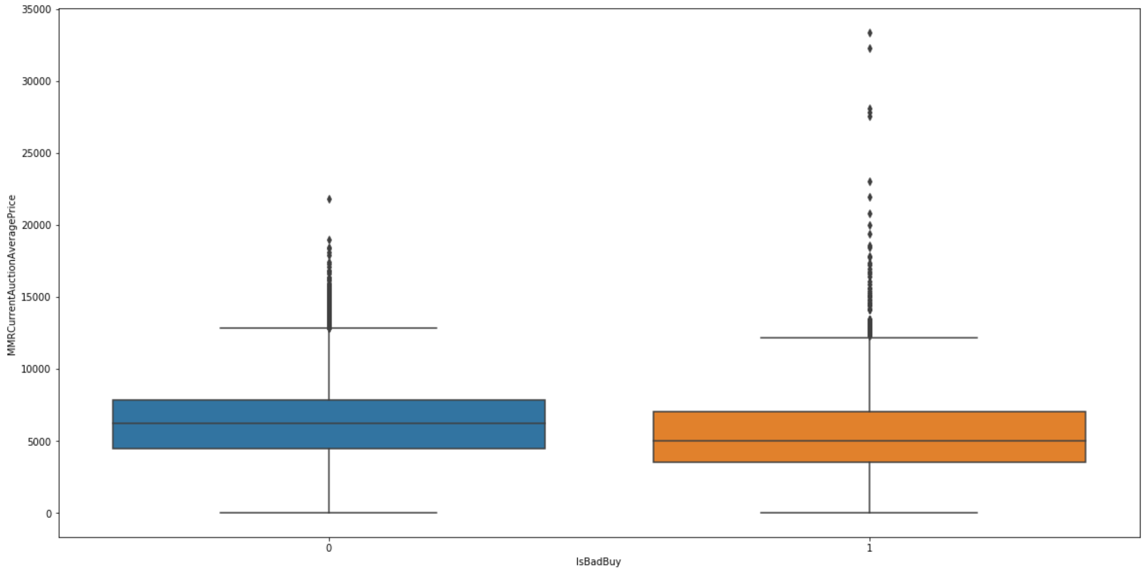

IsBadBuy
0
1
count
12000
10000
8000
6000
4000
2000
0
chrysler
ford
gm
other
TopThreeAmericanName

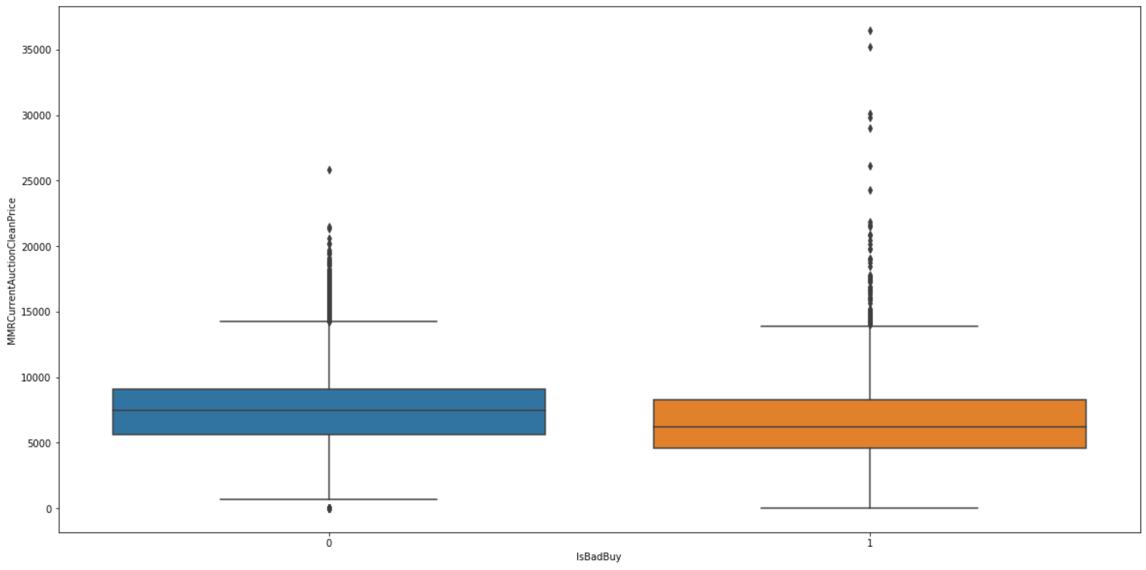

35000
30000
25000
20000
15000
10000
5000
0
MMRAcquisitionAuctionAveragePrice
0
1
IsBadBuy

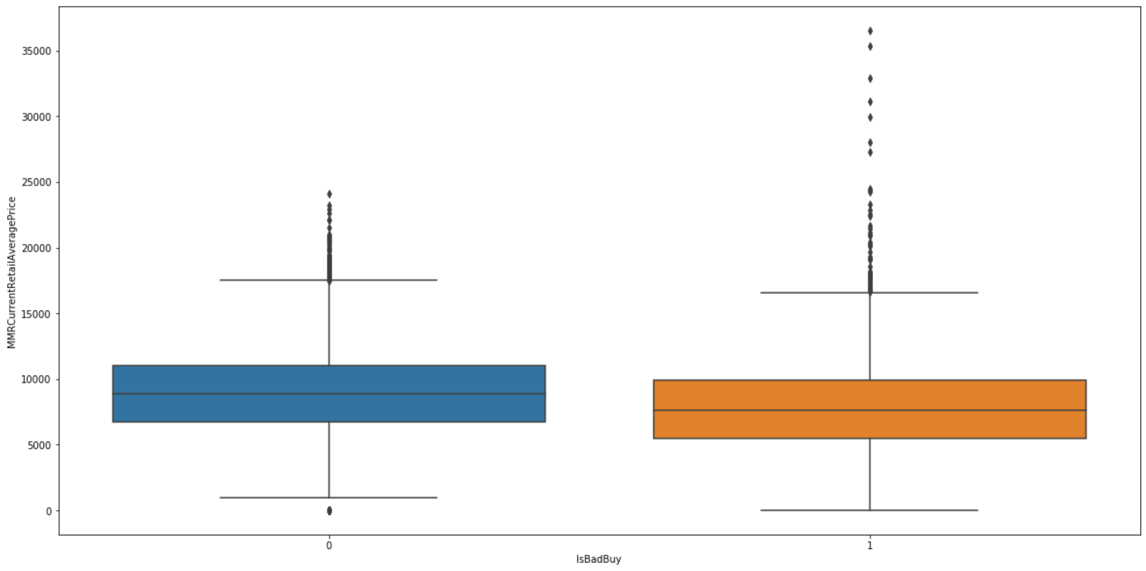

MMRAcquisitionAuctionCleanPrice
35000
30000
25000
20000
15000
10000
5000
0
0
1
IsBadBuy

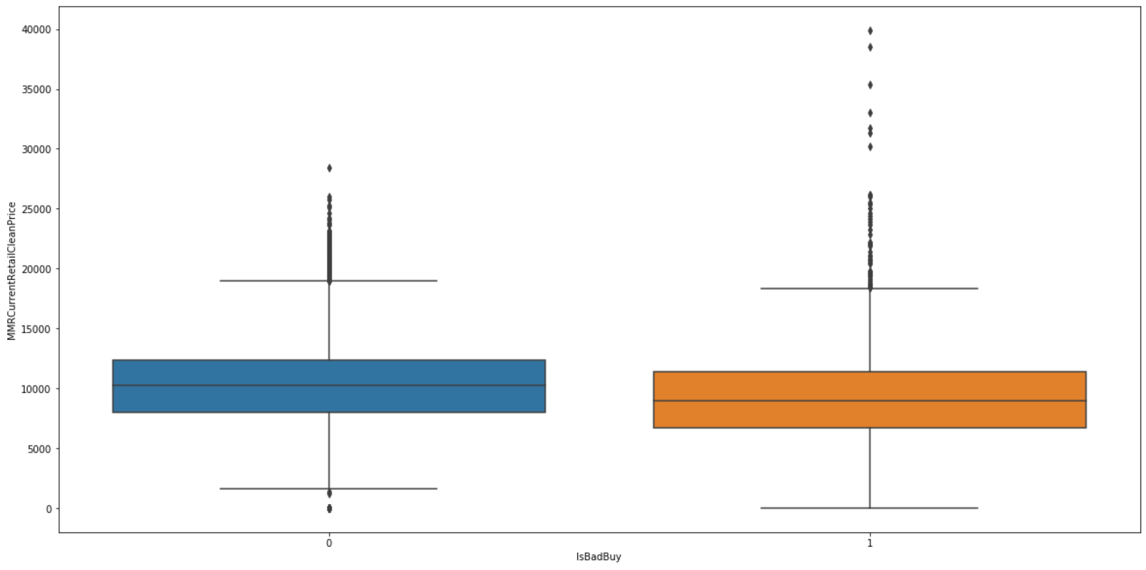

MMRAcquisitionRetailAveragePrice
35000
30000
25000
20000
15000
10000
5000
0
0
1
IsBadBuy

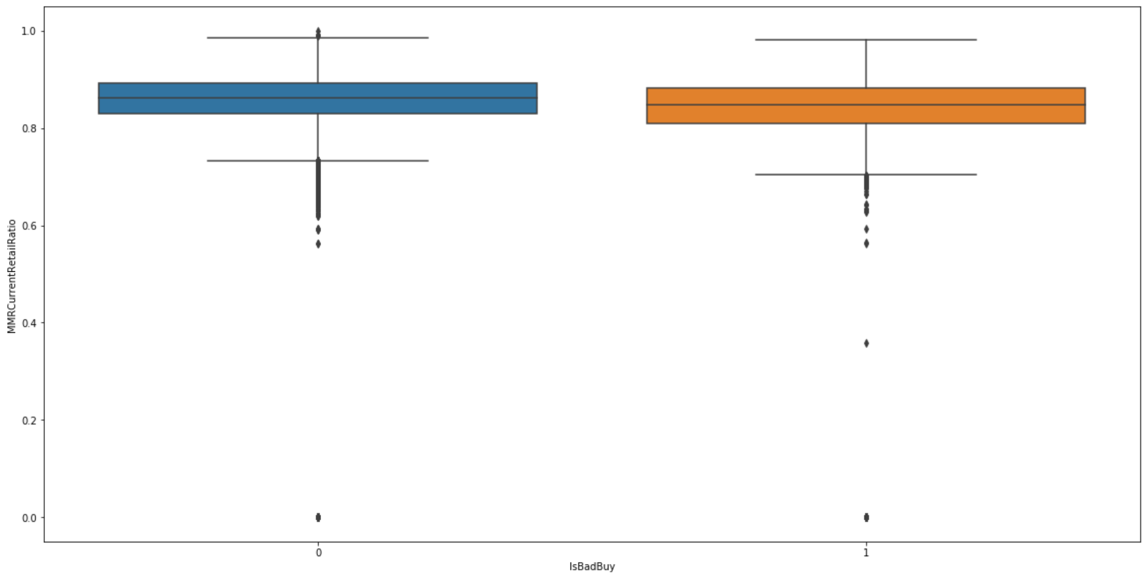

MMRAcquisitonRetailCleanPrice
40000
35000
30000
25000
20000
15000
10000
5000
0
0
1
IsBadBuy

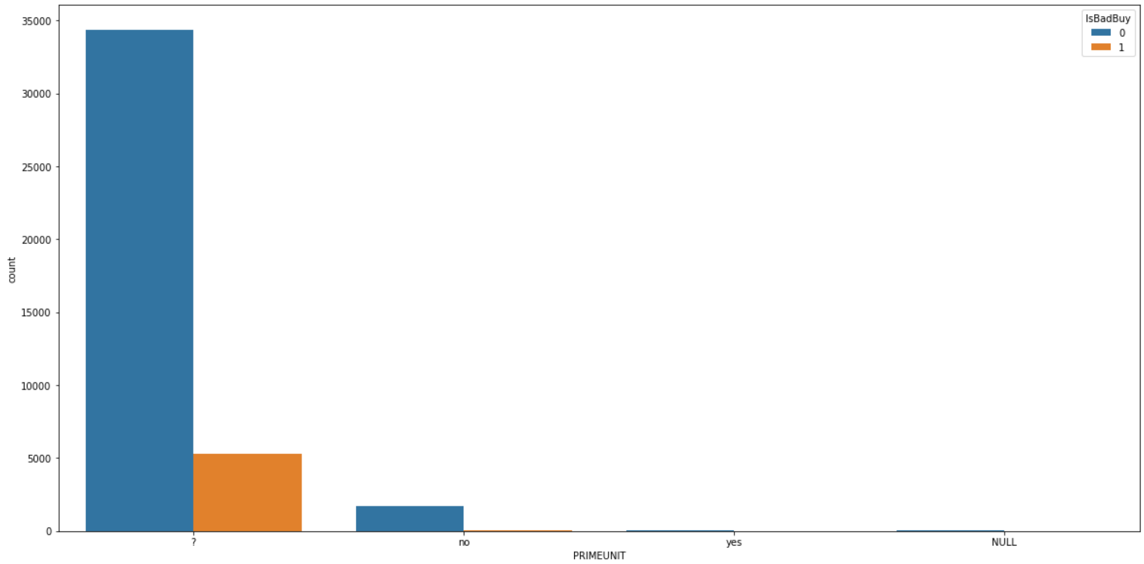

MMRCurrentAuctionAveragePrice
35000
30000
25000
20000
15000
10000
5000
0
0
1
IsBadBuy

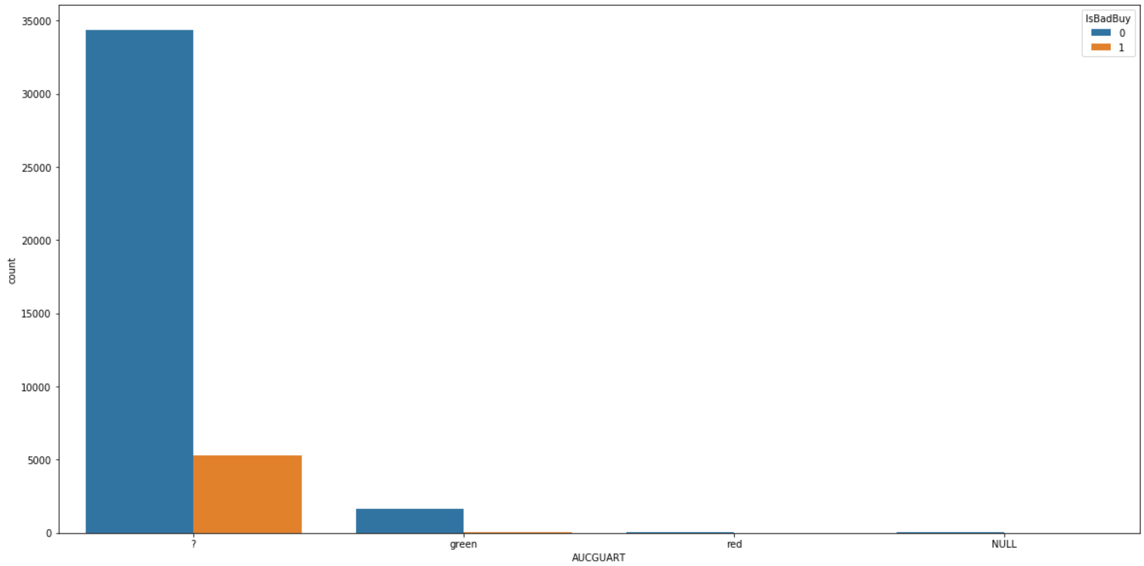

MMRCurrentAuctionCleanPrice
35000
30000
25000
20000
15000
10000
5000
0
0
1
IsBadBuy

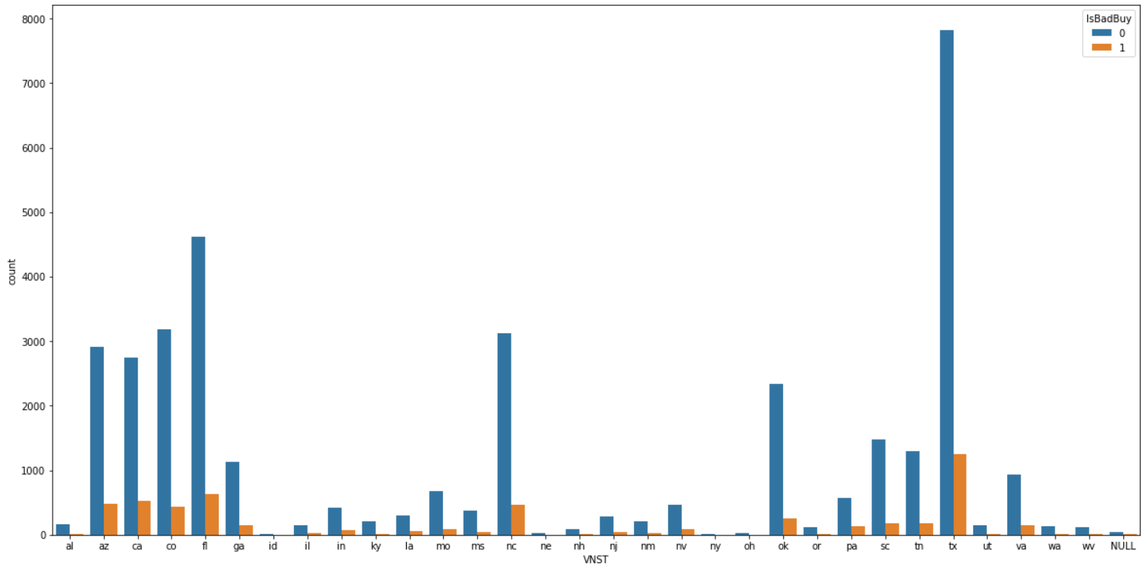

MMRCurrentRetailAveragePrice
35000
30000
25000
20000
15000
10000
5000
0
0
1
IsBadBuy

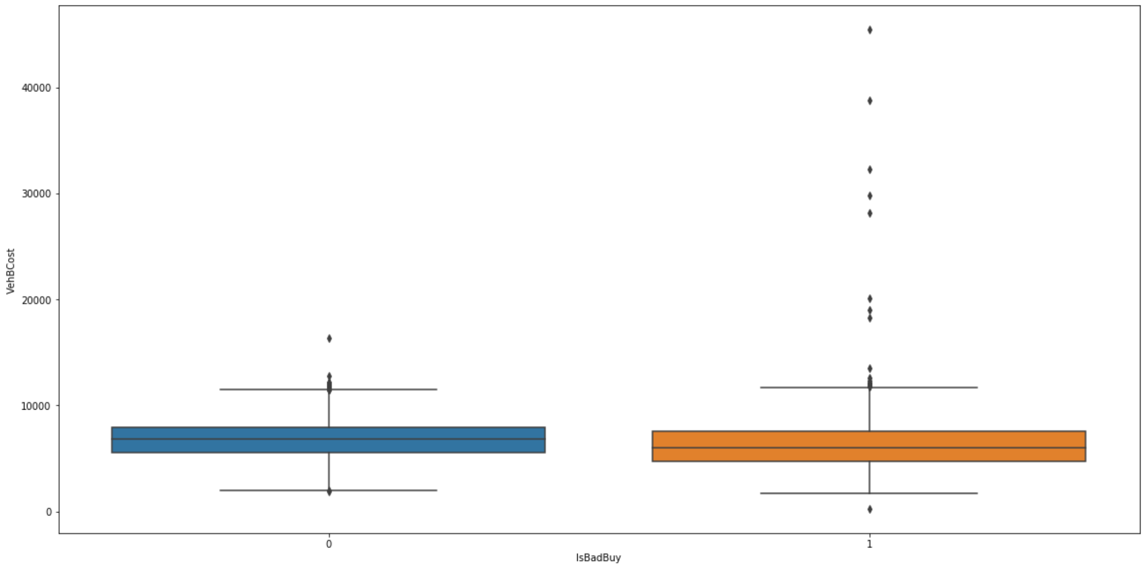

MMRCurrentRetailCleanPrice
40000
35000
30000
25000
20000
15000
10000
5000
0
0
1
IsBadBuy

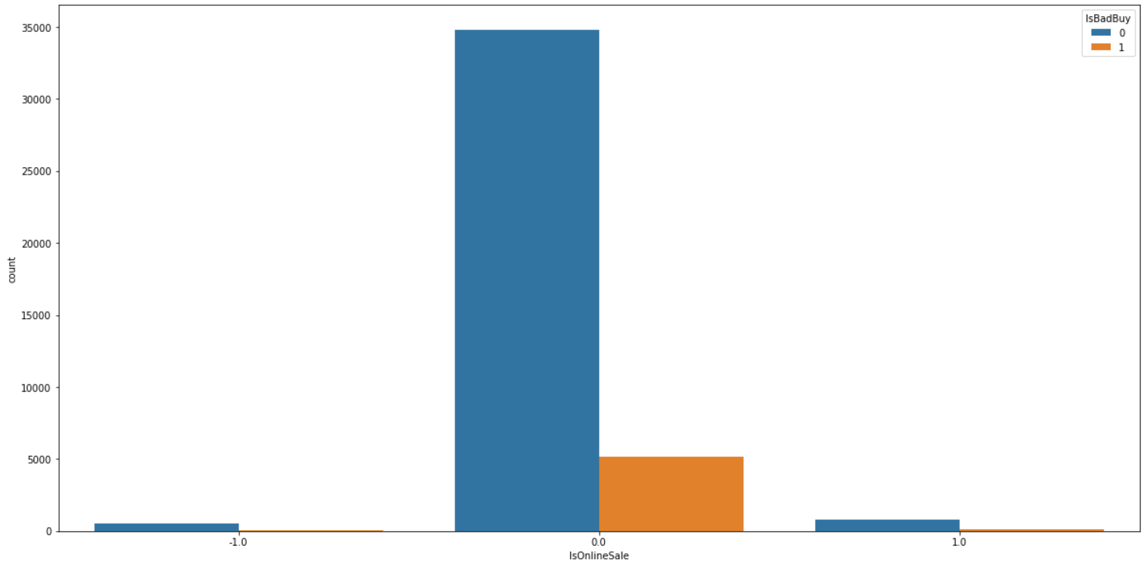

MMRCurrentRetailRatio
1.0
0.8
0.6
0.4
0.2
0.0
0
1
IsBadBuy

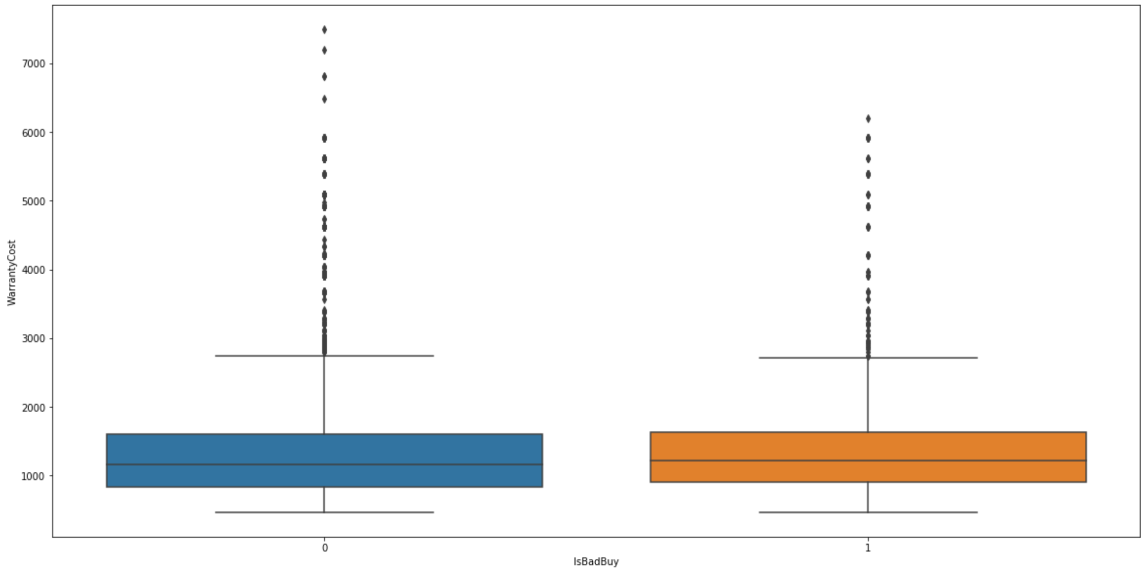

IsBadBuy
0
1
35000
30000
25000
20000
15000
10000
5000
0
count
?
no
yes
NULL
PRIMEUNIT

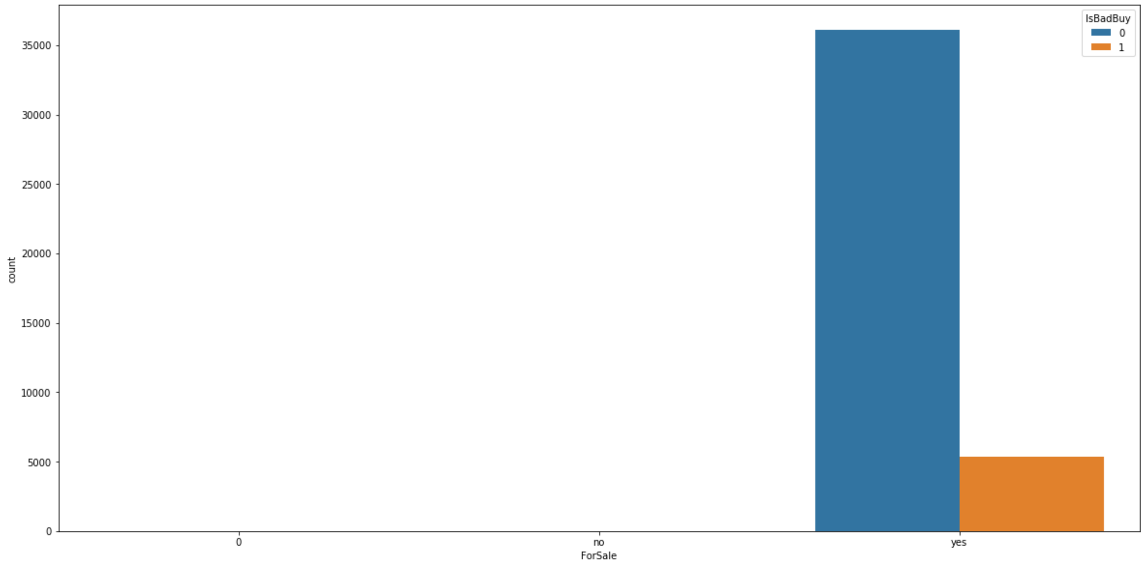

IsBadBuy
0
1
35000
30000
25000
20000
15000
10000
5000
0
count
?
green
red
NULL
AUCGUART


IsBadBuy
0
1
count
8000
7000
6000
5000
4000
3000
2000
1000
0
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
VNST


VehBCost
40000
30000
20000
10000
0
0
1
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 [11]:

```
# Change to the dummy
feature_names_beforDummy = df.drop("IsBadBuy", axis=1).columns

df = pd.get_dummies(df)

feature_names = df.drop("IsBadBuy", axis=1).columns
print("Num of Features:", len(feature_names))
print("\n\n")
print("The variables that included in the training: ")

for name in feature_names:
    print(str(name) + "\n")
```

```
Num of Features: 149



The variables that included in the training:
VehOdo

MMRAcquisitionAuctionAveragePrice

MMRAcquisitionAuctionCleanPrice

MMRAcquisitionRetailAveragePrice

MMRAcquisitonRetailCleanPrice

MMRCurrentAuctionAveragePrice

MMRCurrentAuctionCleanPrice

MMRCurrentRetailAveragePrice

MMRCurrentRetailCleanPrice

MMRCurrentRetailRatio

VehBCost

WarrantyCost

Auction_adesa

Auction_manheim

Auction_other

VehYear_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_2010.0

VehYear_UNKNOWN_VALUE

Make_acura

Make_buick
```

```
Make_cadillac

Make_chevrolet

Make_chrysler

Make_dodge

Make_ford

Make_gmc

Make_honda

Make_hyundai

Make_infiniti

Make_isuzu

Make_jeep

Make_kia

Make_lexus

Make_lincoln

Make_mazda

Make_mercury

Make_mini

Make_mitsubishi

Make_nissan

Make_oldsmobile

Make_pontiac

Make_saturn

Make_scion

Make_subaru

Make_suzuki

Make_toyota

Make_volkswagen

Make_volvo

Color_beige

Color_black

Color_blue
```

```
Color_brown

Color_gold

Color_green

Color_grey

Color_maroon

Color_not avail

Color_orange

Color_other

Color_purple

Color_red

Color_silver

Color_white

Color_yellow

Transmission_auto

Transmission_manual

WheelTypeID_0

WheelTypeID_1

WheelTypeID_2

WheelTypeID_3

WheelTypeID_?

WheelType_?

WheelType_alloy

WheelType_covers

WheelType_special

Nationality_american

Nationality_other

Nationality_other asian

Nationality_top line asian

Size_compact

Size_crossover
```

Size_large

Size_large suv

Size_large truck

Size_medium

Size_medium suv

Size_small suv

Size_small truck

Size_specialty

Size_sports

Size_van

TopThreeAmericanName_chrysler

TopThreeAmericanName_ford

TopThreeAmericanName_gm

TopThreeAmericanName_other

PRIMEUNIT_?

PRIMEUNIT_no

PRIMEUNIT_yes

PRIMEUNIT_NULL

AUCGUART_?

AUCGUART_green

AUCGUART_red

AUCGUART_NULL

VNST_al

VNST_az

VNST_ca

VNST_co

VNST_fl

VNST_ga

VNST_id

VNST_il

VNST_in

```
VNST_ky

VNST_la

VNST_mo

VNST_ms

VNST_nc

VNST_ne

VNST_nh

VNST_nj

VNST_nm

VNST_nv

VNST_ny

VNST_oh

VNST_ok

VNST_or

VNST_pa

VNST_sc

VNST_tn

VNST_tx

VNST_ut

VNST_va

VNST_wa

VNST_wv

VNST_NULL

IsOnlineSale_-1.0

IsOnlineSale_0.0

IsOnlineSale_1.0

ForSale_0

ForSale_no

ForSale_yes
```

In [12]:

```
# Ly

'''
We want to include all the features without droping the information that may be
 useful for the training.
Some columns are droped since they may not provide meaningful information for cl
assifying the kicks, such as the ID, Date and TimeStamp.
'''


# drop_cols = ['PurchaseID','PurchaseDate','PurchaseTimestamp']
```

Out[12]:

```
'\nWe want to include all the features without droping the informati
on that may be useful for the training.\nSome columns are droped sin
ce they may not provide meaningful information for classifying the k
icks, such as the ID, Date and TimeStamp.\n'
```

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

In [13]:

```
# strafying sampling, randomOverSampling -> For training set

'''
We use stratify sampling for splitting the training and the test sets, which mea
ns the portion of kicks
in the training and test set will be the same as the original dataset. Moreover,
in order to deal with the
imbalanced dataset, we use ROS and RUS to test the performance. However, we only
apply ROS and RUS on the training
dataset since we want the test dataset can have the similar distribution to the
 real world cases.
'''
```

Out[13]:

```
'\nWe use stratify sampling for splitting the training and the test
sets, which means the portion of kicks \nin the training and test se
t will be the same as the original dataset. Moreover, in order to de
al with the\nimbalanced dataset, we use ROS and RUS to test the perf
ormance. However, we only apply ROS and RUS on the training\ndataset
since we want the test dataset can have the similar distribution to
the real world cases.\n'
```

In [14]:

```
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")
```

```
Using ROS Resmapling
```

In [15]:

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

```
Number of Training:  50546
Number of Test:  12443
```

Task 2. Predictive Modeling Using Decision Trees

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

In [16]:

```
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 [17]:

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

Out[17]:

```
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 [18]:

```
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))
print("Confusion Matrix: \n", confusion_matrix(y_test, y_pred)) ## Confusion Mat
rix on the TestSet
```

```
Train accuracy: 0.9994856170616864
Test accuracy: 0.8286586835972033
              precision    recall  f1-score   support

           0       0.91      0.90      0.90     10832
           1       0.35      0.37      0.36      1611

   micro avg       0.83      0.83      0.83     12443
   macro avg       0.63      0.63      0.63     12443
weighted avg       0.83      0.83      0.83     12443

Confusion Matrix:
 [[9714 1118]
 [1014  597]]
```

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

In [19]:

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

```
Number of nodes:  6703
```

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

In [20]:

```
def calculate_num_leaves(dt):
    n_nodes = dt.tree_.node_count
    ll = dt.tree_.children_left
    rl = dt.tree_.children_right
    count = 0
    for i in range(0,n_nodes):
        if (ll[i] & rl[i]) == -1:
            count = count + 1
    return count
print("The number of leaves is ",calculate_num_leaves(model));
```

```
The number of leaves is  3352
```

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

In [21]:

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

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

In [22]:

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

```
WheelTypeID_? : 0.13551426074337208
MMRCurrentAuctionAveragePrice : 0.07916633374386034
VehOdo : 0.06681157785792576
VehBCost : 0.06493159964208899
MMRCurrentRetailRatio : 0.06347311733157501
```

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

In [23]:

```
## Discuss the measurement of overfitting
print("Train accuracy:", model.score(X_train, y_train))
print("Test accuracy:", model.score(X_test, y_test))
```

```
Train accuracy: 0.9994856170616864
Test accuracy: 0.8286586835972033
```

Since the accuracy on the training set is much larger than the test set, it may has the overfitting problem. # LY, pls modify this

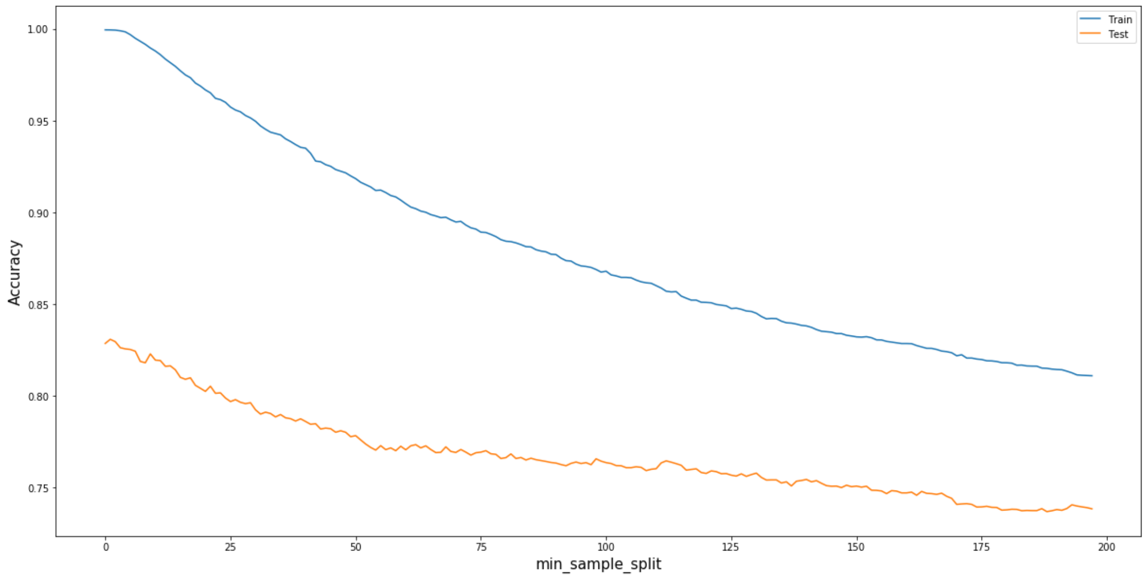
g. Did changing the default setting (i.e., only focus on changing the setting of the number of splits to create a node) help improving the model? Answer the above questions on the best performing tree.

In [24]:

```python
### One tuning on one paramete
'''
The parameter choose is the max_depth
'''

model_accuracies = defaultdict(list)

test_range = list(range(2, 200))
for min_samp in test_range:
    model = DecisionTreeClassifier(random_state=rs, min_samples_split = min_samp
)
    model.fit(X_train, y_train)
    model_accuracies['Train'].append(model.score(X_train, y_train))
    model_accuracies['Test'].append(model.score(X_test, y_test))

plt.figure(figsize=(20,10))
for key in model_accuracies.keys():
    plt.plot(model_accuracies[key], label=key)
plt.ylabel('Accuracy',fontsize=15)
plt.xlabel('min_sample_split',fontsize=15)
plt.legend(loc='upper right')
```

Out[24]:

<matplotlib.legend.Legend at 0x7f4985b78630>



2. Python: Build another decision tree tuned with GridSearchCV

In [25]:

```
# grid search CV
params = {'criterion': ['gini', 'entropy'],
          'max_depth': list(range(1, 500, 50)),
          'splitter': ['best', 'random'],
          'min_samples_leaf': range(1,  4),
          'min_samples_split': [2, 50, 100, 150],
          '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[25]:

```
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':
[1, 51, 101, 151, 201, 251, 301, 351, 401, 451], 'splitter': ['bes
t', 'random'], 'min_samples_leaf': range(1, 4), 'min_samples_split':
[2, 50, 100, 150], 'max_features': ['auto', 'sqrt', 'log2', 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 [26]:

```
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 the confusion matrix
print("Confusion Matrix: \n", confusion_matrix(y_test, y_pred)) ## Confusion Mat
rix on the TestSet

dt_model = cv.best_estimator_
```

```
Train accuracy: 0.9994856170616864
Test accuracy: 0.8236759623884915
              precision    recall  f1-score   support

           0       0.90      0.90      0.90     10832
           1       0.32      0.32      0.32      1611

   micro avg       0.82      0.82      0.82     12443
   macro avg       0.61      0.61      0.61     12443
weighted avg       0.82      0.82      0.82     12443

Confusion Matrix:
 [[9729 1103]
 [1091  520]]
```

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 [27]:

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

```
Number of nodes:  13743
```

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

In [28]:

```
print("The number of leaves is ",calculate_num_leaves(dt_model));
```

```
The number of leaves is  6872
```

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

In [29]:

```
analyse_feature_importance(cv.best_estimator_, feature_names, 1)
print("The competing splits for the first split is: ", model.tree_.threshold[0])
```

```
WheelType_? : 0.10196726739090486
The competing splits for the first split is:  0.5
```

In [30]:

```
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 [31]:

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

```
WheelType_? : 0.10196726739090486
VehBCost : 0.07747480575066952
VehOdo : 0.04975026240861232
MMRAcquisitionAuctionCleanPrice : 0.049539508385422224
MMRCurrentAuctionAveragePrice : 0.04898870588447332
```

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

In [32]:

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

```
Train accuracy: 0.9994856170616864
Test accuracy: 0.8236759623884915
```

Since the accuracy on the training set is much larger than the test set, it may has the overfitting problem. # Ly pls modify this

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

In [33]:

```
print("The best params of DT: ", cv.best_params_)
```

```
The best params of DT:  {'class_weight': 'balanced', 'criterion': 'g
ini', 'max_depth': 101, 'max_features': 'log2', 'min_samples_leaf':
1, 'min_samples_split': 2, 'splitter': 'best'}
```

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 [34]:

```
print("Defualt Model: \n")
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: \n", classification_report(y_test, y_pred))
print("Confusion Matrix: \n", confusion_matrix(y_test, y_pred)) ## Confusion Mat
rix on the TestSet

print("\n\n")

print("GridSearch Model: \n")
print("Train accuracy:", cv.score(X_train, y_train))
print("Test accuracy:", cv.score(X_test, y_test))
y_pred = cv.predict(X_test)
print("Classification report: \n",classification_report(y_test, y_pred))
print("Confusion Matrix: \n", confusion_matrix(y_test, y_pred)) ## Confusion Mat
rix on the TestSet

'''

From the classification report and the confusion matrix

'''



### And anaylse the different from the classification report and the best params
```

```
Defualt Model:

Train accuracy: 0.8110631899655759
Test accuracy: 0.7385678694848509
Classification report:
               precision    recall  f1-score   support

           0       0.92      0.77      0.84     10832
           1       0.25      0.52      0.34      1611

   micro avg       0.74      0.74      0.74     12443
   macro avg       0.58      0.65      0.59     12443
weighted avg       0.83      0.74      0.77     12443

Confusion Matrix:
 [[8351 2481]
 [ 772  839]]



GridSearch Model:

Train accuracy: 0.9994856170616864
Test accuracy: 0.8236759623884915
Classification report:
               precision    recall  f1-score   support

           0       0.90      0.90      0.90     10832
           1       0.32      0.32      0.32      1611

   micro avg       0.82      0.82      0.82     12443
   macro avg       0.61      0.61      0.61     12443
weighted avg       0.82      0.82      0.82     12443

Confusion Matrix:
 [[9729 1103]
 [1091  520]]
```

Out[34]:

```
'\n\nFrom the classification report and the confusion matrix\n\n'
```

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

In [35]:

```
'''
print out all the classified kicks, from y_test to take the x_test out
-> check the length and add the name of features to the value(feature_names).
'''
```

Out[35]:

```
'\nprint out all the classified kicks, from y_test to take the x_tes
t out \n-> check the length and add the name of features to the valu
e(feature_names).\n'
```

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 [36]:

```
'''
We apply imputation on all of the columns except the dropped columns
'''

print("The Columns apply Imputation: \n", list(feature_names_beforDummy))
```

```
The Columns apply Imputation: 
 ['Auction', 'VehYear', 'Make', 'Color', 'Transmission', 'WheelTypeI
D', 'WheelType', 'VehOdo', 'Nationality', 'Size', 'TopThreeAmericanN
ame', 'MMRAcquisitionAuctionAveragePrice', 'MMRAcquisitionAuctionCle
anPrice', 'MMRAcquisitionRetailAveragePrice', 'MMRAcquisitonRetailCl
eanPrice', 'MMRCurrentAuctionAveragePrice', 'MMRCurrentAuctionCleanP
rice', 'MMRCurrentRetailAveragePrice', 'MMRCurrentRetailCleanPrice',
'MMRCurrentRetailRatio', 'PRIMEUNIT', 'AUCGUART', 'VNST', 'VehBCos
t', 'IsOnlineSale', 'WarrantyCost', 'ForSale']
```

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

In [37]:

```
## 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)


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



# 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)
```

```
Using ROS Resmapling
```

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 [38]:

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

Out[38]:

```
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 [39]:

```
## 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[39]:

```
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 [40]:

```
'''
The regression function use the sigmoid function as the activation function at o
utput layer.
'''
```

Out[40]:

```
'\nThe regression function use the sigmoid function as the activatio
n function at output layer.\n'
```

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

In [41]:

```
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: 0.6998773394531713
Test accuracy: 0.7560877601864502
GridSearch Train accuracy: 0.7009456732481304
GridSearch Test accuracy: 0.7552840954753677
```

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

In [42]:

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

```
The best model parameters:  {'C': 1, 'class_weight': 'balanced', 'ma
x_iter': 30, 'solver': 'lbfgs', 'warm_start': True}
```

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

In [43]:

```
# Print all features
print("Features used: \n")

for name in feature_names:
    print( name +", ")
```

```
Features used:

VehOdo,
MMRAcquisitionAuctionAveragePrice,
MMRAcquisitionAuctionCleanPrice,
MMRAcquisitionRetailAveragePrice,
MMRAcquisitonRetailCleanPrice,
MMRCurrentAuctionAveragePrice,
MMRCurrentAuctionCleanPrice,
MMRCurrentRetailAveragePrice,
MMRCurrentRetailCleanPrice,
MMRCurrentRetailRatio,
VehBCost,
WarrantyCost,
Auction_adesa,
Auction_manheim,
Auction_other,
VehYear_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_2010.0,
VehYear_UNKNOWN_VALUE,
Make_acura,
Make_buick,
Make_cadillac,
Make_chevrolet,
Make_chrysler,
Make_dodge,
Make_ford,
Make_gmc,
Make_honda,
Make_hyundai,
Make_infiniti,
Make_isuzu,
Make_jeep,
Make_kia,
Make_lexus,
Make_lincoln,
Make_mazda,
Make_mercury,
Make_mini,
Make_mitsubishi,
Make_nissan,
Make_oldsmobile,
Make_pontiac,
Make_saturn,
Make_scion,
Make_subaru,
Make_suzuki,
Make_toyota,
Make_volkswagen,
Make_volvo,
Color_beige,
Color_black,
Color_blue,
```

```
    Color_brown,
    Color_gold,
    Color_green,
    Color_grey,
    Color_maroon,
    Color_not avail,
    Color_orange,
    Color_other,
    Color_purple,
    Color_red,
    Color_silver,
    Color_white,
    Color_yellow,
    Transmission_auto,
    Transmission_manual,
    WheelTypeID_0,
    WheelTypeID_1,
    WheelTypeID_2,
    WheelTypeID_3,
    WheelTypeID_?,
    WheelType_?,
    WheelType_alloy,
    WheelType_covers,
    WheelType_special,
    Nationality_american,
    Nationality_other,
    Nationality_other asian,
    Nationality_top line asian,
    Size_compact,
    Size_crossover,
    Size_large,
    Size_large suv,
    Size_large truck,
    Size_medium,
    Size_medium suv,
    Size_small suv,
    Size_small truck,
    Size_specialty,
    Size_sports,
    Size_van,
    TopThreeAmericanName_chrysler,
    TopThreeAmericanName_ford,
    TopThreeAmericanName_gm,
    TopThreeAmericanName_other,
    PRIMEUNIT_?,
    PRIMEUNIT_no,
    PRIMEUNIT_yes,
    PRIMEUNIT_NULL,
    AUCGUART_?,
    AUCGUART_green,
    AUCGUART_red,
    AUCGUART_NULL,
    VNST_al,
    VNST_az,
    VNST_ca,
    VNST_co,
    VNST_fl,
    VNST_ga,
    VNST_id,
    VNST_il,
    VNST_in,
```

```
VNST_ky,
VNST_la,
VNST_mo,
VNST_ms,
VNST_nc,
VNST_ne,
VNST_nh,
VNST_nj,
VNST_nm,
VNST_nv,
VNST_ny,
VNST_oh,
VNST_ok,
VNST_or,
VNST_pa,
VNST_sc,
VNST_tn,
VNST_tx,
VNST_ut,
VNST_va,
VNST_wa,
VNST_wv,
VNST_NULL,
IsOnlineSale_-1.0,
IsOnlineSale_0.0,
IsOnlineSale_1.0,
ForSale_0,
ForSale_no,
ForSale_yes,
```

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

In [44]:

```python
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 [45]:

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

```
MMRAcquisitionAuctionAveragePrice : -1.8301352716819697
MMRAcquisitionRetailAveragePrice : 1.556335135697774
MMRCurrentRetailCleanPrice : -1.1608985500248494
WheelTypeID_? : 0.7647388496623555
MMRCurrentAuctionAveragePrice : 0.7090035140103588
```

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

In [46]:

```
y_pred = model.predict(X_test_log)
print("Classification Report: \n\n",classification_report(y_test_log, y_pred))
print("Default Model Confusion Matrix: \n", confusion_matrix(y_test, y_pred))

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

```
Classification Report: 

               precision    recall  f1-score   support

           0       0.93      0.78      0.85     10832
           1       0.29      0.61      0.39      1611

   micro avg       0.76      0.76      0.76     12443
   macro avg       0.61      0.69      0.62     12443
weighted avg       0.85      0.76      0.79     12443

Default Model Confusion Matrix: 
 [[8430 2402]
 [ 633  978]]
GridSearch Classification Report: 

               precision    recall  f1-score   support

           0       0.93      0.78      0.85     10832
           1       0.29      0.61      0.39      1611

   micro avg       0.76      0.76      0.76     12443
   macro avg       0.61      0.69      0.62     12443
weighted avg       0.85      0.76      0.79     12443

GridSearch Confusion Matrix:
  [[8422 2410]
 [ 635  976]]
```

n. Report any sign of overfitting.

In [47]:

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

```
GridSearch Train accuracy: 0.7009456732481304
GridSearch Test accuracy: 0.7552840954753677
```

In [48]:

```
'''
According to the training and test accuracy, the overfitting doesn't occur.
'''
```

Out[48]:

```
"\nAccording to the training and test accuracy, the overfitting does
n't occur.\n"
```

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

In [49]:

```
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 [50]:

```
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:  126
Number of selectFromModel features:  24
```

In [51]:

```
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', 'MMRCurrentRetailAveragePrice', 'MMRCurrentRetailCleanP
rice', 'MMRCurrentRetailRatio', 'VehBCost', 'WarrantyCost', 'Auction
_adesa', 'Auction_manheim', 'Auction_other', 'VehYear_2001.0', 'VehY
ear_2002.0', 'VehYear_2003.0', 'VehYear_2004.0', 'VehYear_2005.0',
'VehYear_2006.0', 'VehYear_2007.0', 'VehYear_2008.0', 'VehYear_2009.
0', 'VehYear_2010.0', 'VehYear_UNKNOWN_VALUE', 'Make_acura', 'Make_b
uick', 'Make_chevrolet', 'Make_chrysler', 'Make_dodge', 'Make_ford',
'Make_honda', 'Make_infiniti', 'Make_isuzu', 'Make_jeep', 'Make_ki
a', 'Make_lexus', 'Make_lincoln', 'Make_mini', 'Make_mitsubishi', 'M
ake_nissan', 'Make_oldsmobile', 'Make_pontiac', 'Make_saturn', 'Make
_scion', 'Make_subaru', 'Make_suzuki', 'Make_toyota', 'Make_volvo',
'Color_beige', 'Color_black', 'Color_brown', 'Color_gold', 'Color_gr
een', 'Color_grey', 'Color_not avail', 'Color_orange', 'Color_othe
r', 'Color_purple', 'Color_red', 'Color_silver', 'Color_white', 'Col
or_yellow', 'Transmission_auto', 'Transmission_manual', 'WheelTypeID
_0', 'WheelTypeID_1', 'WheelTypeID_2', 'WheelTypeID_3', 'WheelTypeID
_?', 'WheelType_?', 'WheelType_alloy', 'WheelType_covers', 'WheelTyp
e_special', 'Nationality_american', 'Nationality_other', 'Nationalit
y_other asian', 'Nationality_top line asian', 'Size_compact', 'Size_
crossover', 'Size_large', 'Size_large suv', 'Size_large truck', 'Siz
e_medium', 'Size_medium suv', 'Size_small suv', 'Size_specialty', 'S
ize_sports', 'Size_van', 'TopThreeAmericanName_chrysler', 'TopThreeA
mericanName_gm', 'TopThreeAmericanName_other', 'PRIMEUNIT_?', 'PRIME
UNIT_no', 'PRIMEUNIT_yes', 'PRIMEUNIT_NULL', 'AUCGUART_?', 'AUCGUART
_green', 'AUCGUART_NULL', 'VNST_al', 'VNST_az', 'VNST_co', 'VNST_f
l', 'VNST_ga', 'VNST_id', 'VNST_in', 'VNST_ky', 'VNST_la', 'VNST_n
c', 'VNST_ne', 'VNST_nh', 'VNST_nj', 'VNST_nm', 'VNST_ny', 'VNST_o
r', 'VNST_pa', 'VNST_sc', 'VNST_tn', 'VNST_tx', 'VNST_ut', 'VNST_NUL
L', 'IsOnlineSale_1.0', 'ForSale_0', 'ForSale_no', 'ForSale_yes']



The SelectFromModel features:

 ['VehOdo', 'MMRAcquisitionAuctionAveragePrice', 'MMRAcquisitionAuct
ionCleanPrice', 'MMRAcquisitionRetailAveragePrice', 'MMRAcquisitonRe
tailCleanPrice', 'MMRCurrentAuctionAveragePrice', 'MMRCurrentAuction
CleanPrice', 'MMRCurrentRetailAveragePrice', 'MMRCurrentRetailCleanP
rice', 'MMRCurrentRetailRatio', 'VehBCost', 'WarrantyCost', 'Auction
_manheim', 'VehYear_2004.0', 'Make_chevrolet', 'Make_dodge', 'Color_
silver', 'Color_white', 'WheelTypeID_2', 'WheelType_?', 'WheelType_c
overs', 'TopThreeAmericanName_chrysler', 'TopThreeAmericanName_gm',
'VNST_tx']
```

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

In [52]:

```
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)
```

```
[LibLinear]

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurr
ent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.6s finishe
d
```

Out[52]:

```
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 [53]:

```
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:

MMRAcquisitionAuctionAveragePrice : -1.2007986138089202
MMRAcquisitionRetailAveragePrice : 1.1707944988856998
MMRCurrentRetailCleanPrice : -0.5862338769571586
Color_white : 0.5771408731924557
MMRAcquisitonRetailCleanPrice : 0.5560971039889662



Top-5 important variables for selectModel

MMRCurrentRetailAveragePrice : -3.155872487409825
MMRCurrentRetailCleanPrice : 2.2997683935748934
MMRAcquisitionAuctionAveragePrice : -1.8616373108354378
VehYear_2005.0 : 1.2396144583206734
MMRAcquisitonRetailCleanPrice : 0.9311113016898371
```

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

In [54]:

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

```
Optimal Parameters for RFE {'C': 0.1, 'class_weight': 'balanced', 'm
ax_iter': 30, 'solver': 'liblinear', 'warm_start': True}
Optimal Parameters for selectModel {'C': 1, 'class_weight': 'balance
d', 'max_iter': 30, 'solver': 'newton-cg', 'warm_start': True}
```

d. Report any sign of overfitting

In [55]:

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

```
GridSearch Train accuracy: 0.7009456732481304
GridSearch Test accuracy: 0.7552840954753677
```

In [56]:

```
'''
No Overfitting occurs in this model ## Ly modify this
'''
```

Out[56]:

```
'\nNo Overfitting occurs in this model ## Ly modify this\n'
```

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

In [57]:

```
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: 0.7009456732481304
GridSearch Test accuracy: 0.7552840954753677


RFE:

Train accuracy: 0.7000949630039963
Test accuracy: 0.7568914248975327


selectModel:

Train accuracy: 0.6835951410596288
Test accuracy: 0.7648477055372499
```

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

In [58]:

```
y_pred = rfe_cv.predict(X_test_rfe)
print("REF classification report: \n",classification_report(y_test, y_pred))
print("REF Confusion Matrix: \n", confusion_matrix(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))
print("selectModel Confusion Matrix: \n", confusion_matrix(y_test, y_pred))
```

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

           0       0.93      0.78      0.85     10832
           1       0.29      0.60      0.39      1611

   micro avg       0.76      0.76      0.76     12443
   macro avg       0.61      0.69      0.62     12443
weighted avg       0.85      0.76      0.79     12443

REF Confusion Matrix:
 [[8444 2388]
 [ 637  974]]



selectModel classification report:
               precision    recall  f1-score   support

           0       0.92      0.79      0.85     10832
           1       0.29      0.57      0.38      1611

   micro avg       0.76      0.76      0.76     12443
   macro avg       0.61      0.68      0.62     12443
weighted avg       0.84      0.76      0.79     12443

selectModel Confusion Matrix:
 [[8606 2226]
 [ 700  911]]
```

In [59]:

```
'''
The performance...

'''
```

Out[59]:

```
'\nThe performance...\n\n'
```

Task4 - Predicting using neural network

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

In [60]:

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

Out[60]:

```
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 [61]:

```
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 [62]:

```
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 [63]:

```
print("MLP Train accuracy:", model.score(X_train, y_train))
print("MLP Test accuracy:", model.score(X_test, y_test))
# No overfitting sign in this model ## Ly modify this
```

```
MLP Train accuracy: 0.459660507260713
MLP Test accuracy: 0.6925982480109298
```

In [64]:

```
'''
The training accuracy and the test accuracy ...
'''
```

Out[64]:

```
'\nThe training accuracy and the test accuracy ...\n'
```

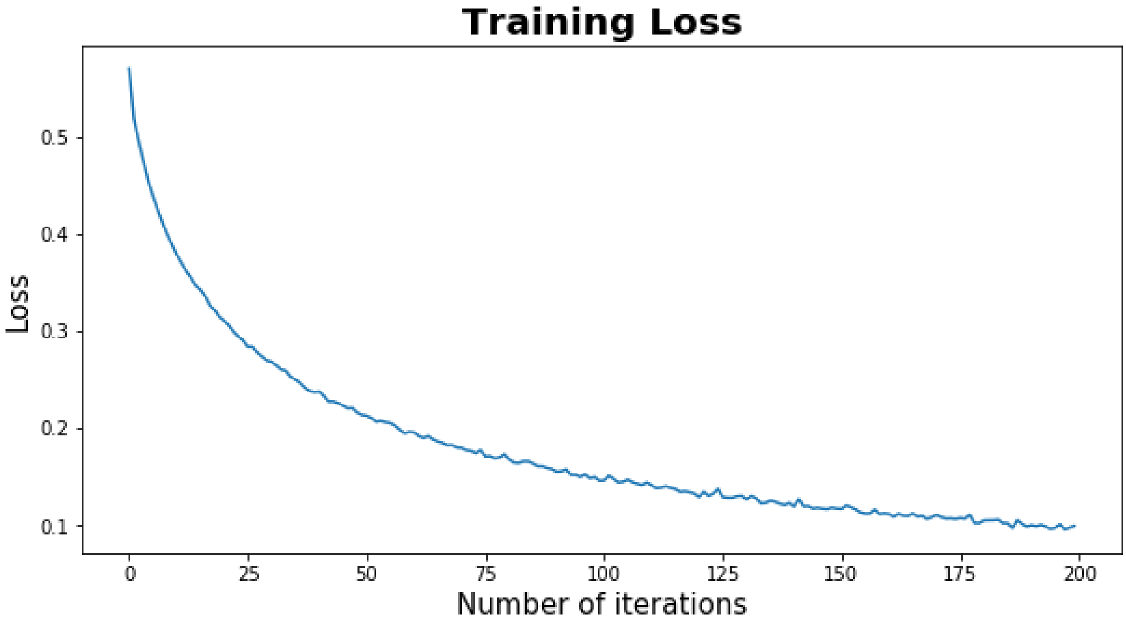
d. Did the training process converge and resulted in the best model?

In [65]:

```
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[65]:

```
[<matplotlib.lines.Line2D at 0x7f493ca15ba8>]
```



In [66]:

```
'''
The loss curve is still decreasing. Therefore, it may not converge to the local
 minima yet.
'''
```

Out[66]:

```
'\nThe loss curve is still decreasing. Therefore, it may not converg
e to the local minima yet.\n'
```

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

In [67]:

```
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))
print("MLP Confusion Matrix: \n", confusion_matrix(y_test, y_pred))
```

```
MLP Train accuracy: 0.459660507260713
MLP Test accuracy: 0.6925982480109298



MLP classification report:
               precision    recall  f1-score   support

           0       0.86      0.77      0.81     10832
           1       0.09      0.14      0.11      1611

   micro avg       0.69      0.69      0.69     12443
   macro avg       0.47      0.46      0.46     12443
weighted avg       0.76      0.69      0.72     12443

MLP Confusion Matrix:
 [[8388 2444]
 [1381  230]]
```

2. Refine this network by tuning it with GridSearchCV.

In [68]:

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

params = [
    {
        'hidden_layer_sizes': [(128, 64, 32, 16), (128, 64,)],
        'activation': ['relu'],
        'solver' : ['adam',],
        'batch_size': [64],
        'shuffle': [True],
        'learning_rate_init': [0.001],
        'n_iter_no_change': [10],
        'max_iter':[200],
        'warm_start': [True],
        'early_stopping': [True],
        'alpha': [0.01, 0.001],
    },
]


cv = GridSearchCV(param_grid=params, estimator=MLPClassifier(random_state=rs, ve
rbose=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 = 0.54707512
Validation score: 0.736301
Iteration 2, loss = 0.47857332
Validation score: 0.777448
Iteration 3, loss = 0.42324346
Validation score: 0.797428
Iteration 4, loss = 0.37362828
Validation score: 0.814441
Iteration 5, loss = 0.32970120
Validation score: 0.841345
Iteration 6, loss = 0.29536147
Validation score: 0.845895
Iteration 7, loss = 0.26196995
Validation score: 0.866271
Iteration 8, loss = 0.23640891
Validation score: 0.867854
Iteration 9, loss = 0.21630361
Validation score: 0.891592
Iteration 10, loss = 0.19812908
Validation score: 0.897527
Iteration 11, loss = 0.17922477
Validation score: 0.892186
Iteration 12, loss = 0.17157421
Validation score: 0.893966
Iteration 13, loss = 0.15687880
Validation score: 0.898912
Iteration 14, loss = 0.15013150
Validation score: 0.908012
Iteration 15, loss = 0.14213609
Validation score: 0.913551
Iteration 16, loss = 0.13710825
Validation score: 0.908803
Iteration 17, loss = 0.12941752
Validation score: 0.916123
Iteration 18, loss = 0.12335300
Validation score: 0.908605
Iteration 19, loss = 0.12127017
Validation score: 0.920475
Iteration 20, loss = 0.11510558
Validation score: 0.918299
Iteration 21, loss = 0.10792456
Validation score: 0.916716
Iteration 22, loss = 0.11128821
Validation score: 0.923838
Iteration 23, loss = 0.10161774
Validation score: 0.915727
Iteration 24, loss = 0.10311017
Validation score: 0.923244
Iteration 25, loss = 0.09677756
Validation score: 0.916123
Iteration 26, loss = 0.09564818
Validation score: 0.919881
Iteration 27, loss = 0.09391351
Validation score: 0.919881
Iteration 28, loss = 0.09325189
Validation score: 0.920673
Iteration 29, loss = 0.08933597
Validation score: 0.919090
Iteration 30, loss = 0.08553687
Validation score: 0.927003
Iteration 31, loss = 0.08509835
```

```
Validation score: 0.920870
Iteration 32, loss = 0.08890293
Validation score: 0.929575
Iteration 33, loss = 0.08273223
Validation score: 0.927399
Iteration 34, loss = 0.08393377
Validation score: 0.919683
Iteration 35, loss = 0.08182656
Validation score: 0.934520
Iteration 36, loss = 0.07923991
Validation score: 0.929377
Iteration 37, loss = 0.07911647
Validation score: 0.924036
Iteration 38, loss = 0.07507023
Validation score: 0.918892
Iteration 39, loss = 0.07546001
Validation score: 0.932938
Iteration 40, loss = 0.07573450
Validation score: 0.925618
Iteration 41, loss = 0.07798078
Validation score: 0.935707
Iteration 42, loss = 0.07570306
Validation score: 0.931553
Iteration 43, loss = 0.07707894
Validation score: 0.923046
Iteration 44, loss = 0.07104559
Validation score: 0.932938
Iteration 45, loss = 0.07088950
Validation score: 0.929575
Iteration 46, loss = 0.07306730
Validation score: 0.930959
Iteration 47, loss = 0.06642030
Validation score: 0.939268
Iteration 48, loss = 0.07605865
Validation score: 0.931157
Iteration 49, loss = 0.07145894
Validation score: 0.933531
Iteration 50, loss = 0.07031683
Validation score: 0.932146
Iteration 51, loss = 0.06679548
Validation score: 0.929377
Iteration 52, loss = 0.06558132
Validation score: 0.928388
Iteration 53, loss = 0.06718902
Validation score: 0.936103
Iteration 54, loss = 0.06389646
Validation score: 0.933927
Iteration 55, loss = 0.06966706
Validation score: 0.924629
Iteration 56, loss = 0.06919731
Validation score: 0.924431
Iteration 57, loss = 0.06199414
Validation score: 0.939268
Iteration 58, loss = 0.06546817
Validation score: 0.931751
Validation score did not improve more than tol=0.000100 for 10 conse
cutive epochs. Stopping.
```

Out[68]:

```
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=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=True, warm_start=False),
       fit_params=None, iid='warn', n_jobs=-1,
       param_grid=[{'hidden_layer_sizes': [(128, 64, 32, 16), (128,
64)], 'activation': ['relu'], 'solver': ['adam'], 'batch_size': [6
4], 'shuffle': [True], 'learning_rate_init': [0.001], 'n_iter_no_cha
nge': [10], 'max_iter': [200], 'warm_start': [True], 'early_stoppin
g': [True], 'alpha': [0.01, 0.001]}],
       pre_dispatch='2*n_jobs', refit=True, return_train_score='war
n',
       scoring=None, verbose=0)
```

a. What is the network architecture?

In [69]:

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

```
Best Parameters of NN:  {'activation': 'relu', 'alpha': 0.001, 'batc
h_size': 64, 'early_stopping': True, 'hidden_layer_sizes': (128, 6
4), 'learning_rate_init': 0.001, 'max_iter': 200, 'n_iter_no_chang
e': 10, 'shuffle': True, 'solver': 'adam', 'warm_start': True}
```

In [70]:

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

```
Number of Layers:  4
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 64
4 Layer with hidden size 1
The activation function:  relu
```

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

In [71]:

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

```
Number of iterations it ran:  58
```

c. Sign of overfitting?

In [72]:

```
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))
# Since training accuracy is much larger than the test accuracy, it has the sign
of overfitting.
```

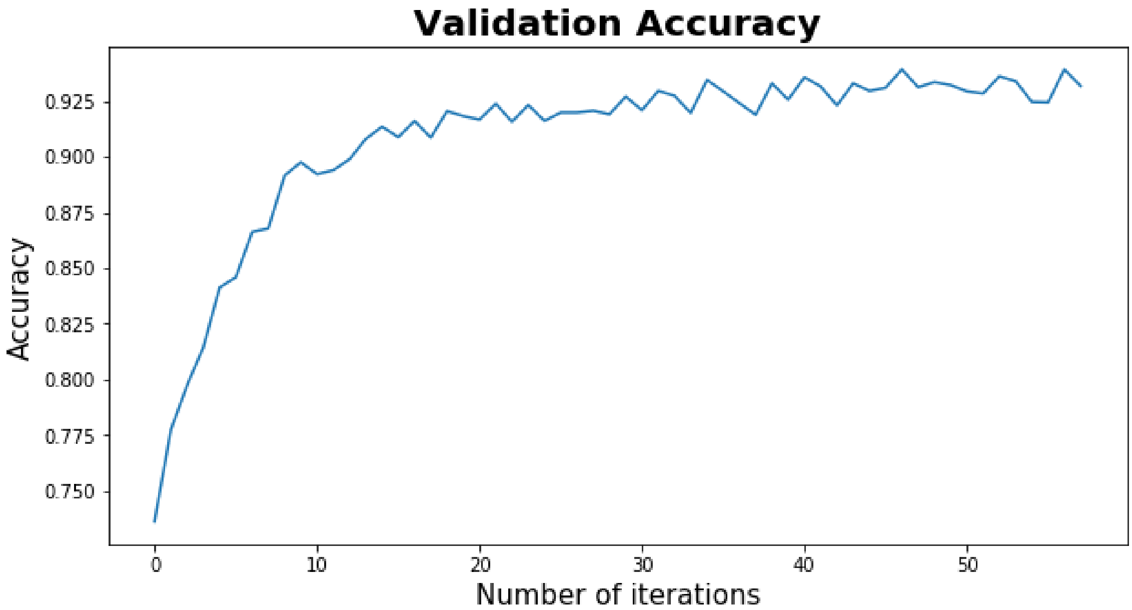
```
GridSearch NN Train accuracy: 0.9777430459383532
GridSearch NN Test accuracy: 0.8370167965924616
```

In [73]:

```
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[73]:

```
[<matplotlib.lines.Line2D at 0x7f494c473780>]
```



In [74]:

```
'''
The training accuracy and the test accuracy...

Also, according to the validation accuracy curve

'''
```

Out[74]:

```
'\nThe training accuracy and the test accuracy...\n\nAlso, according
to the validation accuracy curve\n\n'
```

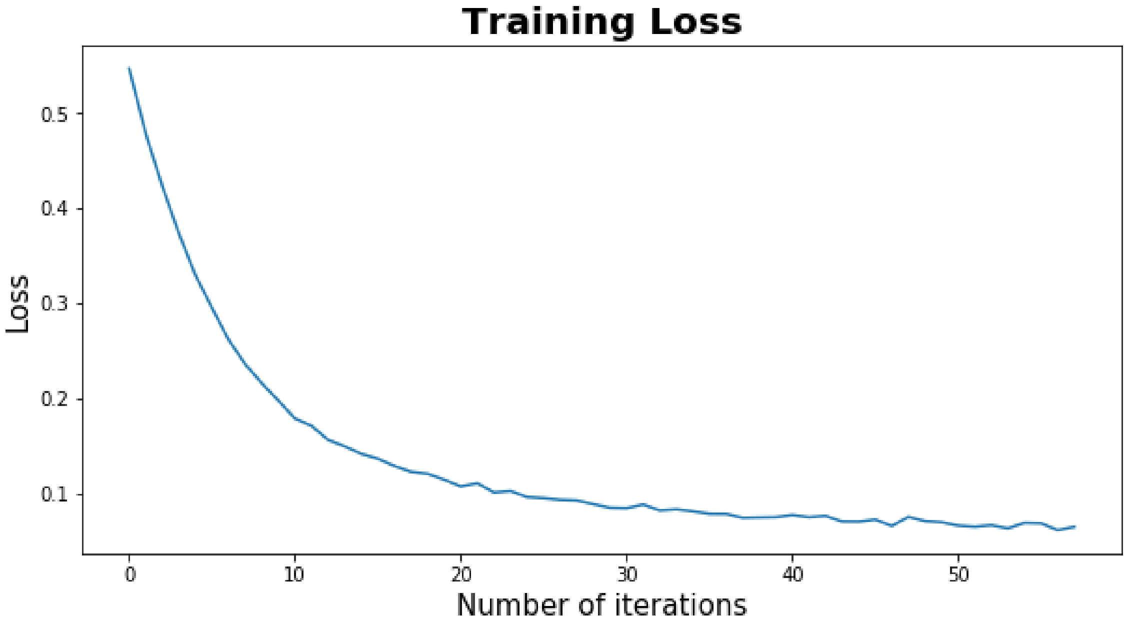
d. Did the training process converge and resulted in the best model?

In [75]:

```
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[75]:

```
[<matplotlib.lines.Line2D at 0x7f4940c82390>]
```



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

In [76]:

```
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("GridSearch NN Confusion Matrix: \n", confusion_matrix(y_test, y_pred))


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

```
GridSearch NN Train accuracy: 0.9777430459383532
GridSearch NN Test accuracy: 0.8370167965924616



GridSearch NN Classification Report: 
               precision    recall  f1-score   support

           0       0.90      0.91      0.91     10832
           1       0.36      0.35      0.36      1611

   micro avg       0.84      0.84      0.84     12443
   macro avg       0.63      0.63      0.63     12443
weighted avg       0.83      0.84      0.84     12443

GridSearch NN Confusion Matrix: 
 [[9855  977]
 [1051  560]]
Best Parameters of NN:  {'activation': 'relu', 'alpha': 0.001, 'batc
h_size': 64, 'early_stopping': True, 'hidden_layer_sizes': (128, 6
4), 'learning_rate_init': 0.001, 'max_iter': 200, 'n_iter_no_chang
e': 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 [77]:

```
params = [
    {
        'hidden_layer_sizes': [(128, 64, 32, 16)],
        'activation': ['relu'],
        'solver' : ['adam',],
        'batch_size': [64],
        'shuffle': [True],
        'learning_rate_init': [0.001],
        'n_iter_no_change': [10],
        'max_iter':[200],
        'warm_start': [True],
        'early_stopping': [True],
        'alpha': [0.01, 0.001],
    },
]



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 = 0.55006225
Validation score: 0.735905
Iteration 2, loss = 0.47984706
Validation score: 0.774679
Iteration 3, loss = 0.42210915
Validation score: 0.797230
Iteration 4, loss = 0.36983292
Validation score: 0.820969
Iteration 5, loss = 0.31999334
Validation score: 0.844115
Iteration 6, loss = 0.28662503
Validation score: 0.860534
Iteration 7, loss = 0.25221418
Validation score: 0.863106
Iteration 8, loss = 0.22846455
Validation score: 0.882295
Iteration 9, loss = 0.20889836
Validation score: 0.883877
Iteration 10, loss = 0.19368282
Validation score: 0.897725
Iteration 11, loss = 0.18013278
Validation score: 0.901484
Iteration 12, loss = 0.16860607
Validation score: 0.909199
Iteration 13, loss = 0.15794195
Validation score: 0.914936
Iteration 14, loss = 0.15202148
Validation score: 0.912562
Iteration 15, loss = 0.14529811
Validation score: 0.917112
Iteration 16, loss = 0.13881151
Validation score: 0.920475
Iteration 17, loss = 0.13177450
Validation score: 0.921266
Iteration 18, loss = 0.13015646
Validation score: 0.914936
Iteration 19, loss = 0.12068699
Validation score: 0.921266
Iteration 20, loss = 0.12192105
Validation score: 0.911573
Iteration 21, loss = 0.12003132
Validation score: 0.912760
Iteration 22, loss = 0.10943126
Validation score: 0.919090
Iteration 23, loss = 0.11078525
Validation score: 0.916518
Iteration 24, loss = 0.10566946
Validation score: 0.927992
Iteration 25, loss = 0.11202087
Validation score: 0.923046
Iteration 26, loss = 0.10638109
Validation score: 0.929377
Iteration 27, loss = 0.09962065
Validation score: 0.924629
Iteration 28, loss = 0.09700691
Validation score: 0.925816
Iteration 29, loss = 0.09746313
Validation score: 0.925025
Iteration 30, loss = 0.09158414
Validation score: 0.924431
Iteration 31, loss = 0.09335132
```

```
Validation score: 0.927399
Iteration 32, loss = 0.09351754
Validation score: 0.931355
Iteration 33, loss = 0.08600295
Validation score: 0.923244
Iteration 34, loss = 0.09347434
Validation score: 0.929377
Iteration 35, loss = 0.08853674
Validation score: 0.922651
Iteration 36, loss = 0.08196959
Validation score: 0.926409
Iteration 37, loss = 0.08012877
Validation score: 0.928783
Iteration 38, loss = 0.09383859
Validation score: 0.933531
Iteration 39, loss = 0.08376033
Validation score: 0.933729
Iteration 40, loss = 0.07836819
Validation score: 0.932344
Iteration 41, loss = 0.08074103
Validation score: 0.926607
Iteration 42, loss = 0.07440560
Validation score: 0.939862
Iteration 43, loss = 0.08109114
Validation score: 0.928388
Iteration 44, loss = 0.07462885
Validation score: 0.930168
Iteration 45, loss = 0.07518688
Validation score: 0.927399
Iteration 46, loss = 0.07608733
Validation score: 0.939664
Iteration 47, loss = 0.07364330
Validation score: 0.931355
Iteration 48, loss = 0.07672526
Validation score: 0.923046
Iteration 49, loss = 0.07709252
Validation score: 0.933927
Iteration 50, loss = 0.06671595
Validation score: 0.923640
Iteration 51, loss = 0.07414392
Validation score: 0.937883
Iteration 52, loss = 0.07204053
Validation score: 0.938081
Iteration 53, loss = 0.06907075
Validation score: 0.933333
Validation score did not improve more than tol=0.000100 for 10 conse
cutive epochs. Stopping.
Iteration 1, loss = 0.58931856
Validation score: 0.693571
Iteration 2, loss = 0.56887711
Validation score: 0.699505
Iteration 3, loss = 0.55819986
Validation score: 0.711573
Iteration 4, loss = 0.54857577
Validation score: 0.712562
Iteration 5, loss = 0.53951080
Validation score: 0.706825
Iteration 6, loss = 0.52938380
Validation score: 0.729970
Iteration 7, loss = 0.51787453
Validation score: 0.736103
```

```
Iteration 8, loss = 0.50666618
Validation score: 0.734916
Iteration 9, loss = 0.49512273
Validation score: 0.752918
Iteration 10, loss = 0.48210883
Validation score: 0.750148
Iteration 11, loss = 0.47066528
Validation score: 0.762611
Iteration 12, loss = 0.45801744
Validation score: 0.768150
Iteration 13, loss = 0.44639482
Validation score: 0.781207
Iteration 14, loss = 0.43490931
Validation score: 0.791889
Iteration 15, loss = 0.42346839
Validation score: 0.792878
Iteration 16, loss = 0.41141134
Validation score: 0.790900
Iteration 17, loss = 0.39994190
Validation score: 0.793670
Iteration 18, loss = 0.39189962
Validation score: 0.801780
Iteration 19, loss = 0.38206891
Validation score: 0.813848
Iteration 20, loss = 0.36943156
Validation score: 0.819782
Iteration 21, loss = 0.36455669
Validation score: 0.830069
Iteration 22, loss = 0.35445151
Validation score: 0.825124
Iteration 23, loss = 0.34878006
Validation score: 0.826904
Iteration 24, loss = 0.34101956
Validation score: 0.819585
Iteration 25, loss = 0.33374843
Validation score: 0.847873
Iteration 26, loss = 0.32613844
Validation score: 0.840158
Iteration 27, loss = 0.32176930
Validation score: 0.835806
Iteration 28, loss = 0.31857059
Validation score: 0.846489
Iteration 29, loss = 0.31287286
Validation score: 0.838773
Iteration 30, loss = 0.30436497
Validation score: 0.849654
Iteration 31, loss = 0.30198760
Validation score: 0.850841
Iteration 32, loss = 0.30087002
Validation score: 0.860732
Iteration 33, loss = 0.29313105
Validation score: 0.858754
Iteration 34, loss = 0.29074290
Validation score: 0.863501
Iteration 35, loss = 0.28659173
Validation score: 0.865084
Iteration 36, loss = 0.28468743
Validation score: 0.863897
Iteration 37, loss = 0.27922393
Validation score: 0.853808
Iteration 38, loss = 0.27723243
```

```
Validation score: 0.874777
Iteration 39, loss = 0.27425093
Validation score: 0.866469
Iteration 40, loss = 0.26962243
Validation score: 0.858556
Iteration 41, loss = 0.27381055
Validation score: 0.866667
Iteration 42, loss = 0.26329959
Validation score: 0.864293
Iteration 43, loss = 0.26343270
Validation score: 0.876360
Iteration 44, loss = 0.26306010
Validation score: 0.880514
Iteration 45, loss = 0.25601979
Validation score: 0.876558
Iteration 46, loss = 0.25369687
Validation score: 0.872404
Iteration 47, loss = 0.25203873
Validation score: 0.878932
Iteration 48, loss = 0.25291981
Validation score: 0.879525
Iteration 49, loss = 0.24999345
Validation score: 0.880910
Iteration 50, loss = 0.24931593
Validation score: 0.879921
Iteration 51, loss = 0.24346036
Validation score: 0.874580
Iteration 52, loss = 0.24572028
Validation score: 0.883482
Iteration 53, loss = 0.24360502
Validation score: 0.875173
Iteration 54, loss = 0.24368640
Validation score: 0.882493
Iteration 55, loss = 0.23456952
Validation score: 0.886647
Iteration 56, loss = 0.23760133
Validation score: 0.887834
Iteration 57, loss = 0.23968095
Validation score: 0.886845
Iteration 58, loss = 0.23138386
Validation score: 0.894164
Iteration 59, loss = 0.23478211
Validation score: 0.880910
Iteration 60, loss = 0.23339679
Validation score: 0.885856
Iteration 61, loss = 0.23410777
Validation score: 0.885064
Iteration 62, loss = 0.22575438
Validation score: 0.883877
Iteration 63, loss = 0.23073593
Validation score: 0.886053
Iteration 64, loss = 0.22946885
Validation score: 0.890406
Iteration 65, loss = 0.22570186
Validation score: 0.898318
Iteration 66, loss = 0.22435437
Validation score: 0.888625
Iteration 67, loss = 0.22642374
Validation score: 0.896340
Iteration 68, loss = 0.21981893
Validation score: 0.884471
```

```
Iteration 69, loss = 0.22127705
Validation score: 0.888823
Iteration 70, loss = 0.21953378
Validation score: 0.886647
Iteration 71, loss = 0.22184895
Validation score: 0.893571
Iteration 72, loss = 0.22176538
Validation score: 0.891197
Iteration 73, loss = 0.22033268
Validation score: 0.892977
Iteration 74, loss = 0.21330043
Validation score: 0.894955
Iteration 75, loss = 0.22060284
Validation score: 0.890801
Iteration 76, loss = 0.21763188
Validation score: 0.901484
Iteration 77, loss = 0.21340858
Validation score: 0.896142
Iteration 78, loss = 0.21158515
Validation score: 0.892582
Iteration 79, loss = 0.21324669
Validation score: 0.896934
Iteration 80, loss = 0.20937251
Validation score: 0.900099
Iteration 81, loss = 0.21220144
Validation score: 0.885856
Iteration 82, loss = 0.21527536
Validation score: 0.885460
Iteration 83, loss = 0.20524368
Validation score: 0.900297
Iteration 84, loss = 0.20888267
Validation score: 0.890999
Iteration 85, loss = 0.21373057
Validation score: 0.896934
Iteration 86, loss = 0.20771093
Validation score: 0.895351
Iteration 87, loss = 0.21593178
Validation score: 0.906034
Iteration 88, loss = 0.20590610
Validation score: 0.887834
Iteration 89, loss = 0.20553455
Validation score: 0.899901
Iteration 90, loss = 0.20402395
Validation score: 0.895747
Iteration 91, loss = 0.20452099
Validation score: 0.893175
Iteration 92, loss = 0.20004052
Validation score: 0.905045
Iteration 93, loss = 0.20396928
Validation score: 0.897923
Iteration 94, loss = 0.20366547
Validation score: 0.902671
Iteration 95, loss = 0.20040389
Validation score: 0.900099
Iteration 96, loss = 0.20262043
Validation score: 0.903462
Iteration 97, loss = 0.20333923
Validation score: 0.903264
Iteration 98, loss = 0.20024018
Validation score: 0.904253
```

```
Validation score did not improve more than tol=0.000100 for 10 conse
cutive epochs. Stopping.
```

Out[77]:

```
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, 64, 32, 16)], 'acti
vation': ['relu'], 'solver': ['adam'], 'batch_size': [64], 'shuffl
e': [True], 'learning_rate_init': [0.001], 'n_iter_no_change': [10],
'max_iter': [200], 'warm_start': [True], 'early_stopping': [True],
'alpha': [0.01, 0.001]}],
       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 [78]:

```
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': 'relu', 'alpha': 0.001, 'batc
h_size': 64, 'early_stopping': True, 'hidden_layer_sizes': (128, 6
4), 'learning_rate_init': 0.001, 'max_iter': 200, 'n_iter_no_chang
e': 10, 'shuffle': True, 'solver': 'adam', 'warm_start': True}
Best Parameters of RFE NN:  {'activation': 'relu', 'alpha': 0.001,
'batch_size': 64, 'early_stopping': True, 'hidden_layer_sizes': (12
8, 64, 32, 16), 'learning_rate_init': 0.001, 'max_iter': 200, 'n_ite
r_no_change': 10, 'shuffle': True, 'solver': 'adam', 'warm_start': T
rue}
Best Parameters of modelSelect NN:  {'activation': 'relu', 'alpha':
0.01, 'batch_size': 64, 'early_stopping': True, 'hidden_layer_size
s': (128, 64, 32, 16), 'learning_rate_init': 0.001, 'max_iter': 200,
'n_iter_no_change': 10, 'shuffle': True, 'solver': 'adam', 'warm_sta
rt': True}



GridSearch:
Number of Layers:  4
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 64
4 Layer with hidden size 1
The activation function:  relu


RFE:
Number of Layers:  6
The First layer is Input Layer, and the last layer is the output lay
er
1 Layer with hidden size 126
2 Layer with hidden size 128
3 Layer with hidden size 64
4 Layer with hidden size 32
5 Layer with hidden size 16
6 Layer with hidden size 1
The activation function:  relu


modelSelect:
Number of Layers:  6
The First layer is Input Layer, and the last layer is the output lay
er
1 Layer with hidden size 24
2 Layer with hidden size 128
3 Layer with hidden size 64
4 Layer with hidden size 32
5 Layer with hidden size 16
6 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 [79]:

```
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: 0.9777430459383532
GridSearch NN Test accuracy: 0.8370167965924616
RFE NN Train accuracy: 0.9760218414909192
RFE NNTest accuracy: 0.8263280559350639
modelSelect NN Train accuracy: 0.9487199778419657
modelSelect NN Test accuracmodelSelect_cvy: 0.7963513622116852
```

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

In [80]:

```
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:  58
Number of iterations rfe ran:  53
Number of iterations modelSelect ran:  98
```

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

In [81]:

```
## From the training and test accuracy, we can see that both RFE NN and model_se
lected NN has the sign of overfitting

## Ly pls modify this.
```

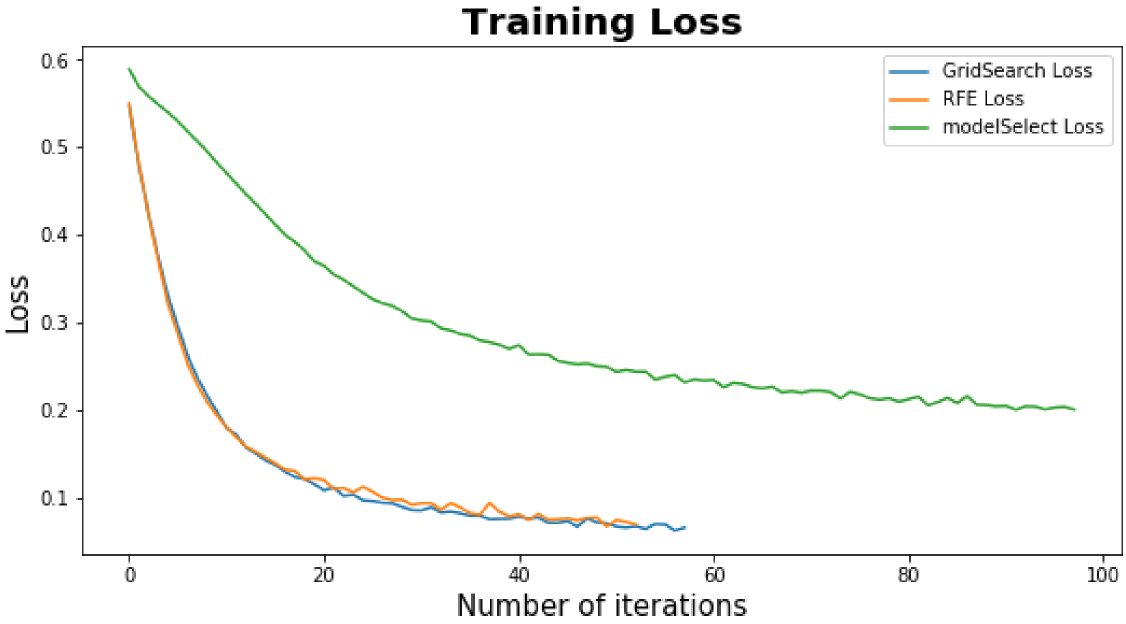
e. Did the training process converge and resulted in the best model?

In [82]:

```
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[82]:

```
<matplotlib.legend.Legend at 0x7f4951aa14a8>
```



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 [83]:

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

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

           0       0.90      0.91      0.91     10832
           1       0.36      0.35      0.36      1611

   micro avg       0.84      0.84      0.84     12443
   macro avg       0.63      0.63      0.63     12443
weighted avg       0.83      0.84      0.84     12443

Confusion Matrix: 
 [[9855  977]
 [1051  560]]


RFE Classification Report: 
              precision    recall  f1-score   support

           0       0.90      0.90      0.90     10832
           1       0.34      0.36      0.35      1611

   micro avg       0.83      0.83      0.83     12443
   macro avg       0.62      0.63      0.62     12443
weighted avg       0.83      0.83      0.83     12443

Confusion Matrix: 
 [[9704 1128]
 [1033  578]]


modelSelect Classification Report: 
              precision    recall  f1-score   support

           0       0.91      0.85      0.88     10832
           1       0.29      0.41      0.34      1611

   micro avg       0.80      0.80      0.80     12443
   macro avg       0.60      0.63      0.61     12443
weighted avg       0.83      0.80      0.81     12443

Confusion Matrix: 
 [[9250 1582]
 [ 952  659]]
```

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 [84]:

```
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 = 0.54707512
Validation score: 0.736301
Iteration 2, loss = 0.47857332
Validation score: 0.777448
Iteration 3, loss = 0.42324346
Validation score: 0.797428
Iteration 4, loss = 0.37362828
Validation score: 0.814441
Iteration 5, loss = 0.32970120
Validation score: 0.841345
Iteration 6, loss = 0.29536147
Validation score: 0.845895
Iteration 7, loss = 0.26196995
Validation score: 0.866271
Iteration 8, loss = 0.23640891
Validation score: 0.867854
Iteration 9, loss = 0.21630361
Validation score: 0.891592
Iteration 10, loss = 0.19812908
Validation score: 0.897527
Iteration 11, loss = 0.17922477
Validation score: 0.892186
Iteration 12, loss = 0.17157421
Validation score: 0.893966
Iteration 13, loss = 0.15687880
Validation score: 0.898912
Iteration 14, loss = 0.15013150
Validation score: 0.908012
Iteration 15, loss = 0.14213609
Validation score: 0.913551
Iteration 16, loss = 0.13710825
Validation score: 0.908803
Iteration 17, loss = 0.12941752
Validation score: 0.916123
Iteration 18, loss = 0.12335300
Validation score: 0.908605
Iteration 19, loss = 0.12127017
Validation score: 0.920475
Iteration 20, loss = 0.11510558
Validation score: 0.918299
Iteration 21, loss = 0.10792456
Validation score: 0.916716
Iteration 22, loss = 0.11128821
Validation score: 0.923838
Iteration 23, loss = 0.10161774
Validation score: 0.915727
Iteration 24, loss = 0.10311017
Validation score: 0.923244
Iteration 25, loss = 0.09677756
Validation score: 0.916123
Iteration 26, loss = 0.09564818
Validation score: 0.919881
Iteration 27, loss = 0.09391351
Validation score: 0.919881
Iteration 28, loss = 0.09325189
Validation score: 0.920673
Iteration 29, loss = 0.08933597
Validation score: 0.919090
Iteration 30, loss = 0.08553687
Validation score: 0.927003
Iteration 31, loss = 0.08509835
```

```
Validation score: 0.920870
Iteration 32, loss = 0.08890293
Validation score: 0.929575
Iteration 33, loss = 0.08273223
Validation score: 0.927399
Iteration 34, loss = 0.08393377
Validation score: 0.919683
Iteration 35, loss = 0.08182656
Validation score: 0.934520
Iteration 36, loss = 0.07923991
Validation score: 0.929377
Iteration 37, loss = 0.07911647
Validation score: 0.924036
Iteration 38, loss = 0.07507023
Validation score: 0.918892
Iteration 39, loss = 0.07546001
Validation score: 0.932938
Iteration 40, loss = 0.07573450
Validation score: 0.925618
Iteration 41, loss = 0.07798078
Validation score: 0.935707
Iteration 42, loss = 0.07570306
Validation score: 0.931553
Iteration 43, loss = 0.07707894
Validation score: 0.923046
Iteration 44, loss = 0.07104559
Validation score: 0.932938
Iteration 45, loss = 0.07088950
Validation score: 0.929575
Iteration 46, loss = 0.07306730
Validation score: 0.930959
Iteration 47, loss = 0.06642030
Validation score: 0.939268
Iteration 48, loss = 0.07605865
Validation score: 0.931157
Iteration 49, loss = 0.07145894
Validation score: 0.933531
Iteration 50, loss = 0.07031683
Validation score: 0.932146
Iteration 51, loss = 0.06679548
Validation score: 0.929377
Iteration 52, loss = 0.06558132
Validation score: 0.928388
Iteration 53, loss = 0.06718902
Validation score: 0.936103
Iteration 54, loss = 0.06389646
Validation score: 0.933927
Iteration 55, loss = 0.06966706
Validation score: 0.924629
Iteration 56, loss = 0.06919731
Validation score: 0.924431
Iteration 57, loss = 0.06199414
Validation score: 0.939268
Iteration 58, loss = 0.06546817
Validation score: 0.931751
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 [85]:

```
print("Report for DT: \n",classification_report(y_test_log, y_pred_dt))
print("DT Confusion Matrix: \n", confusion_matrix(y_test, y_pred_dt))

print("\nReport for Logistic Regression: \n",classification_report(y_test_log, y
_pred_log_reg))
print("Logistic Regression Confusion Matrix: \n", confusion_matrix(y_test, y_pre
d_log_reg))

print("\nReport for NN: \n",classification_report(y_test_log, y_pred_nn))
print("NN Confusion Matrix: \n", confusion_matrix(y_test, y_pred_nn))

print("\nReport for Ensemble: \n",classification_report(y_test_log, y_pred_ensem
ble))
print("Ensemble Confusion Matrix: \n", confusion_matrix(y_test, y_pred_ensemble
))
```

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

           0       0.87      0.95      0.91     10832
           1       0.16      0.07      0.10      1611

   micro avg       0.83      0.83      0.83     12443
   macro avg       0.52      0.51      0.50     12443
weighted avg       0.78      0.83      0.80     12443

DT Confusion Matrix:
 [[10279   553]
 [ 1502   109]]

Report for Logistic Regression:
               precision    recall  f1-score   support

           0       0.93      0.78      0.85     10832
           1       0.29      0.61      0.39      1611

   micro avg       0.76      0.76      0.76     12443
   macro avg       0.61      0.69      0.62     12443
weighted avg       0.85      0.76      0.79     12443

Logistic Regression Confusion Matrix:
 [[8422 2410]
 [ 635  976]]

Report for NN:
               precision    recall  f1-score   support

           0       0.90      0.91      0.91     10832
           1       0.36      0.35      0.36      1611

   micro avg       0.84      0.84      0.84     12443
   macro avg       0.63      0.63      0.63     12443
weighted avg       0.83      0.84      0.84     12443

NN Confusion Matrix:
 [[9855  977]
 [1051  560]]

Report for Ensemble:
               precision    recall  f1-score   support

           0       0.91      0.93      0.92     10832
           1       0.43      0.38      0.40      1611

   micro avg       0.86      0.86      0.86     12443
   macro avg       0.67      0.65      0.66     12443
weighted avg       0.85      0.86      0.85     12443

Ensemble Confusion Matrix:
 [[10038   794]
 [ 1005   606]]
```

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.

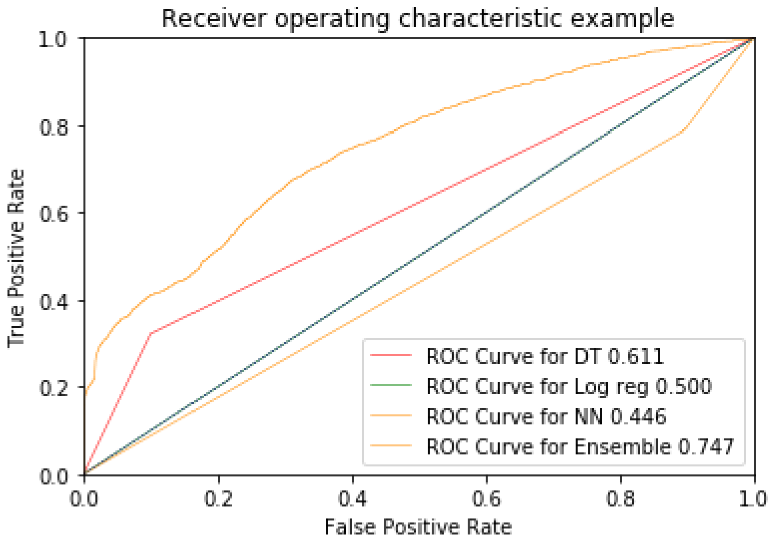
a. Discuss the findings led by (a) ROC Chart (and Index); (b) Score Ranking (or Accuracy Score); (c) Fit Statistics; (or Classification report) and (4) Output.

(a) ROC Chart (and Index)

In [86]:

```
#### 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: 0.6106552750339935
ROC index on test for logistic regression: 0.4997357932951725
ROC index on test for NN: 0.44552339116139317
ROC index on voting classifier: 0.7473490506094089
```



(b) Score Ranking (or Accuracy Score)

In [87]:

```
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: 0.8348469018725387
Accuracy score on test for Logistic Regression: 0.7552840954753677
Accuracy score on test for NN: 0.8370167965924616
Accuracy score on test for Ensemble: 0.8554207184762517
```

(c) Classification report

In [88]:

```
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       0.87      0.95      0.91     10832
           1       0.16      0.07      0.10      1611

   micro avg       0.83      0.83      0.83     12443
   macro avg       0.52      0.51      0.50     12443
weighted avg       0.78      0.83      0.80     12443


Report for Logistic Regression:
               precision    recall  f1-score   support

           0       0.93      0.78      0.85     10832
           1       0.29      0.61      0.39      1611

   micro avg       0.76      0.76      0.76     12443
   macro avg       0.61      0.69      0.62     12443
weighted avg       0.85      0.76      0.79     12443


Report for NN:
               precision    recall  f1-score   support

           0       0.90      0.91      0.91     10832
           1       0.36      0.35      0.36      1611

   micro avg       0.84      0.84      0.84     12443
   macro avg       0.63      0.63      0.63     12443
weighted avg       0.83      0.84      0.84     12443


Report for Ensemble:
               precision    recall  f1-score   support

           0       0.91      0.93      0.92     10832
           1       0.43      0.38      0.40      1611

   micro avg       0.86      0.86      0.86     12443
   macro avg       0.67      0.65      0.66     12443
weighted avg       0.85      0.86      0.85     12443
```

(d) Output

In [89]:

```
### what's the the output? the confusion matrix or just the y_pred? ## Ly pls he
lp me to answer this.
```

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

In [90]:

```
# what's this? ## Ly pls help me to answer this.
```

Task 6. Final Remarks: Decision Making

1. Finally, based on all models and analysis, is there a particular model you will use in decision making? Justify your choice.

We will choose the ensemble model for making decision since it has the highest accuracy. Moreover, the ensemble model has 0.44 precission on the kicks, which means 0.44 it has 44% accuracy when it classify an observation as a kicks. Other model has a lower precision and recall, which means those models can't efficiently detect the "Kicks". If we want to apply this model in the real world, we would expect this model to detect suspecious cases, then apply further investigation on those cases.

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

The NN need more training time and the logistic model need more training time, the decision model and NN model has more serious overfitting problem. However, these two overfitting model have a higher accuracy on the test set. The logisit regression model and th

In [91]:

```
# Add the measurement time to the basic model,

# Also talk about that NN has lots of hyper-params, so need more time for search
ing params
```

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

The decision maker can use the ensemble model for detecting the suspecious deals.

In [ ]:

```

```

In [ ]:

```

```