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 = None #['ros', 'rus']
if NEW_STATEGY:
    print("Using New Preprocessing Strategy")
    using_cat = False
    categorial_cols = ['Auction', 'VehYear', 'Make', 'Color', 'Transmission','Wh
eelTypeID', 'WheelType', 'Nationality', 'Size', 'TopThreeAmericanName','PRIMEUNI
T','AUCGUART','VNST','IsOnlineSale','ForSale' ] # Replaced by the most common
    interval_cols = ['VehOdo','MMRAcquisitionAuctionAveragePrice','MMRAcquisitio
nAuctionCleanPrice','MMRAcquisitionRetailAveragePrice','MMRAcquisitonRetailClean
Price','VehBCost','WarrantyCost' ]
    drop_cols = ['PurchaseID','PurchaseDate','PurchaseTimestamp']
    questionMark_data = ['MMRCurrentAuctionAveragePrice','MMRCurrentAuctionClean
Price','MMRCurrentRetailAveragePrice','MMRCurrentRetailCleanPrice','MMRCurrentRe
tailRatio']
    replaced_vals = ['?', '#VALUE!']
    if using_cat:
        categorial_cols += questionMark_data
        print("See [MMRCurrentAuctionAveragePrice" +
              "MMRCurrentAuctionCleanPrice, MMRCurrentRetailAveragePrice," +
              " MMRCurrentRetailCleanPrice, MMRCurrentRetailRatio] as Categorial
Data")
    else:
        interval_cols += questionMark_data
        print("See [MMRCurrentAuctionAveragePrice" +
              "MMRCurrentAuctionCleanPrice, MMRCurrentRetailAveragePrice," +
              " MMRCurrentRetailCleanPrice, MMRCurrentRetailRatio] as Interval D
ata")
else:
    print("Using Old Preprocessing Strategy")
    drop_cols = ['PurchaseID','PurchaseDate']
    categorial_cols = ['Auction', 'VehYear', 'Make', 'Color', 'Transmission','Wh
eelTypeID', 'WheelType', 'Nationality', 'Size', 'TopThreeAmericanName','PRIMEUNI
T','AUCGUART','VNST','IsOnlineSale','ForSale' ] # Replaced by the most common
    interval_cols = ['PurchaseTimestamp', 'VehOdo','MMRAcquisitionAuctionAverage
Price','MMRAcquisitionAuctionCleanPrice','MMRAcquisitionRetailAveragePrice','MMR
AcquisitonRetailCleanPrice','MMRCurrentAuctionAveragePrice','MMRCurrentAuctionCl
eanPrice','MMRCurrentRetailAveragePrice','MMRCurrentRetailCleanPrice','MMRCurren
tRetailRatio','VehBCost','WarrantyCost' ] # Replaced by the mean
    replaced_vals = ['?', '#VALUE!']

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

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

In [7]:

```
def printColumnInfo():

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

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

```
======================= PurchaseID =======================
----------------------- FIRST FIVE -----------------------
0    0
1    1
2    2
3    3
4    4
Name: PurchaseID, dtype: int64
----------------------- DESCIRBE -----------------------
count    41476.000000
mean     20737.500000
std      11973.234219
min          0.000000
25%      10368.750000
50%      20737.500000
75%      31106.250000
max      41475.000000
Name: PurchaseID, dtype: float64
----------------------- COUNTS -----------------------
Five Most Common:  [2047, 11567, 15693, 13644, 3403]
Num of NULL:  0
Number of ? : 0
Number of #VALUE! : 0
======================= PurchaseTimestamp =======================
----------------------- FIRST FIVE -----------------------
0    1253232000
1    1253232000
2    1253232000
3    1253232000
4    1253232000
Name: PurchaseTimestamp, dtype: int64
----------------------- DESCIRBE -----------------------
count    4.147600e+04
mean     1.262260e+09
std      1.796895e+07
min      1.231114e+09
25%      1.247530e+09
50%      1.262045e+09
75%      1.277770e+09
max      1.293667e+09
Name: PurchaseTimestamp, dtype: float64
----------------------- COUNTS -----------------------
Five Most Common:  [1235520000, 1259020800, 1234396800, 1264032000,
1287014400]
Num of NULL:  0
Number of ? : 0
Number of #VALUE! : 0
======================= PurchaseDate =======================
----------------------- FIRST FIVE -----------------------
0    18/09/2009 10:00
1    18/09/2009 10:00
2    18/09/2009 10:00
3    18/09/2009 10:00
4    18/09/2009 10:00
Name: PurchaseDate, dtype: object
----------------------- DESCIRBE -----------------------
count                41476
unique                 497
top       24/11/2009 10:00
freq                   242
Name: PurchaseDate, dtype: object
```

```
---------------------- COUNTS ----------------------
Five Most Common:  ['24/11/2009 10:00', '12/02/2009 10:00', '25/02/2
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
CADILLAC        17
SUBARU          17
LEXUS           13
VOLVO           12
Name: Make, dtype: int64
Num of NULL:  44
Number of ? : 0
Number of #VALUE! : 0
====================== Color ======================
----------------------- FIRST FIVE -----------------------
0       RED
1       RED
2    SILVER
3       RED
4    SILVER
Name: Color, dtype: object
----------------------- DESCIRBE -----------------------
```

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

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

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

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

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

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

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

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

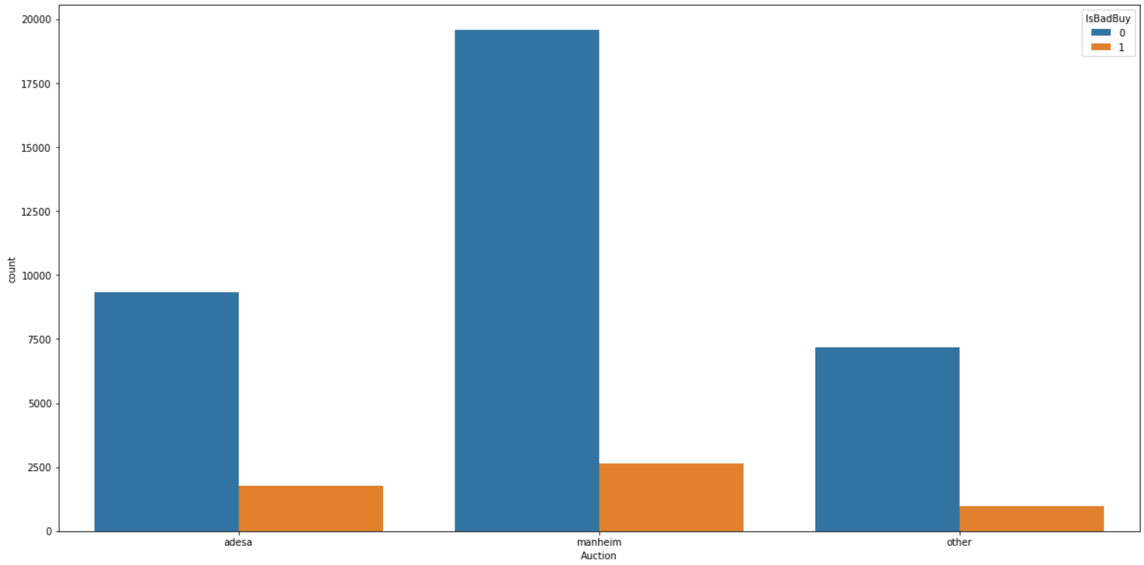
3. Can you identify any clear patterns by initial exploration of the data using histogram or box plot?

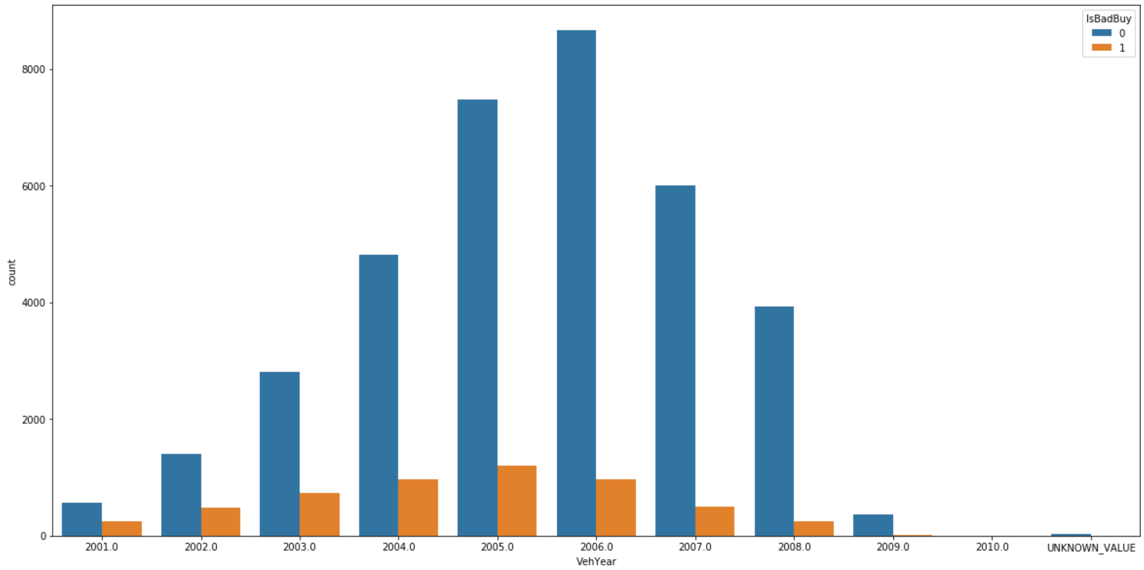
In [9]:

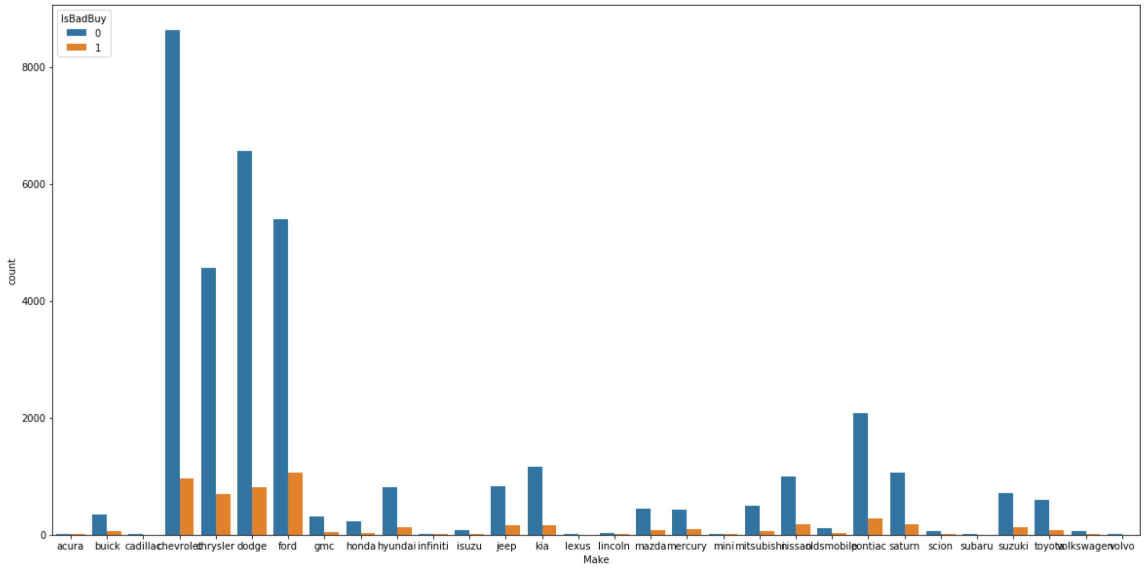
```
def plotAllCols (df):
    for colName in df.columns:
        plt.figure(figsize=(20,10))
        if colName in categorial_cols:
            ### if it's categorial column, plot hist diagram
            sns.countplot(x=colName, data = df, hue="IsBadBuy")
        elif colName in interval_cols:
    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

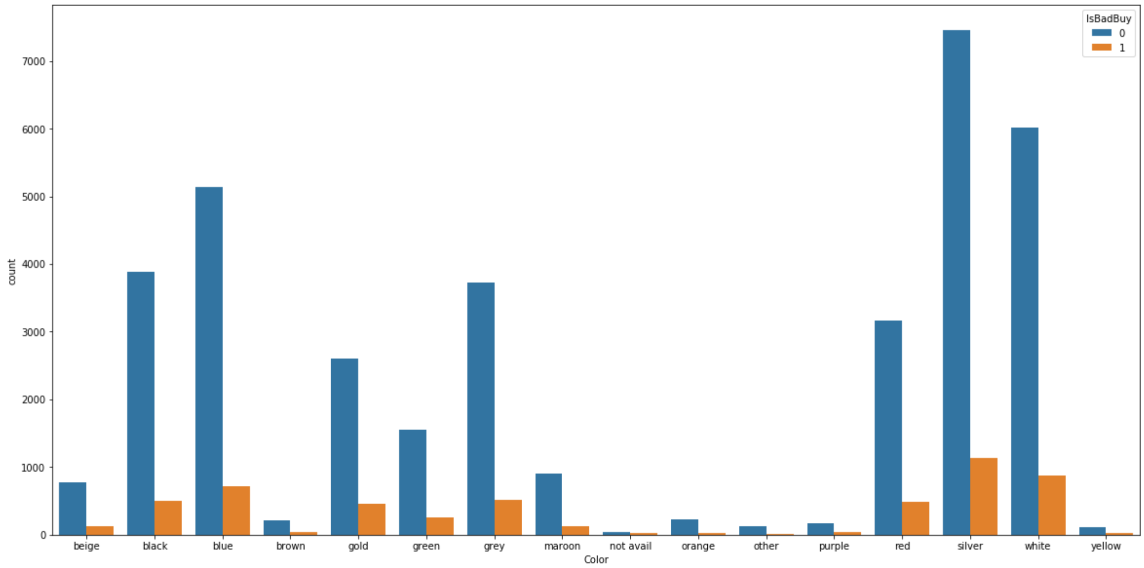
In [10]:

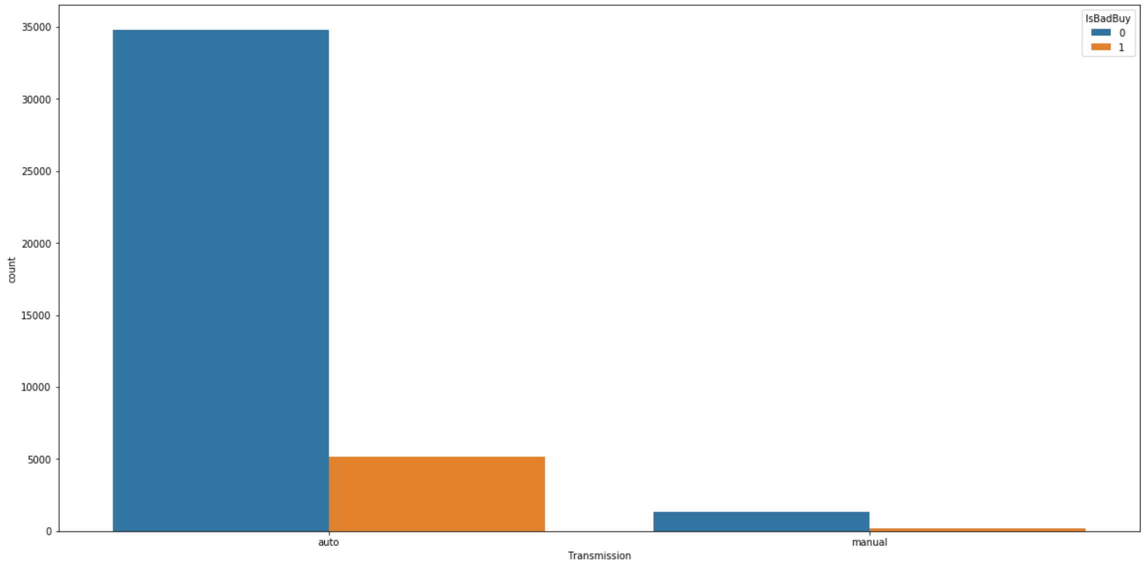
```
plotAllCols(df)
```

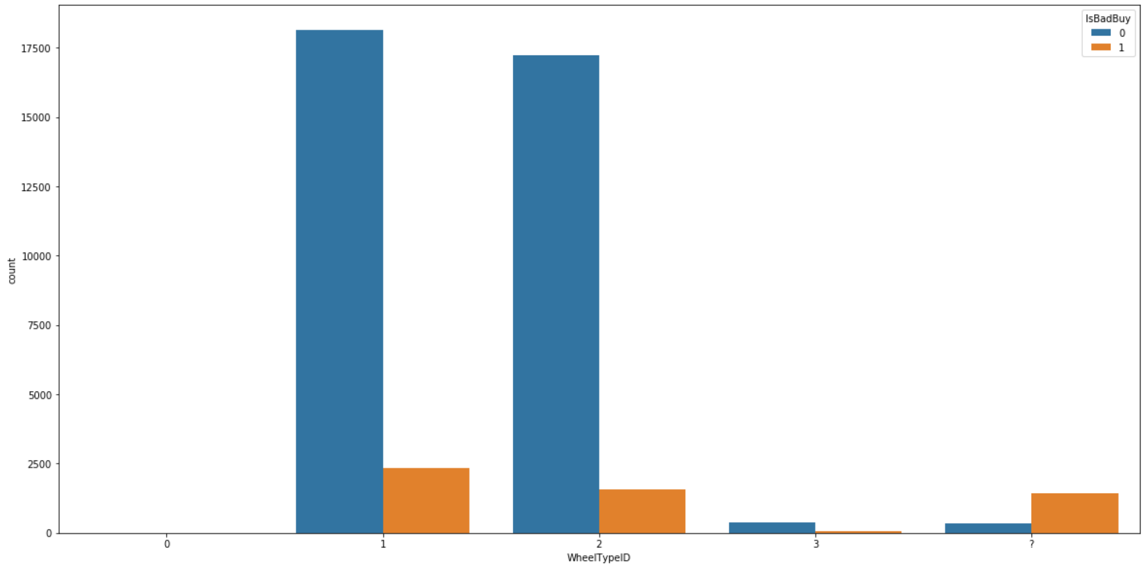

IsBadBuy
0
1
20000
17500
15000
12500
10000
7500
5000
2500
0
count
adesa
manheim
other
Auction

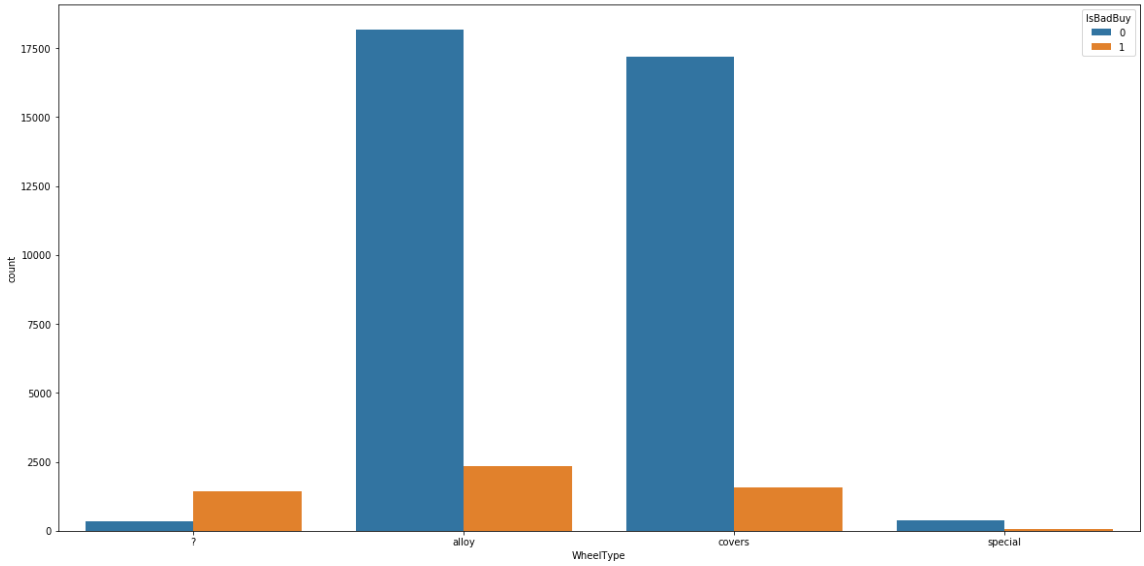

IsBadBuy
0
1
8000
6000
4000
2000
0
count
2001.0
2002.0
2003.0
2004.0
2005.0
2006.0
2007.0
2008.0
2009.0
2010.0
UNKNOWN_VALUE
VehYear

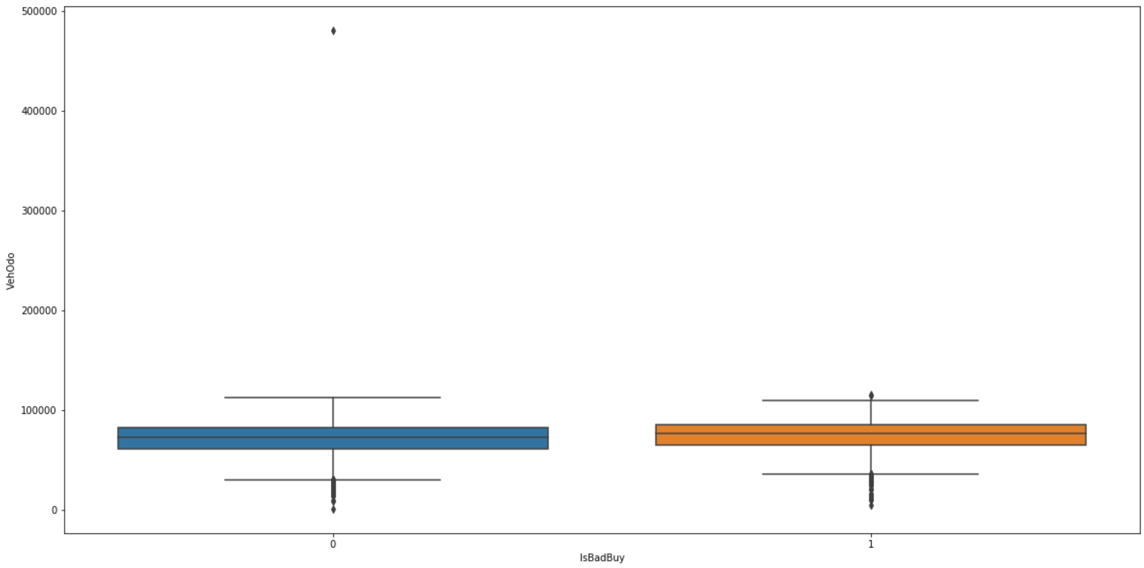

IsBadBuy
0
1
count
Make
acura
buick
cadillac
chevrolet
chrysler
dodge
ford
gmc
honda
hyundai
infiniti
isuzu
jeep
kia
lexus
lincoln
mazda
mercury
mini
mitsubishi
nissan
oldsmobile
pontiac
saturn
scion
subaru
suzuki
toyota
volkswagen
volvo

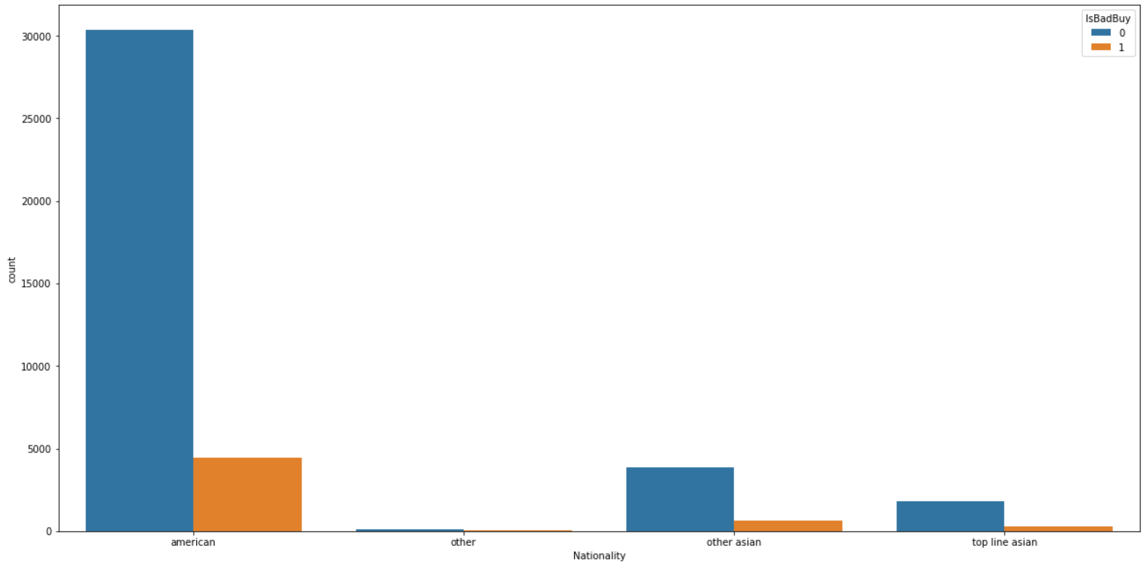

IsBadBuy
0
1
count
Color
beige
black
blue
brown
gold
green
grey
maroon
not avail
orange
other
purple
red
silver
white
yellow

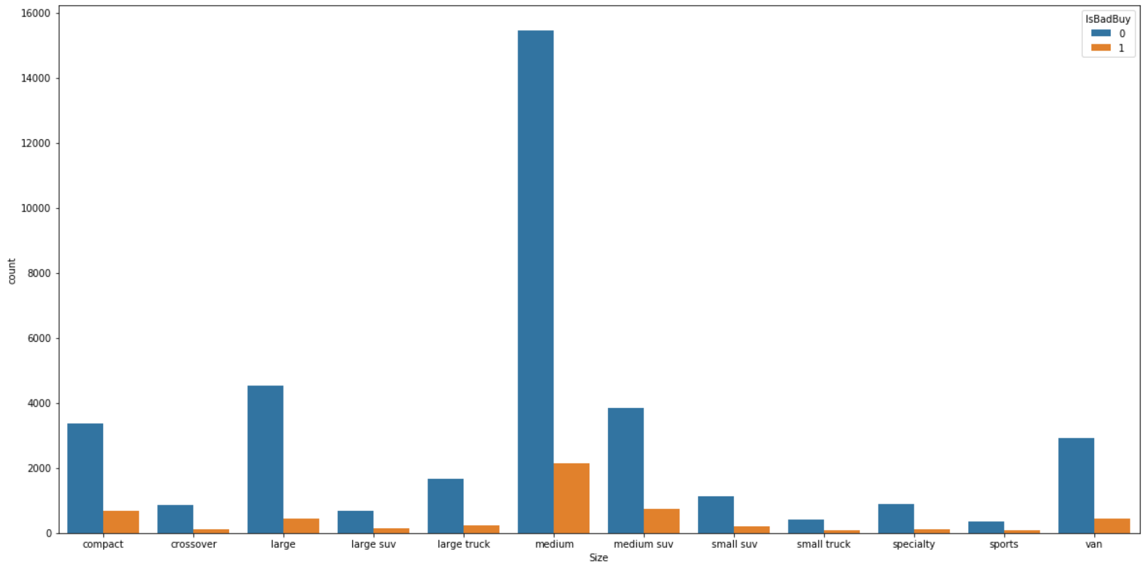

IsBadBuy
0
1
35000
30000
25000
20000
15000
10000
5000
0
count
auto
manual
Transmission

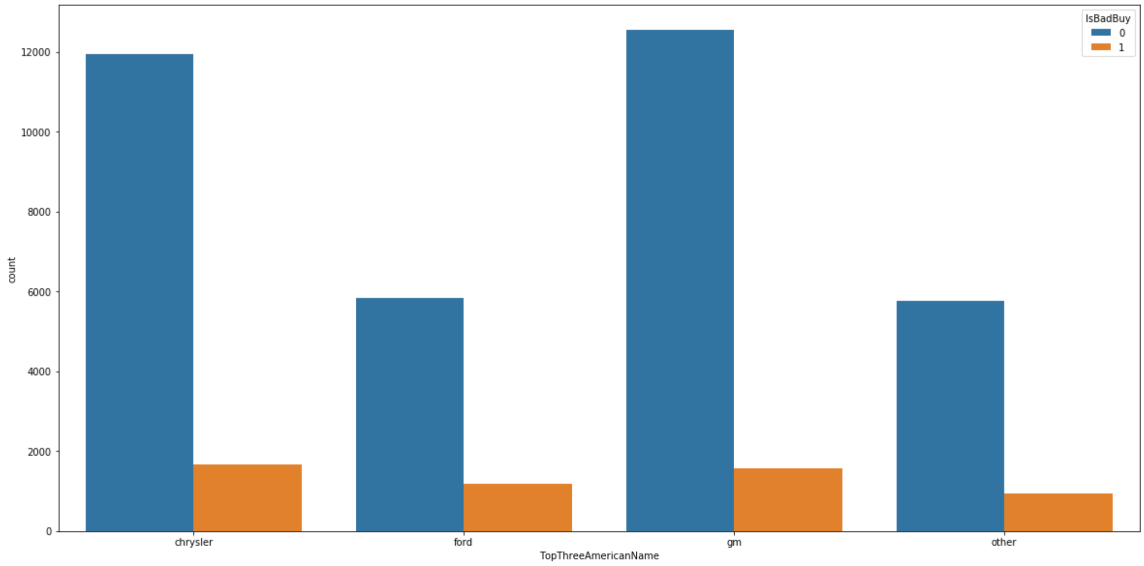

IsBadBuy
0
1
17500
15000
12500
10000
7500
5000
2500
0
count
0
1
2
3
?
WheelTypeID

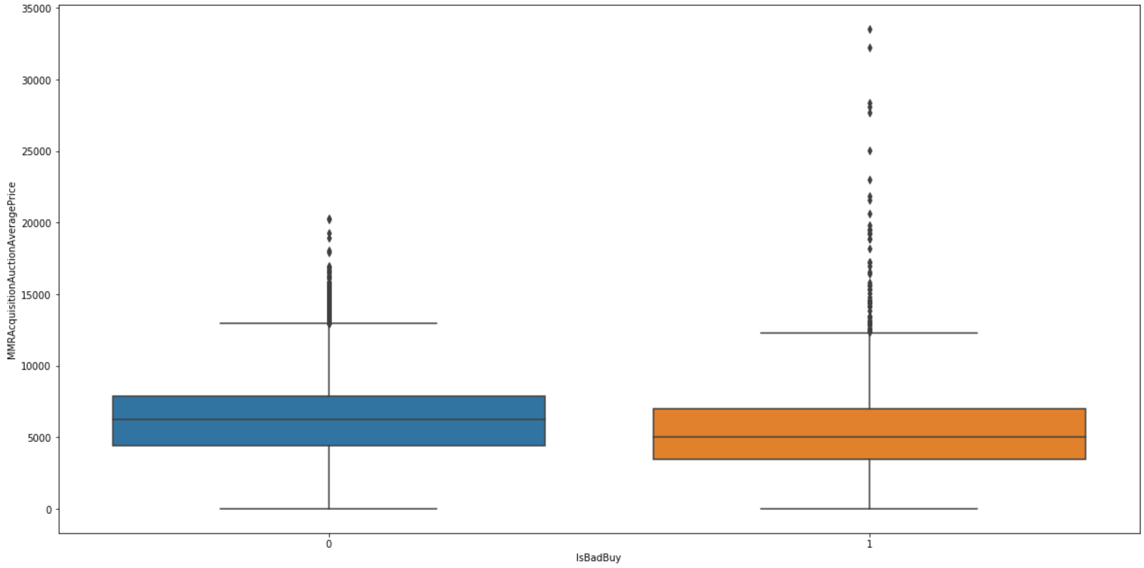

IsBadBuy
0
1
17500
15000
12500
10000
7500
5000
2500
0
count
?
alloy
covers
special
WheelType

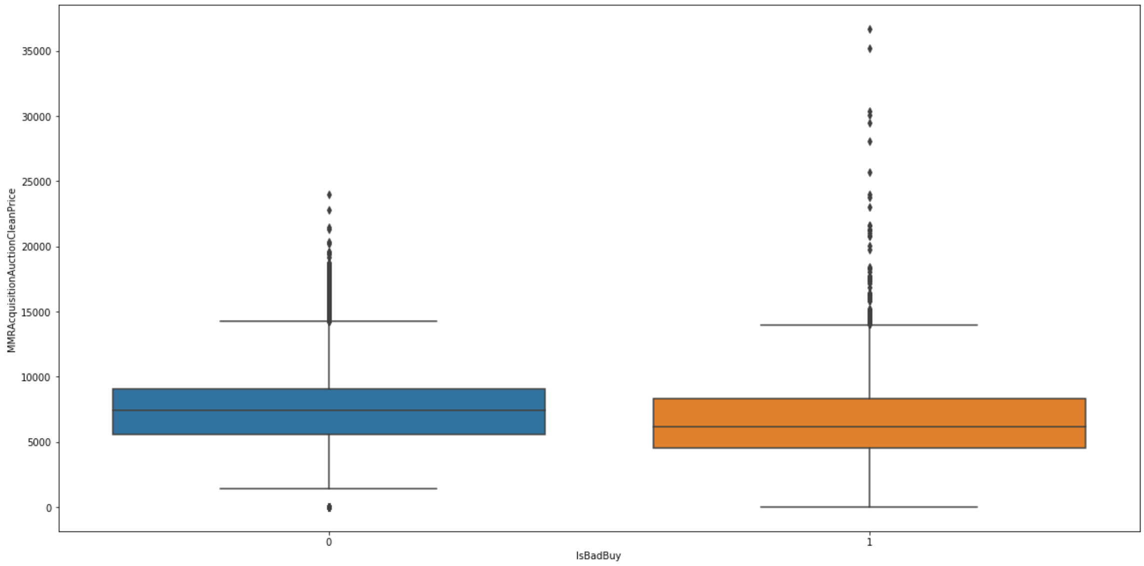

500000
400000
300000
200000
100000
0
VehOdo
0
1
IsBadBuy

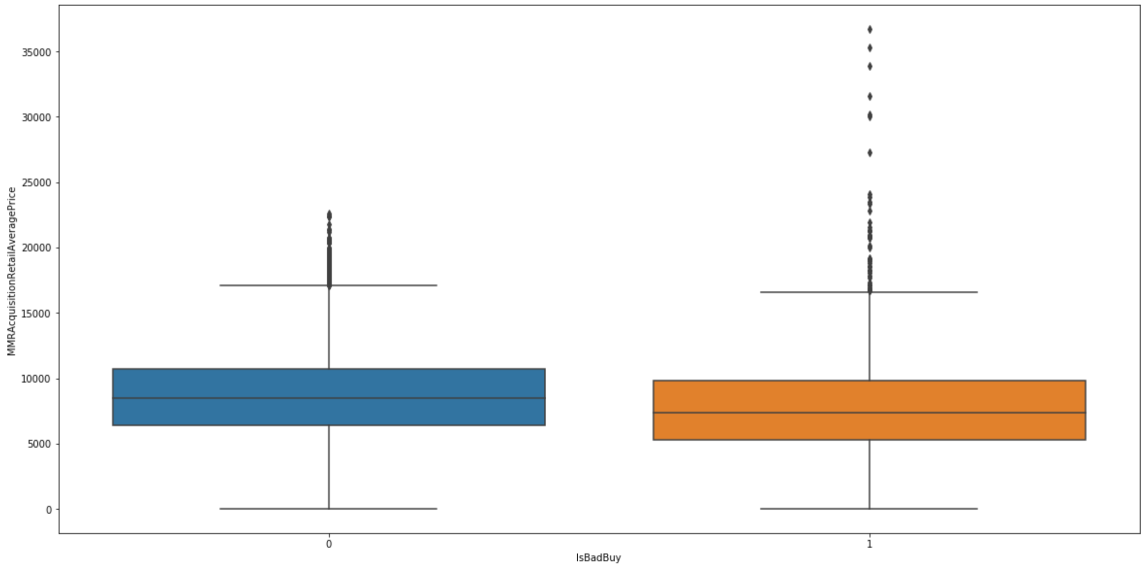

IsBadBuy
0
1
30000
25000
20000
15000
10000
5000
0
count
american
other
other asian
top line asian
Nationality

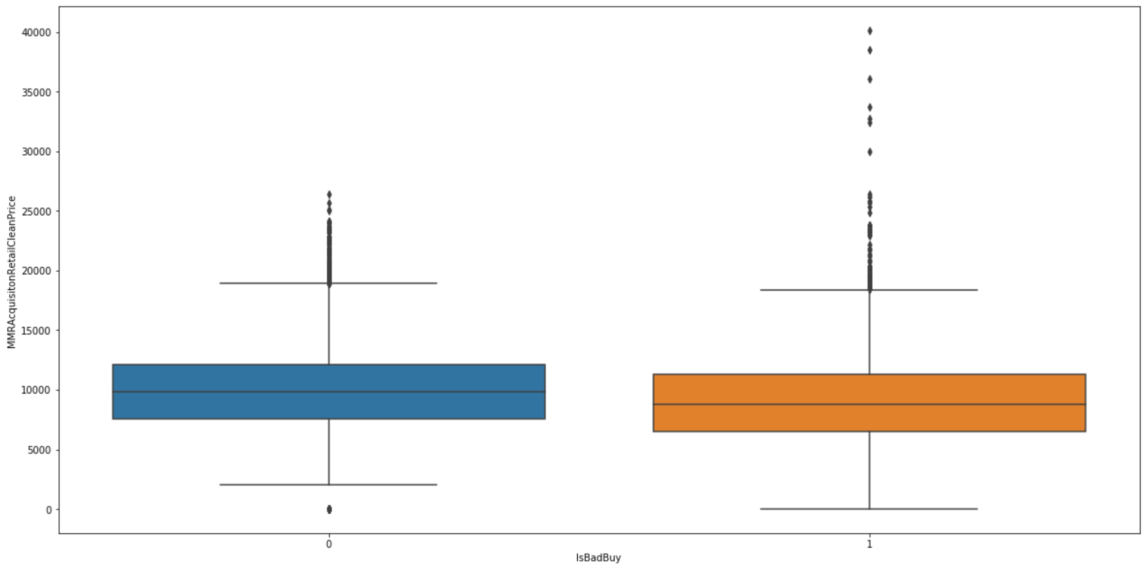

IsBadBuy
0
1
16000
14000
12000
10000
8000
6000
4000
2000
0
count
compact
crossover
large
large suv
large truck
medium
medium suv
small suv
small truck
specialty
sports
van
Size

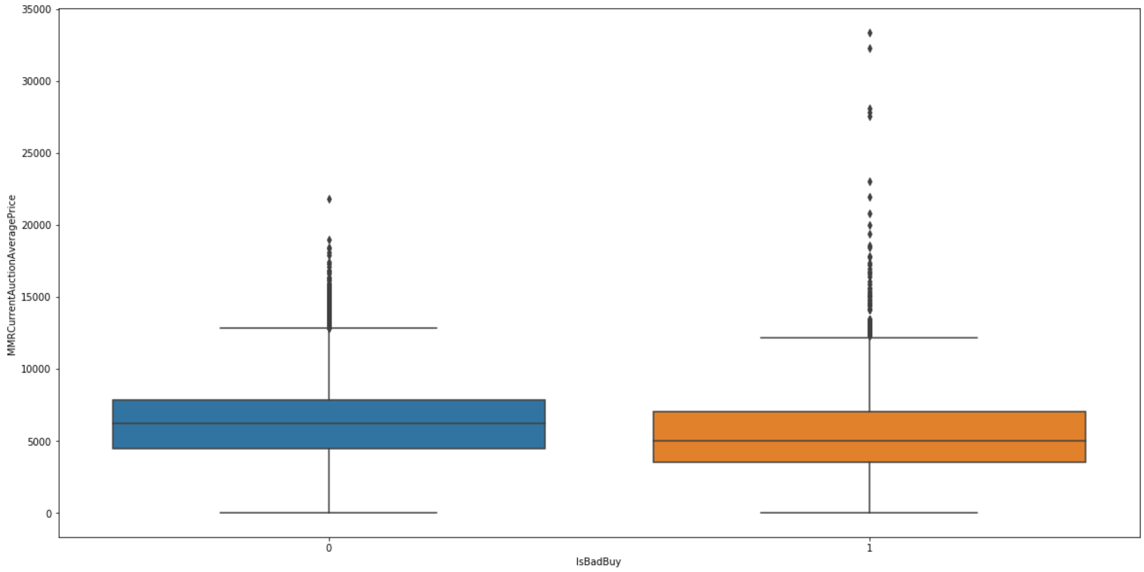

IsBadBuy
0
1
count
12000
10000
8000
6000
4000
2000
0
chrysler
ford
gm
other
TopThreeAmericanName

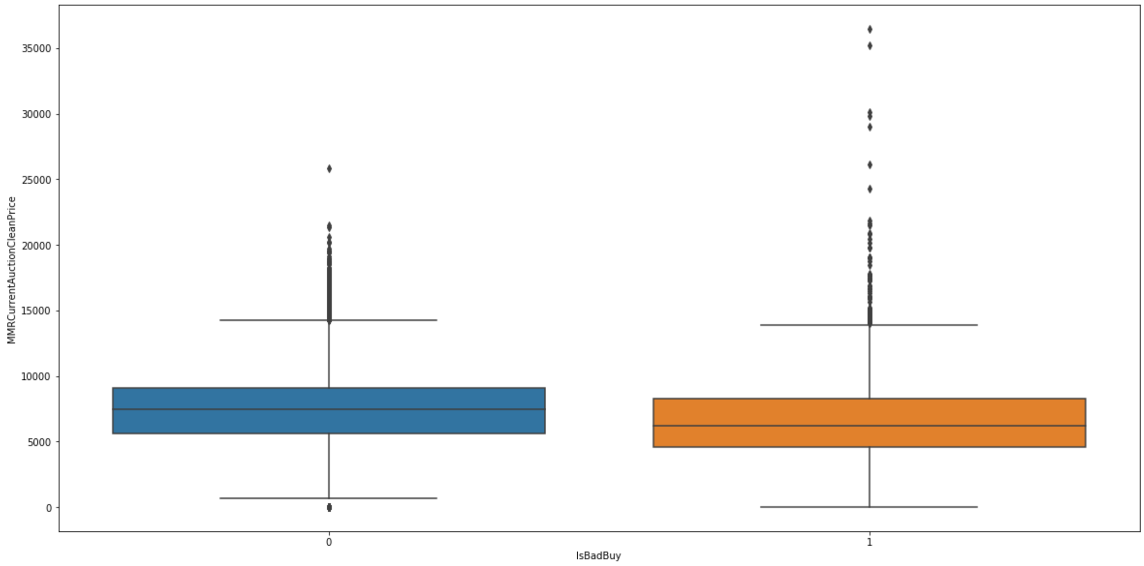

MMRAcquisitionAuctionAveragePrice
35000
30000
25000
20000
15000
10000
5000
0
0
1
IsBadBuy

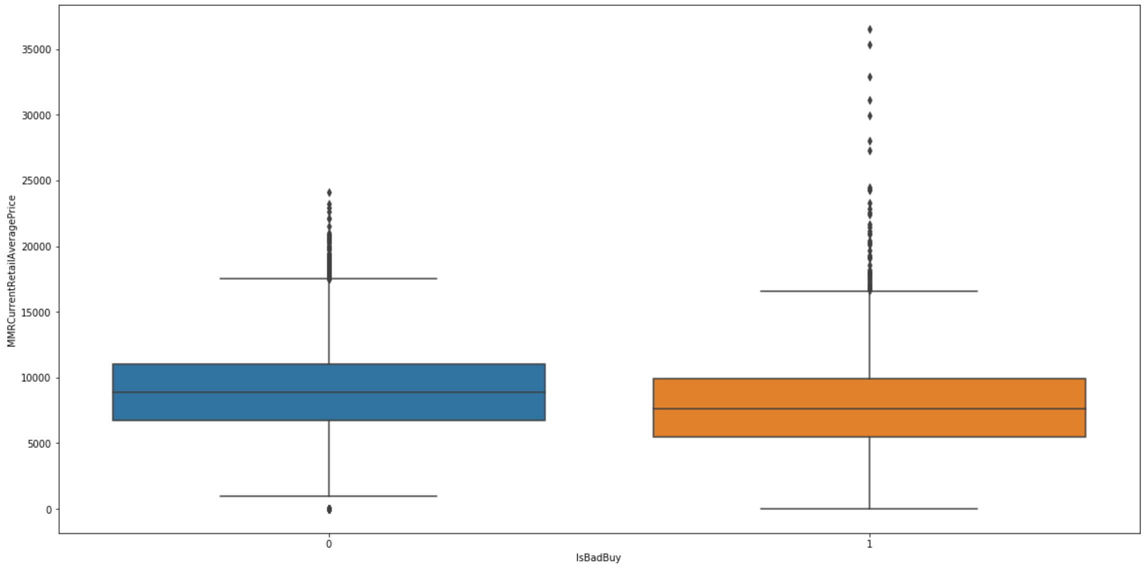

MMRAcquisitionAuctionCleanPrice
IsBadBuy
0
1
0
5000
10000
15000
20000
25000
30000
35000

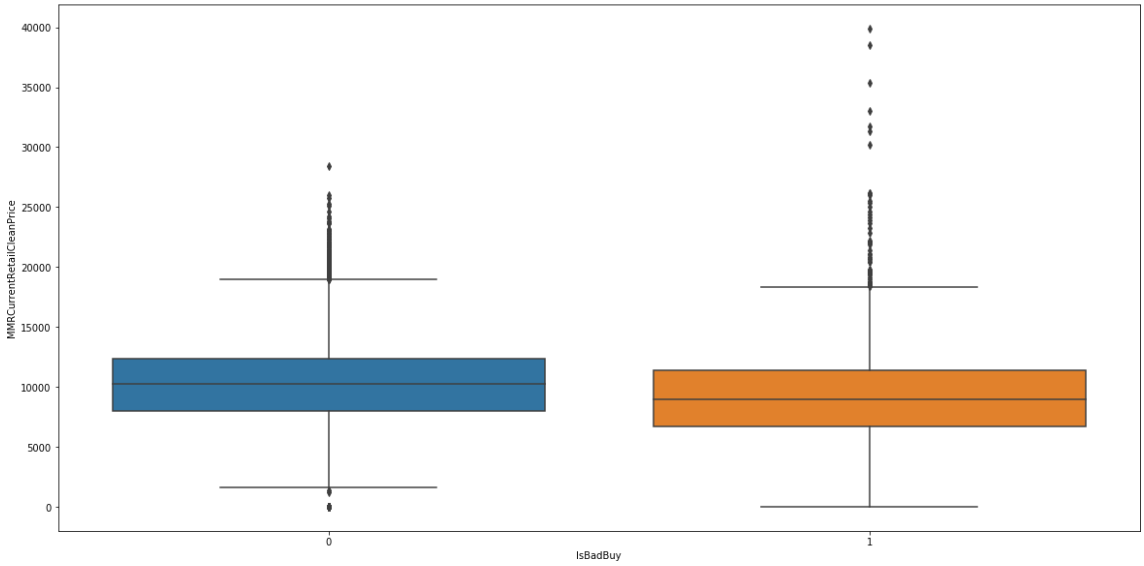

MMRAcquisitionRetailAveragePrice
IsBadBuy
0
1
0
5000
10000
15000
20000
25000
30000
35000

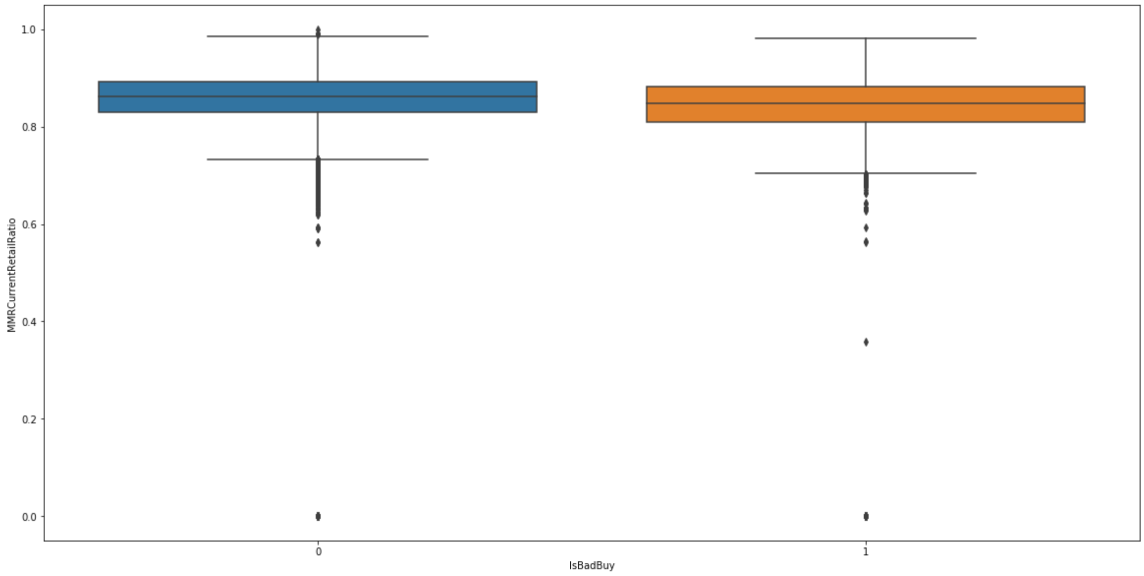

40000
35000
30000
25000
20000
15000
10000
5000
0
MMRAcquisitonRetailCleanPrice
0
1
IsBadBuy

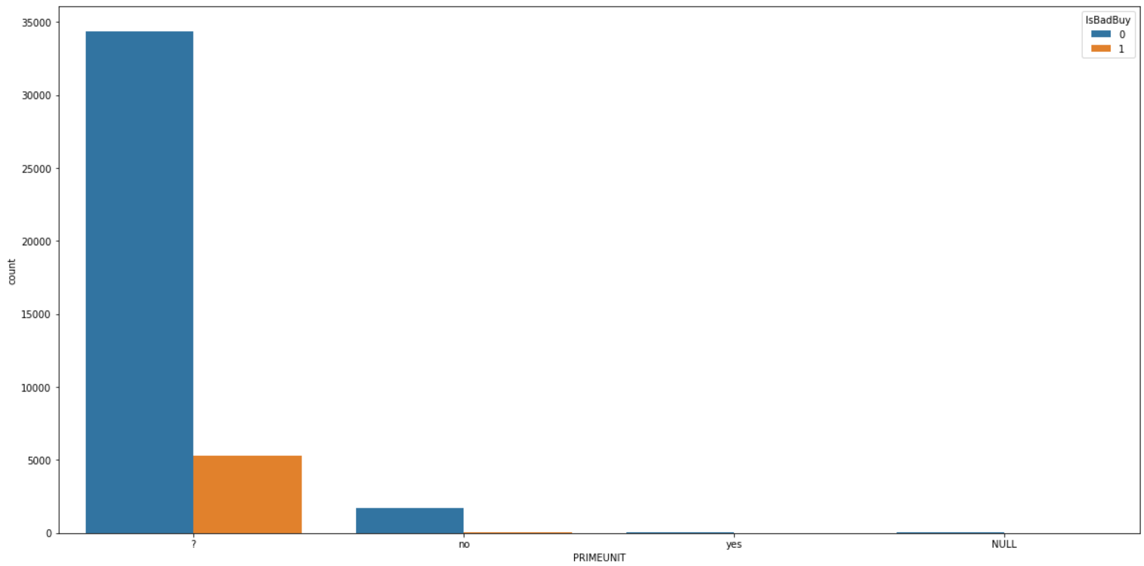

35000
30000
25000
20000
15000
10000
5000
0
MMRCurrentAuctionAveragePrice
0
1
IsBadBuy

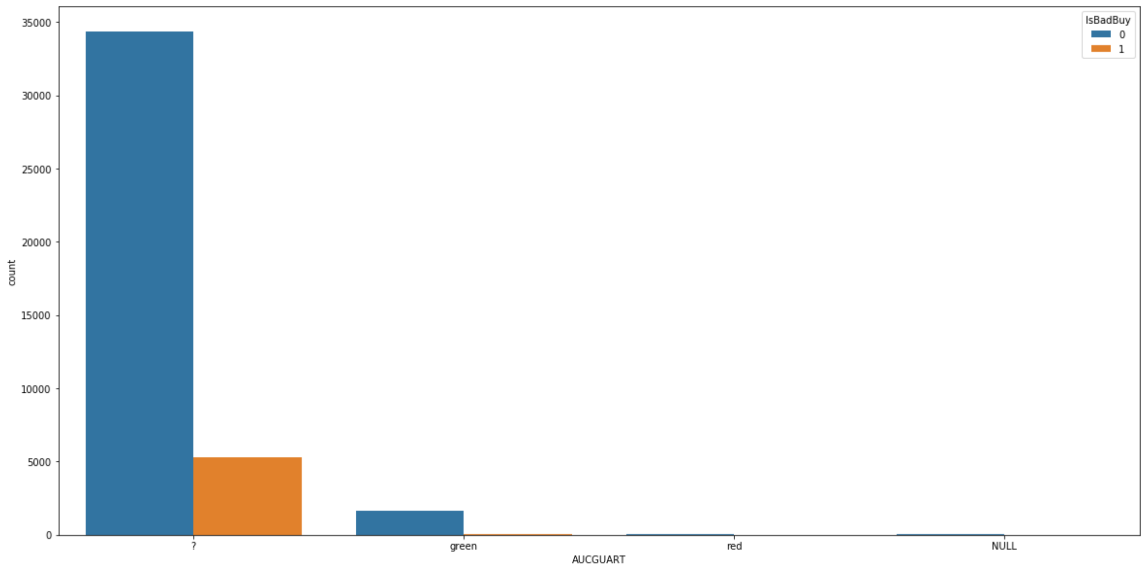

MMRCurrentAuctionCleanPrice
35000
30000
25000
20000
15000
10000
5000
0
0
1
IsBadBuy

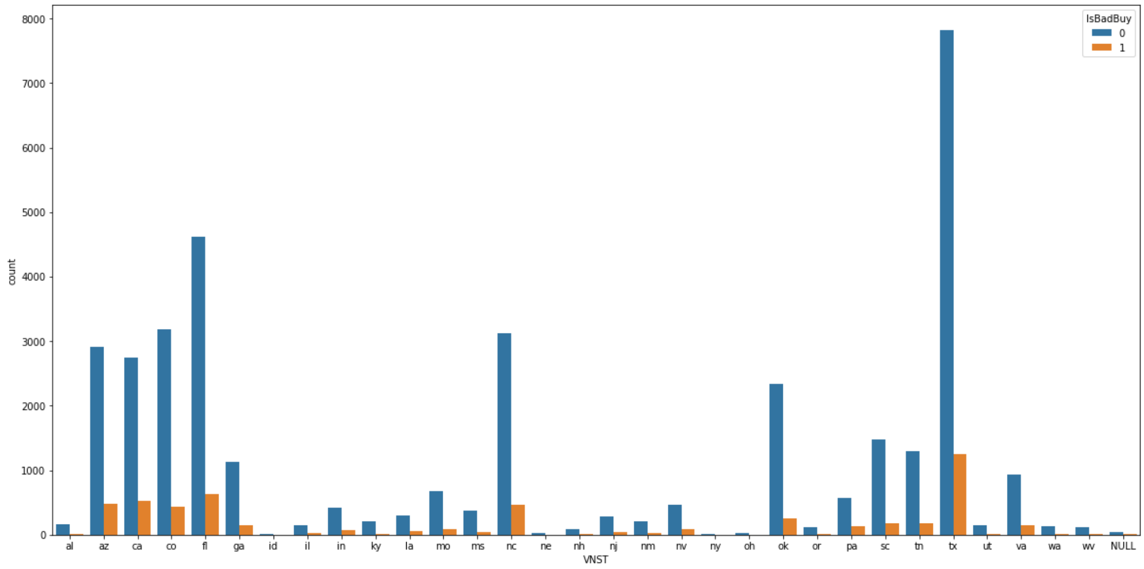

MMRCurrentRetailAveragePrice
35000
30000
25000
20000
15000
10000
5000
0
0
1
IsBadBuy

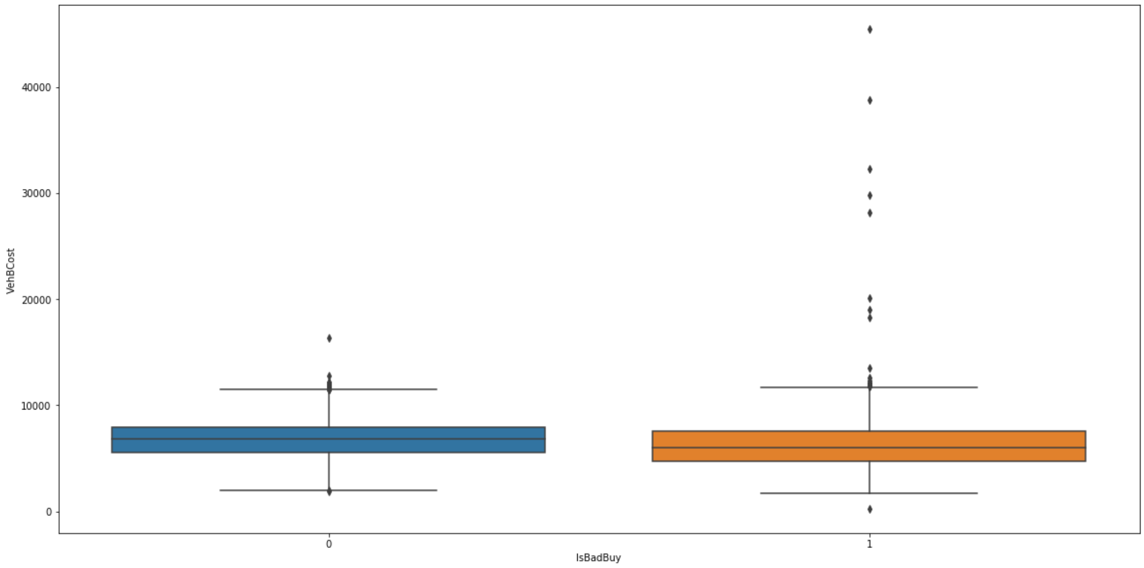

MMRCurrentRetailCleanPrice
40000
35000
30000
25000
20000
15000
10000
5000
0
0
1
IsBadBuy

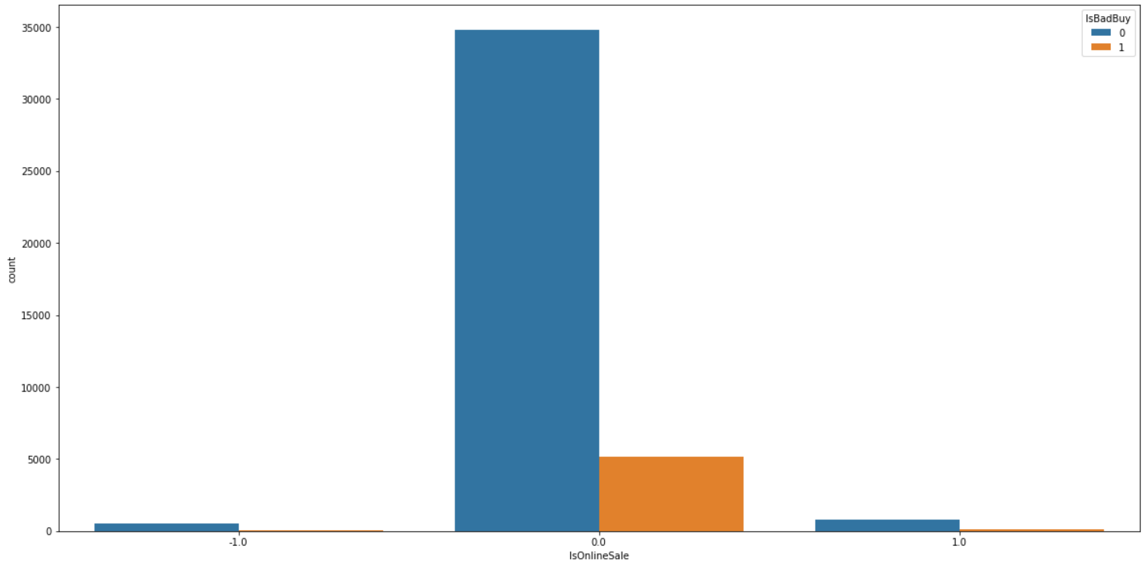

MMRCurrentRetailRatio
1.0
0.8
0.6
0.4
0.2
0.0
0
1
IsBadBuy

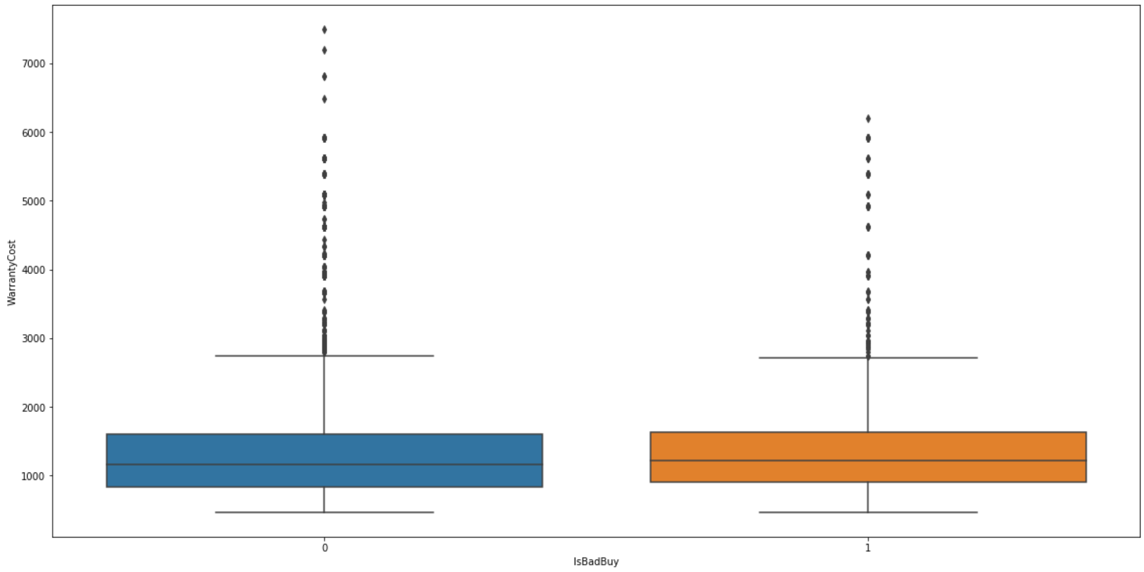

IsBadBuy
0
1
count
35000
30000
25000
20000
15000
10000
5000
0
?
no
yes
NULL
PRIMEUNIT

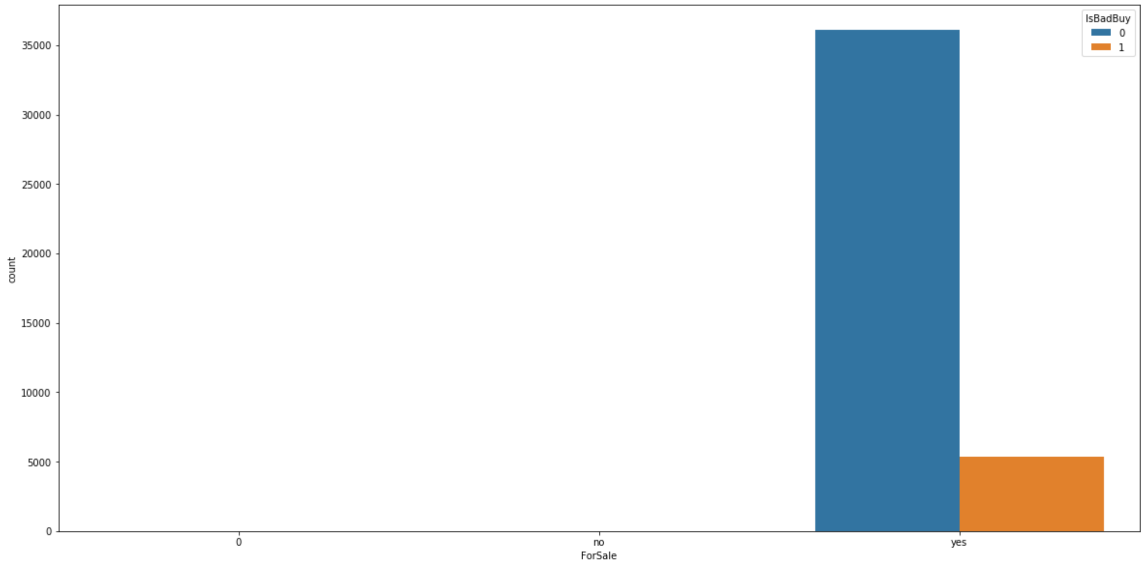

IsBadBuy
0
1
count
35000
30000
25000
20000
15000
10000
5000
0
?
green
red
NULL
AUCGUART


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


VehBCost
IsBadBuy


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


WarrantyCost
7000
6000
5000
4000
3000
2000
1000
0
1
IsBadBuy




```
<Figure size 1440x720 with 0 Axes>
```

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

In [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")
```

```
No Resampling Method Used
```

In [15]:

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

```
Number of Training:  29033
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: 0.9997933386146799
Test accuracy: 0.8183717752953468
              precision    recall  f1-score   support

           0       0.90      0.89      0.89     10832
           1       0.32      0.36      0.34      1611

   micro avg       0.82      0.82      0.82     12443
   macro avg       0.61      0.62      0.62     12443
weighted avg       0.83      0.82      0.82     12443

Confusion Matrix: 
 [[9607 1225]
 [1035  576]]
```

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:  6283
```

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  3142
```

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_? : 0.17480794060968133
VehBCost : 0.07239787499385038
VehOdo : 0.06613050988349085
MMRCurrentRetailRatio : 0.0576476049154974
MMRCurrentAuctionAveragePrice : 0.0473151545321075
```

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: 0.9997933386146799
Test accuracy: 0.8183717752953468
```

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

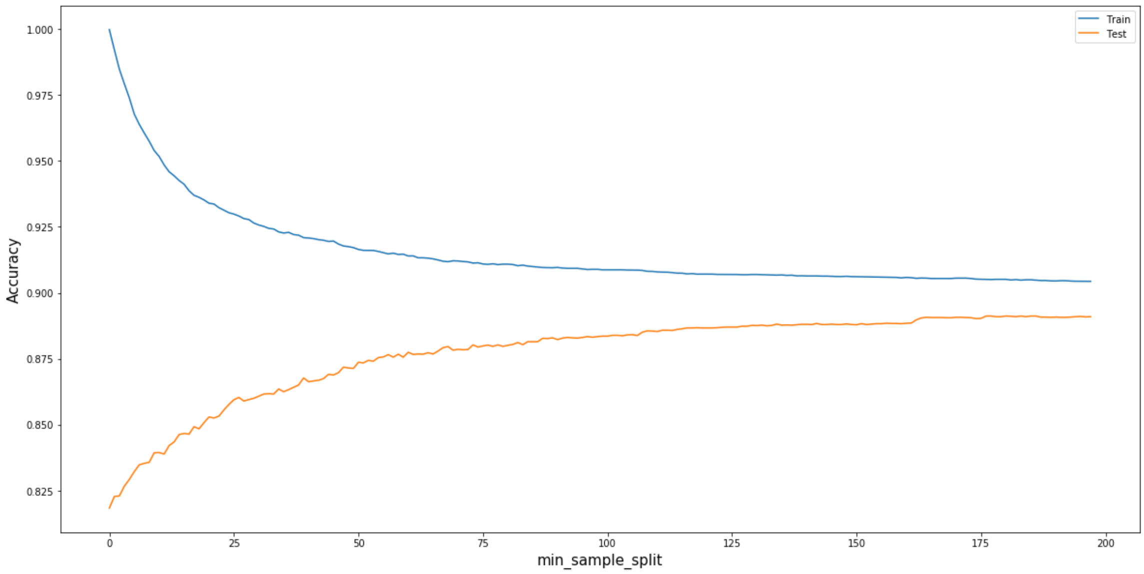
g. Did changing the default setting (i.e., only focus on changing the setting of the number of splits to create a node) help improving the model? Answer the above questions on the best performing tree.

In [24]:

```
### 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 0x7ff206dbebe0>
```



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: 0.8954982261564427
Test accuracy: 0.8980953146347344
              precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.83      0.27      0.40      1611

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

Confusion Matrix:
 [[10746    86]
 [ 1182   429]]
```

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:  3
```

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  2
```

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])
```

```
WheelTypeID_? : 1.0
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)
```

```
WheelTypeID_? : 1.0
ForSale_yes : 0.0
Make_nissan : 0.0
Make_toyota : 0.0
Make_suzuki : 0.0
```

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: 0.8954982261564427
Test accuracy: 0.8980953146347344
```

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': 1, 'max_features': 'auto', '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: 0.9043502221609893
Test accuracy: 0.8909426987060998
Classification report:
               precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.73      0.25      0.37      1611

   micro avg       0.89      0.89      0.89     12443
   macro avg       0.81      0.62      0.66     12443
weighted avg       0.88      0.89      0.87     12443

Confusion Matrix:
 [[10680   152]
 [ 1205   406]]



GridSearch Model:

Train accuracy: 0.8954982261564427
Test accuracy: 0.8980953146347344
Classification report:
               precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.83      0.27      0.40      1611

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

Confusion Matrix:
 [[10746    86]
 [ 1182   429]]
```

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)
```

```
No Resampling Method Used
```

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

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

In [42]:

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

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

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

In [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]:

```
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]:

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

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

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

In [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       0.90      0.99      0.94     10832
           1       0.84      0.26      0.40      1611

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

Default Model Confusion Matrix: 
 [[10751    81]
 [ 1185   426]]
GridSearch Classification Report: 

               precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.84      0.27      0.40      1611

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

GridSearch Confusion Matrix:
  [[10752    80]
 [ 1184   427]]
```

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: 0.8961526538766231
GridSearch Test accuracy: 0.8984167805191674
```

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:  80
Number of selectFromModel features:  1
```

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



The SelectFromModel features: 

 ['WheelTypeID_?']
```

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)
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurr
ent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.3s finishe
d
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurr
ent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.1s 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:

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



Top-5 important variables for selectModel

VehOdo : 0.31924891470356176
```

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': 0.01, 'class_weight': None, 'max_it
er': 50, 'solver': 'lbfgs', 'warm_start': True}
Optimal Parameters for selectModel {'C': 0.0001, 'class_weight': 'ba
lanced', '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: 0.8961526538766231
GridSearch Test accuracy: 0.8984167805191674
```

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: 0.8961526538766231
GridSearch Test accuracy: 0.8984167805191674


RFE:

Train accuracy: 0.8965659766472635
Test accuracy: 0.8984971469902756


selectModel:

Train accuracy: 0.8954982261564427
Test accuracy: 0.8980953146347344
```

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       0.90      0.99      0.94     10832
           1       0.85      0.26      0.40      1611

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

REF Confusion Matrix: 
 [[10755    77]
 [ 1186   425]]



selectModel classification report: 
               precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.83      0.27      0.40      1611

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

selectModel Confusion Matrix: 
 [[10746    86]
 [ 1182   429]]
```

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: 0.8705266420969242
MLP Test accuracy: 0.8705296150446034
```

In [64]:

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

Out[64]:

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

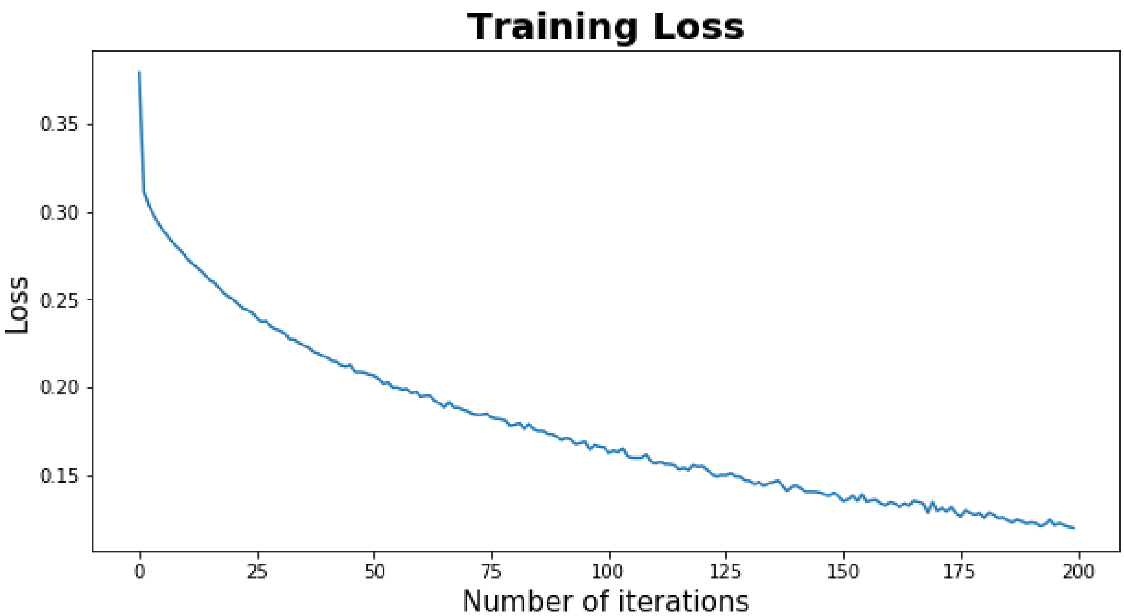
d. Did the training process converge and resulted in the best model?

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 0x7ff206b2b128>]
```



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: 0.8705266420969242
MLP Test accuracy: 0.8705296150446034



MLP classification report:
               precision    recall  f1-score   support

           0       0.87      1.00      0.93     10832
           1       0.00      0.00      0.00      1611

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

MLP Confusion Matrix:
 [[10832     0]
 [ 1611     0]]
```

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 = 0.34152411
Validation score: 0.888774
Iteration 2, loss = 0.30489392
Validation score: 0.890496
Iteration 3, loss = 0.29600840
Validation score: 0.887741
Iteration 4, loss = 0.29005114
Validation score: 0.888085
Iteration 5, loss = 0.28150971
Validation score: 0.889118
Iteration 6, loss = 0.27452219
Validation score: 0.885675
Iteration 7, loss = 0.26559310
Validation score: 0.889118
Iteration 8, loss = 0.25675482
Validation score: 0.880165
Iteration 9, loss = 0.24729563
Validation score: 0.879821
Iteration 10, loss = 0.23670188
Validation score: 0.880510
Iteration 11, loss = 0.22793108
Validation score: 0.874656
Iteration 12, loss = 0.21581747
Validation score: 0.865014
Iteration 13, loss = 0.20664276
Validation score: 0.867080
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, 64,
32, 16), 'learning_rate_init': 0.001, 'max_iter': 200, 'n_iter_no_ch
ange': 10, 'shuffle': True, 'solver': 'adam', 'warm_start': True}
```

In [70]:

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

```
Number of Layers:  6
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 32
5 Layer with hidden size 16
6 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:  13
```

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.
```

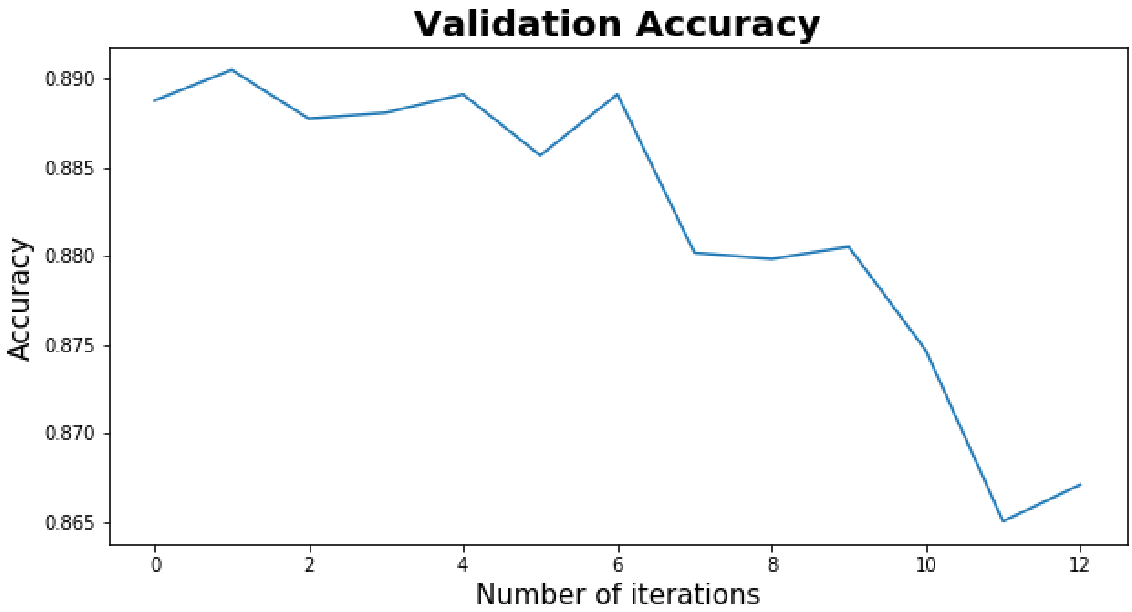
```
GridSearch NN Train accuracy: 0.898598146936245
GridSearch NN Test accuracy: 0.8972112834525436
```

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 0x7ff206b2eba8>]
```



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'
```

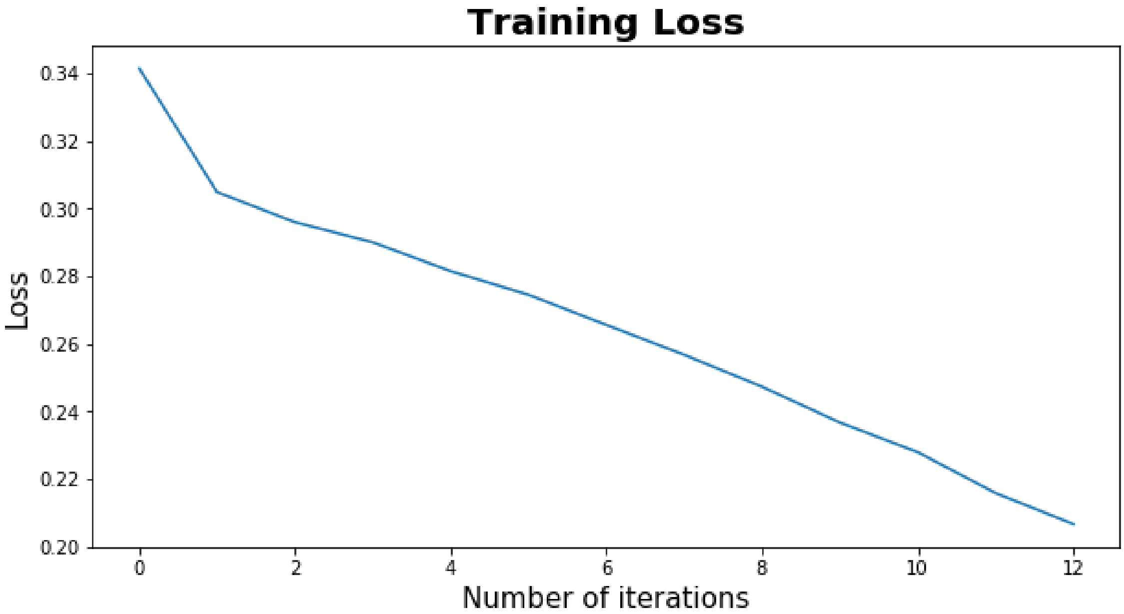
d. Did the training process converge and resulted in the best model?

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 0x7ff206b2d208>]
```



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: 0.898598146936245
GridSearch NN Test accuracy: 0.8972112834525436



GridSearch NN Classification Report: 
               precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.86      0.25      0.38      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.88      0.62      0.66     12443
weighted avg       0.89      0.90      0.87     12443

GridSearch NN Confusion Matrix: 
 [[10765    67]
 [ 1212   399]]
Best Parameters of NN:  {'activation': 'relu', 'alpha': 0.001, 'batc
h_size': 64, 'early_stopping': True, 'hidden_layer_sizes': (128, 64,
32, 16), 'learning_rate_init': 0.001, 'max_iter': 200, 'n_iter_no_ch
ange': 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 = 0.34043827
Validation score: 0.896006
Iteration 2, loss = 0.32038431
Validation score: 0.896694
Iteration 3, loss = 0.31521871
Validation score: 0.896350
Iteration 4, loss = 0.31075403
Validation score: 0.896006
Iteration 5, loss = 0.30691686
Validation score: 0.894628
Iteration 6, loss = 0.30465753
Validation score: 0.895661
Iteration 7, loss = 0.30201420
Validation score: 0.893595
Iteration 8, loss = 0.29882786
Validation score: 0.892906
Iteration 9, loss = 0.29518225
Validation score: 0.894628
Iteration 10, loss = 0.29244294
Validation score: 0.892218
Iteration 11, loss = 0.29027611
Validation score: 0.892906
Iteration 12, loss = 0.28798887
Validation score: 0.892562
Iteration 13, loss = 0.28411061
Validation score: 0.891185
Validation score did not improve more than tol=0.000100 for 10 conse
cutive epochs. Stopping.
Iteration 1, loss = 0.39771178
Validation score: 0.894972
Iteration 2, loss = 0.33653366
Validation score: 0.894972
Iteration 3, loss = 0.33624482
Validation score: 0.894972
Iteration 4, loss = 0.33572577
Validation score: 0.894972
Iteration 5, loss = 0.33538367
Validation score: 0.894972
Iteration 6, loss = 0.33494951
Validation score: 0.894972
Iteration 7, loss = 0.33503967
Validation score: 0.894972
Iteration 8, loss = 0.33463320
Validation score: 0.894972
Iteration 9, loss = 0.33441127
Validation score: 0.894972
Iteration 10, loss = 0.33480210
Validation score: 0.894972
Iteration 11, loss = 0.33426251
Validation score: 0.894972
Iteration 12, loss = 0.33417617
Validation score: 0.894972
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, 64,
32, 16), 'learning_rate_init': 0.001, 'max_iter': 200, 'n_iter_no_ch
ange': 10, 'shuffle': True, 'solver': 'adam', 'warm_start': True}
Best Parameters of RFE NN:  {'activation': 'relu', 'alpha': 0.01, 'b
atch_size': 64, 'early_stopping': True, 'hidden_layer_sizes': (128,
64, 32, 16), 'learning_rate_init': 0.001, 'max_iter': 200, 'n_iter_n
o_change': 10, 'shuffle': True, 'solver': 'adam', 'warm_start': Tru
e}
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:  6
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 32
5 Layer with hidden size 16
6 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 80
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 1
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: 0.898598146936245
GridSearch NN Test accuracy: 0.8972112834525436
RFE NN Train accuracy: 0.898632590500465
RFE NNTest accuracy: 0.8985775134613839
modelSelect NN Train accuracy: 0.8954982261564427
modelSelect NN Test accuracmodelSelect_cvy: 0.8980953146347344
```

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:  13
Number of iterations rfe ran:  13
Number of iterations modelSelect ran:  12
```

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_selected NN has the sign of overfitting

## Ly pls modify this.
```

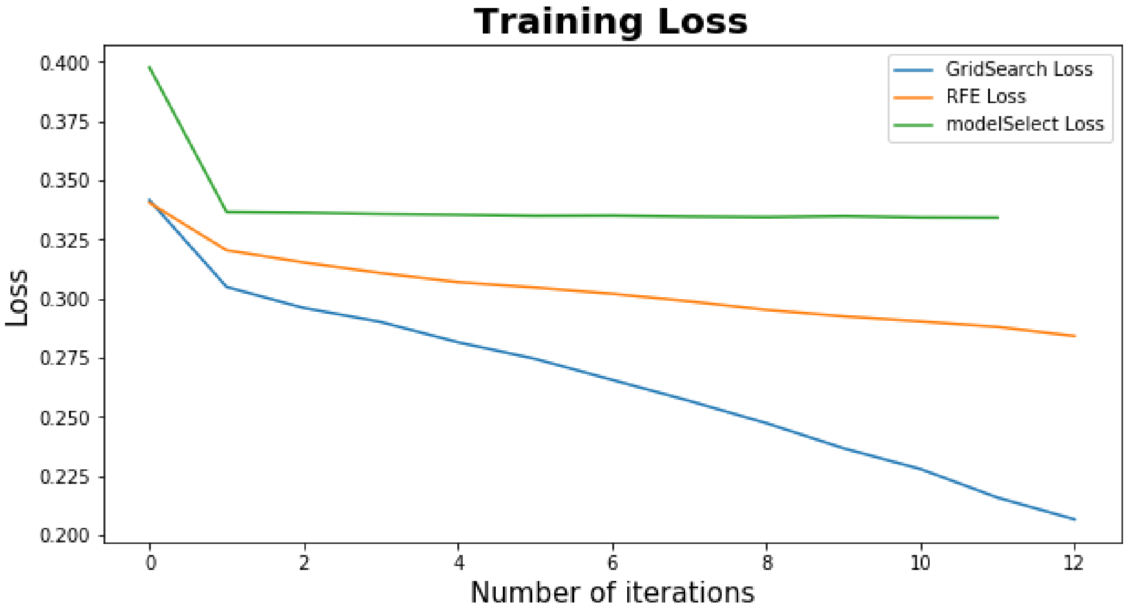
e. Did the training process converge and resulted in the best model?

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 0x7ff20667ac50>
```



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      0.99      0.94     10832
           1       0.86      0.25      0.38      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.88      0.62      0.66     12443
weighted avg       0.89      0.90      0.87     12443

Confusion Matrix: 
 [[10765    67]
 [ 1212   399]]


RFE Classification Report: 
              precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.88      0.25      0.39      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.89      0.62      0.67     12443
weighted avg       0.90      0.90      0.87     12443

Confusion Matrix: 
 [[10774    58]
 [ 1204   407]]


modelSelect Classification Report: 
              precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.83      0.27      0.40      1611

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

Confusion Matrix: 
 [[10746    86]
 [ 1182   429]]
```

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 = 0.34152411
Validation score: 0.888774
Iteration 2, loss = 0.30489392
Validation score: 0.890496
Iteration 3, loss = 0.29600840
Validation score: 0.887741
Iteration 4, loss = 0.29005114
Validation score: 0.888085
Iteration 5, loss = 0.28150971
Validation score: 0.889118
Iteration 6, loss = 0.27452219
Validation score: 0.885675
Iteration 7, loss = 0.26559310
Validation score: 0.889118
Iteration 8, loss = 0.25675482
Validation score: 0.880165
Iteration 9, loss = 0.24729563
Validation score: 0.879821
Iteration 10, loss = 0.23670188
Validation score: 0.880510
Iteration 11, loss = 0.22793108
Validation score: 0.874656
Iteration 12, loss = 0.21581747
Validation score: 0.865014
Iteration 13, loss = 0.20664276
Validation score: 0.867080
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       0.90      0.99      0.94     10832
           1       0.83      0.27      0.40      1611

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

DT Confusion Matrix:
 [[10746    86]
 [ 1182   429]]

Report for Logistic Regression:
              precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.84      0.27      0.40      1611

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

Logistic Regression Confusion Matrix:
 [[10752    80]
 [ 1184   427]]

Report for NN:
              precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.86      0.25      0.38      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.88      0.62      0.66     12443
weighted avg       0.89      0.90      0.87     12443

NN Confusion Matrix:
 [[10765    67]
 [ 1212   399]]

Report for Ensemble:
              precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.83      0.27      0.40      1611

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

Ensemble Confusion Matrix:
 [[10746    86]
 [ 1182   429]]
```

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.

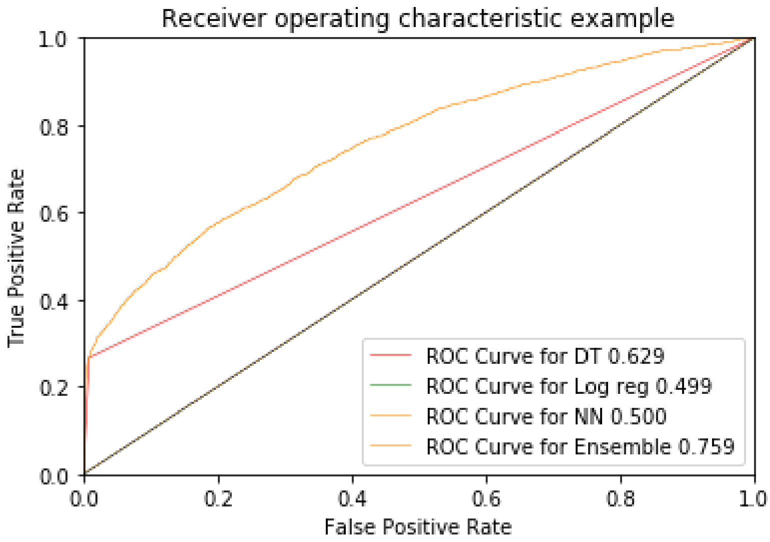
a. Discuss the findings led by (a) ROC Chart (and Index); (b) Score Ranking (or Accuracy Score); (c) Fit Statistics; (or Classification report) and (4) Output.

(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: 0.6291773942439671
ROC index on test for logistic regression: 0.49947161524306216
ROC index on test for NN: 0.5
ROC index on voting classifier: 0.7587285345304209
```



(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: 0.8980953146347344
Accuracy score on test for Logistic Regression: 0.8984167805191674
Accuracy score on test for NN: 0.8972112834525436
Accuracy score on test for Ensemble: 0.8980953146347344
```

(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       0.90      0.99      0.94     10832
           1       0.83      0.27      0.40      1611

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


Report for Logistic Regression:
               precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.84      0.27      0.40      1611

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


Report for NN:
               precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.86      0.25      0.38      1611

   micro avg       0.90      0.90      0.90     12443
   macro avg       0.88      0.62      0.66     12443
weighted avg       0.89      0.90      0.87     12443


Report for Ensemble:
               precision    recall  f1-score   support

           0       0.90      0.99      0.94     10832
           1       0.83      0.27      0.40      1611

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

(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 [ ]:

```

```