# Importing Necessary Libraries

In [1]:

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

'''
TODO:

1. Try to improve

2. Desing the replace_val for each column

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


%matplotlib inline

rs = 101
```

# Task 1. Data Selection and Distribution.

In [2]:

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

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

In [3]:

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

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

In [4]:

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

```
Number of Observations:  41476
```

In [5]:

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

```
The proportion of kicks:  0.1294965763333012
```

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

In [6]:

```
#### PREPROCESSING STATEGY
NEW_STATEGY = True
ResamplingMethod = 'ros' #['ros', 'rus']
if NEW_STATEGY:
    print("Using New Preprocessing Strategy")
    using_cat = False
    categorial_cols = ['Auction', 'VehYear', 'Make', 'Color', 'Transmission','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
SUBARU          17
CADILLAC        17
LEXUS           13
VOLVO           12
Name: Make, dtype: int64
Num of NULL:  44
Number of ? : 0
Number of #VALUE! : 0
====================== Color ======================
---------------------- FIRST FIVE ----------------------
0       RED
1       RED
2    SILVER
3       RED
4    SILVER
Name: Color, dtype: object
---------------------- DESCIRBE ----------------------
```

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

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

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

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

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

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

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

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

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

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

In [8]:

```
if NEW_STATEGY:

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

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

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

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

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

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

    def filterOutRareValue(colName):

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

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

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

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

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

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

            print("Preprocess the col: " + colName)

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

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

        return df

    preprocessStrategy = defaultdict(dict)

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

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

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

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

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

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

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

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

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

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

    }

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

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

    }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
    }

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

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

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

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

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

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

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

```
    newData_prep(df)


else:

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

        # Check the replaced values are not in the dataset

        for colName in df.columns:

            if colName in categorial_cols:

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

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

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

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

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

            if colName in interval_cols:

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

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

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

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

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

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

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

    return df

df = data_prep(df)
```

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

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

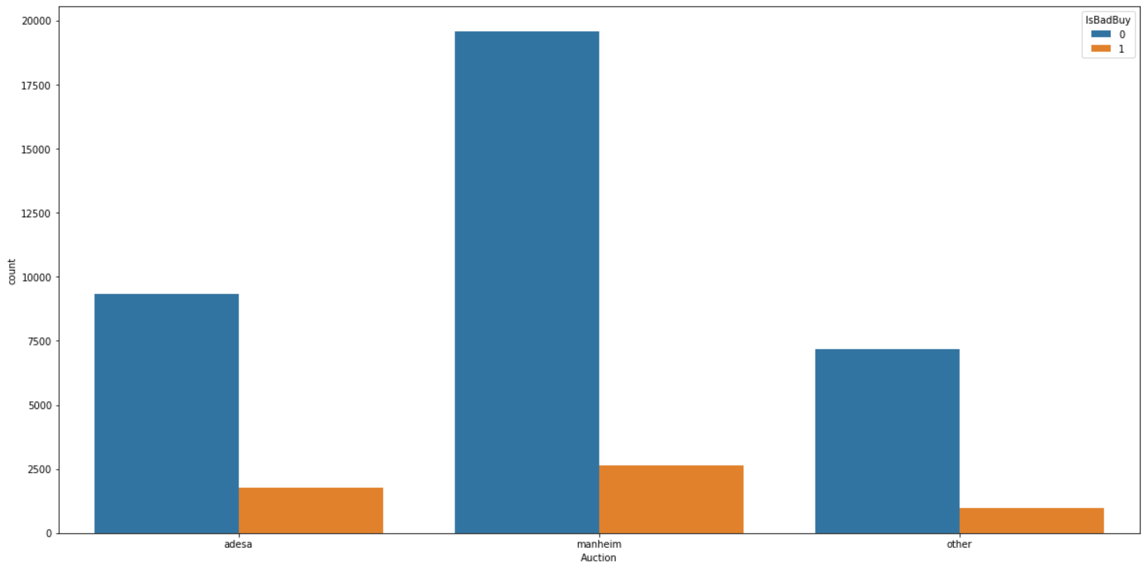
### 3. Can you identify any clear patterns by initial exploration of the data using histogram or box plot?

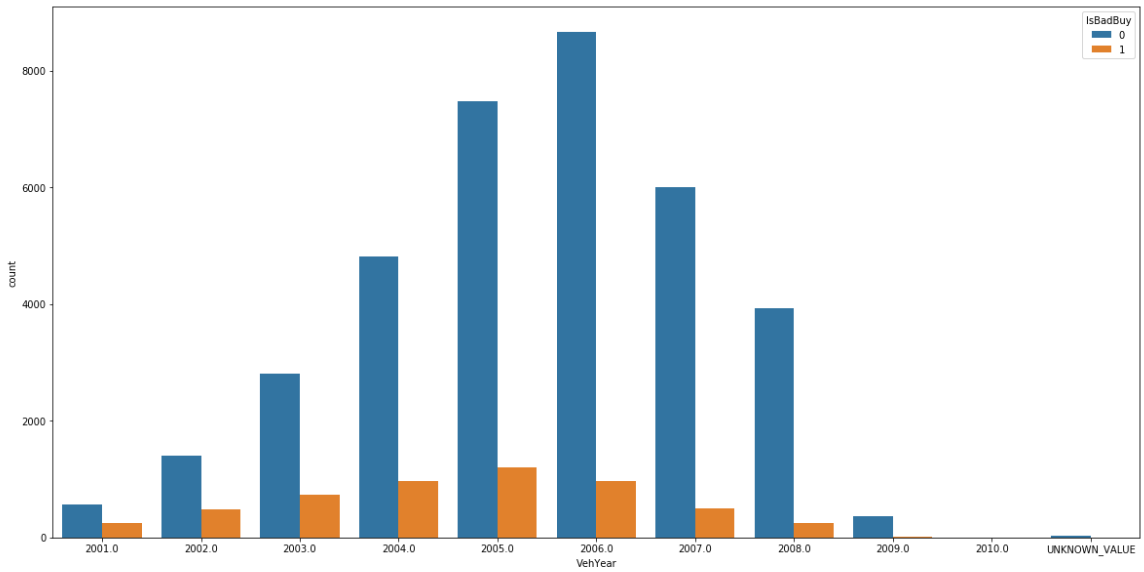
In [9]:

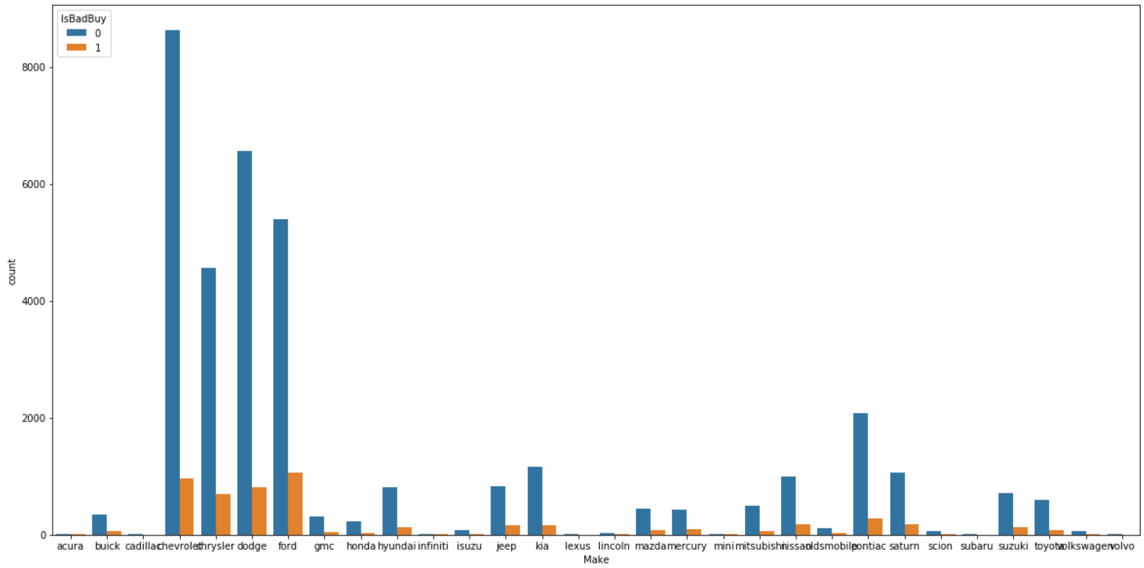
```
def plotAllCols (df):
    for colName in df.columns:
        plt.figure(figsize=(20,10))
        if colName in categorial_cols:
            ### if it's categorial column, plot hist diagram
            sns.countplot(x=colName, data = df, hue="IsBadBuy")
        elif colName in interval_cols:
    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

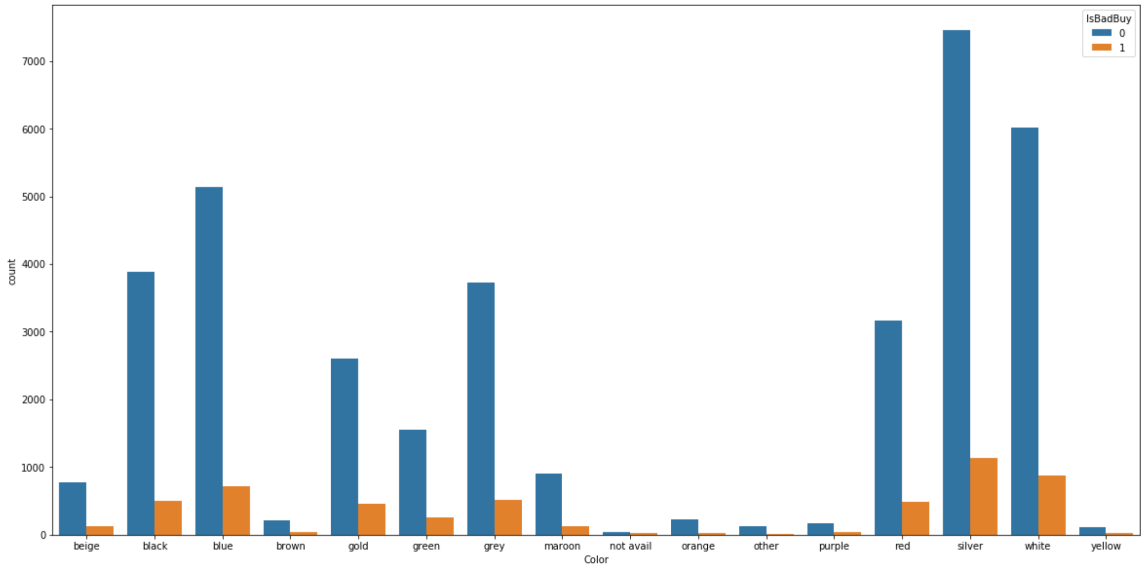
In [10]:

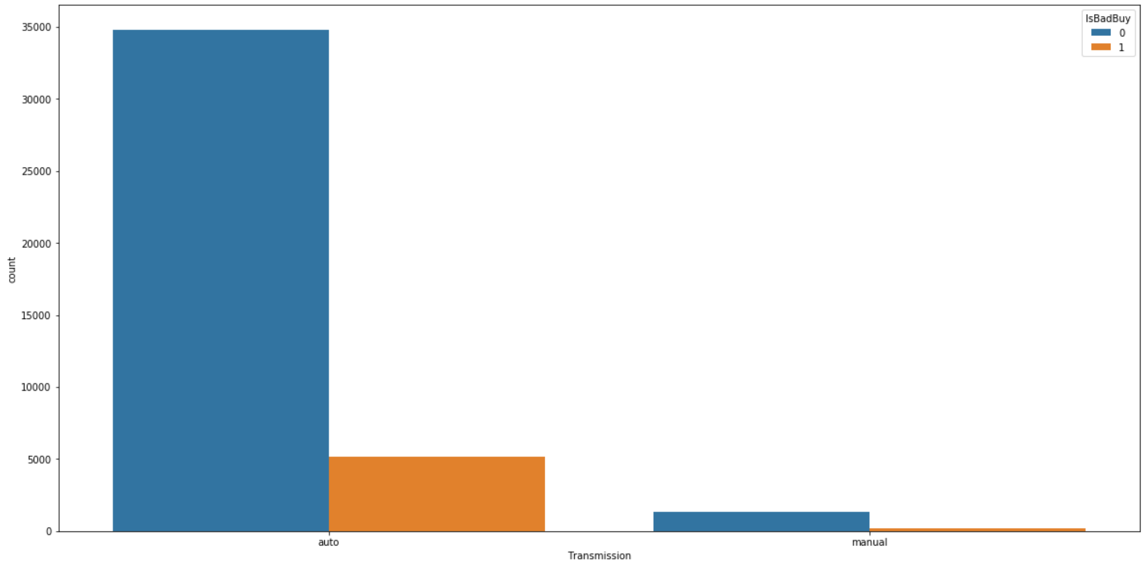
```
plotAllCols(df)
```

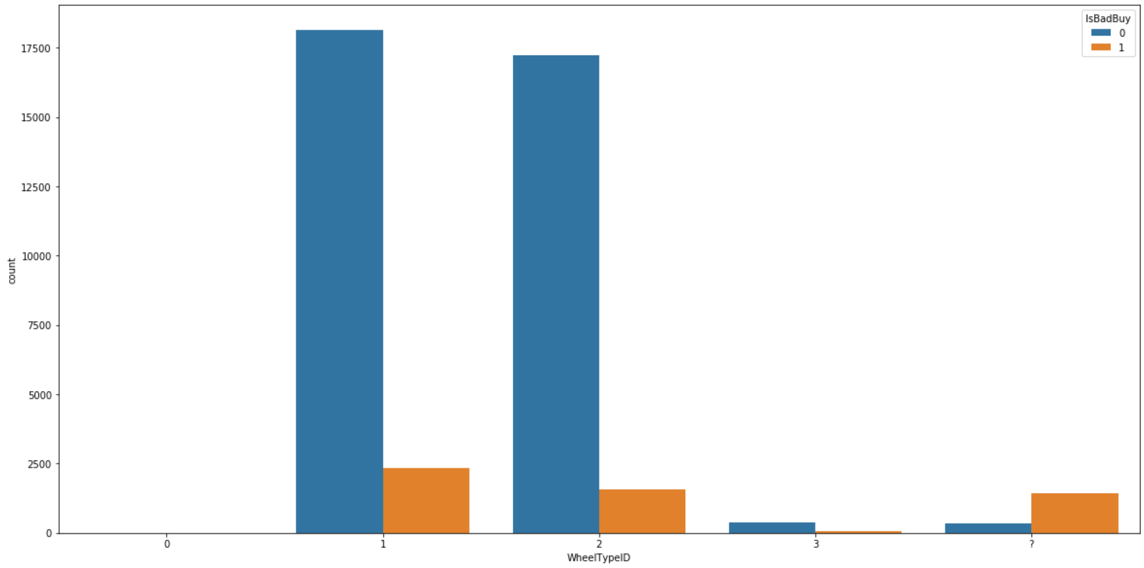

IsBadBuy
0
1
count
20000
17500
15000
12500
10000
7500
5000
2500
0
adesa
manheim
other
Auction

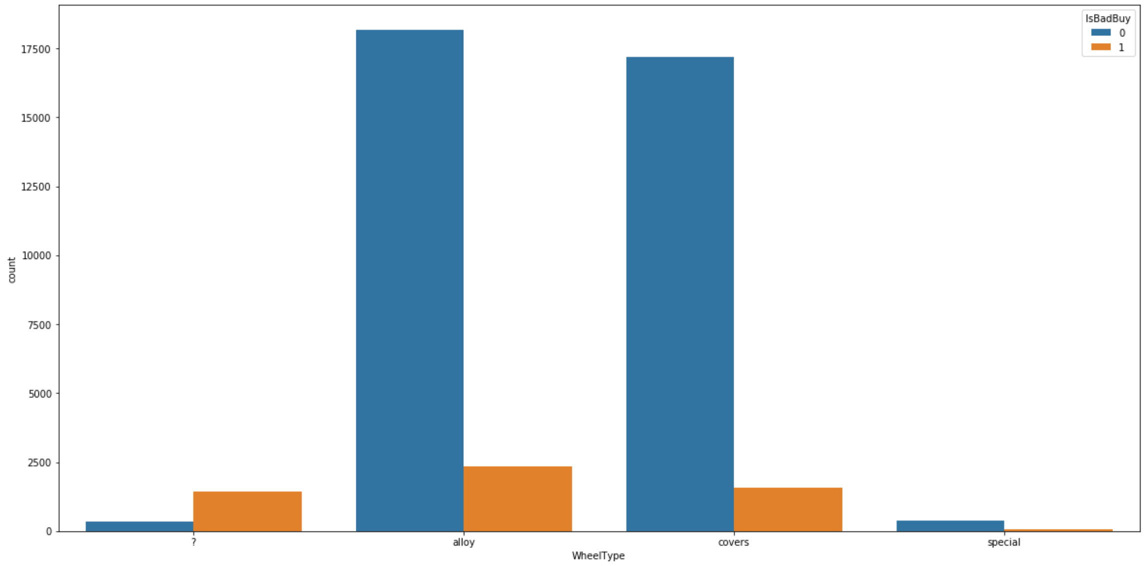

IsBadBuy
0
1
count
8000
6000
4000
2000
0
2001.0
2002.0
2003.0
2004.0
2005.0
2006.0
2007.0
2008.0
2009.0
2010.0
UNKNOWN_VALUE
VehYear

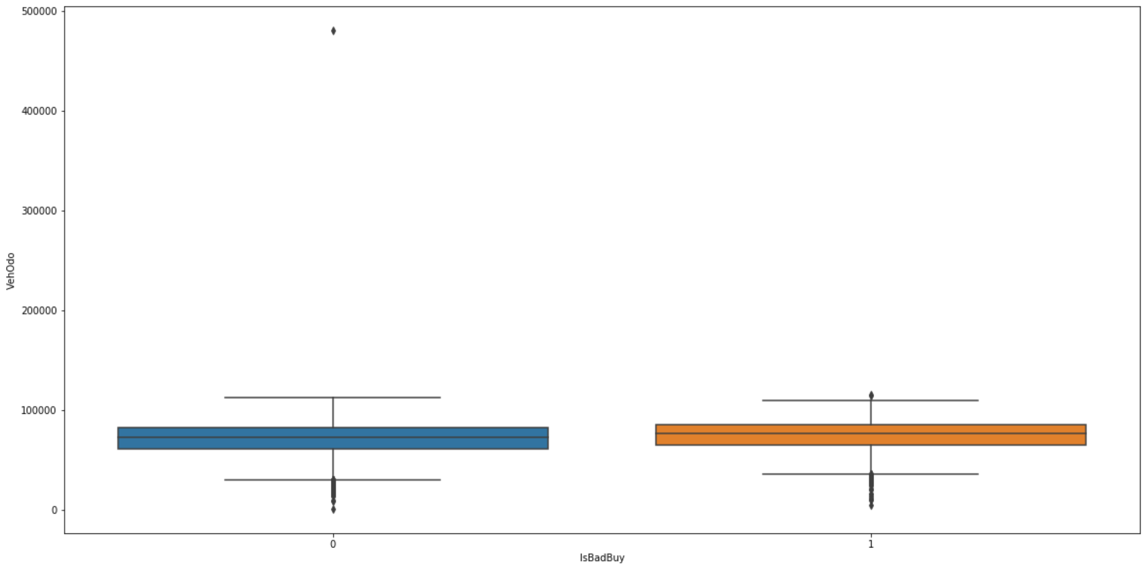

IsBadBuy
0
1
count
Make
acura
buick
cadillac
chevrolet
chrysler
dodge
ford
gmc
honda
hyundai
infiniti
isuzu
jeep
kia
lexus
lincoln
mazda
mercury
mini
mitsubishi
nissan
oldsmobile
pontiac
saturn
scion
subaru
suzuki
toyota
volkswagen
volvo

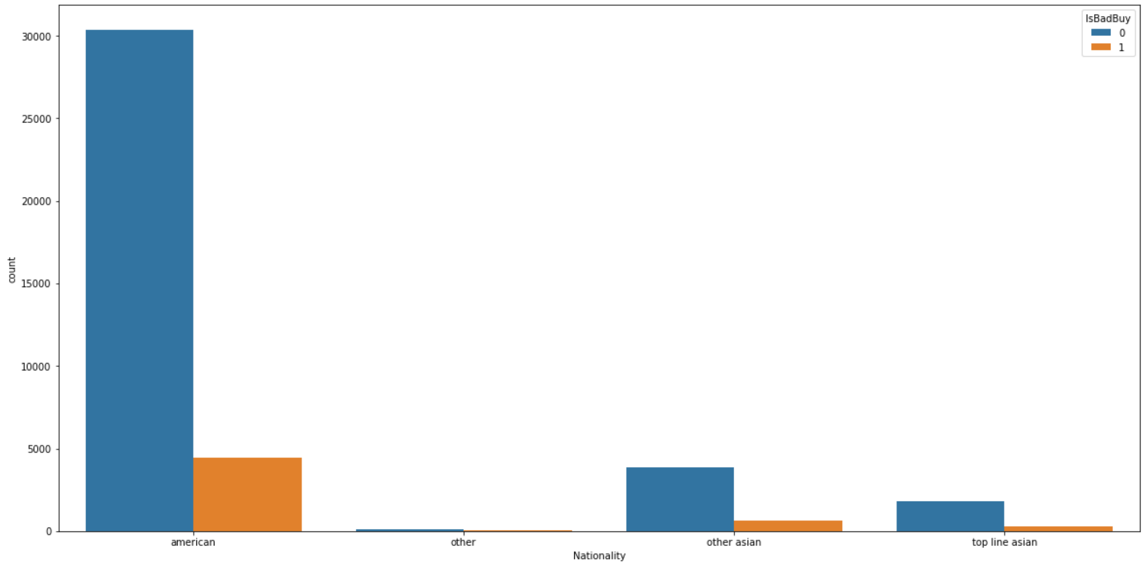

IsBadBuy
0
1
count
Color
beige
black
blue
brown
gold
green
grey
maroon
not avail
orange
other
purple
red
silver
white
yellow

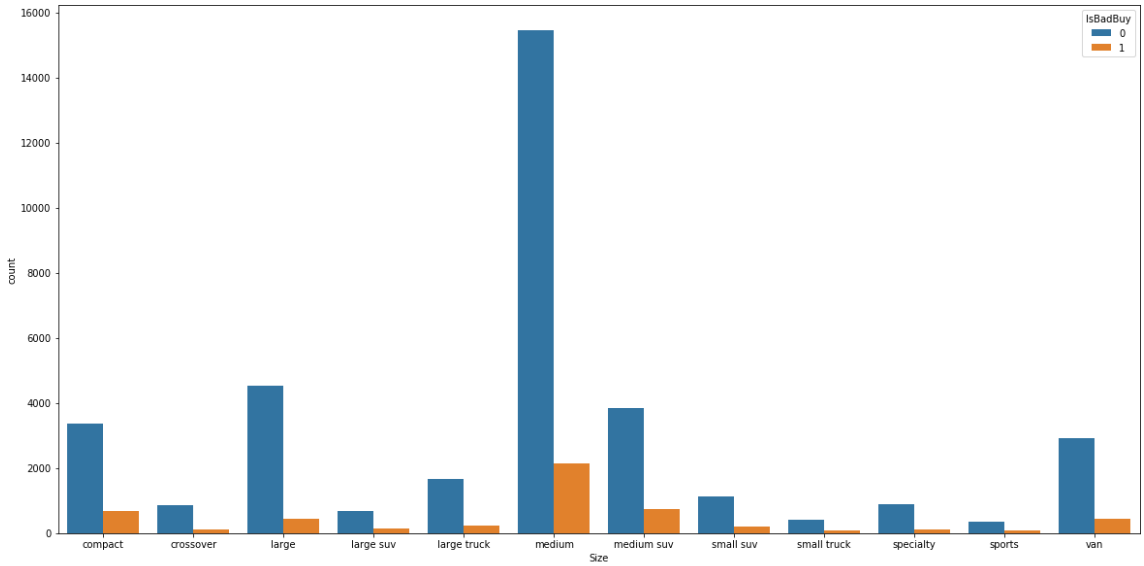

IsBadBuy
0
1
35000
30000
25000
20000
15000
10000
5000
0
count
auto
manual
Transmission

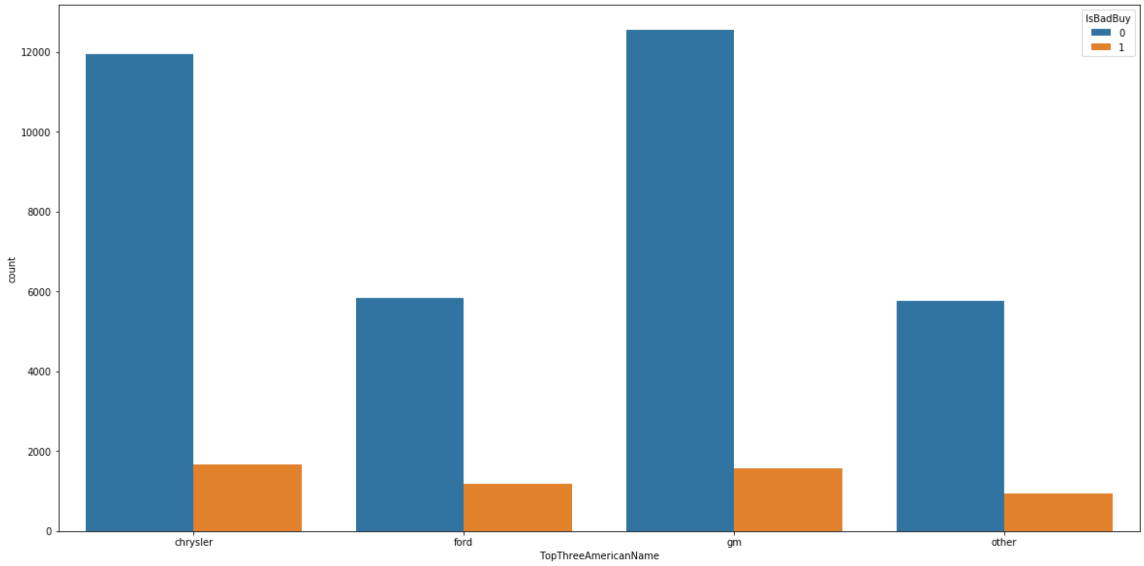

IsBadBuy
0
1
17500
15000
12500
10000
7500
5000
2500
0
count
0
1
2
3
?
WheelTypeID

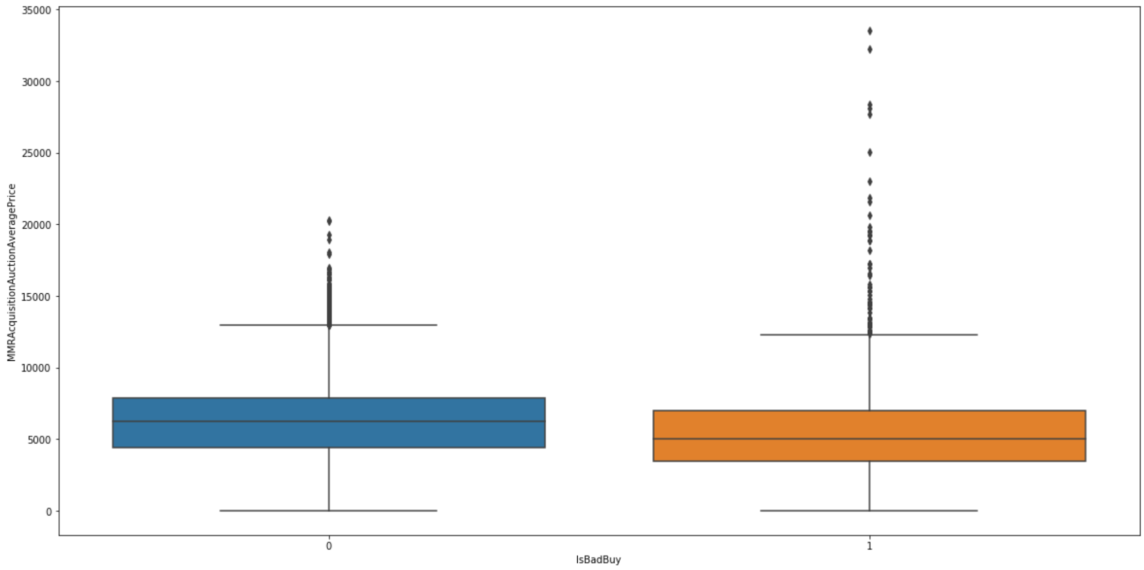

IsBadBuy
0
1
count
17500
15000
12500
10000
7500
5000
2500
0
?
alloy
covers
special
WheelType

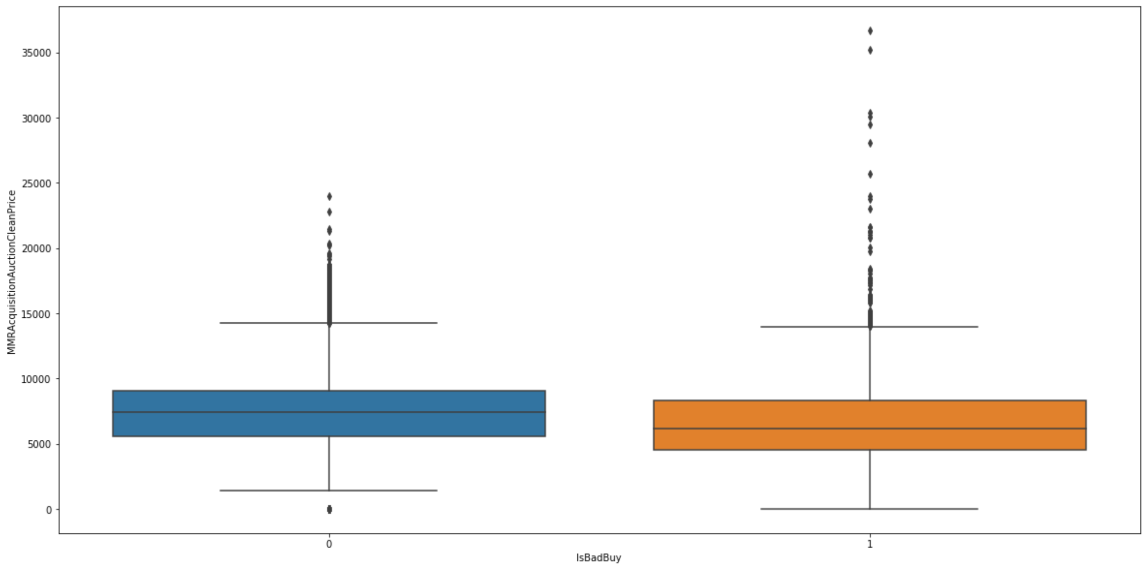

VehOdo
500000
400000
300000
200000
100000
0
0
1
IsBadBuy

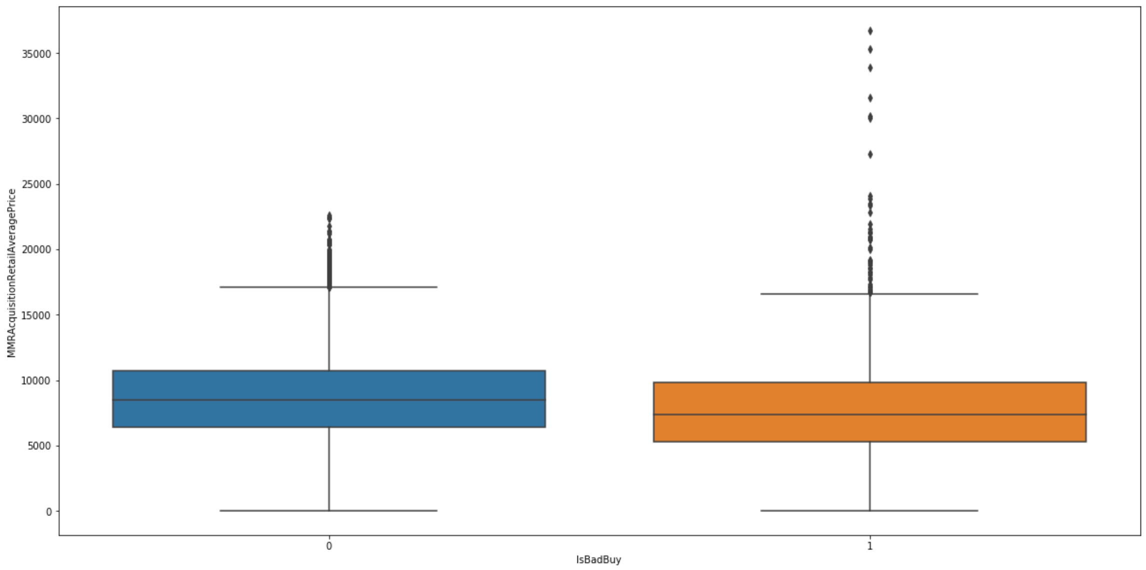

IsBadBuy
0
1
count
30000
25000
20000
15000
10000
5000
0
american
other
other asian
top line asian
Nationality

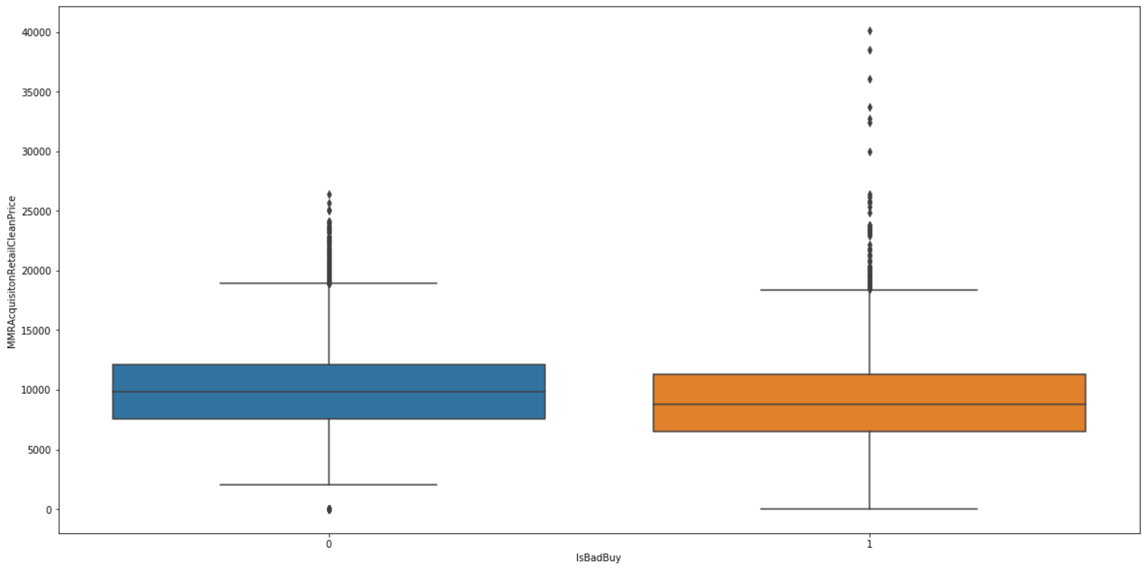

IsBadBuy
0
1
count
16000
14000
12000
10000
8000
6000
4000
2000
0
compact
crossover
large
large suv
large truck
medium
medium suv
small suv
small truck
specialty
sports
van
Size

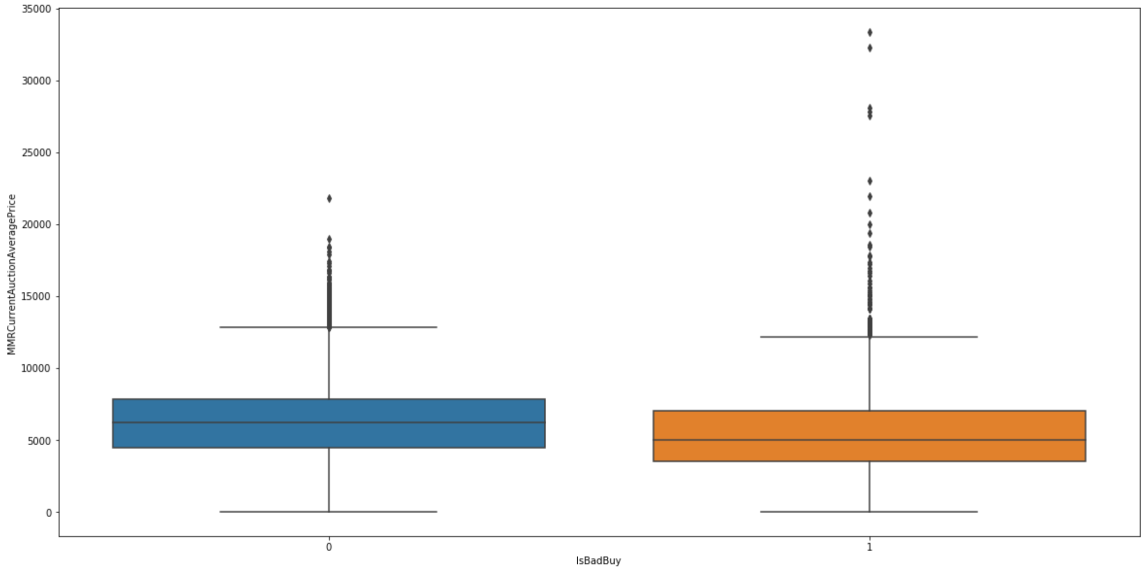

IsBadBuy
0
1
count
12000
10000
8000
6000
4000
2000
0
chrysler
ford
gm
other
TopThreeAmericanName

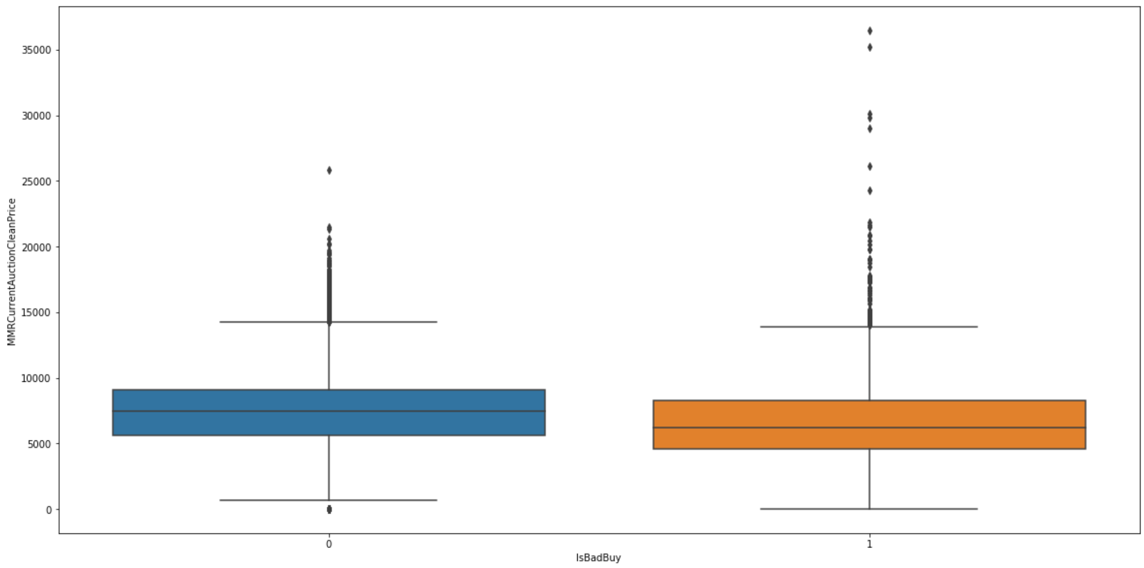

35000
30000
25000
20000
15000
10000
5000
0
MMRAcquisitionAuctionAveragePrice
0
1
IsBadBuy

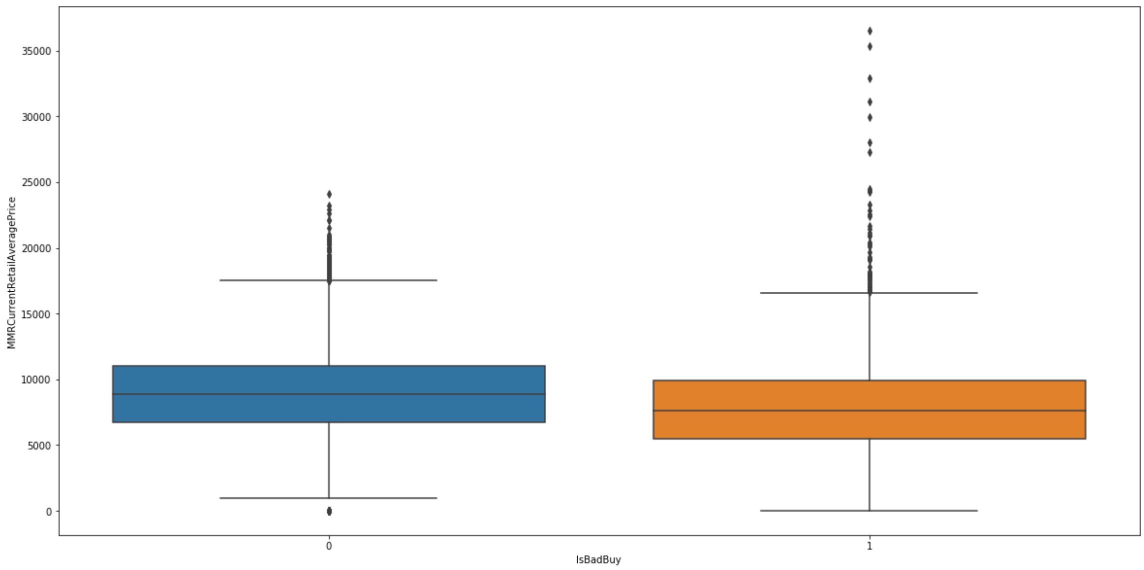

35000
30000
25000
20000
15000
10000
5000
0
MMRAcquisitionAuctionCleanPrice
0
1
IsBadBuy

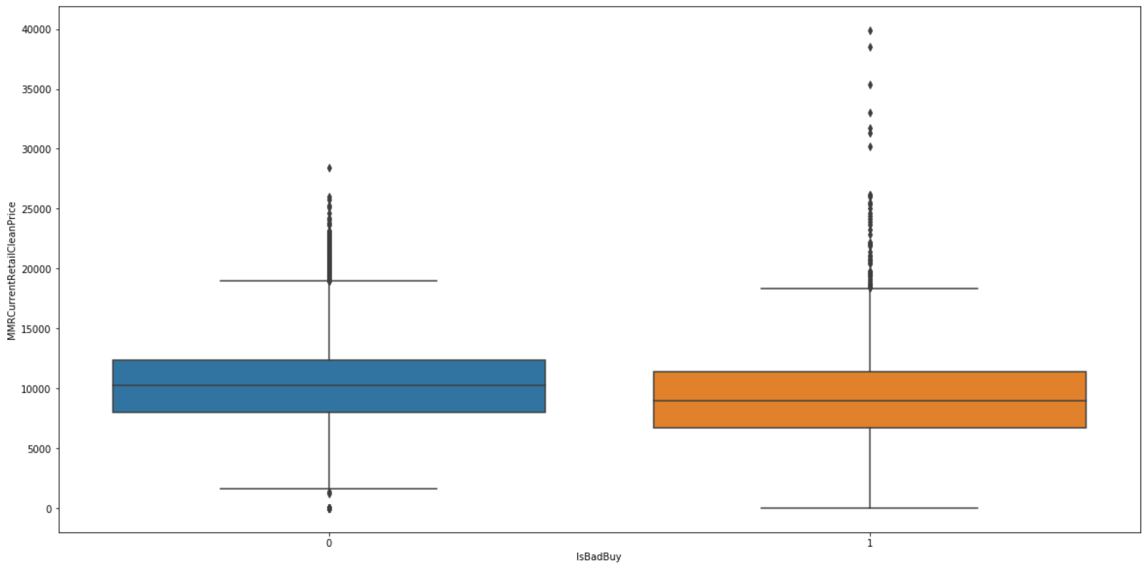

35000
30000
25000
20000
15000
10000
5000
0
MMRAcquisitionRetailAveragePrice
0
1
IsBadBuy


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

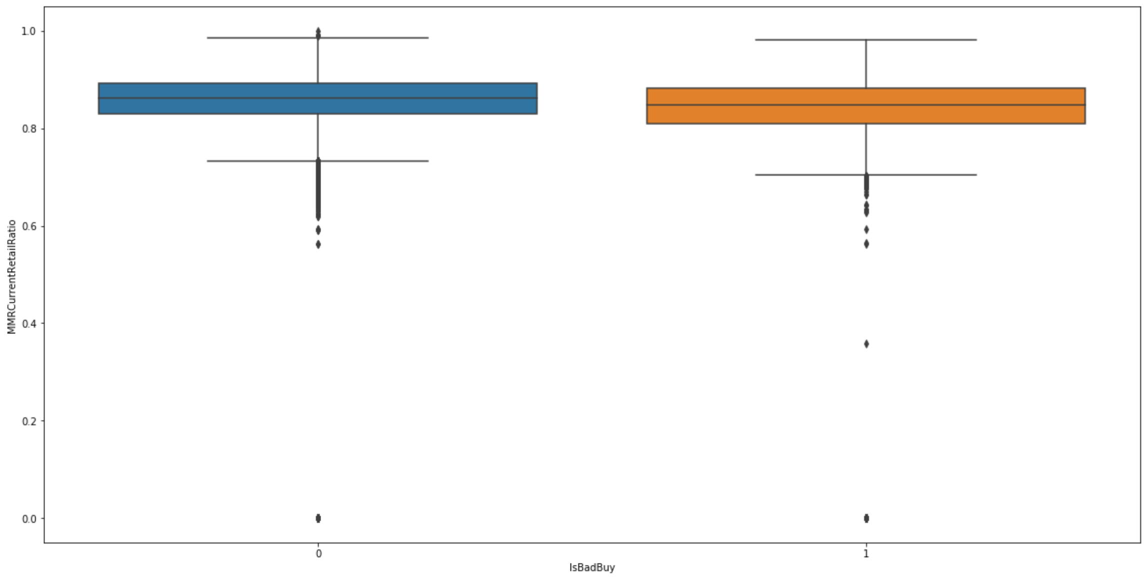

MMRCurrentAuctionAveragePrice
0
5000
10000
15000
20000
25000
30000
35000
0
1
IsBadBuy


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

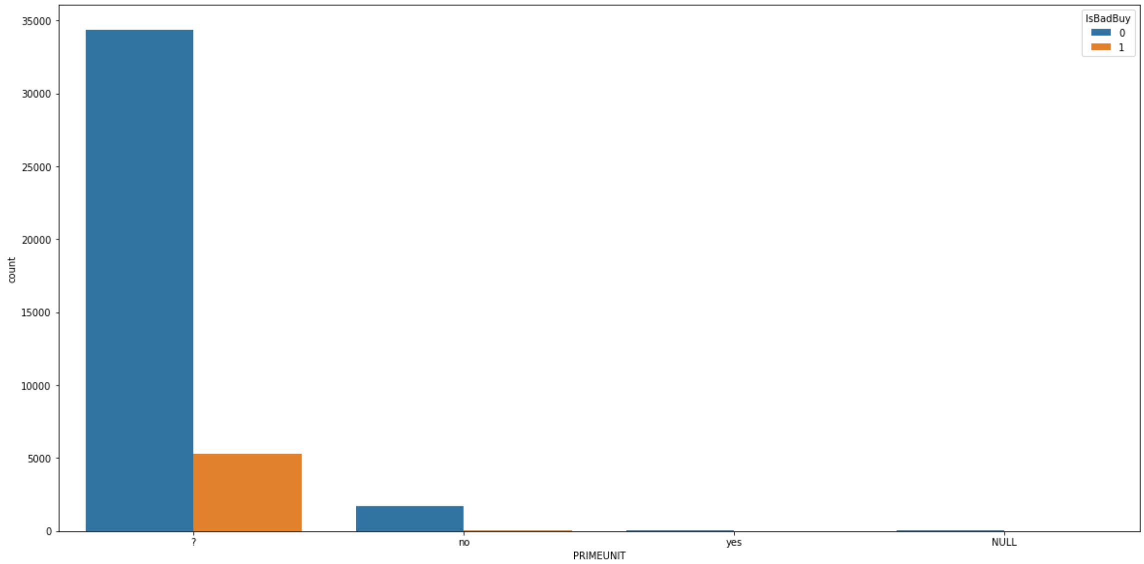

MMRCurrentRetailAveragePrice
35000
30000
25000
20000
15000
10000
5000
0
0
1
IsBadBuy

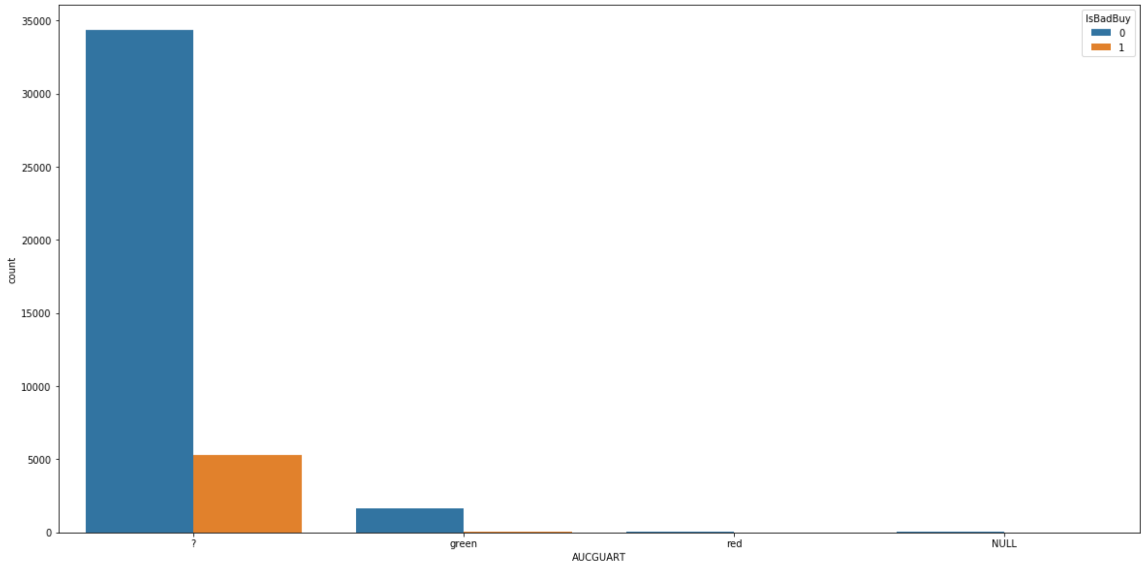


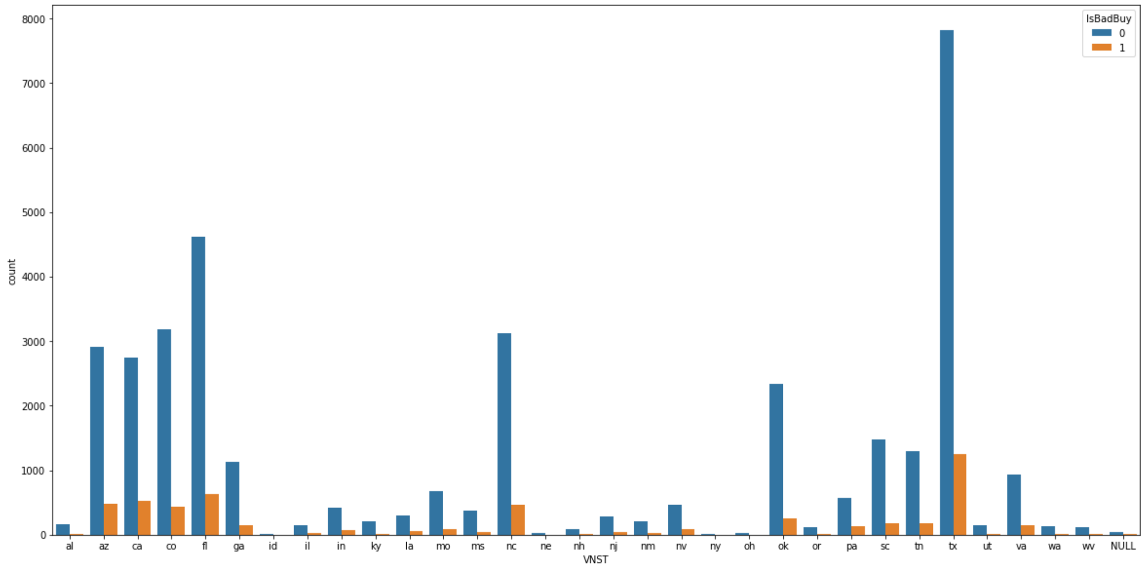
MMRCurrentRetailCleanPrice
IsBadBuy
0
1
40000
35000
30000
25000
20000
15000
10000
5000
0

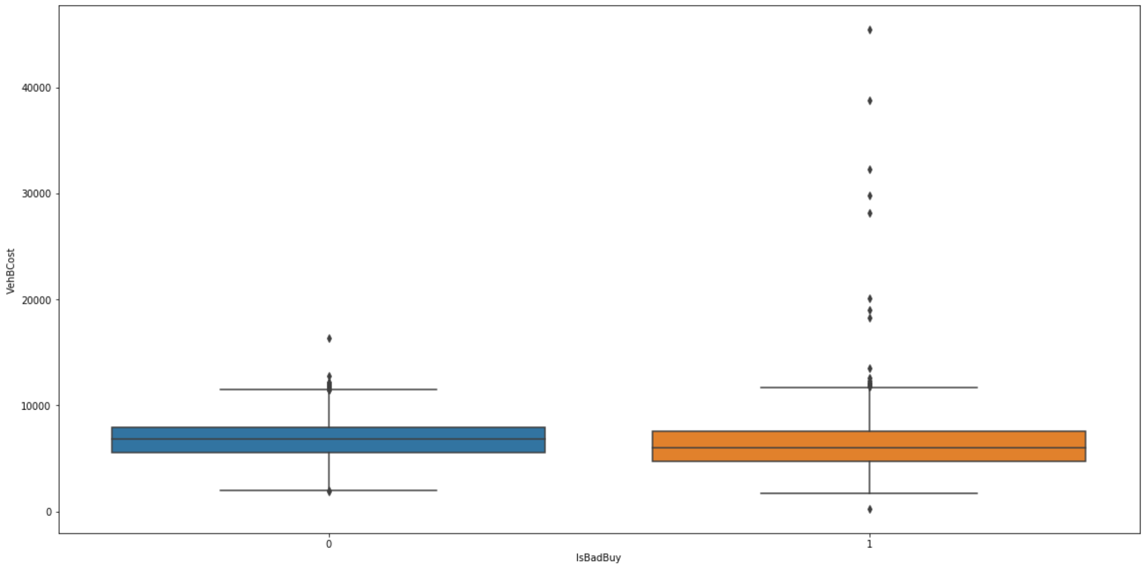



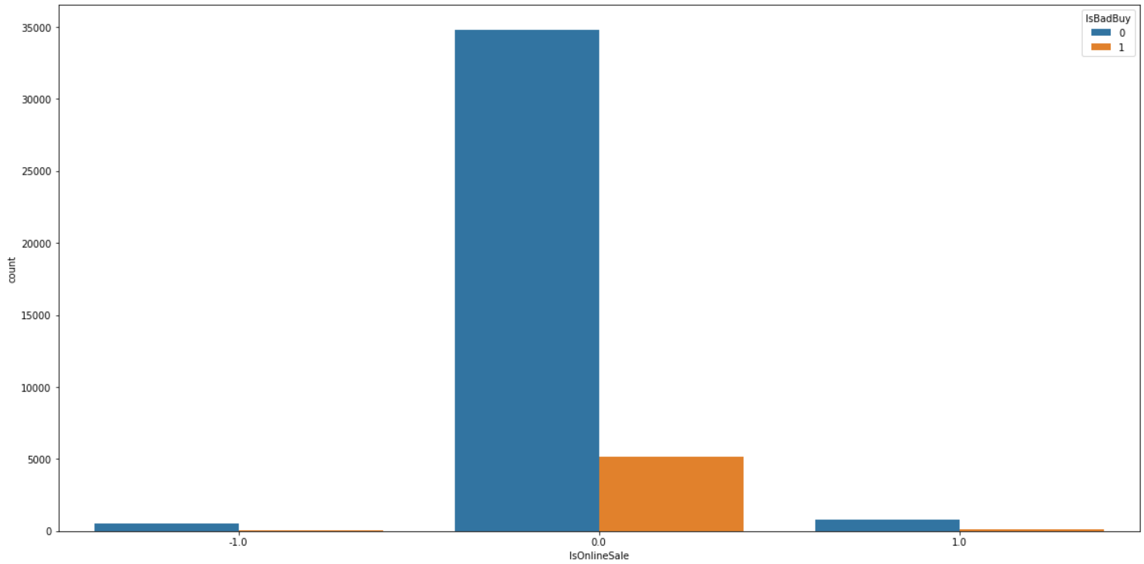
MMRCurrentRetailRatio
IsBadBuy
0
1
1.0
0.8
0.6
0.4
0.2
0.0

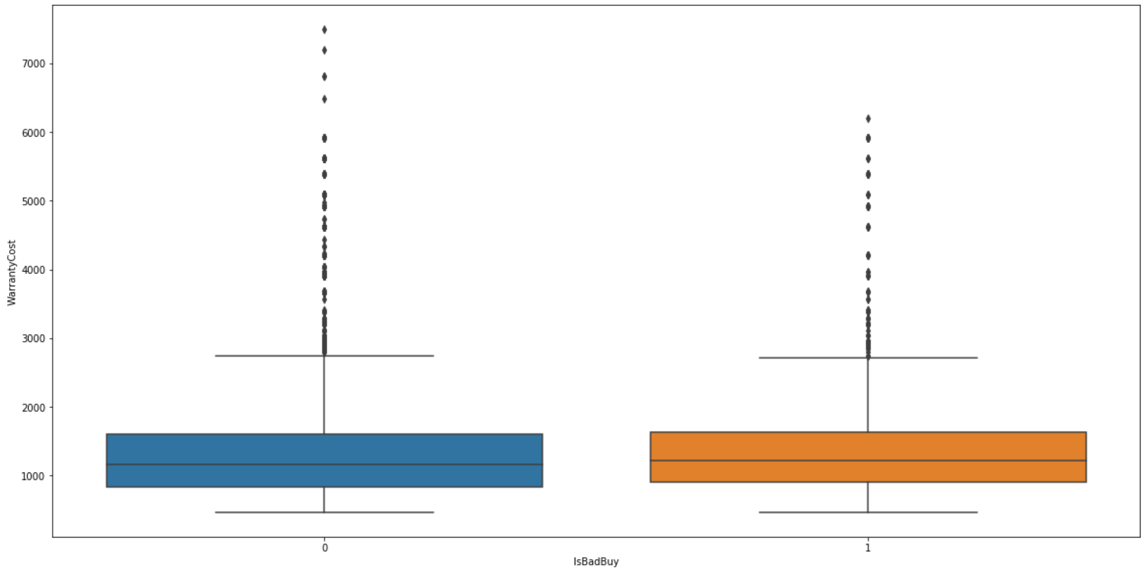

IsBadBuy
0
1
count
35000
30000
25000
20000
15000
10000
5000
0
?
no
yes
NULL
PRIMEUNIT

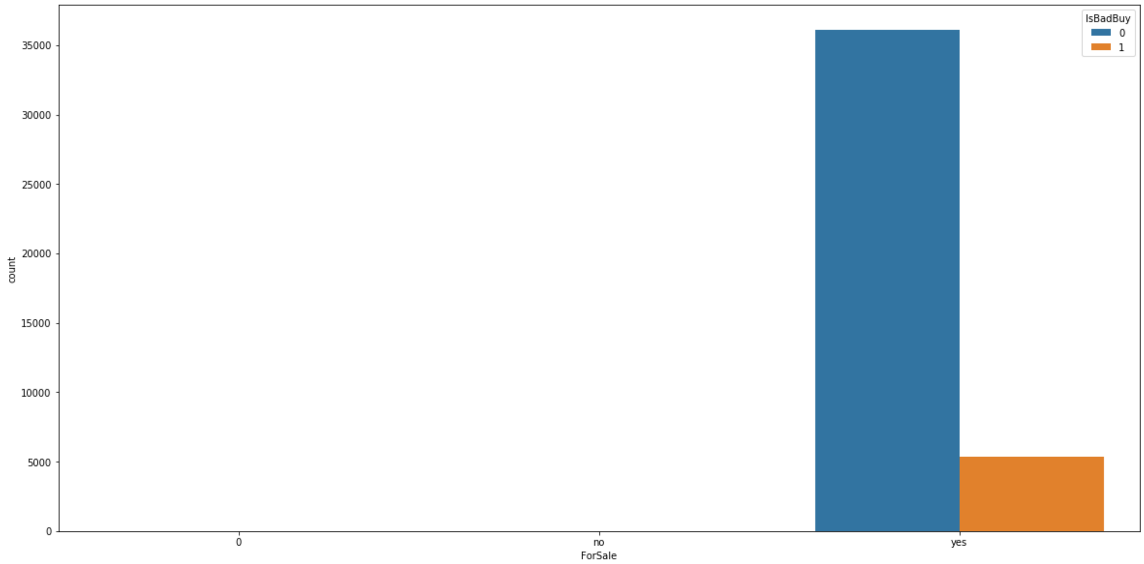

IsBadBuy
0
1
count
35000
30000
25000
20000
15000
10000
5000
0
?
green
red
NULL
AUCGUART


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
0
1


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


7000
6000
5000
4000
3000
2000
1000
WarrantyCost
0
1
IsBadBuy




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

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

In [ ]:

```

```

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

In [11]:

```
# Change to the dummy

df = pd.get_dummies(df)

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

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

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

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

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

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

```
Num of Features: 149
Using ROS Resmapling
```

In [12]:

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

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

# Task 2. Predictive Modeling Using Decision Trees

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

In [13]:

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

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

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

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

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

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

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

In [14]:

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

Out[14]:

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

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

In [15]:

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

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

           0       0.91      0.90      0.90     10832
           1       0.35      0.37      0.36      1611

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

Out[15]:

```
array([[9714, 1118],
       [1014,  597]])
```

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

In [16]:

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

```
Number of nodes:  6703
```

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

In [ ]:

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

In [17]:

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

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

In [18]:

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

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

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

In [ ]:

```

```

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

In [ ]:

```

```

## 2. Python: Build another decision tree tuned with GridSearchCV

In [ ]:

```

```

In [19]:

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

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

Out[19]:

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

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

In [20]:

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

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

dt_model = cv.best_estimator_
```

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

           0       0.90      0.90      0.90     10832
           1       0.32      0.32      0.32      1611

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

{'class_weight': 'balanced', 'criterion': 'gini', 'max_depth': 200,
'max_features': 'log2', 'min_samples_leaf': 1, 'min_samples_split':
2, 'splitter': 'best'}
```

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

In [21]:

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

```
Number of nodes:  13743
```

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

In [ ]:

```

```

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

In [22]:

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

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

In [23]:

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

```
WheelType_? : 0.10196726739090486
VehBCost : 0.07747480575066952
VehOdo : 0.04975026240861232
MMRAcquisitionAuctionCleanPrice : 0.04953950838542224
MMRCurrentAuctionAveragePrice : 0.04898870588447332
```

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

In [ ]:

```

```

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

In [ ]:

```

```

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

In [ ]:

```

```

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

In [ ]:

```

```

In [ ]:

```

```

# Task 3. Predictive Modeling Using Regression

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

In [24]:

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

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

In [25]:

```
## Doing the log transformation


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

def logTransformation(df):

    df_log = df.copy()


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


    return df_log

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


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



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

```
Using ROS Resmapling
```

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

In [26]:

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

Out[26]:

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

In [27]:

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

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

Out[27]:

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

### h. Name the regression function used.

In [ ]:

```

```

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

In [28]:

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

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

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

In [29]:

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

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

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

In [ ]:

```

```

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

In [30]:

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

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

In [31]:

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

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

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

In [32]:

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

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

```
Classification Report:

              precision    recall  f1-score   support

           0       0.93      0.78      0.85     10832
           1       0.29      0.61      0.39      1611

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

GridSearch Classification Report:

              precision    recall  f1-score   support

           0       0.93      0.78      0.85     10832
           1       0.29      0.61      0.39      1611

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

### n. Report any sign of overfitting.

In [33]:

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

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

In [34]:

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

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

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

In [35]:

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

```
Original feature set 149
Number of RFE-selected features:  126
Number of selectFromModel features:  24
```

In [36]:

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

```
The RFE-selected features: 

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



The SelectFromModel features: 

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

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

In [37]:

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

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

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

```
[LibLinear]

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

Out[37]:

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

In [38]:

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

```
Top-5 important variables for RFE: 

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



Top-5 important variables for selectModel 

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

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

In [39]:

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

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

### d. Report any sign of overfitting

In [ ]:

```
```

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

In [40]:

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

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


RFE:

Train accuracy: 0.7000949630039963
Test accuracy: 0.7568914248975327


selectModel:

Train accuracy: 0.6835951410596288
Test accuracy: 0.7648477055372499
```

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

In [41]:

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

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

           0       0.93      0.78      0.85     10832
           1       0.29      0.60      0.39      1611

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




selectModel classification report: 
               precision    recall  f1-score   support

           0       0.92      0.79      0.85     10832
           1       0.29      0.57      0.38      1611

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

# Task4 - Predicting using neural network

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

In [42]:

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

Out[42]:

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

### a. What is the network architecture?

In [43]:

```
def printMLPArchitecture(model):

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

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

printMLPArchitecture(model)
```

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

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

In [44]:

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

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

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

In [45]:

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

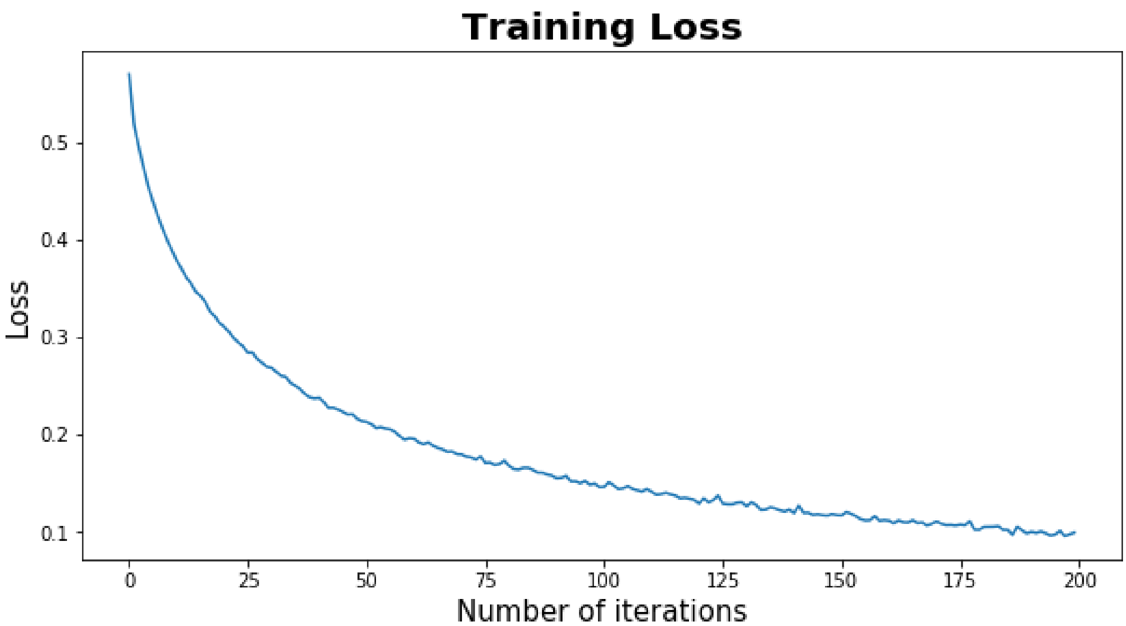
### d. Did the training process converge and resulted in the best model?

In [46]:

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

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

Out[46]:

```
[<matplotlib.lines.Line2D at 0x7f482fe10c18>]
```




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

In [47]:

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

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



MLP classification report: 
               precision    recall  f1-score   support

           0       0.86      0.77      0.81     10832
           1       0.09      0.14      0.11      1611

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

# 2. Refine this network by tuning it with GridSearchCV.

In [ ]:

```

```

In [48]:

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

params = [
    {
        'hidden_layer_sizes': [(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],
    },
]


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 = 0.54693080
Validation score: 0.735312
Iteration 2, loss = 0.47179153
Validation score: 0.777844
Iteration 3, loss = 0.40331981
Validation score: 0.812265
Iteration 4, loss = 0.33787091
Validation score: 0.831454
Iteration 5, loss = 0.29212924
Validation score: 0.867656
Iteration 6, loss = 0.25190921
Validation score: 0.881701
Iteration 7, loss = 0.21855853
Validation score: 0.883680
Iteration 8, loss = 0.19788286
Validation score: 0.902868
Iteration 9, loss = 0.18081237
Validation score: 0.910584
Iteration 10, loss = 0.16247232
Validation score: 0.901879
Iteration 11, loss = 0.15504546
Validation score: 0.905440
Iteration 12, loss = 0.14536334
Validation score: 0.916716
Iteration 13, loss = 0.13576600
Validation score: 0.918101
Iteration 14, loss = 0.13033542
Validation score: 0.916716
Iteration 15, loss = 0.12545532
Validation score: 0.918101
Iteration 16, loss = 0.11629657
Validation score: 0.904055
Iteration 17, loss = 0.11284304
Validation score: 0.921464
Iteration 18, loss = 0.11189941
Validation score: 0.916914
Iteration 19, loss = 0.10325006
Validation score: 0.919881
Iteration 20, loss = 0.10414006
Validation score: 0.922255
Iteration 21, loss = 0.09889908
Validation score: 0.916123
Iteration 22, loss = 0.09608051
Validation score: 0.934125
Iteration 23, loss = 0.09218121
Validation score: 0.921068
Iteration 24, loss = 0.09992786
Validation score: 0.932542
Iteration 25, loss = 0.09121864
Validation score: 0.923046
Iteration 26, loss = 0.09121179
Validation score: 0.927003
Iteration 27, loss = 0.08499365
Validation score: 0.926805
Iteration 28, loss = 0.08393078
Validation score: 0.925618
Iteration 29, loss = 0.08556150
Validation score: 0.926014
Iteration 30, loss = 0.07823876
Validation score: 0.935114
Iteration 31, loss = 0.08434710
```

```
Validation score: 0.932542
Iteration 32, loss = 0.07646444
Validation score: 0.930762
Iteration 33, loss = 0.07609882
Validation score: 0.927596
Iteration 34, loss = 0.07801165
Validation score: 0.936894
Iteration 35, loss = 0.07598078
Validation score: 0.938675
Iteration 36, loss = 0.07703660
Validation score: 0.925025
Iteration 37, loss = 0.07983014
Validation score: 0.931751
Iteration 38, loss = 0.06890338
Validation score: 0.937290
Iteration 39, loss = 0.06798918
Validation score: 0.914738
Iteration 40, loss = 0.07286112
Validation score: 0.929970
Iteration 41, loss = 0.07699081
Validation score: 0.930168
Iteration 42, loss = 0.06542391
Validation score: 0.935509
Iteration 43, loss = 0.06830986
Validation score: 0.941246
Iteration 44, loss = 0.06397221
Validation score: 0.935114
Iteration 45, loss = 0.07137154
Validation score: 0.941048
Iteration 46, loss = 0.06991063
Validation score: 0.935509
Iteration 47, loss = 0.06207058
Validation score: 0.932542
Iteration 48, loss = 0.06402944
Validation score: 0.940455
Iteration 49, loss = 0.05873632
Validation score: 0.939070
Iteration 50, loss = 0.06306456
Validation score: 0.933333
Iteration 51, loss = 0.07488623
Validation score: 0.935509
Iteration 52, loss = 0.06438127
Validation score: 0.934125
Iteration 53, loss = 0.05928741
Validation score: 0.941444
Iteration 54, loss = 0.06207190
Validation score: 0.938081
Iteration 55, loss = 0.06144641
Validation score: 0.920475
Iteration 56, loss = 0.06293869
Validation score: 0.937685
Iteration 57, loss = 0.05972565
Validation score: 0.929179
Iteration 58, loss = 0.05873227
Validation score: 0.932344
Iteration 59, loss = 0.06258876
Validation score: 0.940653
Iteration 60, loss = 0.05570789
Validation score: 0.936696
Iteration 61, loss = 0.05624749
Validation score: 0.940059
```

```
Iteration 62, loss = 0.05877355
Validation score: 0.936696
Iteration 63, loss = 0.06326369
Validation score: 0.939862
Iteration 64, loss = 0.05362035
Validation score: 0.945796
Iteration 65, loss = 0.05962739
Validation score: 0.928981
Iteration 66, loss = 0.05404427
Validation score: 0.944411
Iteration 67, loss = 0.05856537
Validation score: 0.929773
Iteration 68, loss = 0.05812045
Validation score: 0.935509
Iteration 69, loss = 0.05249832
Validation score: 0.937092
Iteration 70, loss = 0.05692958
Validation score: 0.942235
Iteration 71, loss = 0.06169963
Validation score: 0.933927
Iteration 72, loss = 0.05295570
Validation score: 0.948961
Iteration 73, loss = 0.04822423
Validation score: 0.940653
Iteration 74, loss = 0.05481595
Validation score: 0.932542
Iteration 75, loss = 0.05504483
Validation score: 0.938477
Iteration 76, loss = 0.05376706
Validation score: 0.939466
Iteration 77, loss = 0.05802848
Validation score: 0.928783
Iteration 78, loss = 0.05205336
Validation score: 0.942829
Iteration 79, loss = 0.05029906
Validation score: 0.938675
Iteration 80, loss = 0.05840547
Validation score: 0.934125
Iteration 81, loss = 0.05243236
Validation score: 0.943620
Iteration 82, loss = 0.05094091
Validation score: 0.930762
Iteration 83, loss = 0.04746883
Validation score: 0.941444
Validation score did not improve more than tol=0.000100 for 10 conse
cutive epochs. Stopping.
```

Out[48]:

```
GridSearchCV(cv=3, error_score='raise-deprecating',
       estimator=MLPClassifier(activation='relu', alpha=0.0001, batc
h_size='auto', beta_1=0.9,
       beta_2=0.999, early_stopping=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)], '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. What is the network architecture?

In [72]:

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

```
Best Parameters of NN:  {'activation': 'relu', 'alpha': 0.1, 'batch_
size': 64, 'early_stopping': False, 'hidden_layer_sizes': (512, 256,
128, 64, 32), 'learning_rate_init': 0.001, 'max_iter': 200, 'shuffl
e': True, 'solver': 'adam', 'warm_start': True}
GridSearch NN Train accuracy: 0.9638349226447197
GridSearch NN Test accuracy: 0.8021377481314795
```

In [ ]:

In [ ]:

In [ ]:

In [84]:

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

```
Best Parameters of NN:  {'activation': 'relu', 'alpha': 0.001, 'batc
h_size': 64, 'hidden_layer_sizes': (128, 64, 32), 'learning_rate_ini
t': 0.001, 'max_iter': 200, 'shuffle': True, 'solver': 'adam', 'warm
_start': True}
GridSearch NN Train accuracy: 0.9878526490721323
GridSearch NN Test accuracy: 0.8246403600417905
```

In [49]:

```
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 [ ]:

```

```

In [49]:

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

```
Best Parameters of NN:  {'activation': 'relu', 'alpha': 0.0001, 'bat
ch_size': 64, 'early_stopping': True, 'hidden_layer_sizes': (128, 6
4, 32), 'learning_rate_init': 0.001, 'max_iter': 200, 'n_iter_no_cha
nge': 10, 'shuffle': True, 'solver': 'adam', 'warm_start': True}
GridSearch NN Train accuracy: 0.9826494678114984
GridSearch NN Test accuracy: 0.8362131318813791
```

In [50]:

```
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 [51]:

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

```
Number of iterations it ran:  83
```

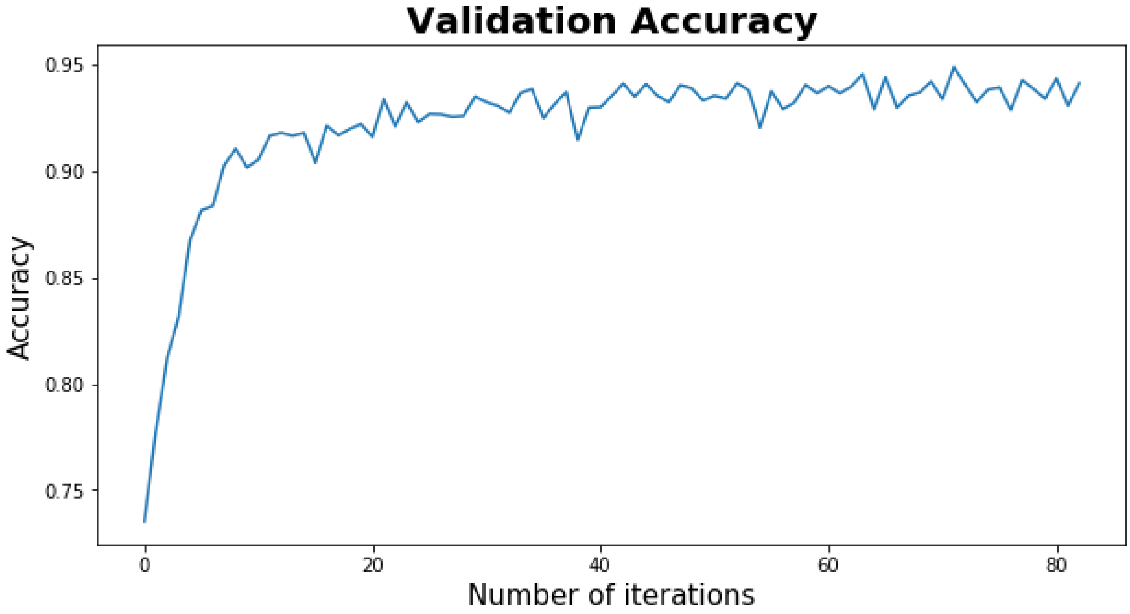
## c. Sign of overfitting?

In [52]:

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

Out[52]:

```
[<matplotlib.lines.Line2D at 0x7f483a049c50>]
```



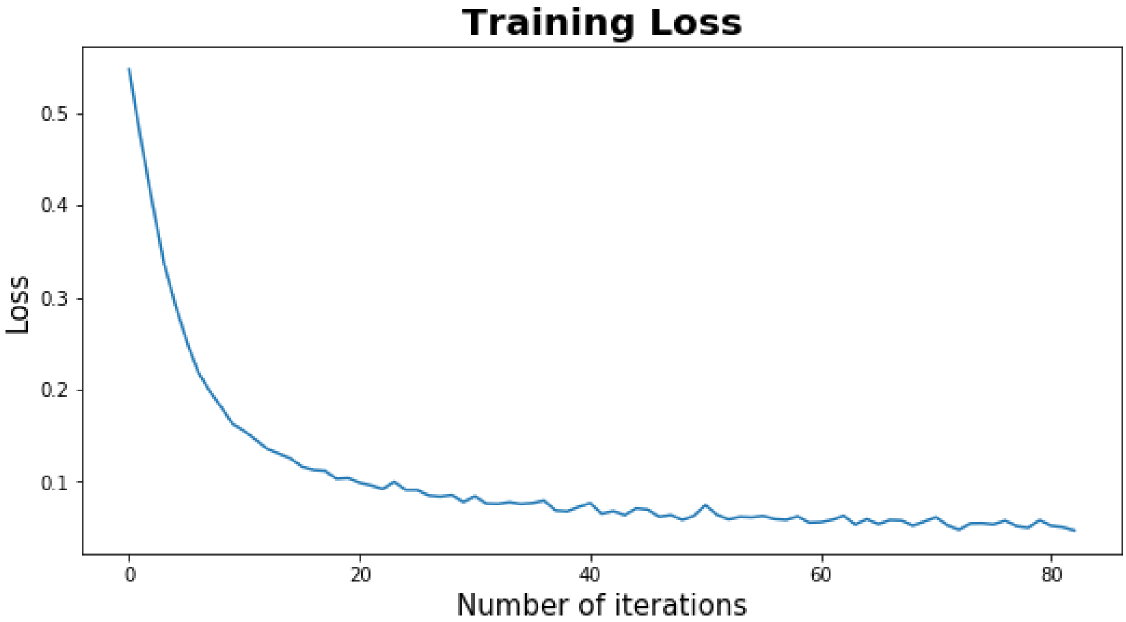

## d. Did the training process converge and resulted in the best model?

In [53]:

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

Out[53]:

```
[<matplotlib.lines.Line2D at 0x7f4870df49e8>]
```




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

In [54]:

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

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

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

```
GridSearch NN Train accuracy: 0.9844893760139279
GridSearch NN Test accuracy: 0.8415976854456321



GridSearch NN Classification Report: 
               precision    recall  f1-score   support

           0       0.90      0.92      0.91     10832
           1       0.37      0.33      0.35      1611

   micro avg       0.84      0.84      0.84     12443
   macro avg       0.64      0.63      0.63     12443
weighted avg       0.83      0.84      0.84     12443

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 [55]:

```
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 = 0.54379255
Validation score: 0.727003
Iteration 2, loss = 0.47608808
Validation score: 0.768150
Iteration 3, loss = 0.42003955
Validation score: 0.799209
Iteration 4, loss = 0.36825116
Validation score: 0.798813
Iteration 5, loss = 0.31963542
Validation score: 0.834224
Iteration 6, loss = 0.27991410
Validation score: 0.857369
Iteration 7, loss = 0.25017291
Validation score: 0.868843
Iteration 8, loss = 0.22515935
Validation score: 0.874777
Iteration 9, loss = 0.20524337
Validation score: 0.881108
Iteration 10, loss = 0.18742855
Validation score: 0.893175
Iteration 11, loss = 0.17456972
Validation score: 0.890999
Iteration 12, loss = 0.16126007
Validation score: 0.902077
Iteration 13, loss = 0.15218531
Validation score: 0.901682
Iteration 14, loss = 0.14374594
Validation score: 0.900495
Iteration 15, loss = 0.13717406
Validation score: 0.908012
Iteration 16, loss = 0.12471034
Validation score: 0.901879
Iteration 17, loss = 0.12454071
Validation score: 0.910188
Iteration 18, loss = 0.11624687
Validation score: 0.921662
Iteration 19, loss = 0.11089894
Validation score: 0.916123
Iteration 20, loss = 0.11077687
Validation score: 0.918497
Iteration 21, loss = 0.10240690
Validation score: 0.917903
Iteration 22, loss = 0.09718684
Validation score: 0.920475
Iteration 23, loss = 0.09371833
Validation score: 0.917705
Iteration 24, loss = 0.09937165
Validation score: 0.913551
Iteration 25, loss = 0.09116725
Validation score: 0.925223
Iteration 26, loss = 0.08686521
Validation score: 0.919486
Iteration 27, loss = 0.08334350
Validation score: 0.922453
Iteration 28, loss = 0.08096201
Validation score: 0.923640
Iteration 29, loss = 0.08178548
Validation score: 0.920870
Iteration 30, loss = 0.07502148
Validation score: 0.922057
Iteration 31, loss = 0.07708381
```

```
Validation score: 0.931949
Iteration 32, loss = 0.08055094
Validation score: 0.917112
Iteration 33, loss = 0.07454518
Validation score: 0.922651
Iteration 34, loss = 0.06402472
Validation score: 0.922651
Iteration 35, loss = 0.07238067
Validation score: 0.926014
Iteration 36, loss = 0.07060910
Validation score: 0.931157
Iteration 37, loss = 0.06786643
Validation score: 0.917507
Iteration 38, loss = 0.06375887
Validation score: 0.929377
Iteration 39, loss = 0.06043171
Validation score: 0.930366
Iteration 40, loss = 0.06331680
Validation score: 0.931355
Iteration 41, loss = 0.07254079
Validation score: 0.923244
Iteration 42, loss = 0.06207245
Validation score: 0.925816
Validation score did not improve more than tol=0.000100 for 10 conse
cutive epochs. Stopping.
Iteration 1, loss = 0.57330803
Validation score: 0.700890
Iteration 2, loss = 0.55231919
Validation score: 0.704055
Iteration 3, loss = 0.53980146
Validation score: 0.713353
Iteration 4, loss = 0.52637715
Validation score: 0.724827
Iteration 5, loss = 0.51236898
Validation score: 0.724827
Iteration 6, loss = 0.49793560
Validation score: 0.737290
Iteration 7, loss = 0.48371777
Validation score: 0.748566
Iteration 8, loss = 0.46784482
Validation score: 0.756677
Iteration 9, loss = 0.45385907
Validation score: 0.753511
Iteration 10, loss = 0.43924093
Validation score: 0.768348
Iteration 11, loss = 0.42754442
Validation score: 0.785559
Iteration 12, loss = 0.41175972
Validation score: 0.793472
Iteration 13, loss = 0.40052821
Validation score: 0.790307
Iteration 14, loss = 0.38851943
Validation score: 0.798220
Iteration 15, loss = 0.37709170
Validation score: 0.796439
Iteration 16, loss = 0.36717006
Validation score: 0.794263
Iteration 17, loss = 0.35841589
Validation score: 0.812463
Iteration 18, loss = 0.34708563
Validation score: 0.813848
```

```
Iteration 19, loss = 0.33810506
Validation score: 0.814243
Iteration 20, loss = 0.32867232
Validation score: 0.821365
Iteration 21, loss = 0.32338271
Validation score: 0.823145
Iteration 22, loss = 0.31209824
Validation score: 0.833630
Iteration 23, loss = 0.30350092
Validation score: 0.825915
Iteration 24, loss = 0.29766153
Validation score: 0.837389
Iteration 25, loss = 0.29023196
Validation score: 0.843521
Iteration 26, loss = 0.28204334
Validation score: 0.836597
Iteration 27, loss = 0.28335248
Validation score: 0.848863
Iteration 28, loss = 0.27437432
Validation score: 0.850247
Iteration 29, loss = 0.27288003
Validation score: 0.850445
Iteration 30, loss = 0.26339030
Validation score: 0.864688
Iteration 31, loss = 0.25732065
Validation score: 0.853412
Iteration 32, loss = 0.25328139
Validation score: 0.858952
Iteration 33, loss = 0.24786158
Validation score: 0.856380
Iteration 34, loss = 0.24501986
Validation score: 0.865084
Iteration 35, loss = 0.23920381
Validation score: 0.858952
Iteration 36, loss = 0.23644814
Validation score: 0.858160
Iteration 37, loss = 0.23185303
Validation score: 0.869634
Iteration 38, loss = 0.22630183
Validation score: 0.869238
Iteration 39, loss = 0.22163310
Validation score: 0.871019
Iteration 40, loss = 0.21871218
Validation score: 0.872404
Iteration 41, loss = 0.21845417
Validation score: 0.873393
Iteration 42, loss = 0.21218798
Validation score: 0.874184
Iteration 43, loss = 0.20696662
Validation score: 0.875964
Iteration 44, loss = 0.20450217
Validation score: 0.876954
Iteration 45, loss = 0.20609946
Validation score: 0.874382
Iteration 46, loss = 0.19943176
Validation score: 0.874580
Iteration 47, loss = 0.19374065
Validation score: 0.885658
Iteration 48, loss = 0.19283788
Validation score: 0.876954
Iteration 49, loss = 0.19071235
```

```
Validation score: 0.878734
Iteration 50, loss = 0.18617896
Validation score: 0.883877
Iteration 51, loss = 0.18739931
Validation score: 0.877547
Iteration 52, loss = 0.18388099
Validation score: 0.886449
Iteration 53, loss = 0.17887319
Validation score: 0.892582
Iteration 54, loss = 0.17510454
Validation score: 0.886053
Iteration 55, loss = 0.17872255
Validation score: 0.883877
Iteration 56, loss = 0.17457621
Validation score: 0.888625
Iteration 57, loss = 0.17125972
Validation score: 0.885658
Iteration 58, loss = 0.16580691
Validation score: 0.885658
Iteration 59, loss = 0.16687167
Validation score: 0.896934
Iteration 60, loss = 0.16285759
Validation score: 0.895351
Iteration 61, loss = 0.16409401
Validation score: 0.893966
Iteration 62, loss = 0.16133912
Validation score: 0.889416
Iteration 63, loss = 0.15426549
Validation score: 0.894955
Iteration 64, loss = 0.15494800
Validation score: 0.883877
Iteration 65, loss = 0.15468493
Validation score: 0.903660
Iteration 66, loss = 0.15522799
Validation score: 0.893769
Iteration 67, loss = 0.15022172
Validation score: 0.892582
Iteration 68, loss = 0.14927851
Validation score: 0.901286
Iteration 69, loss = 0.14947810
Validation score: 0.896142
Iteration 70, loss = 0.14900039
Validation score: 0.893966
Iteration 71, loss = 0.13982624
Validation score: 0.903066
Iteration 72, loss = 0.14389187
Validation score: 0.900890
Iteration 73, loss = 0.14128673
Validation score: 0.904055
Iteration 74, loss = 0.14085949
Validation score: 0.896538
Iteration 75, loss = 0.13723825
Validation score: 0.903858
Iteration 76, loss = 0.13505717
Validation score: 0.904055
Iteration 77, loss = 0.13541222
Validation score: 0.901484
Iteration 78, loss = 0.13465241
Validation score: 0.893769
Iteration 79, loss = 0.13992227
Validation score: 0.901088
```

```
Iteration 80, loss = 0.12931261
Validation score: 0.903066
Iteration 81, loss = 0.12465855
Validation score: 0.909199
Iteration 82, loss = 0.12909149
Validation score: 0.911375
Iteration 83, loss = 0.13236172
Validation score: 0.909594
Iteration 84, loss = 0.12432258
Validation score: 0.908803
Iteration 85, loss = 0.12641086
Validation score: 0.910979
Iteration 86, loss = 0.12442374
Validation score: 0.913947
Iteration 87, loss = 0.12835073
Validation score: 0.913947
Iteration 88, loss = 0.12301191
Validation score: 0.915134
Iteration 89, loss = 0.12069449
Validation score: 0.903462
Iteration 90, loss = 0.11777278
Validation score: 0.903462
Iteration 91, loss = 0.12404978
Validation score: 0.916320
Iteration 92, loss = 0.11887503
Validation score: 0.914342
Iteration 93, loss = 0.11366074
Validation score: 0.899505
Iteration 94, loss = 0.12110635
Validation score: 0.903066
Iteration 95, loss = 0.11552910
Validation score: 0.910781
Iteration 96, loss = 0.11419314
Validation score: 0.911968
Iteration 97, loss = 0.12256008
Validation score: 0.905242
Iteration 98, loss = 0.11431645
Validation score: 0.912760
Iteration 99, loss = 0.11201425
Validation score: 0.915529
Iteration 100, loss = 0.10848371
Validation score: 0.909397
Iteration 101, loss = 0.10614389
Validation score: 0.904649
Iteration 102, loss = 0.11164762
Validation score: 0.916914
Iteration 103, loss = 0.11209671
Validation score: 0.905836
Iteration 104, loss = 0.10906200
Validation score: 0.910188
Iteration 105, loss = 0.10476748
Validation score: 0.909990
Iteration 106, loss = 0.11122429
Validation score: 0.909792
Iteration 107, loss = 0.10502549
Validation score: 0.919288
Iteration 108, loss = 0.10180094
Validation score: 0.918892
Iteration 109, loss = 0.10136023
Validation score: 0.908012
Iteration 110, loss = 0.10664119
```

```
Validation score: 0.917507
Iteration 111, loss = 0.10293620
Validation score: 0.916320
Iteration 112, loss = 0.11947076
Validation score: 0.894164
Iteration 113, loss = 0.11190286
Validation score: 0.921860
Iteration 114, loss = 0.09398701
Validation score: 0.919881
Iteration 115, loss = 0.09901050
Validation score: 0.906825
Iteration 116, loss = 0.10237304
Validation score: 0.916914
Iteration 117, loss = 0.10077175
Validation score: 0.916518
Iteration 118, loss = 0.10207948
Validation score: 0.907221
Iteration 119, loss = 0.10196449
Validation score: 0.917903
Iteration 120, loss = 0.09959117
Validation score: 0.917310
Iteration 121, loss = 0.09460524
Validation score: 0.926014
Iteration 122, loss = 0.09721610
Validation score: 0.917310
Iteration 123, loss = 0.09614555
Validation score: 0.919683
Iteration 124, loss = 0.09664914
Validation score: 0.915331
Iteration 125, loss = 0.09612944
Validation score: 0.910386
Iteration 126, loss = 0.09315450
Validation score: 0.915727
Iteration 127, loss = 0.09338246
Validation score: 0.909397
Iteration 128, loss = 0.09998416
Validation score: 0.920673
Iteration 129, loss = 0.08686188
Validation score: 0.903660
Iteration 130, loss = 0.09772999
Validation score: 0.920673
Iteration 131, loss = 0.09810758
Validation score: 0.922255
Iteration 132, loss = 0.08309455
Validation score: 0.924036
Validation score did not improve more than tol=0.000100 for 10 conse
cutive epochs. Stopping.
```

Out[55]:

```
GridSearchCV(cv=3, error_score='raise-deprecating',
       estimator=MLPClassifier(activation='relu', alpha=0.0001, batc
h_size='auto', beta_1=0.9,
       beta_2=0.999, early_stopping=True, epsilon=1e-08,
       hidden_layer_sizes=(100,), learning_rate='constant',
       learning_rate_init=0.001, max_iter=200, momentum=0.9,
       n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
       random_state=101, shuffle=True, solver='adam', tol=0.0001,
       validation_fraction=0.1, verbose=True, warm_start=False),
       fit_params=None, iid='warn', n_jobs=-1,
       param_grid=[{'hidden_layer_sizes': [(128,), (128, 64, 32)],
'activation': ['logistic', 'relu'], 'solver': ['adam'], 'batch_siz
e': [64], 'shuffle': [True], 'learning_rate_init': [0.001], 'n_iter_
no_change': [10], 'max_iter': [200], 'warm_start': [True], 'early_st
opping': [True], 'alpha': [0.1, 0.0001...er_no_change': [10], 'warm_
start': [True], 'early_stopping': [True], 'alpha': [0.1, 0.0001, 1e-
06]}],
       pre_dispatch='2*n_jobs', refit=True, return_train_score='war
n',
       scoring=None, verbose=0)
```

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

In [56]:

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

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

```
Best Parameters of NN:  {'activation': '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': 1e-06,
'batch_size': 64, 'early_stopping': True, 'hidden_layer_sizes': (12
8, 64, 32), 'learning_rate_init': 0.001, 'max_iter': 200, 'n_iter_no
_change': 10, 'shuffle': True, 'solver': 'adam', 'warm_start': True}
Best Parameters of modelSelect NN:  {'activation': 'relu', 'alpha':
0.0001, 'batch_size': 64, 'early_stopping': True, 'hidden_layer_size
s': (128, 64, 32), 'learning_rate_init': 0.001, 'max_iter': 200, 'n_
iter_no_change': 10, 'shuffle': True, 'solver': 'adam', 'warm_star
t': 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:  5
The First layer is Input Layer, and the last layer is the output lay
er
1 Layer with hidden size 126
2 Layer with hidden size 128
3 Layer with hidden size 64
4 Layer with hidden size 32
5 Layer with hidden size 1
The activation function:  relu


modelSelect:
Number of Layers:  5
The First layer is Input Layer, and the last layer is the output lay
er
1 Layer with hidden size 24
2 Layer with hidden size 128
3 Layer with hidden size 64
4 Layer with hidden size 32
5 Layer with hidden size 1
The activation function:  relu
```

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

In [57]:

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

```
GridSearch NN Train accuracy: 0.9844893760139279
GridSearch NN Test accuracy: 0.8415976854456321
RFE NN Train accuracy: 0.9759624896134215
RFE NNTest accuracy: 0.8274531865305794
modelSelect NN Train accuracy: 0.9666640288054446
modelSelect NN Test accuracmodelSelect_cvy: 0.8104958611267379
```

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

In [58]:

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

```
Number of iterations GS ran:  83
Number of iterations rfe ran:  42
Number of iterations modelSelect ran:  132
```

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

In [ ]:

```

```

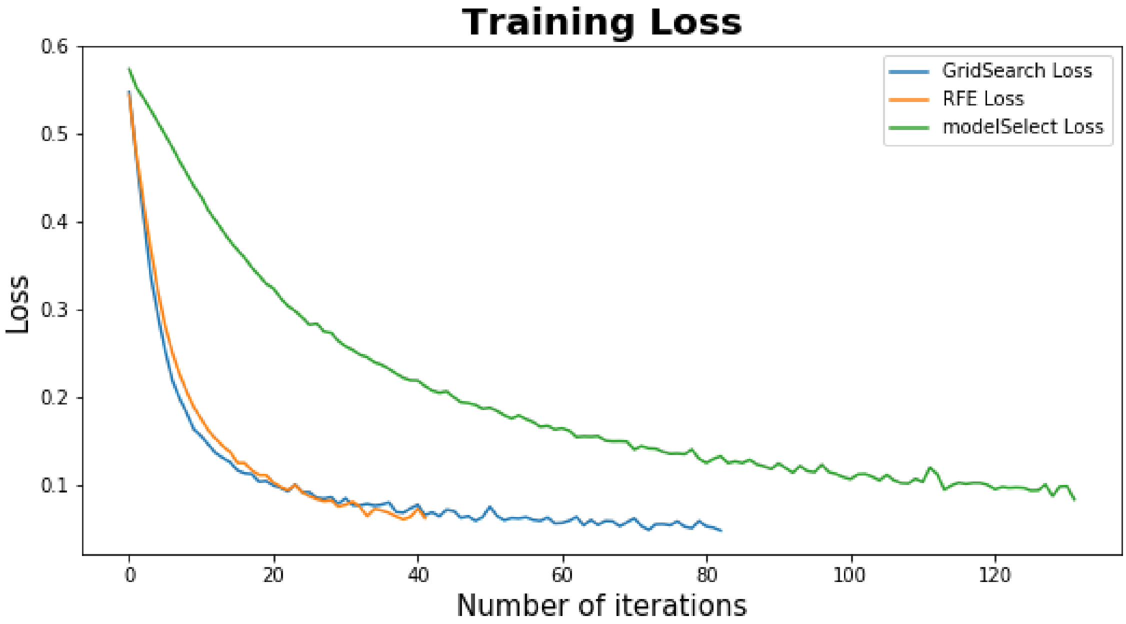
### e. Did the training process converge and resulted in the best model?

In [59]:

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

Out[59]:

<matplotlib.legend.Legend at 0x7f483786e390>




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

In [60]:

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

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

           0       0.90      0.92      0.91     10832
           1       0.37      0.33      0.35      1611

   micro avg       0.84      0.84      0.84     12443
   macro avg       0.64      0.63      0.63     12443
weighted avg       0.83      0.84      0.84     12443



RFE Classification Report: 
              precision    recall  f1-score   support

           0       0.90      0.90      0.90     10832
           1       0.34      0.35      0.35      1611

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



modelSelect Classification Report: 
              precision    recall  f1-score   support

           0       0.90      0.88      0.89     10832
           1       0.30      0.35      0.33      1611

   micro avg       0.81      0.81      0.81     12443
   macro avg       0.60      0.62      0.61     12443
weighted avg       0.82      0.81      0.82     12443
```

# Task 5. Generating an Ensemble Model and Comparing Models

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

In [61]:

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

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

```
Iteration 1, loss = 0.54693080
Validation score: 0.735312
Iteration 2, loss = 0.47179153
Validation score: 0.777844
Iteration 3, loss = 0.40331981
Validation score: 0.812265
Iteration 4, loss = 0.33787091
Validation score: 0.831454
Iteration 5, loss = 0.29212924
Validation score: 0.867656
Iteration 6, loss = 0.25190921
Validation score: 0.881701
Iteration 7, loss = 0.21855853
Validation score: 0.883680
Iteration 8, loss = 0.19788286
Validation score: 0.902868
Iteration 9, loss = 0.18081237
Validation score: 0.910584
Iteration 10, loss = 0.16247232
Validation score: 0.901879
Iteration 11, loss = 0.15504546
Validation score: 0.905440
Iteration 12, loss = 0.14536334
Validation score: 0.916716
Iteration 13, loss = 0.13576600
Validation score: 0.918101
Iteration 14, loss = 0.13033542
Validation score: 0.916716
Iteration 15, loss = 0.12545532
Validation score: 0.918101
Iteration 16, loss = 0.11629657
Validation score: 0.904055
Iteration 17, loss = 0.11284304
Validation score: 0.921464
Iteration 18, loss = 0.11189941
Validation score: 0.916914
Iteration 19, loss = 0.10325006
Validation score: 0.919881
Iteration 20, loss = 0.10414006
Validation score: 0.922255
Iteration 21, loss = 0.09889908
Validation score: 0.916123
Iteration 22, loss = 0.09608051
Validation score: 0.934125
Iteration 23, loss = 0.09218121
Validation score: 0.921068
Iteration 24, loss = 0.09992786
Validation score: 0.932542
Iteration 25, loss = 0.09121864
Validation score: 0.923046
Iteration 26, loss = 0.09121179
Validation score: 0.927003
Iteration 27, loss = 0.08499365
Validation score: 0.926805
Iteration 28, loss = 0.08393078
Validation score: 0.925618
Iteration 29, loss = 0.08556150
Validation score: 0.926014
Iteration 30, loss = 0.07823876
Validation score: 0.935114
Iteration 31, loss = 0.08434710
```

```
Validation score: 0.932542
Iteration 32, loss = 0.07646444
Validation score: 0.930762
Iteration 33, loss = 0.07609882
Validation score: 0.927596
Iteration 34, loss = 0.07801165
Validation score: 0.936894
Iteration 35, loss = 0.07598078
Validation score: 0.938675
Iteration 36, loss = 0.07703660
Validation score: 0.925025
Iteration 37, loss = 0.07983014
Validation score: 0.931751
Iteration 38, loss = 0.06890338
Validation score: 0.937290
Iteration 39, loss = 0.06798918
Validation score: 0.914738
Iteration 40, loss = 0.07286112
Validation score: 0.929970
Iteration 41, loss = 0.07699081
Validation score: 0.930168
Iteration 42, loss = 0.06542391
Validation score: 0.935509
Iteration 43, loss = 0.06830986
Validation score: 0.941246
Iteration 44, loss = 0.06397221
Validation score: 0.935114
Iteration 45, loss = 0.07137154
Validation score: 0.941048
Iteration 46, loss = 0.06991063
Validation score: 0.935509
Iteration 47, loss = 0.06207058
Validation score: 0.932542
Iteration 48, loss = 0.06402944
Validation score: 0.940455
Iteration 49, loss = 0.05873632
Validation score: 0.939070
Iteration 50, loss = 0.06306456
Validation score: 0.933333
Iteration 51, loss = 0.07488623
Validation score: 0.935509
Iteration 52, loss = 0.06438127
Validation score: 0.934125
Iteration 53, loss = 0.05928741
Validation score: 0.941444
Iteration 54, loss = 0.06207190
Validation score: 0.938081
Iteration 55, loss = 0.06144641
Validation score: 0.920475
Iteration 56, loss = 0.06293869
Validation score: 0.937685
Iteration 57, loss = 0.05972565
Validation score: 0.929179
Iteration 58, loss = 0.05873227
Validation score: 0.932344
Iteration 59, loss = 0.06258876
Validation score: 0.940653
Iteration 60, loss = 0.05570789
Validation score: 0.936696
Iteration 61, loss = 0.05624749
Validation score: 0.940059
```

```
Iteration 62, loss = 0.05877355
Validation score: 0.936696
Iteration 63, loss = 0.06326369
Validation score: 0.939862
Iteration 64, loss = 0.05362035
Validation score: 0.945796
Iteration 65, loss = 0.05962739
Validation score: 0.928981
Iteration 66, loss = 0.05404427
Validation score: 0.944411
Iteration 67, loss = 0.05856537
Validation score: 0.929773
Iteration 68, loss = 0.05812045
Validation score: 0.935509
Iteration 69, loss = 0.05249832
Validation score: 0.937092
Iteration 70, loss = 0.05692958
Validation score: 0.942235
Iteration 71, loss = 0.06169963
Validation score: 0.933927
Iteration 72, loss = 0.05295570
Validation score: 0.948961
Iteration 73, loss = 0.04822423
Validation score: 0.940653
Iteration 74, loss = 0.05481595
Validation score: 0.932542
Iteration 75, loss = 0.05504483
Validation score: 0.938477
Iteration 76, loss = 0.05376706
Validation score: 0.939466
Iteration 77, loss = 0.05802848
Validation score: 0.928783
Iteration 78, loss = 0.05205336
Validation score: 0.942829
Iteration 79, loss = 0.05029906
Validation score: 0.938675
Iteration 80, loss = 0.05840547
Validation score: 0.934125
Iteration 81, loss = 0.05243236
Validation score: 0.943620
Iteration 82, loss = 0.05094091
Validation score: 0.930762
Iteration 83, loss = 0.04746883
Validation score: 0.941444
Validation score did not improve more than tol=0.000100 for 10 conse
cutive epochs. Stopping.
```

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

In [62]:

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

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

           0       0.87      0.95      0.91     10832
           1       0.16      0.07      0.10      1611

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


Report for Logistic Regression:
               precision    recall  f1-score   support

           0       0.93      0.78      0.85     10832
           1       0.29      0.61      0.39      1611

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


Report for NN:
               precision    recall  f1-score   support

           0       0.90      0.92      0.91     10832
           1       0.37      0.33      0.35      1611

   micro avg       0.84      0.84      0.84     12443
   macro avg       0.64      0.63      0.63     12443
weighted avg       0.83      0.84      0.84     12443


Report for Ensemble:
               precision    recall  f1-score   support

           0       0.91      0.93      0.92     10832
           1       0.44      0.38      0.41      1611

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

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

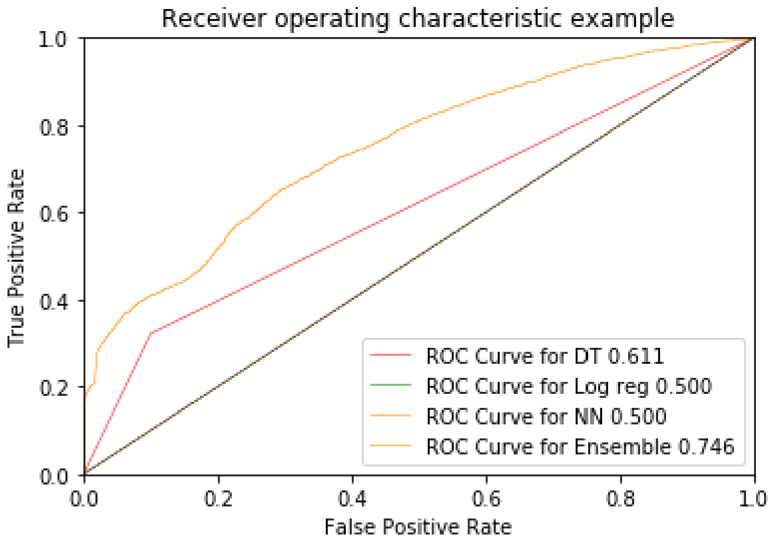
### a. Discuss the findings led by (a) ROC Chart (and Index); (b) Score Ranking (or Accuracy Score); (c) Fit Statistics; (or Classification report) and (4) Output.

(a) ROC Chart (and Index)

In [63]:

```
#### ROC

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

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


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


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


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

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

```
ROC index on test for DT: 0.6106552750339935
ROC index on test for logistic regression: 0.4997357932951725
ROC index on test for NN: 0.5
ROC index on voting classifier: 0.7459614568233351
```




(b) Score Ranking (or Accuracy Score)

In [64]:

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

```
Accuracy score on test for DT: 0.8348469018725387
Accuracy score on test for Logistic Regression: 0.7552840954753677
Accuracy score on test for NN: 0.8415976854456321
Accuracy score on test for Ensemble: 0.856465482600659
```

(c) Classification report

In [65]:

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

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

           0       0.87      0.95      0.91     10832
           1       0.16      0.07      0.10      1611

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


Report for Logistic Regression:
               precision    recall  f1-score   support

           0       0.93      0.78      0.85     10832
           1       0.29      0.61      0.39      1611

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


Report for NN:
               precision    recall  f1-score   support

           0       0.90      0.92      0.91     10832
           1       0.37      0.33      0.35      1611

   micro avg       0.84      0.84      0.84     12443
   macro avg       0.64      0.63      0.63     12443
weighted avg       0.83      0.84      0.84     12443


Report for Ensemble:
               precision    recall  f1-score   support

           0       0.91      0.93      0.92     10832
           1       0.44      0.38      0.41      1611

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

(d) Output

In [ ]:

```
```

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

In [ ]:

# Task 6. Final Remarks: Decision Making

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

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

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

In [ ]:

In [ ]:

In [ ]: