MOTI_Assignment_4_complete

May 30, 2021

1 Assignment 01: Data Preperation

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
pd.set_option('display.width', None)
pd.set_option('display.max_colwidth', None)
```

1.1 Requirement for Unit 01

1. Download the HMEQ Data set
2. Read the data into Python
3. Explore both the input and target variables using statistical techniques.
4. Explore both the input and target variables using graphs and other visualization.
5. Look for relationships between the input variables and the targets.
6. Fix (impute) all missing data. Note: For numerical data, create a flag variable to indicate if the value was missing
7. Convert all categorical variables numeric variables

1.2 Questions 1 and 2

[ ]:
```
# 1 and 2
mydata = "/content/drive/MyDrive/Chromebook DESK/NU MSDS/422 Pract Machine␣
 ↪Learning/Unit 02/HMEQ_Loss.csv"

df = pd.read_csv(mydata)

TARGET_F = "TARGET_BAD_FLAG"
```

```
TARGET_A = "TARGET_LOSS_AMT"
```

[ ]:
```
df.head().T
```

[ ]:

	0	1	2	3	4
TARGET_BAD_FLAG	1	1	1	1	0
TARGET_LOSS_AMT	641	1109	767	1425	NaN
LOAN	1100	1300	1500	1500	1700
MORTDUE	25860	70053	13500	NaN	97800
VALUE	39025	68400	16700	NaN	112000
REASON	HomeImp	HomeImp	HomeImp	NaN	HomeImp
JOB	Other	Other	Other	NaN	Office
YOJ	10.5	7	4	NaN	3
DEROG	0	0	0	NaN	0
DELINQ	0	2	0	NaN	0
CLAGE	94.3667	121.833	149.467	NaN	93.3333
NINQ	1	0	1	NaN	0
CLNO	9	14	10	NaN	14
DEBTINC	NaN	NaN	NaN	NaN	NaN

1.3 Question 3: Explore both the input and target variables using statistical techniques

[ ]:
```
df.describe().T
```

[ ]:

	count	mean	std	min	\
TARGET_BAD_FLAG	5960.0	0.199497	0.399656	0.000000	
TARGET_LOSS_AMT	1189.0	13414.576955	10839.455965	224.000000	
LOAN	5960.0	18607.969799	11207.480417	1100.000000	
MORTDUE	5442.0	73760.817200	44457.609458	2063.000000	
VALUE	5848.0	101776.048741	57385.775334	8000.000000	
YOJ	5445.0	8.922268	7.573982	0.000000	
DEROG	5252.0	0.254570	0.846047	0.000000	
DELINQ	5380.0	0.449442	1.127266	0.000000	
CLAGE	5652.0	179.766275	85.810092	0.000000	
NINQ	5450.0	1.186055	1.728675	0.000000	
CLNO	5738.0	21.296096	10.138933	0.000000	
DEBTINC	4693.0	33.779915	8.601746	0.524499	

	25%	50%	75%	max
TARGET_BAD_FLAG	0.000000	0.000000	0.000000	1.000000
TARGET_LOSS_AMT	5639.000000	11003.000000	17634.000000	78987.000000
LOAN	11100.000000	16300.000000	23300.000000	89900.000000
MORTDUE	46276.000000	65019.000000	91488.000000	399550.000000
VALUE	66075.500000	89235.500000	119824.250000	855909.000000
YOJ	3.000000	7.000000	13.000000	41.000000
DEROG	0.000000	0.000000	0.000000	10.000000

DELINQ	0.000000	0.000000	0.000000	15.000000
CLAGE	115.116702	173.466667	231.562278	1168.233561
NINQ	0.000000	1.000000	2.000000	17.000000
CLNO	15.000000	20.000000	26.000000	71.000000
DEBTINC	29.140031	34.818262	39.003141	203.312149

1.3.1 Data types?

Indented block

[ ]:
```
# print( df.describe().T )
# print( df.dtypes )

dt = df.dtypes
print( dt )
```

```
TARGET_BAD_FLAG      int64
TARGET_LOSS_AMT    float64
LOAN                 int64
MORTDUE            float64
VALUE              float64
REASON              object
JOB                 object
YOJ                float64
DEROG              float64
DELINQ             float64
CLAGE              float64
NINQ               float64
CLNO               float64
DEBTINC            float64
dtype: object
```

1.3.2 Categorical or Numeric Vars?

[ ]:
```
# identify objects and numbers, put into lists
## define oblist and numlist

objList = []
numList = []
for i in dt.index :
    #print(" here is i .....", i , " ..... and here is the type", dt[i] )
    if i in ( [ TARGET_F, TARGET_A ] ) : continue
    if dt[i] in (["object"]) : objList.append( i )
    if dt[i] in (["float64","int64"]) : numList.append( i )
```

[ ]:
```
print("\n OBJECTS ")
print(" ------- ")
for i in objList :
```

```
   print(i)

print("\n NUMBERS ")
print(" ------- ")
for i in numList :
   print(i)
```

```
 OBJECTS
 -------
REASON
JOB

 NUMBERS
 -------
LOAN
MORTDUE
VALUE
YOJ
DEROG
DELINQ
CLAGE
NINQ
CLNO
DEBTINC
```

[ ]:
```
for i in objList :
    print(" Class = ", i )
    g = df.groupby( i )
    print( g[i].count() )
    x = g[ TARGET_F ].mean()
    print( "Mean Bad Flag", x )
    print( " ................. ")
    x = g[ TARGET_A ].mean()
    print( "Mean Loss Amount", x )
    print(" ===============\n\n\n ")
```

```
 Class =  REASON
REASON
DebtCon    3928
HomeImp    1780
Name: REASON, dtype: int64
Mean Bad Flag REASON
DebtCon    0.189664
HomeImp    0.222472
Name: TARGET_BAD_FLAG, dtype: float64
 …
Mean Loss Amount REASON
```

```
DebtCon    16005.163758
HomeImp     8388.090909
Name: TARGET_LOSS_AMT, dtype: float64
 ===============



 Class =  JOB
JOB
Mgr         767
Office      948
Other      2388
ProfExe    1276
Sales       109
Self        193
Name: JOB, dtype: int64
Mean Bad Flag JOB
Mgr        0.233377
Office     0.131857
Other      0.231993
ProfExe    0.166144
Sales      0.348624
Self       0.300518
Name: TARGET_BAD_FLAG, dtype: float64
 …
Mean Loss Amount JOB
Mgr        14141.536313
Office     13475.304000
Other      11570.102888
ProfExe    14660.966981
Sales      16421.447368
Self       22232.362069
Name: TARGET_LOSS_AMT, dtype: float64
 ===============
```

[ ]:
```
##### Print most common, and ignore the missing. the missing cannot comeback as␣
 ↪the most common
for i in objList :
    print( i )
    print( df[i].unique() )
    g = df.groupby( i )
    print( g[i].count() )
    print( "MOST COMMON = ", df[i].mode()[0] )
    print( "MISSING = ", df[i].isna().sum() )
```

```
    print( "\n\n")
```

```
REASON
['HomeImp' nan 'DebtCon']
REASON
DebtCon    3928
HomeImp    1780
Name: REASON, dtype: int64
MOST COMMON =  DebtCon
MISSING =  252



JOB
['Other' nan 'Office' 'Sales' 'Mgr' 'ProfExe' 'Self']
JOB
Mgr         767
Office      948
Other      2388
ProfExe    1276
Sales       109
Self        193
Name: JOB, dtype: int64
MOST COMMON =  Other
MISSING =  279
```

[ ]:
```
### compare means

for i in objList:
   g = df.groupby( i )
   x = g[ TARGET_F ].median()
   print( "Median Comparison- Loss Amount", x )
   print( " ................. ")
   x = g[ TARGET_A ].median()
   print( "Median Comparison- Bad Flag", x )
   print(" ===============\n\n\n ")
```

```
Median Comparison- Loss Amount REASON
DebtCon    0
HomeImp    0
Name: TARGET_BAD_FLAG, dtype: int64
 …
Median Comparison- Bad Flag REASON
DebtCon    13630.0
HomeImp     5784.5
```

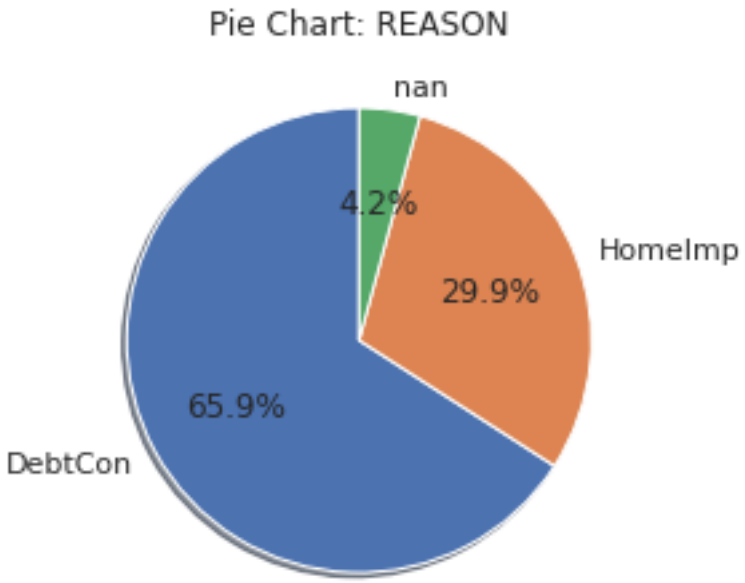
```
Name: TARGET_LOSS_AMT, dtype: float64
 ===============



Median Comparison- Loss Amount JOB
Mgr        0
Office     0
Other      0
ProfExe    0
Sales      0
Self       0
Name: TARGET_BAD_FLAG, dtype: int64
 …
Median Comparison- Bad Flag JOB
Mgr        12779.0
Office     10208.0
Other       9332.0
ProfExe    12438.5
Sales      15614.0
Self       18484.5
Name: TARGET_LOSS_AMT, dtype: float64
 ===============
```

1.4 Question 4: Explore both the input and target variables using graphs and other visualization

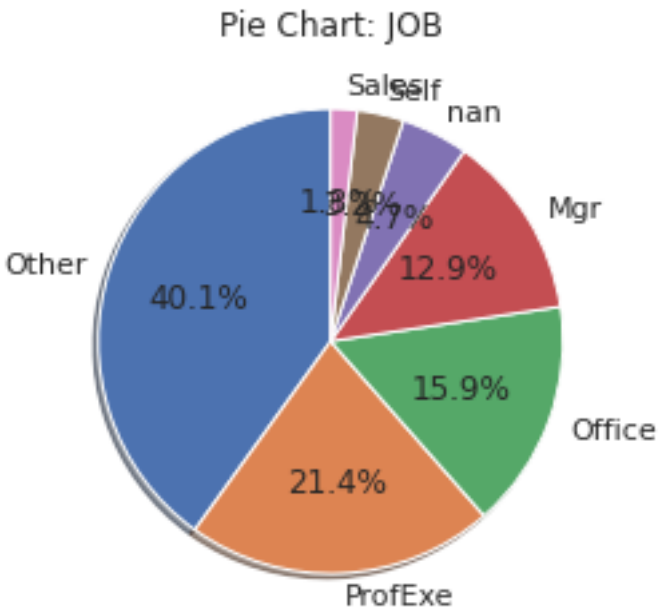
1.4.1 Show me my cat vars

[ ]:
```
for i in objList :
    x = df[ i ].value_counts(dropna=False)
    print( x )
    theLabels = x.axes[0].tolist()
    print( theLabels )


    theSlices = list(x)
    print( theSlices )
    plt.pie( theSlices,
             labels=theLabels,
             startangle = 90,
             shadow=True,
             autopct="%1.1f%%")
    plt.title("Pie Chart: " + i)
    plt.show()
```

```
DebtCon    3928
HomeImp    1780
NaN         252
Name: REASON, dtype: int64
['DebtCon', 'HomeImp', nan]
[3928, 1780, 252]
```

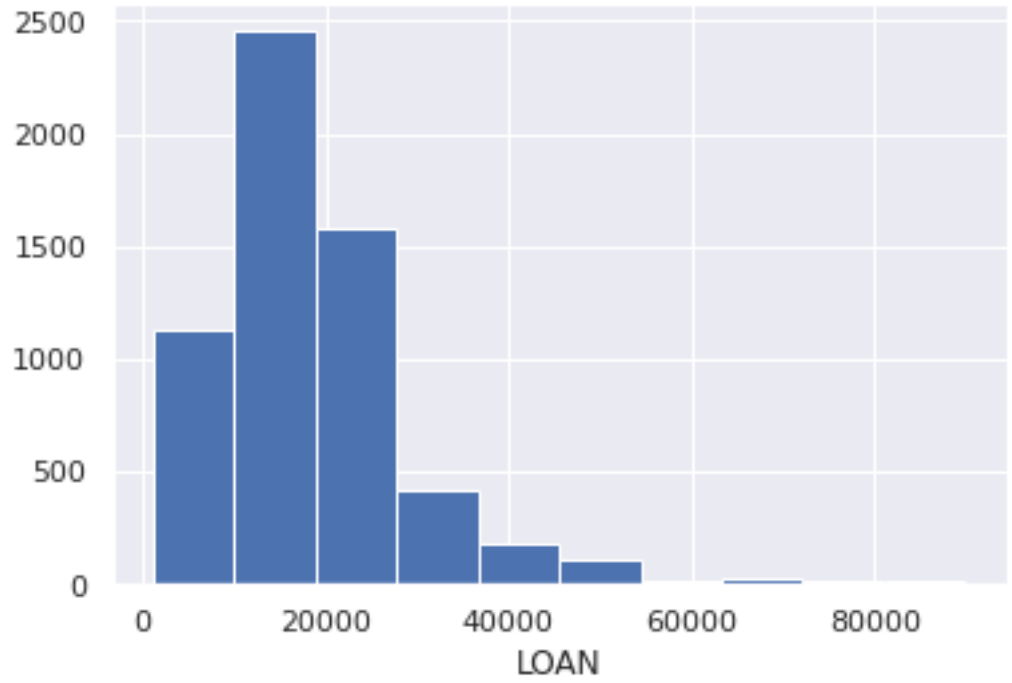


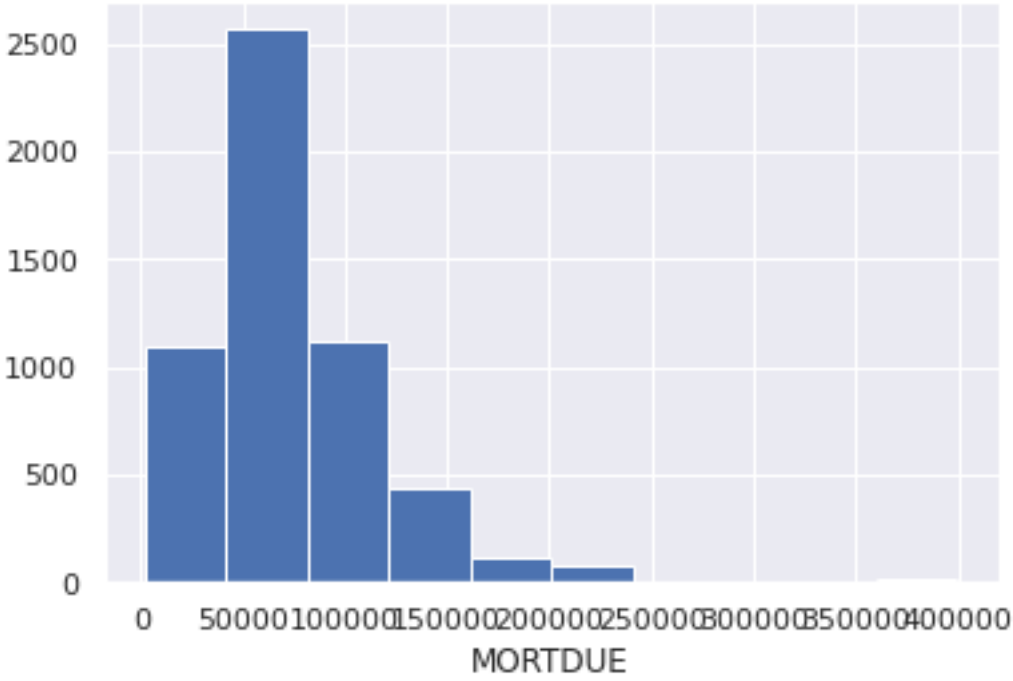

```
Other      2388
ProfExe    1276
Office      948
Mgr         767
NaN         279
Self        193
Sales       109
Name: JOB, dtype: int64
['Other', 'ProfExe', 'Office', 'Mgr', nan, 'Self', 'Sales']
[2388, 1276, 948, 767, 279, 193, 109]
```

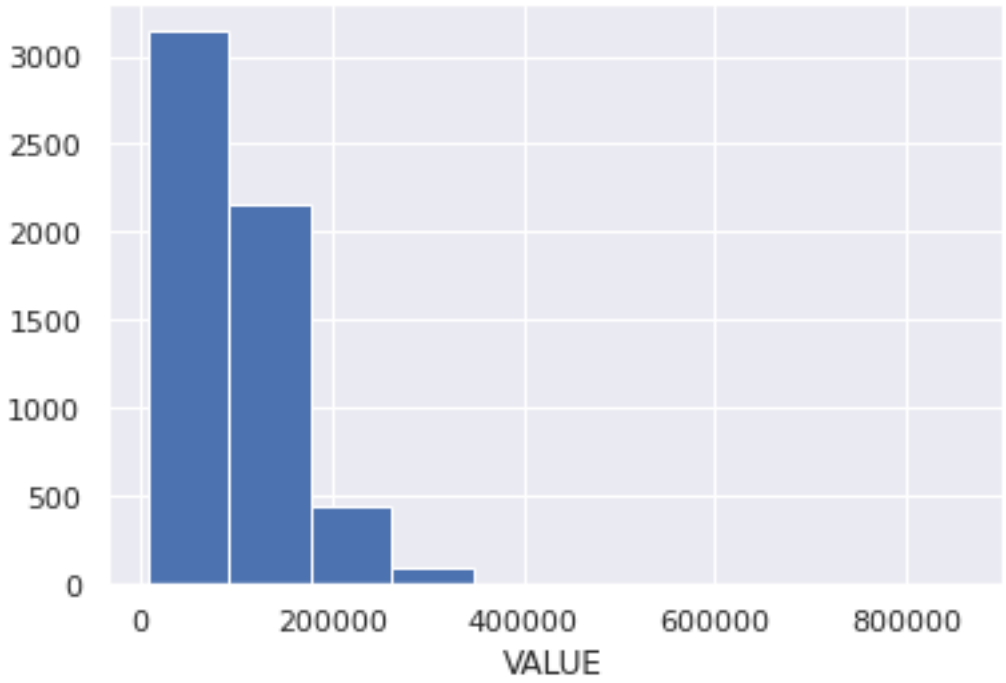


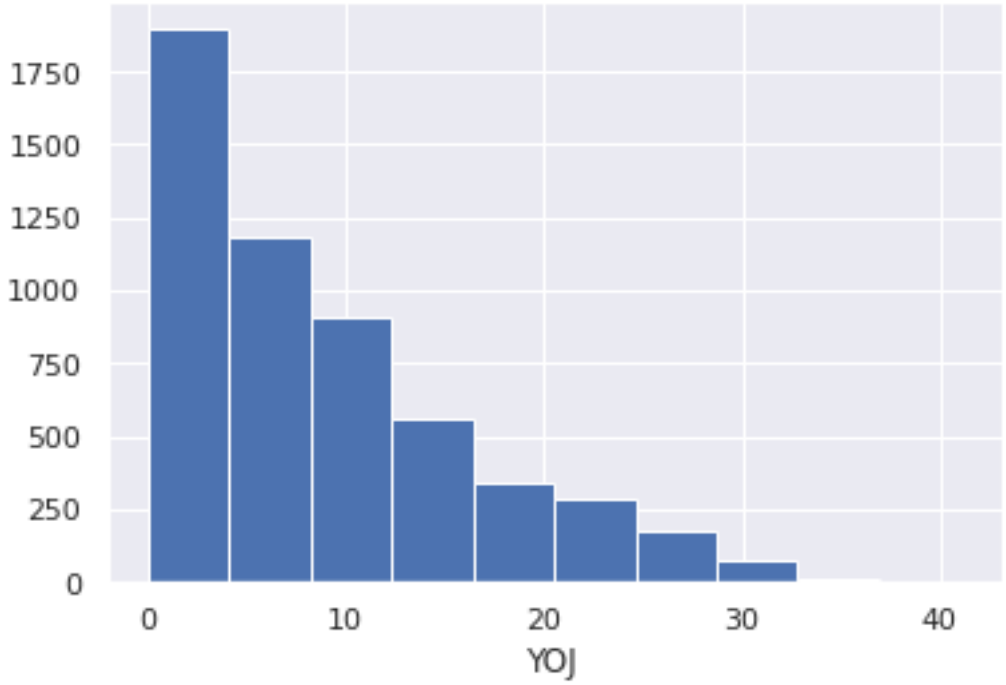
1.4.2 Show me my numeric vars

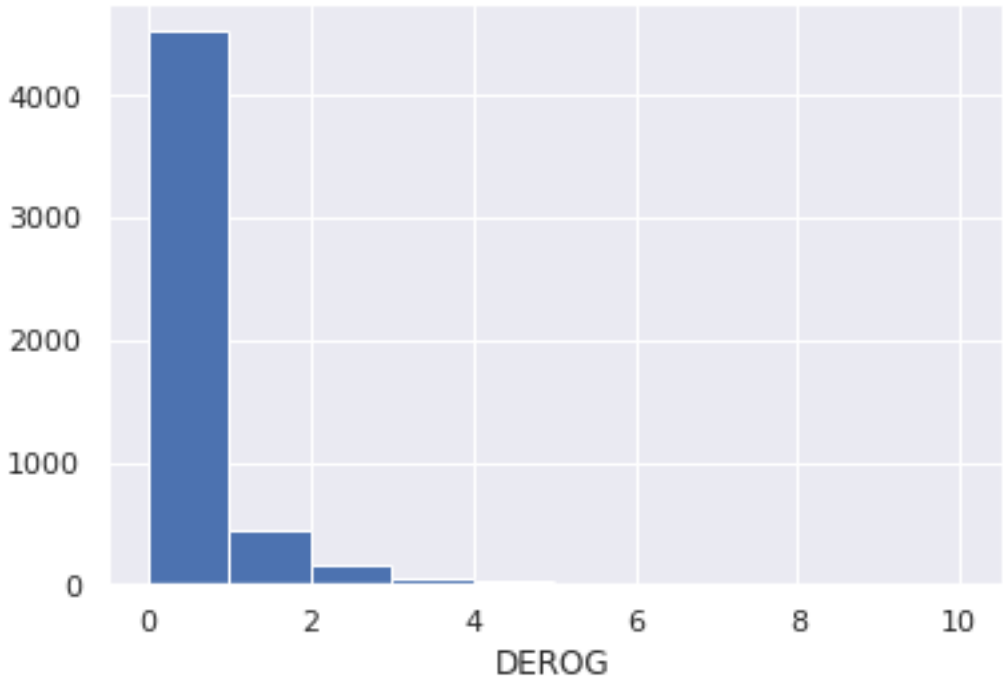
[ ]:
```
for i in numList :
    plt.hist( df[ i ] )
    plt.xlabel( i )
    plt.show()
```

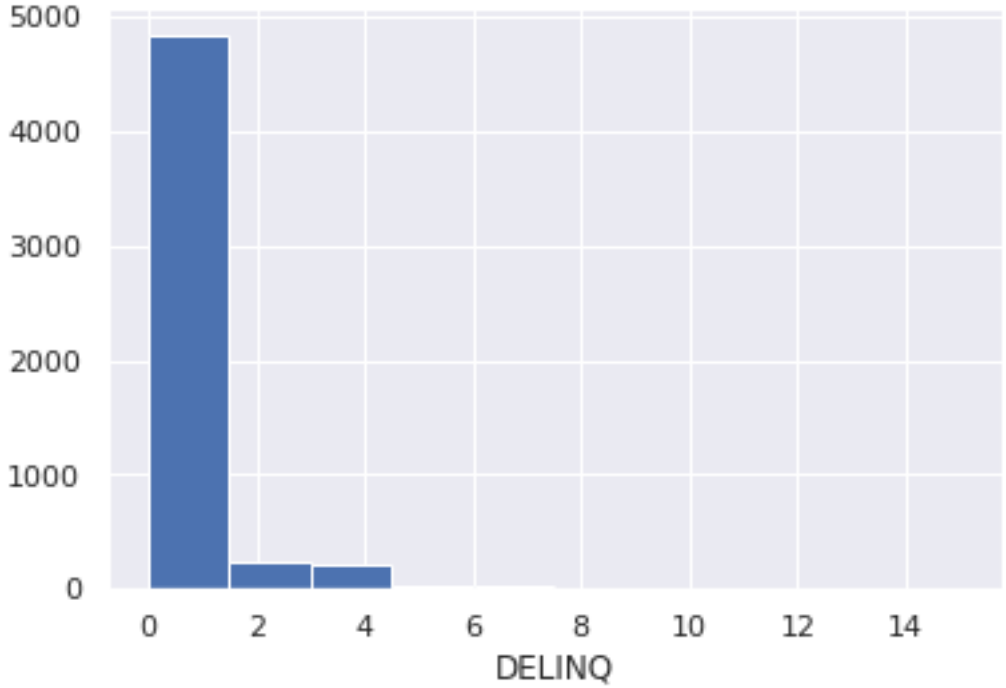

2500
2000
1500
1000
500
0
0
20000
40000
60000
80000
LOAN

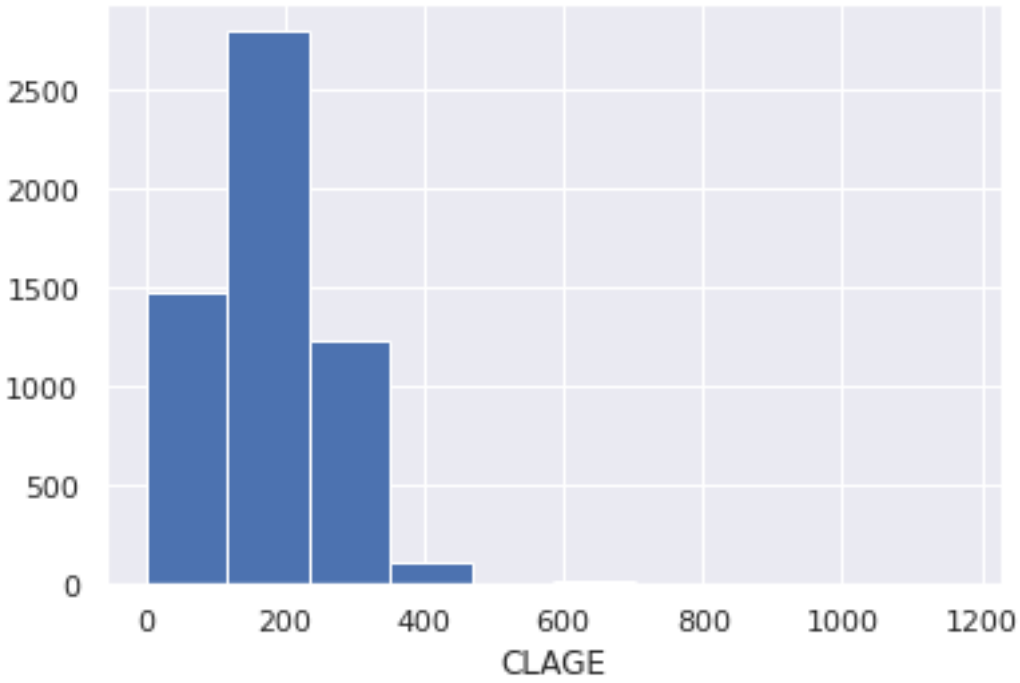

2500
2000
1500
1000
500
0
0
50000
100000
150000
200000
250000
300000
350000
400000
MORTDUE

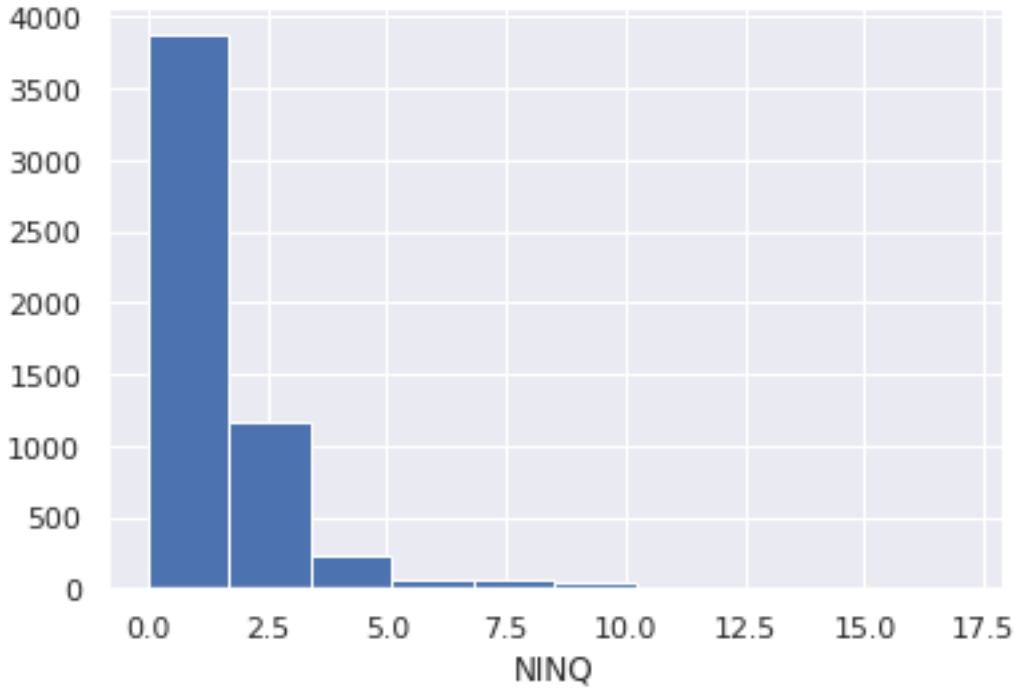

3000
2500
2000
1500
1000
500
0
0
200000
400000
600000
800000
VALUE

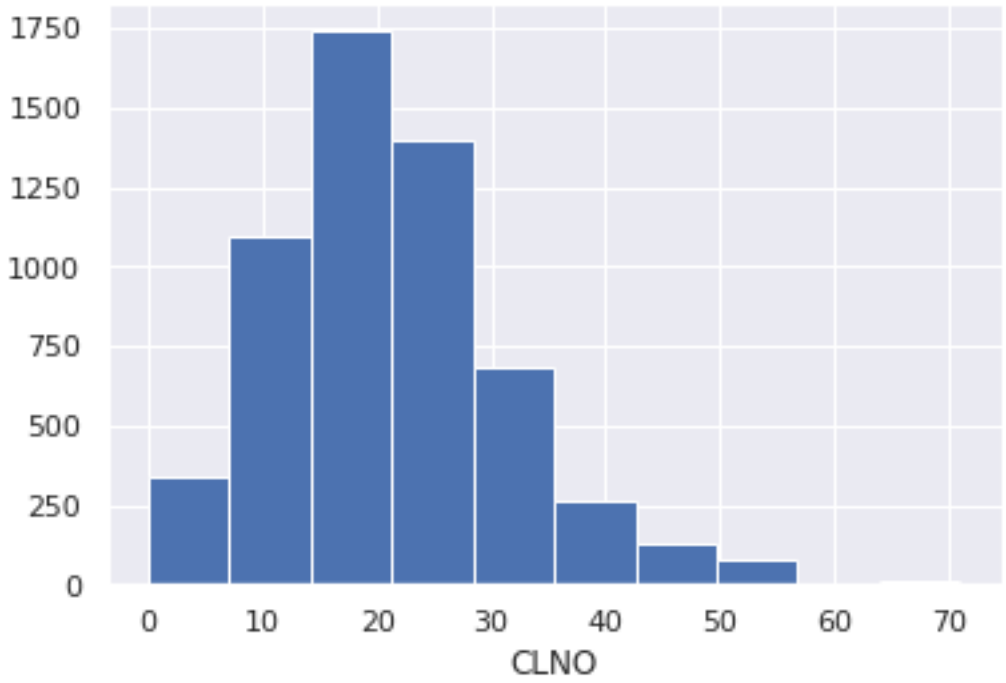

1750
1500
1250
1000
750
500
250
0
0
10
20
30
40
YOJ

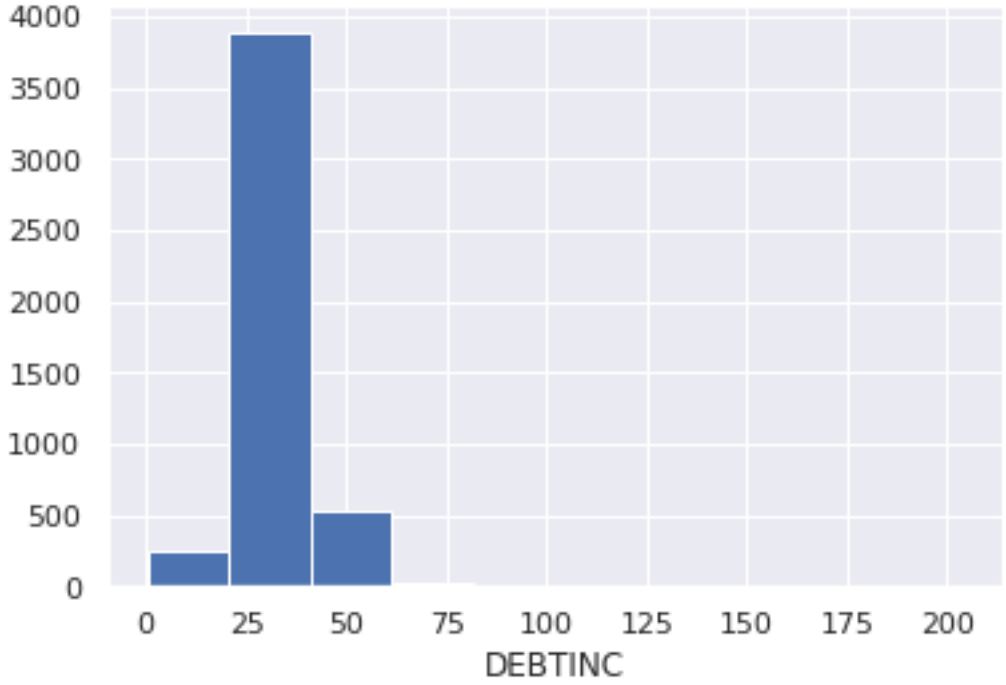

4000
3000
2000
1000
0
0
2
4
6
8
10
DEROG


5000
4000
3000
2000
1000
0
0
2
4
6
8
10
12
14
DELINQ


2500
2000
1500
1000
500
0
0
200
400
600
800
1000
1200
CLAGE

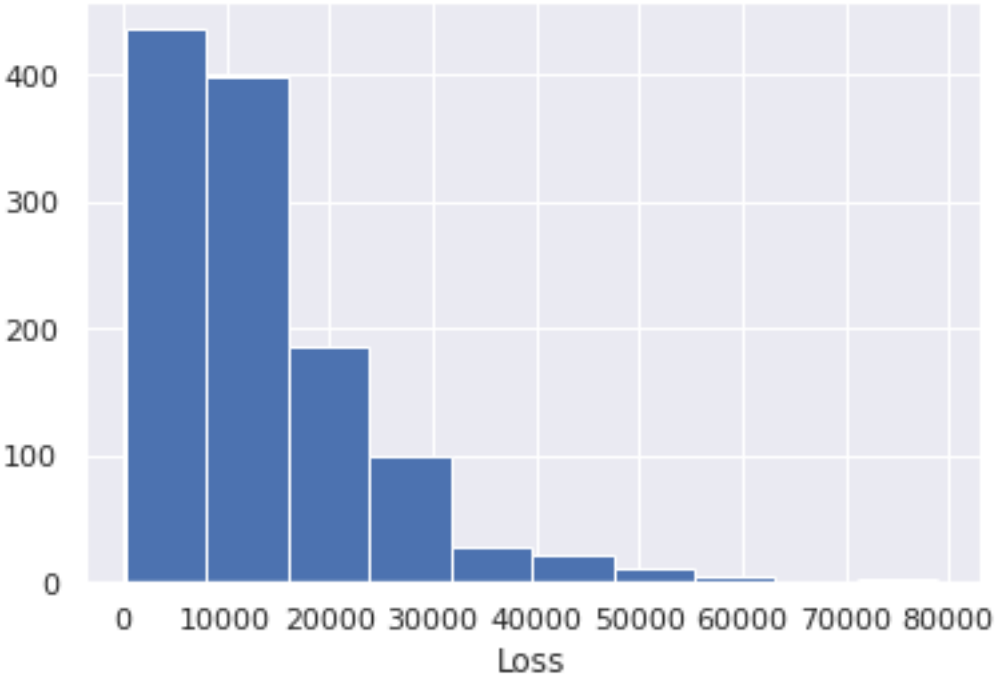

4000
3500
3000
2500
2000
1500
1000
500
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
NINQ


1750
1500
1250
1000
750
500
250
0
0
10
20
30
40
50
60
70
CLNO


4000
3500
3000
2500
2000
1500
1000
500
0
0
25
50
75
100
125
150
175
200
DEBTINC

1.4.3 Histogram my losses

[ ]:
```
plt.hist( df[ TARGET_A ] )
plt.xlabel( "Loss" )
plt.show()
```



1.5 Question 5: Look for relationships between the input variables and the targets

1.5.1 Written summary

[ ]:
```
"""
All these loans are home equity lines of loans.

Most loans were taken out for the purpose of Debt Consolidation (3928 loans).
The second reason was for Home Improvement(not the show on ABC) (1780 loans).

Almost 19% of the debt consolidation loans defaulted.
Around 22% of the home improvement loans defaulted.

The average loss on a default Home Eq loan taken for debt consolidation was $16.
 ↪005.
The average loss on a default Home Eq loan taken for home improvement was␣
 ↪$8,388.
```

```
Most loans were taken out by people who classified their job as "other".

The riskiest loans are to those employed in Sales and Self employment.
The safest loans are to those employed in Office and Prof Executive roles.

252 did not specificy what their loan is for. More likely, that information is␣
↪stored properly.
I cannot imagine someone getting a loan and writing "don't worry about it" when␣
↪the banker asks what it is for.

There are 5959 (5960, because it started at 0) total Home Eq Loans.
"""
```

[ ]:
```
'\nAll these loans are home equity lines of loans. \n\nMost loans were taken out
for the purpose of Debt Consolidation (3928 loans). \nThe second reason was for
Home Improvement(not the show on ABC) (1780 loans).\n\nAlmost 19% of the debt
consolidation loans defaulted. \nAround 22% of the home improvement loans
defaulted. \n\nThe average loss on a default Home Eq loan taken for debt
consolidation was $16.005.\nThe average loss on a default Home Eq loan taken for
home improvement was $8,388. \n\nMost loans were taken out by people who
classified their job as "other".\n\nThe riskiest loans are to those employed in
Sales and Self employment.\nThe safest loans are to those employed in Office and
Prof Executive roles. \n\n252 did not specificy what their loan is for. More
likely, that information is stored properly. \nI cannot imagine someone getting
a loan and writing "don\'t worry about it" when the banker asks what it is for.
\n\nThere are 5959 (5960, because it started at 0) total Home Eq Loans.\n'
```

1.6 Question 6: Impute missing data, both cat and numerical

[ ]:
```
"""
FILL IN MISSING WITH THE CATEGORY "MISSING"
"""
for i in objList :
    if df[i].isna().sum() == 0 : continue
    print( i ) ### This prints what is missing ### you see job and reason␣
↪missing
    #print("HAS MISSING")
    NAME = "IMP_"+i
    #print( NAME )
    df[NAME] = df[i]
    df[NAME] = df[NAME].fillna("MISSING")
    print( "variable",i," has this many missing", df[i].isna().sum() )
    print( "variable",NAME," has this many missing", df[NAME].isna().sum() )
    g = df.groupby( NAME )
    print( g[NAME].count() )
    print( "\n\n")
```

```
    df = df.drop( i, axis=1 )
```

```
REASON
variable REASON  has this many missing 252
variable IMP_REASON  has this many missing 0
IMP_REASON
DebtCon    3928
HomeImp    1780
MISSING     252
Name: IMP_REASON, dtype: int64
```

```
JOB
variable JOB  has this many missing 279
variable IMP_JOB  has this many missing 0
IMP_JOB
MISSING     279
Mgr         767
Office      948
Other      2388
ProfExe    1276
Sales       109
Self        193
Name: IMP_JOB, dtype: int64
```

1.6.1 Rerun the creation of the object list, updated to include imputed values

[ ]:
```
'''
Rerun the creation of the object list, updated to include imputed values
'''
dt = df.dtypes
objList = []
for i in dt.index :
    #print(" here is i .....", i , " ..... and here is the type", dt[i] )
    if i in ( [ TARGET_F, TARGET_A ] ) : continue
    if dt[i] in (["object"]) : objList.append( i )
```

1.7 Question 7: Convert all categorical variables numeric variables

1.7.1 It getting hot in herr (one hot encoding)

[ ]:
```
for i in objList :
    print(" Class = ", i )
    print( df[i].unique() )
```

```
 Class =  IMP_REASON
['HomeImp' 'MISSING' 'DebtCon']
 Class =  IMP_JOB
['Other' 'MISSING' 'Office' 'Sales' 'Mgr' 'ProfExe' 'Self']
```

[ ]:
```
for i in objList :
   print(" Class = ", i )
   g = df.groupby( i )
   print( g[i].count() )
   x = g[ TARGET_F ].mean()
   print( " ................. ")
   print( "Avg Loan Default Prob", x )
   print( " ................. ")
   x = g[ TARGET_A ].mean()
   #x = g[ TARGET_A ].median()
   print( "Avg Loan Default Amount", x )
   print(" ===============\n\n\n ")
```

```
 Class =  IMP_REASON
IMP_REASON
DebtCon    3928
HomeImp    1780
MISSING     252
Name: IMP_REASON, dtype: int64
 …
Avg Loan Default Prob IMP_REASON
DebtCon    0.189664
HomeImp    0.222472
MISSING    0.190476
Name: TARGET_BAD_FLAG, dtype: float64
 …
Avg Loan Default Amount IMP_REASON
DebtCon    16005.163758
HomeImp     8388.090909
MISSING    14675.020833
Name: TARGET_LOSS_AMT, dtype: float64
 ===============



 Class =  IMP_JOB
IMP_JOB
MISSING     279
```

```
Mgr          767
Office       948
Other       2388
ProfExe     1276
Sales        109
Self         193
Name: IMP_JOB, dtype: int64
 …
Avg Loan Default Prob IMP_JOB
MISSING     0.082437
Mgr         0.233377
Office      0.131857
Other       0.231993
ProfExe     0.166144
Sales       0.348624
Self        0.300518
Name: TARGET_BAD_FLAG, dtype: float64
 …
Avg Loan Default Amount IMP_JOB
MISSING     13162.173913
Mgr         14141.536313
Office      13475.304000
Other       11570.102888
ProfExe     14660.966981
Sales       16421.447368
Self        22232.362069
Name: TARGET_LOSS_AMT, dtype: float64
 ===============
```

[ ]:
```python
for i in objList :
    #print(" Class = ", i )
    thePrefix = "z_" + i
    #print( thePrefix )
    y = pd.get_dummies( df[i], prefix=thePrefix, dummy_na=False )
    # USE drop_first=True IF DATA IS BINARY, MINIMIZES NUMBER OF VARIABLES␣
 ↪ADDED
    #y = pd.get_dummies( df[i], prefix=thePrefix, drop_first=True )
    #print( type(y) )
    #print( y.head().T )
    df = pd.concat( [df, y], axis=1 )
    df = df.drop( i, axis=1 )

# print( df.head().T ) #Print variable in a .py file, but we're in using␣
 ↪Notebook
```

```
df.head().T
```

[ ]:

	0	1	2	3	\
TARGET_BAD_FLAG	1.000000	1.000000	1.000000	1.0	
TARGET_LOSS_AMT	641.000000	1109.000000	767.000000	1425.0	
LOAN	1100.000000	1300.000000	1500.000000	1500.0	
MORTDUE	25860.000000	70053.000000	13500.000000	NaN	
VALUE	39025.000000	68400.000000	16700.000000	NaN	
YOJ	10.500000	7.000000	4.000000	NaN	
DEROG	0.000000	0.000000	0.000000	NaN	
DELINQ	0.000000	2.000000	0.000000	NaN	
CLAGE	94.366667	121.833333	149.466667	NaN	
NINQ	1.000000	0.000000	1.000000	NaN	
CLNO	9.000000	14.000000	10.000000	NaN	
DEBTINC	NaN	NaN	NaN	NaN	
z_IMP_REASON_DebtCon	0.000000	0.000000	0.000000	0.0	
z_IMP_REASON_HomeImp	1.000000	1.000000	1.000000	0.0	
z_IMP_REASON_MISSING	0.000000	0.000000	0.000000	1.0	
z_IMP_JOB_MISSING	0.000000	0.000000	0.000000	1.0	
z_IMP_JOB_Mgr	0.000000	0.000000	0.000000	0.0	
z_IMP_JOB_Office	0.000000	0.000000	0.000000	0.0	
z_IMP_JOB_Other	1.000000	1.000000	1.000000	0.0	
z_IMP_JOB_ProfExe	0.000000	0.000000	0.000000	0.0	
z_IMP_JOB_Sales	0.000000	0.000000	0.000000	0.0	
z_IMP_JOB_Self	0.000000	0.000000	0.000000	0.0	

	4
TARGET_BAD_FLAG	0.000000
TARGET_LOSS_AMT	NaN
LOAN	1700.000000
MORTDUE	97800.000000
VALUE	112000.000000
YOJ	3.000000
DEROG	0.000000
DELINQ	0.000000
CLAGE	93.333333
NINQ	0.000000
CLNO	14.000000
DEBTINC	NaN
z_IMP_REASON_DebtCon	0.000000
z_IMP_REASON_HomeImp	1.000000
z_IMP_REASON_MISSING	0.000000
z_IMP_JOB_MISSING	0.000000
z_IMP_JOB_Mgr	0.000000
z_IMP_JOB_Office	1.000000
z_IMP_JOB_Other	0.000000
z_IMP_JOB_ProfExe	0.000000

```
z_IMP_JOB_Sales              0.000000
z_IMP_JOB_Self               0.000000
```

1.7.2 Missing Value Imputation For Numeric Variables

[ ]:
```
for i in numList :
    g = df.groupby( i )
    #print( g[i].count() )
    print(i)
    print( "MEDIAN = ", df[i].median() ) # The [0] returns the index 0
    print( "MISSING = ", df[i].isna().sum() )
    print( "\n")
```

```
LOAN
MEDIAN =  16300.0
MISSING =  0


MORTDUE
MEDIAN =  65019.0
MISSING =  518


VALUE
MEDIAN =  89235.5
MISSING =  112


YOJ
MEDIAN =  7.0
MISSING =  515


DEROG
MEDIAN =  0.0
MISSING =  708


DELINQ
MEDIAN =  0.0
MISSING =  580


CLAGE
MEDIAN =  173.4666667
MISSING =  308
```

```
NINQ
MEDIAN =  1.0
MISSING =  510


CLNO
MEDIAN =  20.0
MISSING =  222


DEBTINC
MEDIAN =  34.81826182
MISSING =  1267
```

[ ]:
```
for i in numList :
    if df[i].isna().sum() == 0 : continue
    FLAG = "M_" + i
    IMP = "IMP_" + i
    #print(i)
    #print( df[i].isna().sum() )
    #print( FLAG )
    #print( IMP )
    #print(" ------- ")
    df[ FLAG ] = df[i].isna() + 0
    df[ IMP ] = df[ i ]
    df.loc[ df[IMP].isna(), IMP ] = df[i].median()
    df = df.drop( i, axis=1 )

df.head().T
```

[ ]:

	0	1	2	3	\
TARGET_BAD_FLAG	1.000000	1.000000	1.000000	1.000000	
TARGET_LOSS_AMT	641.000000	1109.000000	767.000000	1425.000000	
LOAN	1100.000000	1300.000000	1500.000000	1500.000000	
z_IMP_REASON_DebtCon	0.000000	0.000000	0.000000	0.000000	
z_IMP_REASON_HomeImp	1.000000	1.000000	1.000000	0.000000	
z_IMP_REASON_MISSING	0.000000	0.000000	0.000000	1.000000	
z_IMP_JOB_MISSING	0.000000	0.000000	0.000000	1.000000	
z_IMP_JOB_Mgr	0.000000	0.000000	0.000000	0.000000	
z_IMP_JOB_Office	0.000000	0.000000	0.000000	0.000000	
z_IMP_JOB_Other	1.000000	1.000000	1.000000	0.000000	
z_IMP_JOB_ProfExe	0.000000	0.000000	0.000000	0.000000	
z_IMP_JOB_Sales	0.000000	0.000000	0.000000	0.000000	
z_IMP_JOB_Self	0.000000	0.000000	0.000000	0.000000	
M_MORTDUE	0.000000	0.000000	0.000000	1.000000	

IMP_MORTDUE	25860.000000	70053.000000	13500.000000	65019.000000
M_VALUE	0.000000	0.000000	0.000000	1.000000
IMP_VALUE	39025.000000	68400.000000	16700.000000	89235.500000
M_YOJ	0.000000	0.000000	0.000000	1.000000
IMP_YOJ	10.500000	7.000000	4.000000	7.000000
M_DEROG	0.000000	0.000000	0.000000	1.000000
IMP_DEROG	0.000000	0.000000	0.000000	0.000000
M_DELINQ	0.000000	0.000000	0.000000	1.000000
IMP_DELINQ	0.000000	2.000000	0.000000	0.000000
M_CLAGE	0.000000	0.000000	0.000000	1.000000
IMP_CLAGE	94.366667	121.833333	149.466667	173.466667
M_NINQ	0.000000	0.000000	0.000000	1.000000
IMP_NINQ	1.000000	0.000000	1.000000	1.000000
M_CLNO	0.000000	0.000000	0.000000	1.000000
IMP_CLNO	9.000000	14.000000	10.000000	20.000000
M_DEBTINC	1.000000	1.000000	1.000000	1.000000
IMP_DEBTINC	34.818262	34.818262	34.818262	34.818262

	4
TARGET_BAD_FLAG	0.000000
TARGET_LOSS_AMT	NaN
LOAN	1700.000000
z_IMP_REASON_DebtCon	0.000000
z_IMP_REASON_HomeImp	1.000000
z_IMP_REASON_MISSING	0.000000
z_IMP_JOB_MISSING	0.000000
z_IMP_JOB_Mgr	0.000000
z_IMP_JOB_Office	1.000000
z_IMP_JOB_Other	0.000000
z_IMP_JOB_ProfExe	0.000000
z_IMP_JOB_Sales	0.000000
z_IMP_JOB_Self	0.000000
M_MORTDUE	0.000000
IMP_MORTDUE	97800.000000
M_VALUE	0.000000
IMP_VALUE	112000.000000
M_YOJ	0.000000
IMP_YOJ	3.000000
M_DEROG	0.000000
IMP_DEROG	0.000000
M_DELINQ	0.000000
IMP_DELINQ	0.000000
M_CLAGE	0.000000
IMP_CLAGE	93.333333
M_NINQ	0.000000
IMP_NINQ	0.000000
M_CLNO	0.000000

```
IMP_CLNO                 14.000000
M_DEBTINC                 1.000000
IMP_DEBTINC              34.818262
```

[ ]:
```
df.describe().T
```

[ ]:
```
                        count           mean           std          min  \
TARGET_BAD_FLAG        5960.0       0.199497      0.399656     0.000000
TARGET_LOSS_AMT        1189.0   13414.576955  10839.455965   224.000000
LOAN                   5960.0   18607.969799  11207.480417  1100.000000
z_IMP_REASON_DebtCon   5960.0       0.659060      0.474065     0.000000
z_IMP_REASON_HomeImp   5960.0       0.298658      0.457708     0.000000
z_IMP_REASON_MISSING   5960.0       0.042282      0.201248     0.000000
z_IMP_JOB_MISSING      5960.0       0.046812      0.211254     0.000000
z_IMP_JOB_Mgr          5960.0       0.128691      0.334886     0.000000
z_IMP_JOB_Office       5960.0       0.159060      0.365763     0.000000
z_IMP_JOB_Other        5960.0       0.400671      0.490076     0.000000
z_IMP_JOB_ProfExe      5960.0       0.214094      0.410227     0.000000
z_IMP_JOB_Sales        5960.0       0.018289      0.134004     0.000000
z_IMP_JOB_Self         5960.0       0.032383      0.177029     0.000000
M_MORTDUE              5960.0       0.086913      0.281731     0.000000
IMP_MORTDUE            5960.0   73001.041812  42552.726779  2063.000000
M_VALUE                5960.0       0.018792      0.135801     0.000000
IMP_VALUE              5960.0  101540.387423  56869.436682  8000.000000
M_YOJ                  5960.0       0.086409      0.280991     0.000000
IMP_YOJ                5960.0       8.756166      7.259424     0.000000
M_DEROG                5960.0       0.118792      0.323571     0.000000
IMP_DEROG              5960.0       0.224329      0.798458     0.000000
M_DELINQ               5960.0       0.097315      0.296412     0.000000
IMP_DELINQ             5960.0       0.405705      1.079256     0.000000
M_CLAGE                5960.0       0.051678      0.221394     0.000000
IMP_CLAGE              5960.0     179.440725     83.574697     0.000000
M_NINQ                 5960.0       0.085570      0.279752     0.000000
IMP_NINQ               5960.0       1.170134      1.653866     0.000000
M_CLNO                 5960.0       0.037248      0.189386     0.000000
IMP_CLNO               5960.0      21.247819      9.951308     0.000000
M_DEBTINC              5960.0       0.212584      0.409170     0.000000
IMP_DEBTINC            5960.0      34.000651      7.644528     0.524499

                              25%           50%           75%           max
TARGET_BAD_FLAG          0.000000      0.000000      0.000000      1.000000
TARGET_LOSS_AMT       5639.000000  11003.000000  17634.000000  78987.000000
LOAN                 11100.000000  16300.000000  23300.000000  89900.000000
z_IMP_REASON_DebtCon     0.000000      1.000000      1.000000      1.000000
z_IMP_REASON_HomeImp     0.000000      0.000000      1.000000      1.000000
z_IMP_REASON_MISSING     0.000000      0.000000      0.000000      1.000000
z_IMP_JOB_MISSING        0.000000      0.000000      0.000000      1.000000
```

z_IMP_JOB_Mgr	0.000000	0.000000	0.000000	1.000000
z_IMP_JOB_Office	0.000000	0.000000	0.000000	1.000000
z_IMP_JOB_Other	0.000000	0.000000	1.000000	1.000000
z_IMP_JOB_ProfExe	0.000000	0.000000	0.000000	1.000000
z_IMP_JOB_Sales	0.000000	0.000000	0.000000	1.000000
z_IMP_JOB_Self	0.000000	0.000000	0.000000	1.000000
M_MORTDUE	0.000000	0.000000	0.000000	1.000000
IMP_MORTDUE	48139.000000	65019.000000	88200.250000	399550.000000
M_VALUE	0.000000	0.000000	0.000000	1.000000
IMP_VALUE	66489.500000	89235.500000	119004.750000	855909.000000
M_YOJ	0.000000	0.000000	0.000000	1.000000
IMP_YOJ	3.000000	7.000000	12.000000	41.000000
M_DEROG	0.000000	0.000000	0.000000	1.000000
IMP_DEROG	0.000000	0.000000	0.000000	10.000000
M_DELINQ	0.000000	0.000000	0.000000	1.000000
IMP_DELINQ	0.000000	0.000000	0.000000	15.000000
M_CLAGE	0.000000	0.000000	0.000000	1.000000
IMP_CLAGE	117.371430	173.466667	227.143058	1168.233561
M_NINQ	0.000000	0.000000	0.000000	1.000000
IMP_NINQ	0.000000	1.000000	2.000000	17.000000
M_CLNO	0.000000	0.000000	0.000000	1.000000
IMP_CLNO	15.000000	20.000000	26.000000	71.000000
M_DEBTINC	0.000000	0.000000	0.000000	1.000000
IMP_DEBTINC	30.763159	34.818262	37.949892	203.312149

[ ]:

[ ]:
```
##Convert jobs to numerical    ###############
##df["y_JOB_MGR"] = (df.JOB.isin( ["Mgr"] ) + 0 )
##df["y_JOB_Office"] = (df.JOB.isin( ["Office"] ) + 0)
##df["y_JOB_Other"] = (df.JOB.isin( ["Other"] ) + 0)
##df["y_JOB_ProfExe"] = (df.JOB.isin( ["ProfExe"] ) + 0)
##df["y_JOB_Sales"] = (df.JOB.isin( ["Sales"] ) + 0)
##df["y_JOB_Self"] = (df.JOB.isin( ["Self"] ) + 0)

##Convert reason for loan to numerial
##df["y_REASON_HomeImp"] = (df.REASON.isin( ["HomeImp"] ) + 0 )
##df["y_REASON_DebtCon"] = (df.REASON.isin( ["DebtCon"] ) + 0)


##print( df.head().T )
```

2 Assignment 02: Tree based models

[ ]:
```
################### START UNIT 2 ################################################
### Make test and training data ###
```

[ ]:
```
import math
from operator import itemgetter ## grab variables out of random forests

from sklearn.model_selection import train_test_split
import sklearn.metrics as metrics

from sklearn import tree
from sklearn.tree import _tree

from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import RandomForestClassifier

from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import GradientBoostingClassifier
```

[ ]:
```
df.head().T
```

[ ]:

	0	1	2	3	\
TARGET_BAD_FLAG	1.000000	1.000000	1.000000	1.000000	
TARGET_LOSS_AMT	641.000000	1109.000000	767.000000	1425.000000	
LOAN	1100.000000	1300.000000	1500.000000	1500.000000	
z_IMP_REASON_DebtCon	0.000000	0.000000	0.000000	0.000000	
z_IMP_REASON_HomeImp	1.000000	1.000000	1.000000	0.000000	
z_IMP_REASON_MISSING	0.000000	0.000000	0.000000	1.000000	
z_IMP_JOB_MISSING	0.000000	0.000000	0.000000	1.000000	
z_IMP_JOB_Mgr	0.000000	0.000000	0.000000	0.000000	
z_IMP_JOB_Office	0.000000	0.000000	0.000000	0.000000	
z_IMP_JOB_Other	1.000000	1.000000	1.000000	0.000000	
z_IMP_JOB_ProfExe	0.000000	0.000000	0.000000	0.000000	
z_IMP_JOB_Sales	0.000000	0.000000	0.000000	0.000000	
z_IMP_JOB_Self	0.000000	0.000000	0.000000	0.000000	
M_MORTDUE	0.000000	0.000000	0.000000	1.000000	
IMP_MORTDUE	25860.000000	70053.000000	13500.000000	65019.000000	
M_VALUE	0.000000	0.000000	0.000000	1.000000	
IMP_VALUE	39025.000000	68400.000000	16700.000000	89235.500000	
M_YOJ	0.000000	0.000000	0.000000	1.000000	
IMP_YOJ	10.500000	7.000000	4.000000	7.000000	
M_DEROG	0.000000	0.000000	0.000000	1.000000	
IMP_DEROG	0.000000	0.000000	0.000000	0.000000	
M_DELINQ	0.000000	0.000000	0.000000	1.000000	
IMP_DELINQ	0.000000	2.000000	0.000000	0.000000	
M_CLAGE	0.000000	0.000000	0.000000	1.000000	

IMP_CLAGE	94.366667	121.833333	149.466667	173.466667
M_NINQ	0.000000	0.000000	0.000000	1.000000
IMP_NINQ	1.000000	0.000000	1.000000	1.000000
M_CLNO	0.000000	0.000000	0.000000	1.000000
IMP_CLNO	9.000000	14.000000	10.000000	20.000000
M_DEBTINC	1.000000	1.000000	1.000000	1.000000
IMP_DEBTINC	34.818262	34.818262	34.818262	34.818262

	4
TARGET_BAD_FLAG	0.000000
TARGET_LOSS_AMT	NaN
LOAN	1700.000000
z_IMP_REASON_DebtCon	0.000000
z_IMP_REASON_HomeImp	1.000000
z_IMP_REASON_MISSING	0.000000
z_IMP_JOB_MISSING	0.000000
z_IMP_JOB_Mgr	0.000000
z_IMP_JOB_Office	1.000000
z_IMP_JOB_Other	0.000000
z_IMP_JOB_ProfExe	0.000000
z_IMP_JOB_Sales	0.000000
z_IMP_JOB_Self	0.000000
M_MORTDUE	0.000000
IMP_MORTDUE	97800.000000
M_VALUE	0.000000
IMP_VALUE	112000.000000
M_YOJ	0.000000
IMP_YOJ	3.000000
M_DEROG	0.000000
IMP_DEROG	0.000000
M_DELINQ	0.000000
IMP_DELINQ	0.000000
M_CLAGE	0.000000
IMP_CLAGE	93.333333
M_NINQ	0.000000
IMP_NINQ	0.000000
M_CLNO	0.000000
IMP_CLNO	14.000000
M_DEBTINC	1.000000
IMP_DEBTINC	34.818262

2.1 Decision Trees

2.1.1 Develop a decision tree to predict the probability of default

2.1.2 Split Data

[ ]:
```
"""
SPLIT DATA
"""
# remove target data
X = df.copy() ### make exact copy of data
X = X.drop( TARGET_F, axis=1 ) ## drop target
X = X.drop( TARGET_A, axis=1 )  ## drop amount
## X.head().T

Y = df[ [TARGET_F, TARGET_A] ]  ## add new target var
## Y.head().T
```

[ ]:
```
## Make train data. Take 80% for training and randomly take 20% out for test set
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, train_size=0.8,␣
 ↪test_size=0.2, random_state=1)  ## you can fiddle with this to get different␣
 ↪random numbers

print( "FLAG DATA" )
print( "TRAINING = ", X_train.shape )  ## what is the shape if training and␣
 ↪testing data
print( "TEST = ", X_test.shape )  ## will show 29 vars and then training␣
 ↪ammount.
```

```
FLAG DATA
TRAINING =  (4768, 29)
TEST =  (1192, 29)
```

[ ]:
```
## Make a flag var where target amounts are not missing. ~not
F = ~ Y_train[ TARGET_A ].isna() ## is set to TRUE, ~ makes it not true
## create a copy, create new variables, subsetting the x, y train to show only␣
 ↪the true values will go into W. W is input data of only defaults. Z is␣
 ↪subeset of Y.
W_train = X_train[F].copy() ## X train is input
Z_train = Y_train[F].copy()

### have a data set of only people who defaulted
F = ~ Y_test[ TARGET_A ].isna()
W_test = X_test[F].copy()
Z_test = Y_test[F].copy()

## Show me Z training and test data
print( Z_train.describe() )
print( Z_test.describe() )
print( "\n\n")

### Y_train.head().T ## see who defaulted and for how much
```

```
       TARGET_BAD_FLAG  TARGET_LOSS_AMT
count            941.0       941.000000
mean               1.0     13421.645058
std                0.0     10662.481428
min                1.0       224.000000
25%                1.0      5817.000000
50%                1.0     10959.000000
75%                1.0     17635.000000
max                1.0     73946.000000
       TARGET_BAD_FLAG  TARGET_LOSS_AMT
count            248.0       248.000000
mean               1.0     13387.758065
std                0.0     11508.703991
min                1.0       320.000000
25%                1.0      5214.500000
50%                1.0     11336.500000
75%                1.0     16734.000000
max                1.0     78987.000000
```

2.1.3 Actually make decision tree

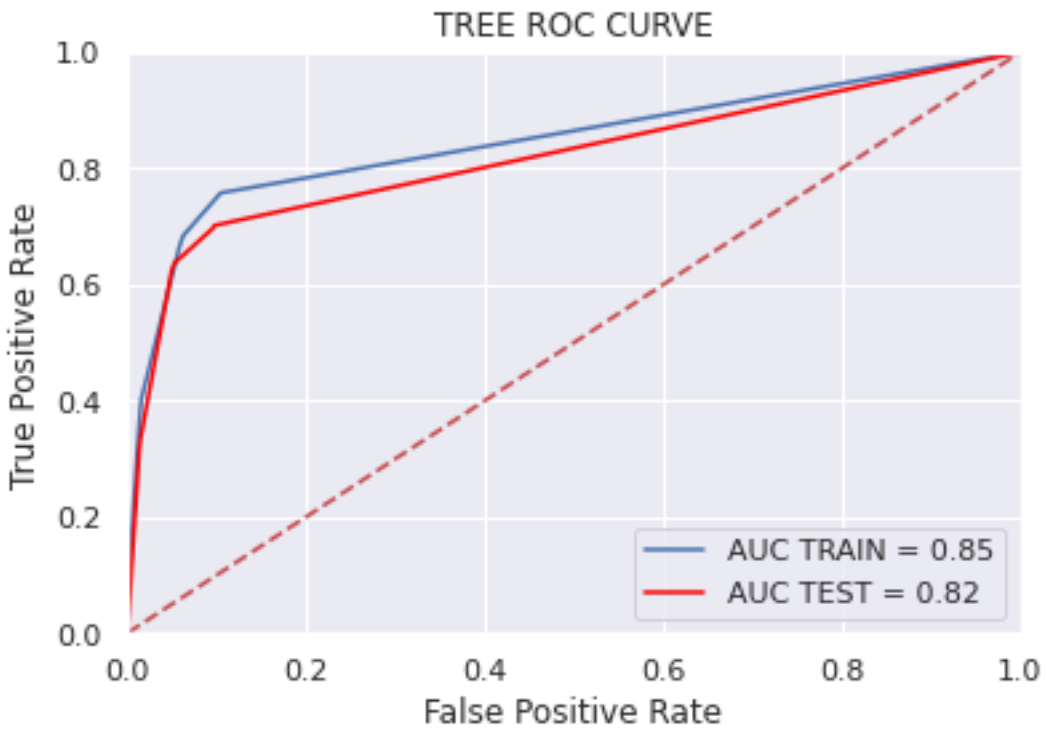
[ ]:
```
fm01_Tree = tree.DecisionTreeClassifier( max_depth=3 ) ## how many levels deep␣
 ↪do you want to go. you could also () and let python decide whats good
fm01_Tree = fm01_Tree.fit( X_train, Y_train[ TARGET_F ] )

Y_Pred_train = fm01_Tree.predict(X_train)
Y_Pred_test = fm01_Tree.predict(X_test)


print("\n=============\n")
print("DECISION TREE\n")
print("Probability of default")
print("Accuracy Train:",metrics.accuracy_score(Y_train[TARGET_F],␣
 ↪Y_Pred_train)) ## Ytrain is real data Ypred is what you predicted
print("Accuracy Test:",metrics.accuracy_score(Y_test[TARGET_F], Y_Pred_test))
print("\n")
```

```
=============

DECISION TREE

Probability of default
Accuracy Train: 0.8873741610738255
Accuracy Test: 0.8825503355704698
```

2.1.4 ROC Curve

[ ]:
```
### pred probibility value
probs = fm01_Tree.predict_proba(X_train)
# probs[0:5]
p1 = probs[:,1]  ## column zero and one.
fpr_train, tpr_train, threshold = metrics.roc_curve( Y_train[TARGET_F], p1)
roc_auc_train = metrics.auc(fpr_train, tpr_train)

probs = fm01_Tree.predict_proba(X_test)
p1 = probs[:,1]
fpr_test, tpr_test, threshold = metrics.roc_curve( Y_test[TARGET_F], p1)
roc_auc_test = metrics.auc(fpr_test, tpr_test)

### save to make comparisons later
fpr_tree = fpr_test
tpr_tree = tpr_test
auc_tree = roc_auc_test
```

[ ]:
```
plt.title('TREE ROC CURVE')
plt.plot(fpr_train, tpr_train, 'b', label = 'AUC TRAIN = %0.2f' % roc_auc_train)
plt.plot(fpr_test, tpr_test, 'b', label = 'AUC TEST = %0.2f' % roc_auc_test,␣
 ↪color="red")
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



2.1.5 Show me decision tree in graphviz

[ ]:
```
## What vars are important? we might have hundreds of vars. whats important?

def getTreeVars( TREE, varNames ) :
    tree_ = TREE.tree_
    varName = [ varNames[i] if i != _tree.TREE_UNDEFINED else "undefined!" for␣
 ↪i in tree_.feature ]

    nameSet = set()
    for i in tree_.feature :
        if i != _tree.TREE_UNDEFINED :
            nameSet.add( i )
    nameList = list( nameSet )
    parameter_list = list()
    for i in nameList :
        parameter_list.append( varNames[i] )
    return parameter_list

feature_cols = list( X.columns.values )
vars_tree_flag = getTreeVars( fm01_Tree, feature_cols )

### List the variables included in the decision tree that predict loss amount.
```

```
for i in vars_tree_flag :
  print(i)
```

```
M_DEROG
IMP_DELINQ
IMP_CLAGE
M_DEBTINC
IMP_DEBTINC
```

[ ]:
```
feature_cols = list( X.columns.values )
tree.export_graphviz(fm01_Tree,out_file='tree_f.txt',filled=True, rounded=True,␣
 ↪feature_names = feature_cols, impurity=False, class_names=["Good","Bad"]  )
vars_tree_flag = getTreeVars( fm01_Tree, feature_cols )
```

2.1.6 What vars are predictive?

2.2 Develop a decision tree to predict the loss amount assuming that the loan defaults

2.2.1 Losses, regression tree

[ ]:
```
### Losses, regression tree
amt_m01_Tree = tree.DecisionTreeRegressor( max_depth= 4 )
amt_m01_Tree = amt_m01_Tree.fit( W_train, Z_train[TARGET_A] )

Z_Pred_train = amt_m01_Tree.predict(W_train)
Z_Pred_test = amt_m01_Tree.predict(W_test)

print( "MEAN Train", Z_train[TARGET_A].mean() )
print( "MEAN Test", Z_test[TARGET_A].mean() )
print( " ----- \n\n" )
```

```
MEAN Train 13421.64505844846
MEAN Test 13387.758064516129
 -----
```

2.2.2 Calculate the RMSE for both the training data set and the test data set

[ ]:
```
## root mean sq error
RMSE_TRAIN = math.sqrt( metrics.mean_squared_error(Z_train[TARGET_A],␣
 ↪Z_Pred_train))
RMSE_TEST = math.sqrt( metrics.mean_squared_error(Z_test[TARGET_A],␣
 ↪Z_Pred_test))
```

```
print("TREE RMSE Train:", RMSE_TRAIN )
print("TREE RMSE Test:", RMSE_TEST )
```

```
TREE RMSE Train: 4561.00912099753
TREE RMSE Test: 5783.259789318173
```

[ ]:
```
### save for laer
RMSE_TREE = RMSE_TEST
```

2.2.3 Display the Decision Tree using a Graphviz program

[ ]:
```
### TREE
feature_cols = list( X.columns.values )
vars_tree_amt = getTreeVars( amt_m01_Tree, feature_cols )
tree.export_graphviz(amt_m01_Tree,out_file='tree_a.txt',filled=True,␣
 ↪rounded=True, feature_names = feature_cols, impurity=False, precision=0  )
```

2.2.4 List the variables included in the decision tree that predict loss amount.

[ ]:
```
print("\n")
for i in vars_tree_amt :
    print(i)
```

```
LOAN
z_IMP_REASON_DebtCon
IMP_VALUE
IMP_CLNO
M_DEBTINC
IMP_DEBTINC
```

2.3 Random Forests

2.3.1 Develop a Random Forest to predict the probability of default

[ ]:
```
"""
RANDOM FOREST
"""
def getEnsembleTreeVars( ENSTREE, varNames ) :
    importance = ENSTREE.feature_importances_
    index = np.argsort(importance)
    theList = []
    for i in index :
        imp_val = importance[i]
        if imp_val > np.average( ENSTREE.feature_importances_ ) :
```

```
            v = int( imp_val / np.max( ENSTREE.feature_importances_ ) * 100 )
            theList.append( ( varNames[i], v ) )
    theList = sorted(theList,key=itemgetter(1),reverse=True)
    return theList
```

[ ]:
```
### random forest classifier, how many trees you want? 25?
fm01_RF = RandomForestClassifier( n_estimators = 25, random_state=1 )
fm01_RF = fm01_RF.fit( X_train, Y_train[ TARGET_F ] )

# predict default or not
Y_Pred_train = fm01_RF.predict(X_train)
Y_Pred_test = fm01_RF.predict(X_test)
```

2.3.2 Calculate the accuracy of the model on both the training and test data set

[ ]:
```
## print acuracy of model

print("\n=============\n")
print("RANDOM FOREST\n")
print("Probability of crash")
print("Accuracy Train:",metrics.accuracy_score(Y_train[TARGET_F], Y_Pred_train))
print("Accuracy Test:",metrics.accuracy_score(Y_test[TARGET_F], Y_Pred_test))
print("\n")
```

```
=============

RANDOM FOREST

Probability of crash
Accuracy Train: 0.9995805369127517
Accuracy Test: 0.9135906040268457
```

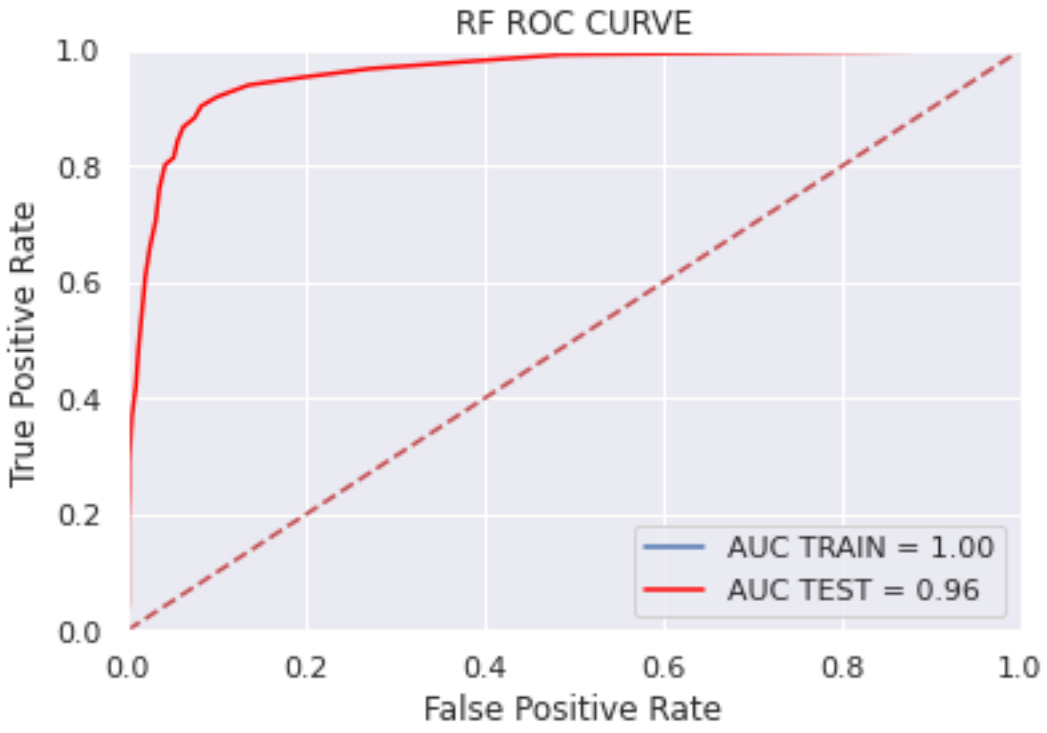
2.3.3 Create a graph that shows the ROC curves for both the training and test data set. Clearly label each curve and display the Area Under the ROC curve.

[ ]:
```
## Roc curve for random forests
## train
probs = fm01_RF.predict_proba(X_train)
p1 = probs[:,1]
fpr_train, tpr_train, threshold = metrics.roc_curve( Y_train[TARGET_F], p1)
roc_auc_train = metrics.auc(fpr_train, tpr_train)

## test
probs = fm01_RF.predict_proba(X_test)
```

```
p1 = probs[:,1]
fpr_test, tpr_test, threshold = metrics.roc_curve( Y_test[TARGET_F], p1)
roc_auc_test = metrics.auc(fpr_test, tpr_test)
```

[ ]:
```
## save test data for later comparison
fpr_RF = fpr_test
tpr_RF = tpr_test
auc_RF = roc_auc_test
```

[ ]:
```
plt.title('RF ROC CURVE')
plt.plot(fpr_train, tpr_train, 'b', label = 'AUC TRAIN = %0.2f' % roc_auc_train)
plt.plot(fpr_test, tpr_test, 'b', label = 'AUC TEST = %0.2f' % roc_auc_test,␣
↪color="red")
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



2.3.4 List the variables included in the Random Forest that predict loan default.

[ ]:
```
feature_cols = list( X.columns.values )
vars_RF_flag = getEnsembleTreeVars( fm01_RF, feature_cols )

for i in vars_RF_flag :  ## how often far was used in the model, that are above␣
 ↪average
    print( i )
```

```
('M_DEBTINC', 100)
('IMP_DEBTINC', 62)
('IMP_CLAGE', 40)
('IMP_DELINQ', 38)
('LOAN', 37)
('IMP_VALUE', 35)
('IMP_CLNO', 32)
('IMP_MORTDUE', 32)
('IMP_YOJ', 25)
('IMP_DEROG', 22)
('IMP_NINQ', 20)
```

2.4 Develop a Random Forest to predict the loss amount assuming that the loan defaults

[ ]:
```
### model to predict losses, regressor
amt_m01_RF = RandomForestRegressor(n_estimators = 100, random_state=1)
amt_m01_RF = amt_m01_RF.fit( W_train, Z_train[TARGET_A] )

Z_Pred_train = amt_m01_RF.predict(W_train)
Z_Pred_test = amt_m01_RF.predict(W_test)
```

2.4.1 Calculate the RMSE for both the training data set and the test data set

[ ]:
```
RMSE_TRAIN = math.sqrt( metrics.mean_squared_error(Z_train[TARGET_A],␣
 ↪Z_Pred_train))
RMSE_TEST = math.sqrt( metrics.mean_squared_error(Z_test[TARGET_A],␣
 ↪Z_Pred_test))

print("RF RMSE Train:", RMSE_TRAIN )
print("RF RMSE Test:", RMSE_TEST )

RMSE_RF = RMSE_TEST
```

```
RF RMSE Train: 1238.42831958866
RF RMSE Test: 3272.2093665087723
```

2.4.2 List the variables included in the Random Forest that predict loss amount.

[ ]:
```
feature_cols = list( X.columns.values )
vars_RF_amt = getEnsembleTreeVars( amt_m01_RF, feature_cols )

for i in vars_RF_amt :
    print( i )
```

```
('LOAN', 100)
('IMP_CLNO', 12)
('IMP_DEBTINC', 5)
```

2.5 Gradient Boosting

2.5.1 Develop a Gradient Boosting model to predict the probability of default

[ ]:
```
fm01_GB = GradientBoostingClassifier( random_state=1 )
fm01_GB = fm01_GB.fit( X_train, Y_train[ TARGET_F ] )

Y_Pred_train = fm01_GB.predict(X_train)
Y_Pred_test = fm01_GB.predict(X_test)

print("\n==============\n")
print("GRADIENT BOOSTING\n")
print("Probability of crash")
print("Accuracy Train:",metrics.accuracy_score(Y_train[TARGET_F], Y_Pred_train))
print("Accuracy Test:",metrics.accuracy_score(Y_test[TARGET_F], Y_Pred_test))
print("\n")
```

```
==============

GRADIENT BOOSTING

Probability of crash
Accuracy Train: 0.9234479865771812
Accuracy Test: 0.9043624161073825
```

2.5.2 Calculate the accuracy of the model on both the training and test data set

[ ]:
```
RMSE_TRAIN = math.sqrt( metrics.mean_squared_error(Z_train[TARGET_A],␣
 ↪Z_Pred_train))
RMSE_TEST = math.sqrt( metrics.mean_squared_error(Z_test[TARGET_A],␣
 ↪Z_Pred_test))

print("GB RMSE Train:", RMSE_TRAIN )
print("GB RMSE Test:", RMSE_TEST )
```

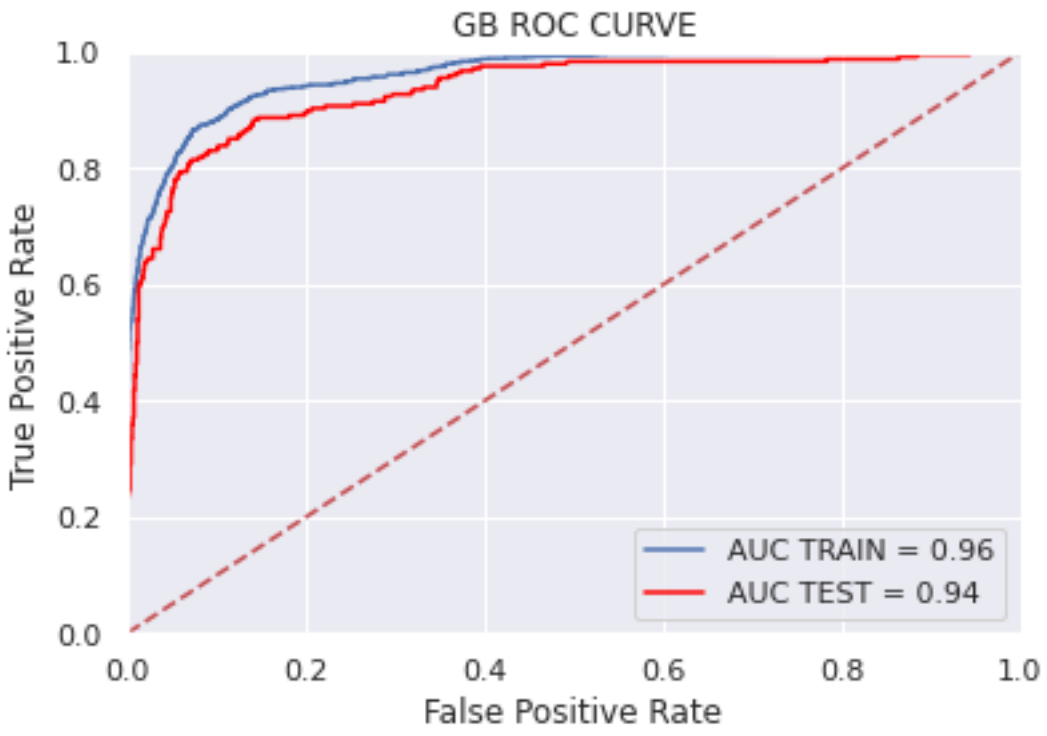
```
GB RMSE Train: 1238.42831958866
GB RMSE Test: 3272.2093665087723
```

2.5.3 Create a graph that shows the ROC curves for both the training and test data set. Clearly label each curve and display the Area Under the ROC curve.

[ ]:
```
## train
probs = fm01_GB.predict_proba(X_train)
p1 = probs[:,1]
fpr_train, tpr_train, threshold = metrics.roc_curve( Y_train[TARGET_F], p1)
roc_auc_train = metrics.auc(fpr_train, tpr_train)

## test
probs = fm01_GB.predict_proba(X_test)
p1 = probs[:,1]
fpr_test, tpr_test, threshold = metrics.roc_curve( Y_test[TARGET_F], p1)
roc_auc_test = metrics.auc(fpr_test, tpr_test)
```

[ ]:
```
## save for special occasion
fpr_GB = fpr_test
tpr_GB = tpr_test
auc_GB = roc_auc_test
```

[ ]:
```
plt.title('GB ROC CURVE')
plt.plot(fpr_train, tpr_train, 'b', label = 'AUC TRAIN = %0.2f' % roc_auc_train)
plt.plot(fpr_test, tpr_test, 'b', label = 'AUC TEST = %0.2f' % roc_auc_test,␣
 ↪color="red")
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



2.5.4 List the variables included in the Gradient Boosting that predict loan default.

[ ]:
```
feature_cols = list( X.columns.values )
vars_GB_flag = getEnsembleTreeVars( fm01_GB, feature_cols )

for i in vars_GB_flag :
    print(i)
```

```
('M_DEBTINC', 100)
('IMP_DEBTINC', 29)
('IMP_DELINQ', 19)
('IMP_CLAGE', 14)
('IMP_DEROG', 7)
```

2.6 Develop a Gradient Boosting to predict the loss amount assuming that the loan defaults

[ ]:
```
amt_m01_GB = GradientBoostingRegressor(random_state=1)
amt_m01_GB = amt_m01_GB.fit( W_train, Z_train[TARGET_A] )

Z_Pred_train = amt_m01_GB.predict(W_train)
Z_Pred_test = amt_m01_GB.predict(W_test)
```

2.6.1 Calculate the RMSE for both the training data set and the test data set

[ ]:
```
RMSE_TRAIN = math.sqrt( metrics.mean_squared_error(Z_train[TARGET_A],␣
 ↪Z_Pred_train))
RMSE_TEST = math.sqrt( metrics.mean_squared_error(Z_test[TARGET_A],␣
 ↪Z_Pred_test))

print("GB RMSE Train:", RMSE_TRAIN )
print("GB RMSE Test:", RMSE_TEST )

RMSE_GB = RMSE_TEST
```

```
GB RMSE Train: 1190.372223849314
GB RMSE Test: 2641.734547206168
```

2.6.2 List the variables included in the Gradient Boosting that predict loss amount.

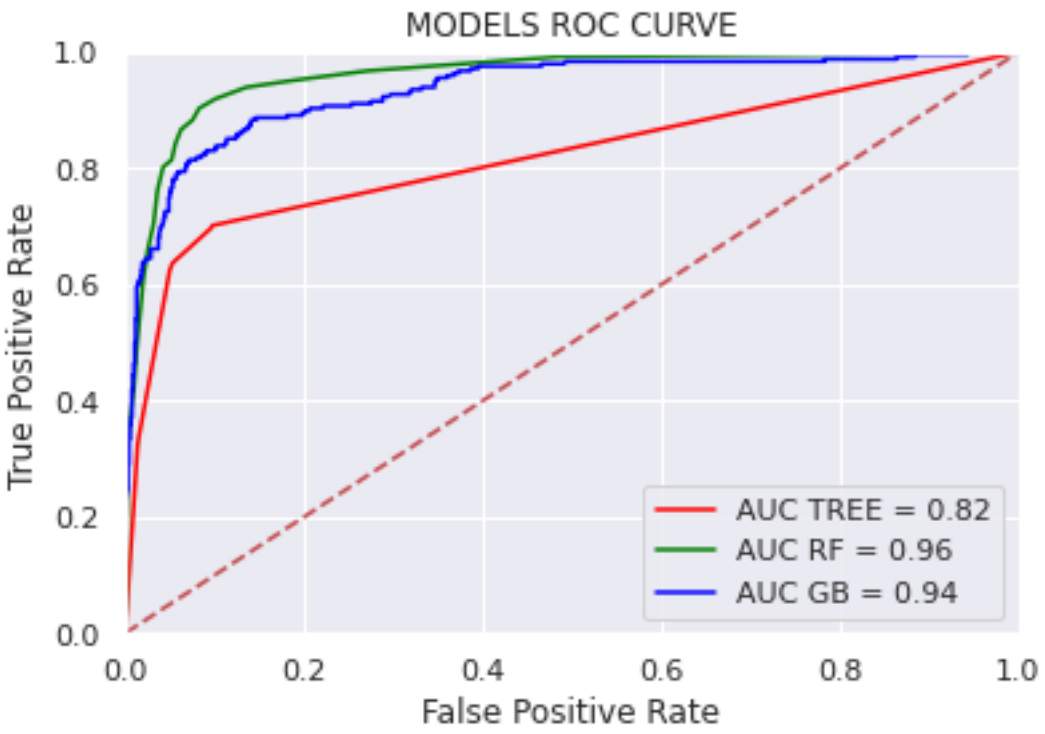
[ ]:
```
feature_cols = list( X.columns.values )
vars_GB_amt = getEnsembleTreeVars( amt_m01_GB, feature_cols )

for i in vars_GB_amt :
    print(i)
```

```
('LOAN', 100)
('IMP_CLNO', 14)
('IMP_DEBTINC', 5)
('M_DEBTINC', 5)
```

2.7 ROC Curves

2.7.1 Generate a ROC curve for the Decision Tree, Random Forest, and Gradient Boosting models using the Test Data Set

[ ]:
```
plt.title('MODELS ROC CURVE')
plt.plot(fpr_tree, tpr_tree, 'b', label = 'AUC TREE = %0.2f' % auc_tree,␣
 ↪color="red")
plt.plot(fpr_RF, tpr_RF, 'b', label = 'AUC RF = %0.2f' % auc_RF, color="green")
plt.plot(fpr_GB, tpr_GB, 'b', label = 'AUC GB = %0.2f' % auc_GB, color="blue")
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



2.8 Interpretations

Discuss any observations and interpretations you have based upon the results of your analysis.

Include a discussion of the Decision Tree diagrams. Do they appear to make sense?

Which variables appear to be most predictive of loan default? Do they make sense?

Which variables appear to be most predictive of loss amount? Do they make sense?

If you were to select one of these models to put into production, which would it be? Why would you select this model?

Note: Simply dumping bunch of output into a file without any thoughtful interpretation will result in a low grade. In the corporate world, you are expected to provide interpretation of any analysis.

Assignment 03: Regression

Assignment 03: Regression

[ ]:
```
import math
import itertools

import pandas as pd
import numpy as np
from operator import itemgetter
```

```
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.preprocessing import OneHotEncoder

from sklearn.model_selection import train_test_split
import sklearn.metrics as metrics

from sklearn import tree
from sklearn.tree import _tree

from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import RandomForestClassifier

from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import GradientBoostingClassifier

from sklearn.linear_model import LinearRegression
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix

from mlxtend.feature_selection import SequentialFeatureSelector as SFS
from mlxtend.plotting import plot_sequential_feature_selection as plot_sfs

import tensorflow as tf

from sklearn.preprocessing import MinMaxScaler

import warnings
warnings.filterwarnings("ignore")

sns.set()
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
pd.set_option('display.width', None)
pd.set_option('display.max_colwidth', None)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/externals/joblib/__init__.py:15:
FutureWarning: sklearn.externals.joblib is deprecated in 0.21 and will be
removed in 0.23. Please import this functionality directly from joblib, which
can be installed with: pip install joblib. If this warning is raised when
loading pickled models, you may need to re-serialize those models with scikit-
learn 0.21+.
  warnings.warn(msg, category=FutureWarning)
```

##Logistic Regression

[ ]:
```
import math
import pandas as pd
import numpy as np
from operator import itemgetter


import matplotlib.pyplot as plt
import seaborn as sns


from sklearn.model_selection import train_test_split
import sklearn.metrics as metrics


from sklearn import tree
from sklearn.tree import _tree

from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import RandomForestClassifier

from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import GradientBoostingClassifier

from sklearn.linear_model import LinearRegression
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix

from mlxtend.feature_selection import SequentialFeatureSelector as SFS
from mlxtend.plotting import plot_sequential_feature_selection as plot_sfs

import warnings
warnings.filterwarnings("ignore")



sns.set()
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
pd.set_option('display.width', None)
pd.set_option('display.max_colwidth', None)
```

[ ]:
```
"""
MODEL ACCURACY METRICS
"""

def getProbAccuracyScores( NAME, MODEL, X, Y ) :
    pred = MODEL.predict( X )
```

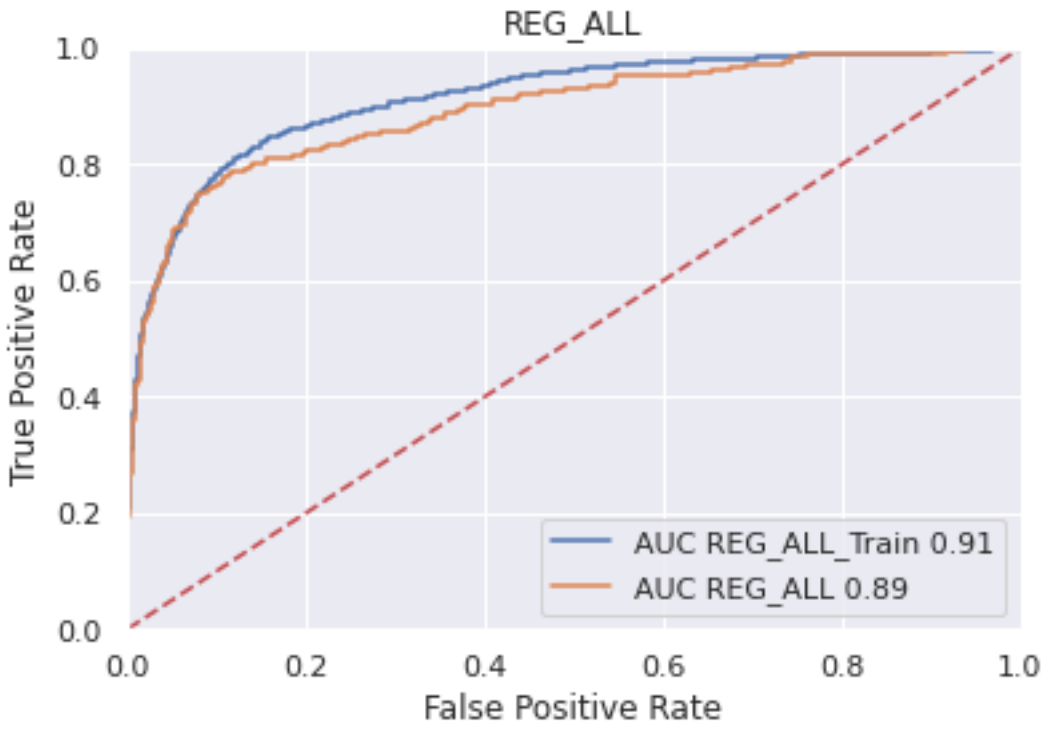
```
    probs = MODEL.predict_proba( X )
    acc_score = metrics.accuracy_score(Y, pred)
    p1 = probs[:,1]
    fpr, tpr, threshold = metrics.roc_curve( Y, p1)
    auc = metrics.auc(fpr,tpr)
    return [NAME, acc_score, fpr, tpr, auc]

def print_ROC_Curve( TITLE, LIST ) :
    fig = plt.figure(figsize=(6,4))
    plt.title( TITLE )
    for theResults in LIST :
        NAME = theResults[0]
        fpr = theResults[2]
        tpr = theResults[3]
        auc = theResults[4]
        theLabel = "AUC " + NAME + ' %0.2f' % auc
        plt.plot(fpr, tpr, label = theLabel )
    plt.legend(loc = 'lower right')
    plt.plot([0, 1], [0, 1],'r--')
    plt.xlim([0, 1])
    plt.ylim([0, 1])
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')
    plt.show()

def print_Accuracy( TITLE, LIST ) :
    print( TITLE )
    print( "======" )
    for theResults in LIST :
        NAME = theResults[0]
        ACC = theResults[1]
        print( NAME, " = ", ACC )
    print( "------\n\n" )

def getAmtAccuracyScores( NAME, MODEL, X, Y ) :
    pred = MODEL.predict( X )
    MEAN = Y.mean()
    RMSE = math.sqrt( metrics.mean_squared_error( Y, pred))
    return [NAME, RMSE, MEAN]
```

[ ]:
```
### Define getCoefLogit and getCoefLinear

def getCoefLogit( MODEL, TRAIN_DATA ) :
    varNames = list( TRAIN_DATA.columns.values )
    coef_dict = {}
    coef_dict["INTERCEPT"] = MODEL.intercept_[0]
    for coef, feat in zip(MODEL.coef_[0],varNames):
```

```
        coef_dict[feat] = coef
    print("\nDEFAULT")
    print("---------")
    print("Total Variables: ", len( coef_dict ) )
    for i in coef_dict :
        print( i, " = ", coef_dict[i]  )



def getCoefLinear( MODEL, TRAIN_DATA ) :
    varNames = list( TRAIN_DATA.columns.values )
    coef_dict = {}
    coef_dict["INTERCEPT"] = MODEL.intercept_
    for coef, feat in zip(MODEL.coef_,varNames):
        coef_dict[feat] = coef
    print("\nLOSS")
    print("---------")
    print("Total Variables: ", len( coef_dict ) )
    for i in coef_dict :
        print( i, " = ", coef_dict[i]  )
```

###Develop a logistic regression model to determine the probability of a loan default. Use all of the variables.

[ ]:
```
## LOG REG ALL
WHO = "REG_ALL"

CLM = LogisticRegression( solver='newton-cg', max_iter=1000 )
CLM = CLM.fit( X_train, Y_train[ TARGET_F ] )

TRAIN_CLM = getProbAccuracyScores( WHO + "_Train", CLM, X_train, Y_train[␣
 ↪TARGET_F ] )
TEST_CLM = getProbAccuracyScores( WHO, CLM, X_test, Y_test[ TARGET_F ] )

print_ROC_Curve( WHO, [ TRAIN_CLM, TEST_CLM ] )
print_Accuracy( WHO + " CLASSIFICATION ACCURACY", [ TRAIN_CLM, TEST_CLM ] )

varNames = list( X_train.columns.values )

### How many variables are there to begin with?
print(len(varNames))

REG_ALL_CLM_COEF = getCoefLogit( CLM, X_train )
REG_ALL_CLM = TEST_CLM.copy()
```



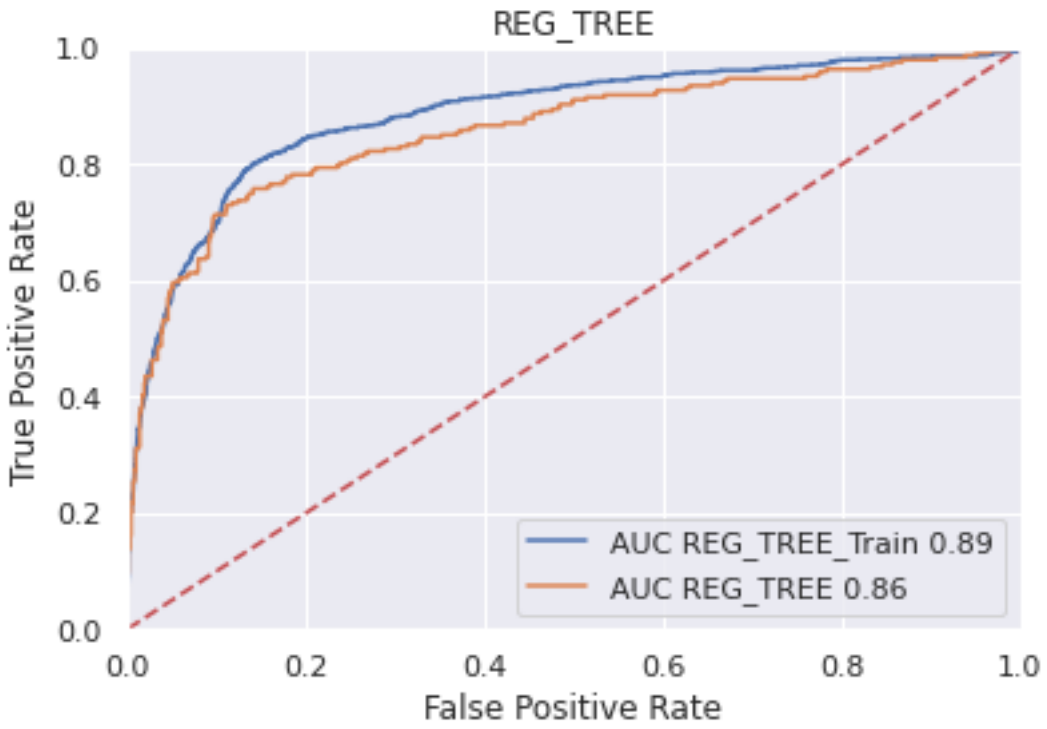

```
REG_ALL CLASSIFICATION ACCURACY
======
REG_ALL_Train  =  0.8936661073825504
REG_ALL  =  0.8901006711409396
------


29

DEFAULT
---------
Total Variables:  30
INTERCEPT  =  -5.311734020622573
LOAN  =  -4.340942519019282e-06
z_IMP_REASON_DebtCon  =  -0.08293101067456012
z_IMP_REASON_HomeImp  =  0.07695499189819174
z_IMP_REASON_MISSING  =  -0.0874722268606013
z_IMP_JOB_MISSING  =  -1.3766329376561408
z_IMP_JOB_Mgr  =  0.1289955631540239
z_IMP_JOB_Office  =  -0.5027387435072825
z_IMP_JOB_Other  =  0.1932150095163468
z_IMP_JOB_ProfExe  =  -0.07885171139996583
z_IMP_JOB_Sales  =  1.2251894883879153
```

```
z_IMP_JOB_Self  =  0.3173750858681247
M_MORTDUE  =  0.24431172633388964
IMP_MORTDUE  =  -2.590132278207286e-06
M_VALUE  =  3.949235505038842
IMP_VALUE  =  2.7977517644692995e-06
M_YOJ  =  -0.6536297245517879
IMP_YOJ  =  -0.01585151127720914
M_DEROG  =  -1.7600556632649478
IMP_DEROG  =  0.5242192958658574
M_DELINQ  =  -0.306043894474021
IMP_DELINQ  =  0.7945598510385935
M_CLAGE  =  1.1282370854937418
IMP_CLAGE  =  -0.005329847265471324
M_NINQ  =  0.024783260369856164
IMP_NINQ  =  0.1405859820286056
M_CLNO  =  2.111900457816619
IMP_CLNO  =  -0.013342955479501118
M_DEBTINC  =  2.65134869971354
IMP_DEBTINC  =  0.1001991368709554
```

###Develop a logistic regression model to determine the probability of a loan default. Use the variables that were selected by a DECISION TREE.

[ ]:
```
"""
LOG REGRESSION DECISION TREE
"""

def getTreeVars( TREE, varNames ) :
    tree_ = TREE.tree_
    varName = [ varNames[i] if i != _tree.TREE_UNDEFINED else "undefined!" for␣
 ↪i in tree_.feature ]

    nameSet = set()
    for i in tree_.feature :
        if i != _tree.TREE_UNDEFINED :
            nameSet.add( i )
    nameList = list( nameSet )
    parameter_list = list()
    for i in nameList :
        parameter_list.append( varNames[i] )
    return parameter_list



WHO = "REG_TREE"

CLM = LogisticRegression( solver='newton-cg', max_iter=1000 )
CLM = CLM.fit( X_train[vars_tree_flag], Y_train[ TARGET_F ] )
```

```
TRAIN_CLM = getProbAccuracyScores( WHO + "_Train", CLM,␣
 ↪X_train[vars_tree_flag], Y_train[ TARGET_F ] )
TEST_CLM = getProbAccuracyScores( WHO, CLM, X_test[vars_tree_flag], Y_test[␣
 ↪TARGET_F ] )

print_ROC_Curve( WHO, [ TRAIN_CLM, TEST_CLM ] )
print_Accuracy( WHO + " CLASSIFICATION ACCURACY", [ TRAIN_CLM, TEST_CLM ] )

varNames = list( X_train.columns.values )

REG_TREE_CLM_COEF = getCoefLogit( CLM, X_train[vars_tree_flag] )

REG_TREE_CLM = TEST_CLM.copy()

TREE_CLM = TEST_CLM.copy()

### NEED this later for STEPWISE vars list. It comes from the DT
```




```
REG_TREE CLASSIFICATION ACCURACY
======
REG_TREE_Train  =  0.8758389261744967
REG_TREE  =  0.8741610738255033
```

```
------
```

```
DEFAULT
---------
Total Variables:  6
INTERCEPT  =  -4.942507685020032
M_DEROG  =  -0.8321416484146855
IMP_DELINQ  =  0.7409836248453477
IMP_CLAGE  =  -0.0063706637859314835
M_DEBTINC  =  2.825781584330456
IMP_DEBTINC  =  0.0938354700697999
```

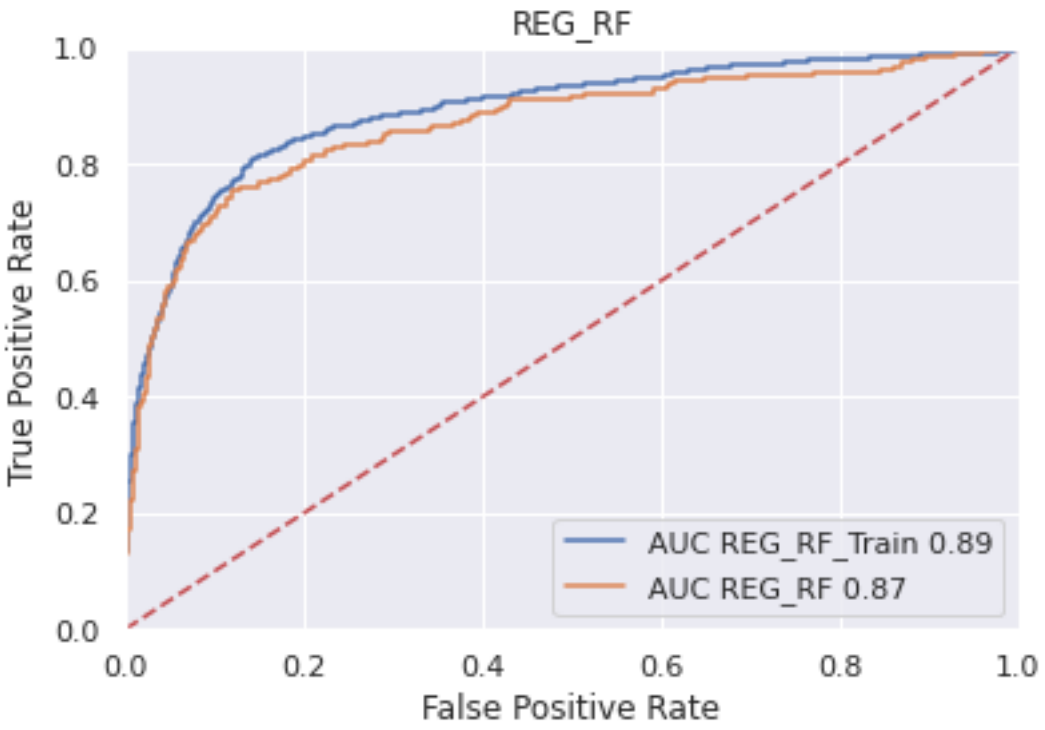
###Develop a logistic regression model to determine the probability of a loan default. Use the variables that were selected by a RANDOM FOREST.

[ ]:
```
"""
LOG REGRESSION RANDOM FOREST
"""

WHO = "REG_RF"


print("\n\n")
RF_flag = []
for i in vars_RF_flag :
    print(i)
    theVar = i[0]
    RF_flag.append( theVar )

print("\n\n")
RF_amt = []
for i in vars_RF_amt :
    print(i)
    theVar = i[0]
    RF_amt.append( theVar )


CLM = LogisticRegression( solver='newton-cg', max_iter=1000 )
CLM = CLM.fit( X_train[RF_flag], Y_train[ TARGET_F ] )

TRAIN_CLM = getProbAccuracyScores( WHO + "_Train", CLM, X_train[RF_flag],␣
 ↪Y_train[ TARGET_F ] )
TEST_CLM = getProbAccuracyScores( WHO, CLM, X_test[RF_flag], Y_test[ TARGET_F ]␣
 ↪)

print_ROC_Curve( WHO, [ TRAIN_CLM, TEST_CLM ] )
```

```
print_Accuracy( WHO + " CLASSIFICATION ACCURACY", [ TRAIN_CLM, TEST_CLM ] )

REG_RF_CLM_COEF = getCoefLogit( CLM, X_train[RF_flag] )

REG_RF_CLM = TEST_CLM.copy()
RF_CLM = TEST_CLM.copy()
```

```
('M_DEBTINC', 100)
('IMP_DEBTINC', 62)
('IMP_CLAGE', 40)
('IMP_DELINQ', 38)
('LOAN', 37)
('IMP_VALUE', 35)
('IMP_CLNO', 32)
('IMP_MORTDUE', 32)
('IMP_YOJ', 25)
('IMP_DEROG', 22)
('IMP_NINQ', 20)
```

```
('LOAN', 100)
('IMP_CLNO', 12)
('IMP_DEBTINC', 5)
```




```
REG_RF CLASSIFICATION ACCURACY
======
REG_RF_Train  =  0.8783557046979866
REG_RF  =  0.8741610738255033
------
```

```
DEFAULT
---------
Total Variables:  12
INTERCEPT  =  -5.021941157003482
M_DEBTINC  =  2.747713669249404
IMP_DEBTINC  =  0.09398734546324619
IMP_CLAGE  =  -0.005217996610183003
IMP_DELINQ  =  0.6989593526928543
LOAN  =  -5.983755463380205e-06
IMP_VALUE  =  2.1088815170575974e-06
IMP_CLNO  =  -0.01782453831458048
IMP_MORTDUE  =  -1.4287679543938208e-06
IMP_YOJ  =  -0.011453290870773715
IMP_DEROG  =  0.5678724008094669
IMP_NINQ  =  0.11315185798874619
```

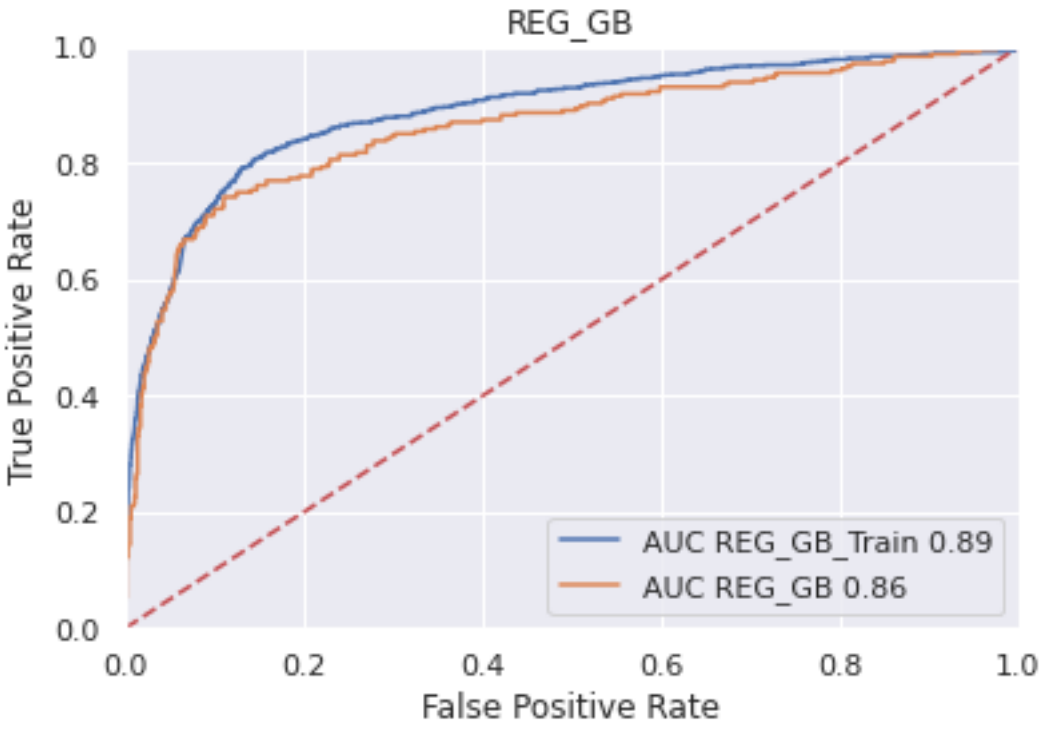
###Develop a logistic regression model to determine the probability of a loan default. Use the variables that were selected by a GRADIENT BOOSTING model.

[ ]:
```
"""
REGRESSION GRADIENT BOOSTING
"""

WHO = "REG_GB"


print("\n\n")
GB_flag = []
for i in vars_GB_flag :
    print(i)
    theVar = i[0]
    GB_flag.append( theVar )

print("\n\n")
GB_amt = []
for i in vars_GB_amt :
    print(i)
    theVar = i[0]
    GB_amt.append( theVar )


CLM = LogisticRegression( solver='newton-cg', max_iter=1000 )
CLM = CLM.fit( X_train[GB_flag], Y_train[ TARGET_F ] )

TRAIN_CLM = getProbAccuracyScores( WHO + "_Train", CLM, X_train[GB_flag],␣
 ↪Y_train[ TARGET_F ] )
TEST_CLM = getProbAccuracyScores( WHO, CLM, X_test[GB_flag], Y_test[ TARGET_F ]␣
 ↪)

print_ROC_Curve( WHO, [ TRAIN_CLM, TEST_CLM ] )
print_Accuracy( WHO + " CLASSIFICATION ACCURACY", [ TRAIN_CLM, TEST_CLM ] )

REG_GB_CLM_COEF = getCoefLogit( CLM, X_train[GB_flag] )

REG_GB_CLM = TEST_CLM.copy()
GB_CLM = TEST_CLM.copy()
```

```
('M_DEBTINC', 100)
('IMP_DEBTINC', 29)
('IMP_DELINQ', 19)
('IMP_CLAGE', 14)
```

```
('IMP_DEROG', 7)
```

```
('LOAN', 100)
('IMP_CLNO', 14)
('IMP_DEBTINC', 5)
('M_DEBTINC', 5)
```




```
REG_GB CLASSIFICATION ACCURACY
======
REG_GB_Train  =  0.8770973154362416
REG_GB  =  0.8699664429530202
------
```

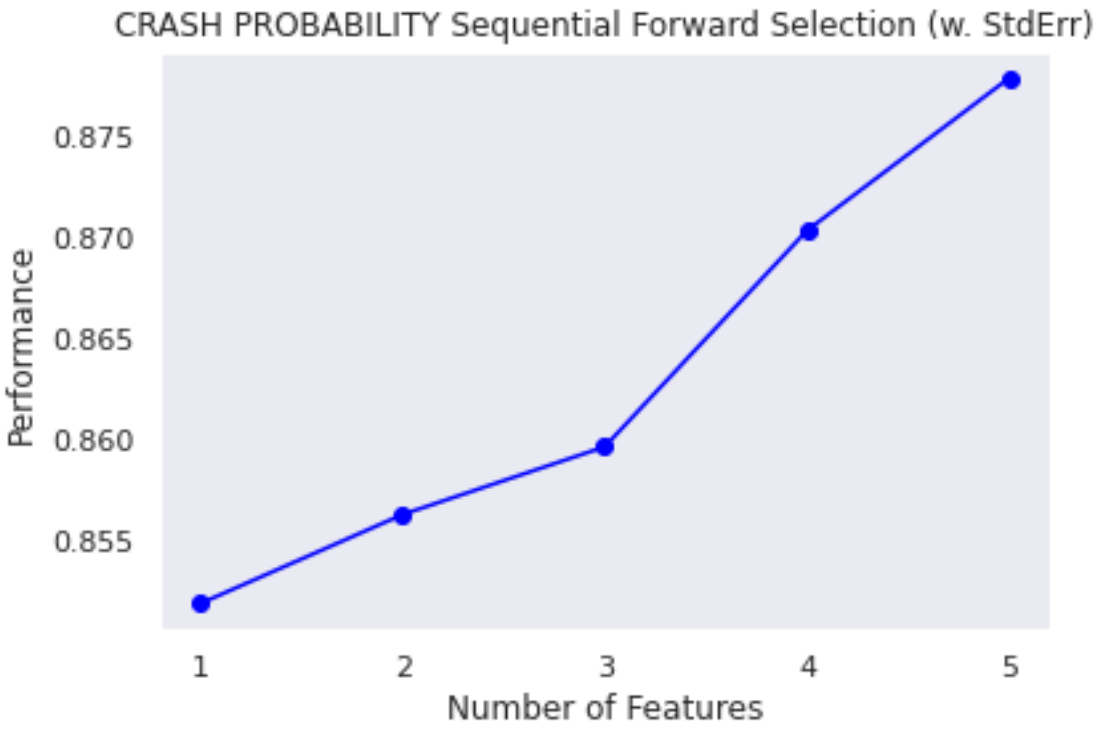
```
DEFAULT
---------
Total Variables:  6
INTERCEPT  =  -5.17479485344813
M_DEBTINC  =  2.7866477905410574
IMP_DEBTINC  =  0.09387326511791212
IMP_DELINQ  =  0.6677893066857544
```

```
IMP_CLAGE   =  -0.006211691874228503
IMP_DEROG   =  0.574138020955196
```

###Develop a logistic regression model to determine the probability of a loan default. Use the variables that were selected by STEPWISE SELECTION.

[ ]:
```
"""
REGRESSION STEPWISE
"""

U_train = X_train[ vars_tree_flag ]
stepVarNames = list( U_train.columns.values )
maxCols = U_train.shape[1]

sfs = SFS( LogisticRegression( solver='newton-cg', max_iter=100 ),
           k_features=( 1, maxCols ),
           forward=True,
           floating=False,
           cv=3
           )
sfs.fit(U_train.values, Y_train[ TARGET_F ].values)

theFigure = plot_sfs(sfs.get_metric_dict(), kind=None )
plt.title('CRASH PROBABILITY Sequential Forward Selection (w. StdErr)')
plt.grid()
plt.show()

dfm = pd.DataFrame.from_dict( sfs.get_metric_dict()).T
dfm = dfm[ ['feature_names', 'avg_score'] ]
dfm.avg_score = dfm.avg_score.astype(float)

print(" ..................... ")
maxIndex = dfm.avg_score.argmax()
print("argmax")
print( dfm.iloc[ maxIndex, ] )
print(" ..................... ")

stepVars = dfm.iloc[ maxIndex, ]
stepVars = stepVars.feature_names
print( stepVars )

finalStepVars = []
for i in stepVars :
    index = int(i)
    try :
        theName = stepVarNames[ index ]
        finalStepVars.append( theName )
    except :
```

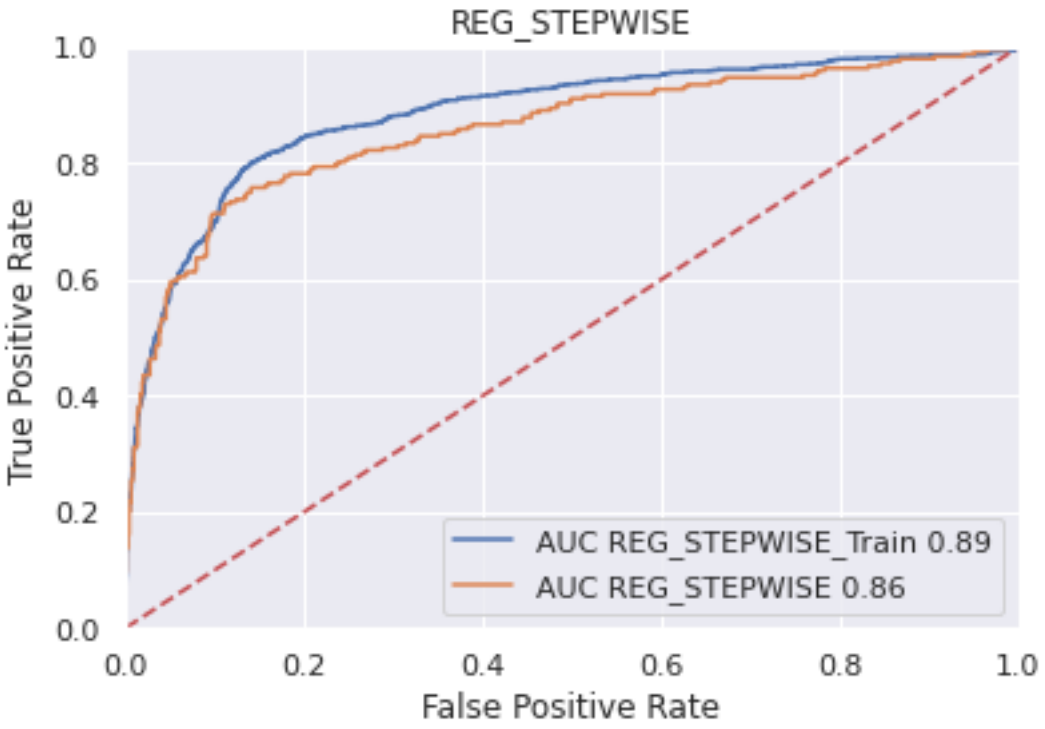
```
        pass

for i in finalStepVars :
    print(i)

U_train = X_train[ finalStepVars ]
U_test = X_test[ finalStepVars ]
```




```
 …
argmax
feature_names    (0, 1, 2, 3, 4)
avg_score               0.877937
Name: 5, dtype: object
 …
('0', '1', '2', '3', '4')
M_DEROG
IMP_DELINQ
IMP_CLAGE
M_DEBTINC
IMP_DEBTINC
```

[ ]:
```
"""
REGRESSION
```

```
"""

WHO = "REG_STEPWISE"

CLM = LogisticRegression( solver='newton-cg', max_iter=1000 )
CLM = CLM.fit( U_train, Y_train[ TARGET_F ] )

TRAIN_CLM = getProbAccuracyScores( WHO + "_Train", CLM, U_train, Y_train[␣
 ↪TARGET_F ] )
TEST_CLM = getProbAccuracyScores( WHO, CLM, U_test, Y_test[ TARGET_F ] )

print_ROC_Curve( WHO, [ TRAIN_CLM, TEST_CLM ] )
print_Accuracy( WHO + " CLASSIFICATION ACCURACY", [ TRAIN_CLM, TEST_CLM ] )

REG_ALL_CLM = TEST_CLM.copy()
REG_STEP_CLM = TEST_CLM.copy()
```

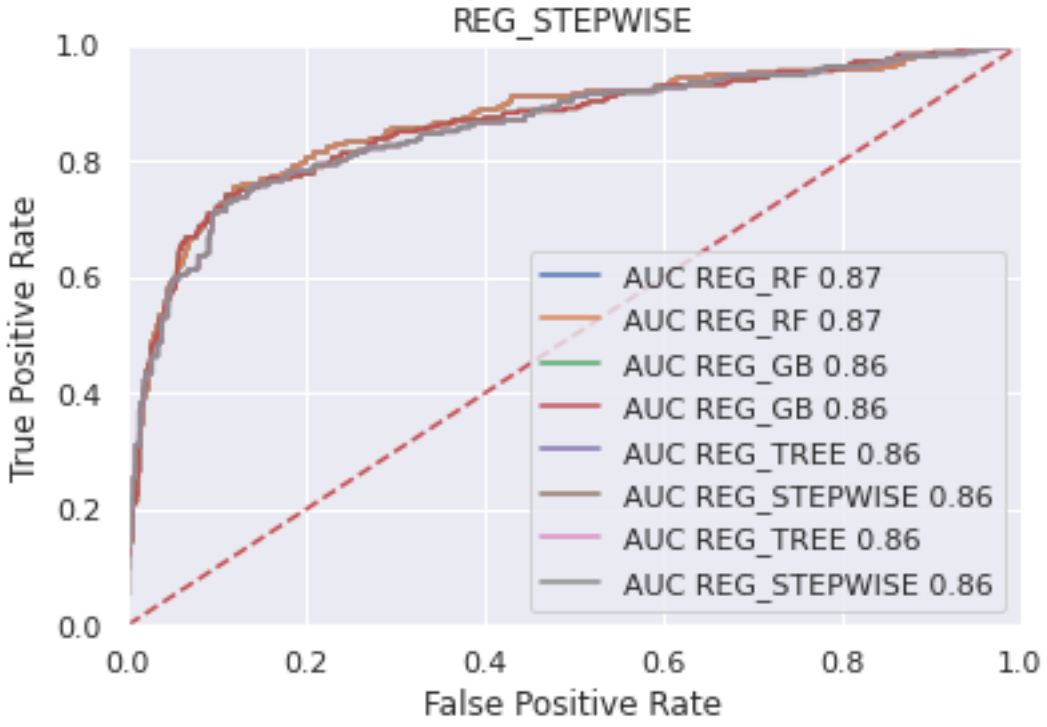



```
REG_STEPWISE CLASSIFICATION ACCURACY
======
REG_STEPWISE_Train  =  0.8758389261744967
REG_STEPWISE  =  0.8741610738255033
------
```

###FOR ALL MODELS…

2.8.1 Display a ROC curve for the test data with all your models on the same graph (tree based and regression).

```
[ ]: ALL_CLM = [ TREE_CLM, RF_CLM, GB_CLM, REG_ALL_CLM, REG_TREE_CLM, REG_RF_CLM,␣
      ↪REG_GB_CLM, REG_STEP_CLM ]

     ALL_CLM = sorted( ALL_CLM, key = lambda x: x[4], reverse=True )
     print_ROC_Curve( WHO, ALL_CLM )

     ALL_CLM = sorted( ALL_CLM, key = lambda x: x[1], reverse=True )
     print_Accuracy( "ALL CLASSIFICATION ACCURACY", ALL_CLM )
```




```
ALL CLASSIFICATION ACCURACY
======
REG_RF  =  0.8741610738255033
REG_RF  =  0.8741610738255033
REG_TREE  =  0.8741610738255033
REG_STEPWISE  =  0.8741610738255033
REG_TREE  =  0.8741610738255033
REG_STEPWISE  =  0.8741610738255033
```

```
REG_GB  =  0.8699664429530202
REG_GB  =  0.8699664429530202
------
```

2.9 Linear Regression

###Develop a linear regression model to determine the expected loss if the loan defaults. Use all of the variables.

[ ]:
```
# REG ALL
# LOSS from default

WHO = "REG_ALL"


AMT = LinearRegression()
AMT = AMT.fit( W_train, Z_train[TARGET_A] )

TRAIN_AMT = getAmtAccuracyScores( WHO + "_Train", AMT, W_train,␣
 ↪Z_train[TARGET_A] )
TEST_AMT = getAmtAccuracyScores( WHO, AMT, W_test, Z_test[TARGET_A] )
print_Accuracy( WHO + " RMSE ACCURACY", [ TRAIN_AMT, TEST_AMT ] )


varNames = list( X_train.columns.values )

REG_ALL_AMT_COEF = getCoefLinear( AMT, X_train )


REG_ALL_AMT = TEST_AMT.copy()
```

```
REG_ALL RMSE ACCURACY
======
REG_ALL_Train  =  3613.472655284998
REG_ALL  =  3493.138389711616
------
```

```
LOSS
---------
Total Variables:  30
INTERCEPT  =  -9191.044754955601
LOAN  =  0.7593493336317956
z_IMP_REASON_DebtCon  =  1356.3801730623113
z_IMP_REASON_HomeImp  =  -568.7781312512033
```

```
z_IMP_REASON_MISSING  =  -787.6020457255566
z_IMP_JOB_MISSING  =  954.5916788813136
z_IMP_JOB_Mgr  =  -733.4895440272803
z_IMP_JOB_Office  =  -507.4749574270879
z_IMP_JOB_Other  =  -417.87721527814995
z_IMP_JOB_ProfExe  =  -917.9573920981579
z_IMP_JOB_Sales  =  785.4546689525307
z_IMP_JOB_Self  =  836.7527609967965
M_MORTDUE  =  -630.3290566699718
IMP_MORTDUE  =  0.006818437769222669
M_VALUE  =  84.82079676912065
IMP_VALUE  =  -0.0047390677659035745
M_YOJ  =  -17.063000354304865
IMP_YOJ  =  -87.19176941480185
M_DEROG  =  797.9397669264924
IMP_DEROG  =  318.5786348672121
M_DELINQ  =  332.2852404137122
IMP_DELINQ  =  733.6384007196855
M_CLAGE  =  -5543.799635261365
IMP_CLAGE  =  -18.664938631871106
M_NINQ  =  -1116.8487119578404
IMP_NINQ  =  -65.04723643840293
M_CLNO  =  7512.904763664946
IMP_CLNO  =  211.57599243783608
M_DEBTINC  =  5599.6348359651265
IMP_DEBTINC  =  108.81151910025544
```

###Develop a linear regression model to determine the expected loss if the loan defaults. Use the variables that were selected by a DECISION TREE.

[ ]:
```
# REG DT
# LOSS from default

WHO = "REG_TREE"

AMT = LinearRegression()
AMT = AMT.fit( W_train[vars_tree_amt], Z_train[TARGET_A] )

TRAIN_AMT = getAmtAccuracyScores( WHO + "_Train", AMT, W_train[vars_tree_amt],␣
 ↪Z_train[TARGET_A] )
TEST_AMT = getAmtAccuracyScores( WHO, AMT, W_test[vars_tree_amt],␣
 ↪Z_test[TARGET_A] )
print_Accuracy( WHO + " RMSE ACCURACY", [ TRAIN_AMT, TEST_AMT ] )


varNames = list( X_train.columns.values )

REG_TREE_AMT_COEF = getCoefLinear( AMT, X_train[vars_tree_amt] )
```

```
REG_TREE_AMT = TEST_AMT.copy()
TREE_AMT = TEST_AMT.copy()
```

```
REG_TREE RMSE ACCURACY
======
REG_TREE_Train  =  4307.627172622767
REG_TREE  =  4307.245008917904
------
```

```
LOSS
---------
Total Variables:  7
INTERCEPT  =  -13074.63468145558
LOAN  =  0.7430228895734364
z_IMP_REASON_DebtCon  =  1927.9757387092066
IMP_VALUE  =  -0.004662869380808843
IMP_CLNO  =  236.6746215402237
M_DEBTINC  =  5775.1576313434225
IMP_DEBTINC  =  117.5266970785724
```

###Develop a linear regression model to determine the expected loss if the loan defaults. Use the variables that were selected by a RANDOM FOREST

[ ]:
```
# LOG REG RF
# LOSS from default

WHO = "REG_RF"

print("\n\n")
RF_amt = []
for i in vars_RF_amt :
    print(i)
    theVar = i[0]
    RF_amt.append( theVar )

AMT = LinearRegression()
AMT = AMT.fit( W_train[RF_amt], Z_train[TARGET_A] )

TRAIN_AMT = getAmtAccuracyScores( WHO + "_Train", AMT, W_train[RF_amt],␣
 ↪Z_train[TARGET_A] )
TEST_AMT = getAmtAccuracyScores( WHO, AMT, W_test[RF_amt], Z_test[TARGET_A] )
print_Accuracy( WHO + " RMSE ACCURACY", [ TRAIN_AMT, TEST_AMT ] )


REG_RF_AMT_COEF = getCoefLinear( AMT, X_train[RF_amt] )
```

```
REG_RF_AMT = TEST_AMT.copy()
RF_AMT = TEST_AMT.copy()
```

```
('LOAN', 100)
('IMP_CLNO', 12)
('IMP_DEBTINC', 5)
REG_RF RMSE ACCURACY
======
REG_RF_Train  =  5128.0038815240705
REG_RF  =  5381.010415951751
------
```

```
LOSS
---------
Total Variables:  4
INTERCEPT  =  -6566.743371416622
LOAN  =  0.7272984265393583
IMP_CLNO  =  255.45668969639542
IMP_DEBTINC  =  62.78970516702118
```

2.9.1 Develop a linear regression model to determine the expected loss if the loan defaults. Use the variables that were selected by a GRADIENT BOOSTING model.

[ ]:
```
# LOG REG GB
# LOSS from default

print("\n\n")
GB_amt = []
for i in vars_GB_amt :
    print(i)
    theVar = i[0]
    GB_amt.append( theVar )

AMT = LinearRegression()
AMT = AMT.fit( W_train[GB_amt], Z_train[TARGET_A] )

TRAIN_AMT = getAmtAccuracyScores( WHO + "_Train", AMT, W_train[GB_amt],␣
 ↪Z_train[TARGET_A] )
TEST_AMT = getAmtAccuracyScores( WHO, AMT, W_test[GB_amt], Z_test[TARGET_A] )
print_Accuracy( WHO + " RMSE ACCURACY", [ TRAIN_AMT, TEST_AMT ] )
```

```
REG_GB_AMT_COEF = getCoefLinear( AMT, X_train[GB_amt] )

REG_GB_AMT = TEST_AMT.copy()
GB_AMT = TEST_AMT.copy()
```

```
('LOAN', 100)
('IMP_CLNO', 14)
('IMP_DEBTINC', 5)
('M_DEBTINC', 5)
REG_RF RMSE ACCURACY
======
REG_RF_Train  =  4408.564694728578
REG_RF  =  4465.163660738355
------
```

```
LOSS
---------
Total Variables:  5
INTERCEPT  =  -12580.151025484236
LOAN  =  0.749219228333857
IMP_CLNO  =  246.2871822540995
IMP_DEBTINC  =  117.66800058738173
M_DEBTINC  =  5736.300969032894
```

###Develop a linear regression model to determine the expected loss if the loan defaults. Use the variables that were selected by STEPWISE SELECTION.

[ ]:
```
# STEPWISE REG

V_train = W_train[ GB_amt ]
stepVarNames = list( V_train.columns.values )
maxCols = V_train.shape[1]

sfs = SFS( LinearRegression(),
           k_features=( 1, maxCols ),
           forward=True,
           floating=False,
           scoring = 'r2',
           cv=5
           )
sfs.fit(V_train.values, Z_train[ TARGET_A ].values)

theFigure = plot_sfs(sfs.get_metric_dict(), kind=None )
```

```
plt.title('LOSSSSSSSSS Sequential Forward Selection (w. StdErr)')
plt.grid()
plt.show()

dfm = pd.DataFrame.from_dict( sfs.get_metric_dict()).T
dfm = dfm[ ['feature_names', 'avg_score'] ]
dfm.avg_score = dfm.avg_score.astype(float)

print(" ..................... ")
maxIndex = dfm.avg_score.argmax()
print("argmax")
print( dfm.iloc[ maxIndex, ] )
print(" ..................... ")

stepVars = dfm.iloc[ maxIndex, ]
stepVars = stepVars.feature_names
print( stepVars )

finalStepVars = []
for i in stepVars :
    index = int(i)
    try :
        theName = stepVarNames[ index ]
        finalStepVars.append( theName )
    except :
        pass

for i in finalStepVars :
    print(i)

V_train = W_train[ finalStepVars ]
V_test = W_test[ finalStepVars ]
```


```
 …
argmax
feature_names    (0, 1, 2, 3)
avg_score            0.811237
Name: 4, dtype: object
 …
('0', '1', '2', '3')
LOAN
IMP_CLNO
IMP_DEBTINC
M_DEBTINC
```

[ ]:
```
AMT = LinearRegression()
AMT = AMT.fit( V_train, Z_train[TARGET_A] )

TRAIN_AMT = getAmtAccuracyScores( WHO + "_Train", AMT, V_train,␣
 ↪Z_train[TARGET_A] )
TEST_AMT = getAmtAccuracyScores( WHO, AMT, V_test, Z_test[TARGET_A] )
print_Accuracy( WHO + " RMSE ACCURACY", [ TRAIN_AMT, TEST_AMT ] )

REG_STEP_CLM_COEF = getCoefLogit( CLM, U_train )
REG_STEP_AMT_COEF = getCoefLinear( AMT, V_train )
```

```
REG_STEP_CLM = TEST_CLM.copy()
REG_STEP_AMT = TEST_AMT.copy()
```

```
REG_RF RMSE ACCURACY
======
REG_RF_Train  =  4408.564694728578
REG_RF  =  4465.163660738355
------
```

```
DEFAULT
---------
Total Variables:  6
INTERCEPT  =  -4.942507685020032
M_DEROG  =  -0.8321416484146855
IMP_DELINQ  =  0.7409836248453477
IMP_CLAGE  =  -0.0063706637859314835
M_DEBTINC  =  2.825781584330456
IMP_DEBTINC  =  0.0938354700697999

LOSS
---------
Total Variables:  5
INTERCEPT  =  -12580.151025484236
LOAN  =  0.749219228333857
IMP_CLNO  =  246.2871822540995
IMP_DEBTINC  =  117.66800058738173
M_DEBTINC  =  5736.300969032894
```

###List the RMSE for the test data set for all of the models created (tree based and regression).

[ ]:
```
ALL_AMT = [ TREE_AMT, RF_AMT, GB_AMT, REG_ALL_AMT, REG_TREE_AMT, REG_RF_AMT,␣
 ↪REG_GB_AMT, REG_STEP_AMT ]
ALL_AMT = sorted( ALL_AMT, key = lambda x: x[1] )
print_Accuracy( "ALL DAMAGE MODEL ACCURACY", ALL_AMT )
```

```
ALL DAMAGE MODEL ACCURACY
======
REG_ALL  =  3493.138389711616
REG_TREE  =  4307.245008917904
REG_TREE  =  4307.245008917904
REG_RF  =  4465.163660738355
REG_RF  =  4465.163660738355
REG_RF  =  4465.163660738355
REG_RF  =  5381.010415951751
REG_RF  =  5381.010415951751
------
```

2.9.2 This model shows me that the regression al all models is the best because it has the lowest RMSE. The lowest RMSE means the random deviation from the true population.

###Print coeff and desribe.

[ ]:
```
"""
REGRESSION
"""

WHO = "REG_STEPWISE"
AMT = LinearRegression()
AMT = AMT.fit( V_train, Z_train[TARGET_A] )

TRAIN_AMT = getAmtAccuracyScores( WHO + "_Train", AMT, V_train,␣
 ↪Z_train[TARGET_A] )
TEST_AMT = getAmtAccuracyScores( WHO, AMT, V_test, Z_test[TARGET_A] )
print_Accuracy( WHO + " RMSE ACCURACY", [ TRAIN_AMT, TEST_AMT ] )

REG_STEP_CLM_COEF = getCoefLogit( CLM, U_train )
REG_STEP_AMT_COEF = getCoefLinear( AMT, V_train )

REG_STEP_CLM = TEST_CLM.copy()
REG_STEP_AMT = TEST_AMT.copy()


### The meaning I am able to draw from this model is that the more variables we␣
 ↪have, the more negative the intercept will be. The negative intercept means␣
 ↪that greater losses on a loan are predicted when we add more variables.␣
 ↪Missing derog information is also a bad sign. It means that if missing␣
 ↪derogatory is mentioned in a creadit report, this person is a riskier␣
 ↪borrowwer and more likely to lead to greater losses. SO yes, this model and␣
 ↪it findings make sense.
```

```
REG_STEPWISE RMSE ACCURACY
======
REG_STEPWISE_Train  =  4408.564694728578
REG_STEPWISE  =  4465.163660738355
------



DEFAULT
---------
Total Variables:  6
INTERCEPT  =  -4.942507685020032
```

```
M_DEROG  =  -0.8321416484146855
IMP_DELINQ  =  0.7409836248453477
IMP_CLAGE  =  -0.0063706637859314835
M_DEBTINC  =  2.825781584330456
IMP_DEBTINC  =  0.0938354700697999

LOSS
---------
Total Variables:  5
INTERCEPT  =  -12580.151025484236
LOAN  =  0.749219228333857
IMP_CLNO  =  246.2871822540995
IMP_DEBTINC  =  117.66800058738173
M_DEBTINC  =  5736.300969032894
```

#Assignment 04: Neural Networks

[ ]:
```
import math
import itertools

import pandas as pd
import numpy as np
from operator import itemgetter

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.preprocessing import OneHotEncoder

from sklearn.model_selection import train_test_split
import sklearn.metrics as metrics

from sklearn import tree
from sklearn.tree import _tree

from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import RandomForestClassifier

from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import GradientBoostingClassifier

from sklearn.linear_model import LinearRegression
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix

from mlxtend.feature_selection import SequentialFeatureSelector as SFS
from mlxtend.plotting import plot_sequential_feature_selection as plot_sfs
```

```
import tensorflow as tf

from sklearn.preprocessing import MinMaxScaler

import warnings
warnings.filterwarnings("ignore")

sns.set()
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
pd.set_option('display.width', None)
pd.set_option('display.max_colwidth', None)
```

3 Tensor Flow

3.0.1 Test and train data

[ ]:
```
import tensorflow as tf

from sklearn.preprocessing import MinMaxScaler  #### Need to scale between zero␣
 ↪and one
theScaler = MinMaxScaler()
theScaler.fit( X_train )



def get_TF_ProbAccuracyScores( NAME, MODEL, X, Y ) :
    probs = MODEL.predict_proba( X )
    pred_list = []
    for p in probs :
        pred_list.append( np.argmax( p ) )
    pred = np.array( pred_list )
    acc_score = metrics.accuracy_score(Y, pred)
    p1 = probs[:,1]
    fpr, tpr, threshold = metrics.roc_curve( Y, p1)
    auc = metrics.auc(fpr,tpr)
    return [NAME, acc_score, fpr, tpr, auc]
```

[ ]:
```
WHO = "Tensor_FLow"

U_train = theScaler.transform( X_train )
U_test = theScaler.transform( X_test )

U_train = pd.DataFrame( U_train )
U_test = pd.DataFrame( U_test )

U_train.columns = list( X_train.columns.values )
```

```
U_test.columns = list( X_train.columns.values )

U_train = U_train[ GB_flag ]
U_test = U_test[ GB_flag ]
```

3.0.2 RELU: Activation RELU

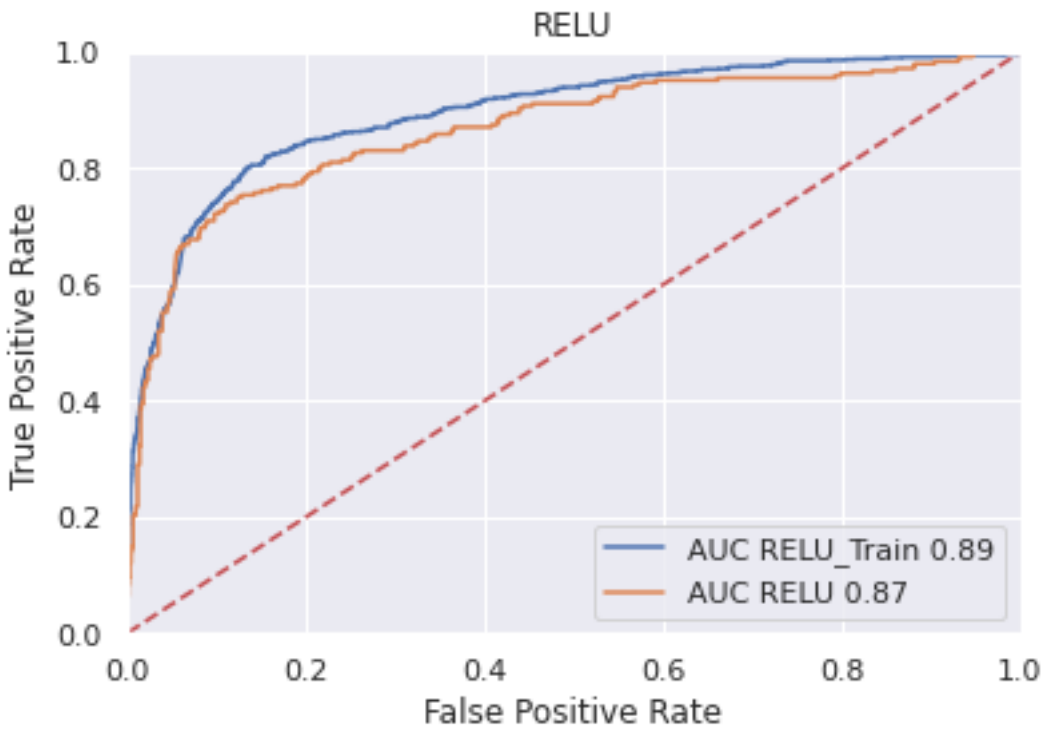
[ ]:
```
F_theShapeSize = U_train.shape[1]
F_theActivation = tf.keras.activations.relu
F_theLossMetric = tf.keras.losses.SparseCategoricalCrossentropy()
F_theOptimizer = tf.keras.optimizers.Adam()
F_theEpochs = 100
```

[ ]:
```
F_theUnits = int( 2*F_theShapeSize ) ## starting off point ## rule is 2x times
```

###Calculate the accuracy of the model on both the training and test data set

###Create a graph that shows the ROC curves for both the training and test data set. Clearly label each curve and display the Area Under the ROC curve.

[ ]:
```
WHO = "RELU"

F_LAYER_01 = tf.keras.layers.Dense( units=F_theUnits,␣
 ↪activation=F_theActivation, input_dim=F_theShapeSize )
F_LAYER_OUTPUT = tf.keras.layers.Dense( units=2, activation=tf.keras.
 ↪activations.softmax )

CLM = tf.keras.Sequential()
CLM.add( F_LAYER_01 )

CLM.add( F_LAYER_OUTPUT )
CLM.compile( loss=F_theLossMetric, optimizer=F_theOptimizer)
CLM.fit( U_train, Y_train[TARGET_F], epochs=F_theEpochs, verbose=False )

TRAIN_CLM = get_TF_ProbAccuracyScores( WHO + "_Train", CLM, U_train, Y_train[␣
 ↪TARGET_F ] )
TEST_CLM = get_TF_ProbAccuracyScores( WHO, CLM, U_test, Y_test[ TARGET_F ] )

print_ROC_Curve( WHO, [ TRAIN_CLM, TEST_CLM ] )
print_Accuracy( WHO + " CLASSIFICATION ACCURACY", [ TRAIN_CLM, TEST_CLM ] )

RELU_CLM = TEST_CLM.copy()
```

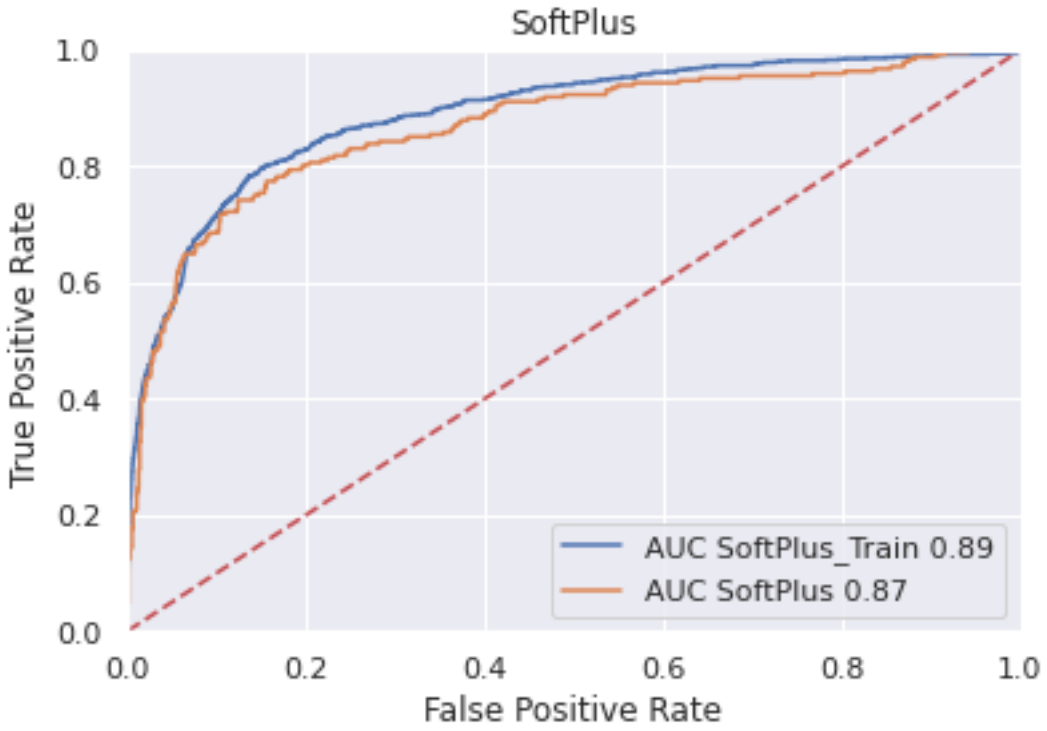



```
RELU CLASSIFICATION ACCURACY
======
RELU_Train  =  0.8779362416107382
RELU  =  0.87248322147651
------
```

3.0.3 Activation SoftPlus

[ ]:
```
WHO = "SoftPlus"

F_theShapeSize = U_train.shape[1]
F_theActivation = tf.keras.activations.softplus
F_theLossMetric = tf.keras.losses.SparseCategoricalCrossentropy()
F_theOptimizer = tf.keras.optimizers.Adam()
F_theEpochs = 100

F_theUnits = int( 2*F_theShapeSize ) ## starting off point ## rule is 2x times

F_LAYER_01 = tf.keras.layers.Dense( units=F_theUnits,␣
 ↪activation=F_theActivation, input_dim=F_theShapeSize )
```

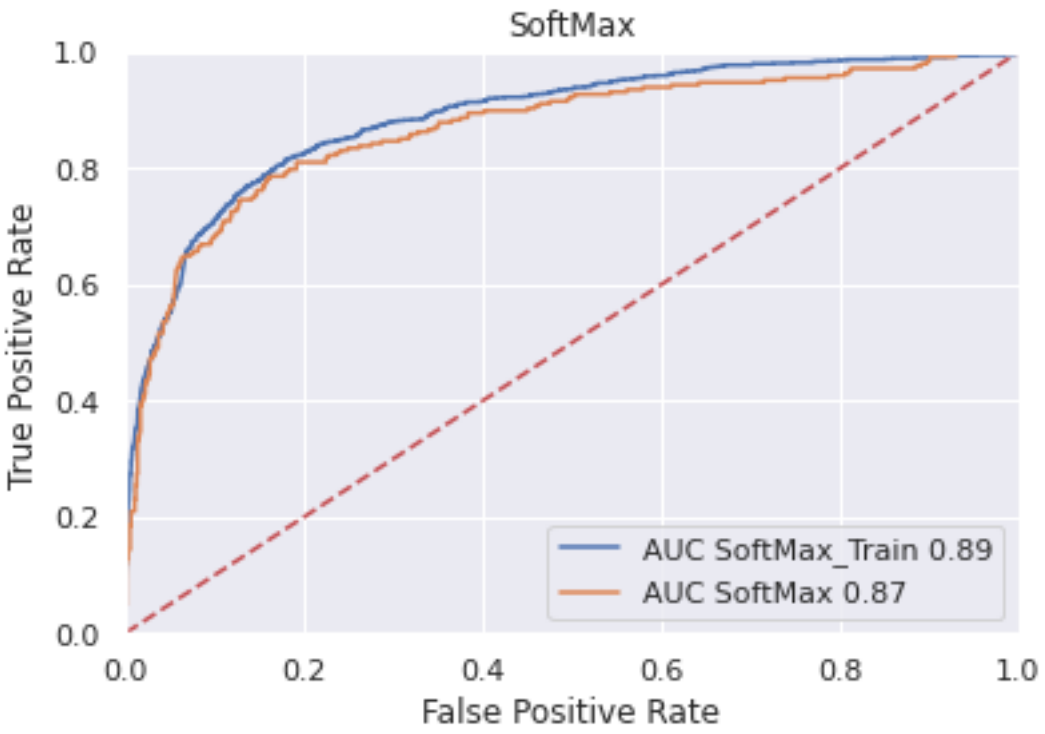
```
F_LAYER_OUTPUT = tf.keras.layers.Dense( units=2, activation=tf.keras.
 ↪activations.softmax )


CLM = tf.keras.Sequential()
CLM.add( F_LAYER_01 )
CLM.add( F_LAYER_OUTPUT )
CLM.compile( loss=F_theLossMetric, optimizer=F_theOptimizer)
CLM.fit( U_train, Y_train[TARGET_F], epochs=F_theEpochs, verbose=False )

TRAIN_CLM = get_TF_ProbAccuracyScores( WHO + "_Train", CLM, U_train, Y_train[␣
 ↪TARGET_F ] )
TEST_CLM = get_TF_ProbAccuracyScores( WHO, CLM, U_test, Y_test[ TARGET_F ] )

print_ROC_Curve( WHO, [ TRAIN_CLM, TEST_CLM ] )
print_Accuracy( WHO + " CLASSIFICATION ACCURACY", [ TRAIN_CLM, TEST_CLM ] )

SoftPlus_CLM = TEST_CLM.copy()
```




```
SoftPlus CLASSIFICATION ACCURACY
======
SoftPlus_Train  =  0.87248322147651
SoftPlus  =  0.8674496644295302
```

3.0.4 Activation: SoftMax

[ ]:
```
WHO = "SoftMax"

F_theShapeSize = U_train.shape[1]
F_theActivation = tf.keras.activations.softmax
F_theLossMetric = tf.keras.losses.SparseCategoricalCrossentropy()
F_theOptimizer = tf.keras.optimizers.Adam()
F_theEpochs = 100

F_theUnits = int( 2*F_theShapeSize ) ## starting off point ## rule is 2x times

F_LAYER_01 = tf.keras.layers.Dense( units=F_theUnits,␣
 ↪activation=F_theActivation, input_dim=F_theShapeSize )
F_LAYER_OUTPUT = tf.keras.layers.Dense( units=2, activation=tf.keras.
 ↪activations.softmax )


CLM = tf.keras.Sequential()
CLM.add( F_LAYER_01 )
CLM.add( F_LAYER_OUTPUT )
CLM.compile( loss=F_theLossMetric, optimizer=F_theOptimizer)
CLM.fit( U_train, Y_train[TARGET_F], epochs=F_theEpochs, verbose=False )

TRAIN_CLM = get_TF_ProbAccuracyScores( WHO + "_Train", CLM, U_train, Y_train[␣
 ↪TARGET_F ] )
TEST_CLM = get_TF_ProbAccuracyScores( WHO, CLM, U_test, Y_test[ TARGET_F ] )

print_ROC_Curve( WHO, [ TRAIN_CLM, TEST_CLM ] )
print_Accuracy( WHO + " CLASSIFICATION ACCURACY", [ TRAIN_CLM, TEST_CLM ] )

SoftMax_CLM = TEST_CLM.copy()
```

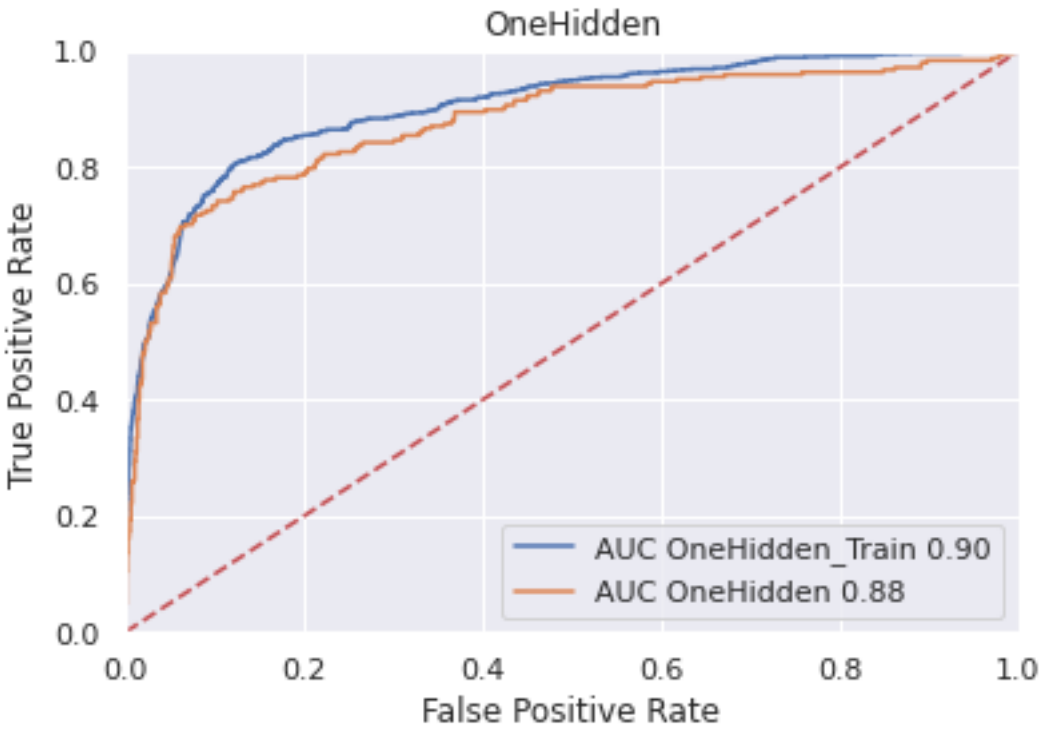



```
SoftMax CLASSIFICATION ACCURACY
======
SoftMax_Train  =  0.8714345637583892
SoftMax  =  0.8682885906040269
------
```

3.0.5 One hidden layer dip

[ ]:
```
WHO = "OneHidden"

F_theShapeSize = U_train.shape[1]
F_theActivation = tf.keras.activations.relu
F_theLossMetric = tf.keras.losses.SparseCategoricalCrossentropy()
F_theOptimizer = tf.keras.optimizers.Adam()
F_theEpochs = 100

F_theUnits = int( 2*F_theShapeSize ) ## starting off point ## rule is 2x times

F_LAYER_01 = tf.keras.layers.Dense( units=F_theUnits,␣
 ↪activation=F_theActivation, input_dim=F_theShapeSize )
```

```
F_LAYER_02 = tf.keras.layers.Dense( units=F_theUnits,␣
 ↪activation=F_theActivation ) ### hidden

F_LAYER_OUTPUT = tf.keras.layers.Dense( units=2, activation=tf.keras.
 ↪activations.softmax )

CLM = tf.keras.Sequential()
CLM.add( F_LAYER_01 )
CLM.add( F_LAYER_02 )
CLM.add( F_LAYER_OUTPUT )
CLM.compile( loss=F_theLossMetric, optimizer=F_theOptimizer)
CLM.fit( U_train, Y_train[TARGET_F], epochs=F_theEpochs, verbose=False )

TRAIN_CLM = get_TF_ProbAccuracyScores( WHO + "_Train", CLM, U_train, Y_train[␣
 ↪TARGET_F ] )
TEST_CLM = get_TF_ProbAccuracyScores( WHO, CLM, U_test, Y_test[ TARGET_F ] )

print_ROC_Curve( WHO, [ TRAIN_CLM, TEST_CLM ] )
print_Accuracy( WHO + " CLASSIFICATION ACCURACY", [ TRAIN_CLM, TEST_CLM ] )

OneHidden_CLM = TEST_CLM.copy()
```

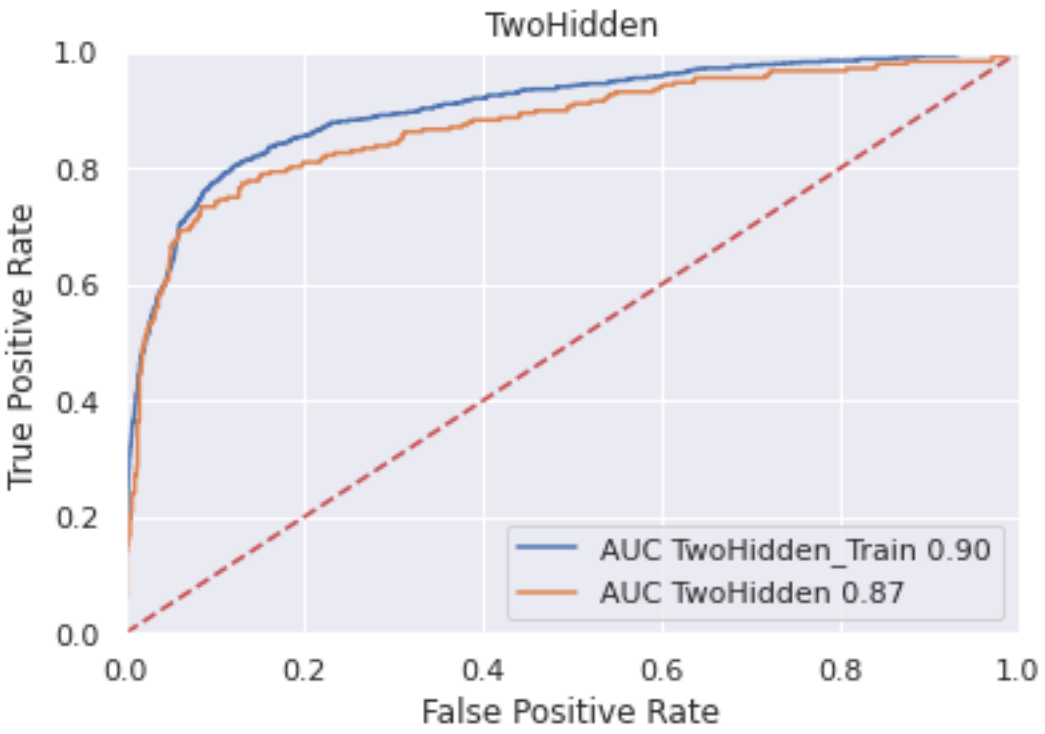



```
OneHidden CLASSIFICATION ACCURACY
```

```
======
OneHidden_Train  =  0.8856963087248322
OneHidden  =  0.8783557046979866
------
```

###Two Hidden

[ ]:
```
WHO = "TwoHidden"

F_theShapeSize = U_train.shape[1]
F_theActivation = tf.keras.activations.relu
F_theLossMetric = tf.keras.losses.SparseCategoricalCrossentropy()
F_theOptimizer = tf.keras.optimizers.Adam()
F_theEpochs = 100

F_theUnits = int( 2*F_theShapeSize ) ## starting off point ## rule is 2x times

F_LAYER_01 = tf.keras.layers.Dense( units=F_theUnits,␣
 ↪activation=F_theActivation, input_dim=F_theShapeSize )
F_LAYER_02 = tf.keras.layers.Dense( units=F_theUnits,␣
 ↪activation=F_theActivation ) ### 1 hidden
F_LAYER_03 = tf.keras.layers.Dense( units=F_theUnits,␣
 ↪activation=F_theActivation ) ### 2 hidden, should probably use this if you␣
 ↪have pictures, otherwise probably do not need go this "Deep"
F_LAYER_OUTPUT = tf.keras.layers.Dense( units=2, activation=tf.keras.
 ↪activations.softmax )

CLM = tf.keras.Sequential()
CLM.add( F_LAYER_01 )
CLM.add( F_LAYER_02 )
CLM.add( F_LAYER_03 )
CLM.add( F_LAYER_OUTPUT )
CLM.compile( loss=F_theLossMetric, optimizer=F_theOptimizer)
CLM.fit( U_train, Y_train[TARGET_F], epochs=F_theEpochs, verbose=False )

TRAIN_CLM = get_TF_ProbAccuracyScores( WHO + "_Train", CLM, U_train, Y_train[␣
 ↪TARGET_F ] )
TEST_CLM = get_TF_ProbAccuracyScores( WHO, CLM, U_test, Y_test[ TARGET_F ] )

print_ROC_Curve( WHO, [ TRAIN_CLM, TEST_CLM ] )
print_Accuracy( WHO + " CLASSIFICATION ACCURACY", [ TRAIN_CLM, TEST_CLM ] )

TwoHidden_CLM = TEST_CLM.copy()
```

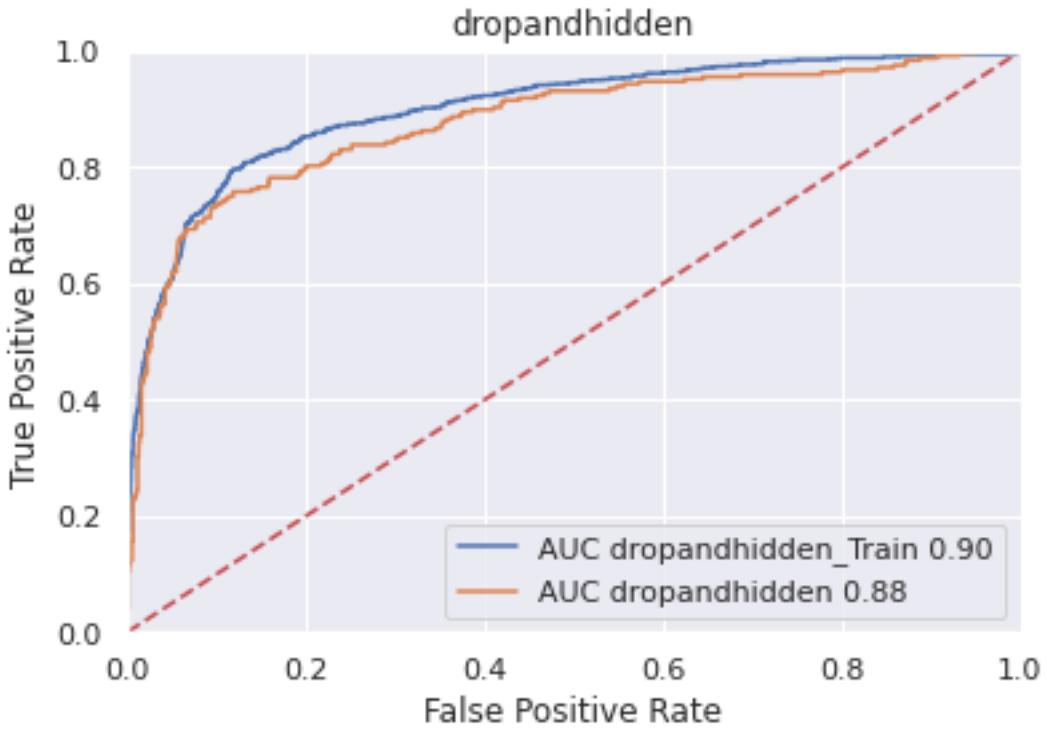



```
TwoHidden CLASSIFICATION ACCURACY
======
TwoHidden_Train  =  0.889261744966443
TwoHidden  =  0.8859060402684564
------
```

3.0.6 1 Dropout

[ ]:
```
WHO = "dropandhidden"

F_theShapeSize = U_train.shape[1]
F_theActivation = tf.keras.activations.relu
F_theLossMetric = tf.keras.losses.SparseCategoricalCrossentropy()
F_theOptimizer = tf.keras.optimizers.Adam()
F_theEpochs = 100

F_theUnits = int( 2*F_theShapeSize)

F_LAYER_01 = tf.keras.layers.Dense( units=F_theUnits,␣
 ↪activation=F_theActivation, input_dim=F_theShapeSize )
F_LAYER_DROP = tf.keras.layers.Dropout( 0.2 )
```

```
F_LAYER_02 = tf.keras.layers.Dense( units=F_theUnits,␣
 ↪activation=F_theActivation )
F_LAYER_OUTPUT = tf.keras.layers.Dense( units=2, activation=tf.keras.
 ↪activations.softmax )


CLM = tf.keras.Sequential()
CLM.add( F_LAYER_01 )
CLM.add( F_LAYER_DROP )
CLM.add( F_LAYER_02 )
CLM.add( F_LAYER_OUTPUT )
CLM.compile( loss=F_theLossMetric, optimizer=F_theOptimizer)
CLM.fit( U_train, Y_train[TARGET_F], epochs=F_theEpochs, verbose=False )

TRAIN_CLM = get_TF_ProbAccuracyScores( WHO + "_Train", CLM, U_train, Y_train[␣
 ↪TARGET_F ] )
TEST_CLM = get_TF_ProbAccuracyScores( WHO, CLM, U_test, Y_test[ TARGET_F ] )

print_ROC_Curve( WHO, [ TRAIN_CLM, TEST_CLM ] )
print_Accuracy( WHO + " CLASSIFICATION ACCURACY", [ TRAIN_CLM, TEST_CLM ] )

dropandhidden_CLM = TEST_CLM.copy()
```



```
dropandhidden CLASSIFICATION ACCURACY
======
dropandhidden_Train  =  0.8850671140939598
dropandhidden  =  0.8791946308724832
------
```

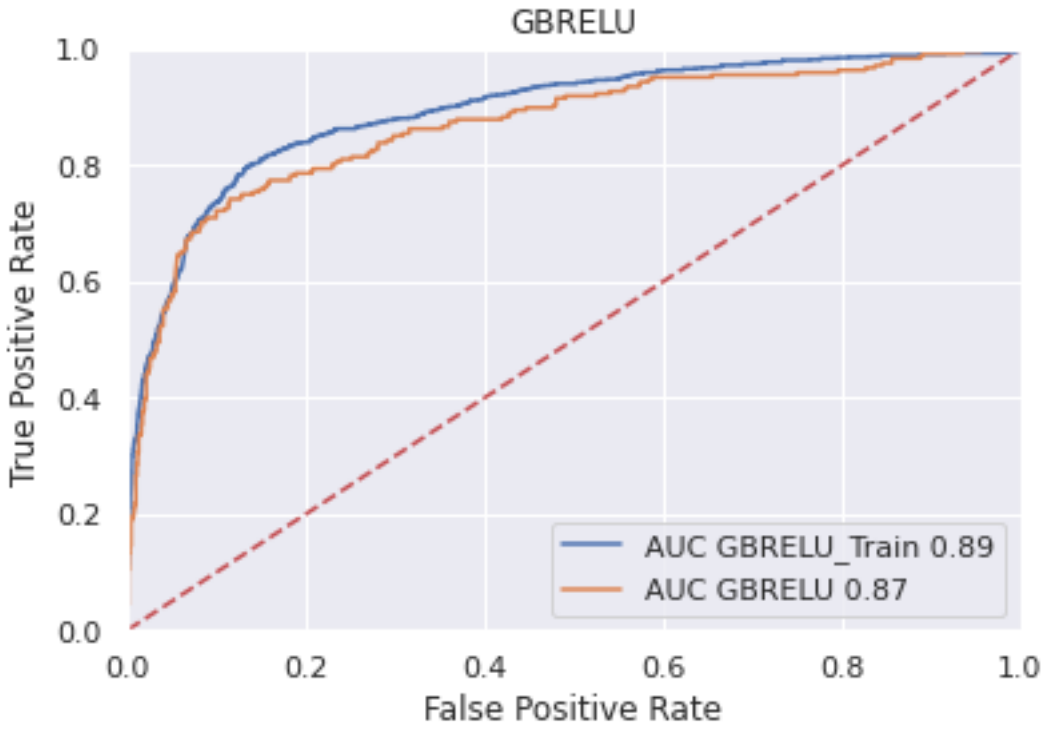
###Explore using a variable selection technique

[ ]:
```
GB_flag = []
for i in vars_GB_flag :
    print(i)
    theVar = i[0]
    GB_flag.append( theVar )

G_train = U_train[ GB_flag ]
G_test = U_test[ GB_flag ]
```

```
('M_DEBTINC', 100)
('IMP_DEBTINC', 29)
('IMP_DELINQ', 19)
('IMP_CLAGE', 14)
('IMP_DEROG', 7)
```

[ ]:
```
WHO = "GBRELU"

F_theShapeSize = U_train.shape[1]
F_theActivation = tf.keras.activations.relu
F_theLossMetric = tf.keras.losses.SparseCategoricalCrossentropy()
F_theOptimizer = tf.keras.optimizers.Adam()
F_theEpochs = 100

F_theUnits = int( 2*F_theShapeSize ) ## starting off point ## rule is 2x times

F_LAYER_01 = tf.keras.layers.Dense( units=F_theUnits,␣
 ↪activation=F_theActivation, input_dim=F_theShapeSize )
F_LAYER_OUTPUT = tf.keras.layers.Dense( units=2, activation=tf.keras.
 ↪activations.softmax )


CLM = tf.keras.Sequential()
CLM.add( F_LAYER_01 )
CLM.add( F_LAYER_OUTPUT )
CLM.compile( loss=F_theLossMetric, optimizer=F_theOptimizer)
CLM.fit( U_train, Y_train[TARGET_F], epochs=F_theEpochs, verbose=False )
```

```
TRAIN_CLM = get_TF_ProbAccuracyScores( WHO + "_Train", CLM, G_train, Y_train[␣
 ↪TARGET_F ] )
TEST_CLM = get_TF_ProbAccuracyScores( WHO, CLM, G_test, Y_test[ TARGET_F ] )

print_ROC_Curve( WHO, [ TRAIN_CLM, TEST_CLM ] )
print_Accuracy( WHO + " CLASSIFICATION ACCURACY", [ TRAIN_CLM, TEST_CLM ] )

GBRELU_CLM = TEST_CLM.copy()
```

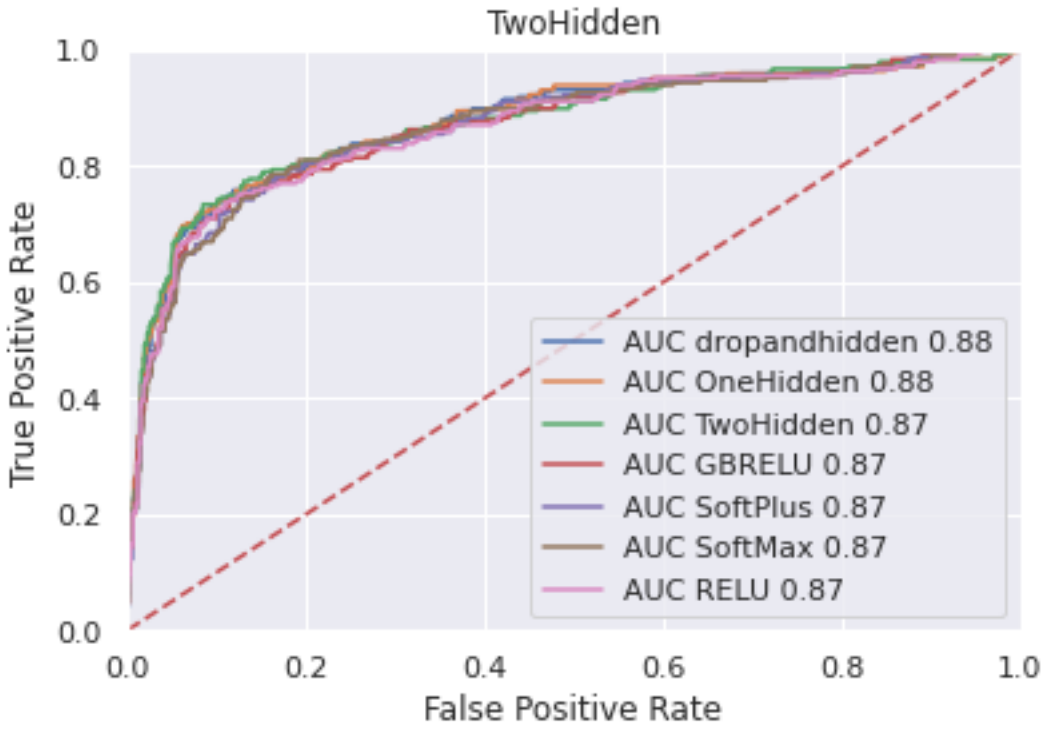



```
GBRELU CLASSIFICATION ACCURACY
======
GBRELU_Train  =  0.8777265100671141
GBRELU  =  0.8691275167785235
------
```

3.0.7 Display a ROC curve for the test data with all your models on the same graph

[ ]:
```
ALL_CLM = [ RELU_CLM, SoftPlus_CLM, SoftMax_CLM, OneHidden_CLM, TwoHidden_CLM,␣
 ↪dropandhidden_CLM, GBRELU_CLM ]

ALL_CLM = sorted( ALL_CLM, key = lambda x: x[4], reverse=True )
```

```
print_ROC_Curve( WHO, ALL_CLM )

ALL_CLM = sorted( ALL_CLM, key = lambda x: x[1], reverse=True )
print_Accuracy( "ALL CLASSIFICATION ACCURACY", ALL_CLM )
```




```
ALL CLASSIFICATION ACCURACY
======
TwoHidden  =  0.8859060402684564
dropandhidden  =  0.8791946308724832
OneHidden  =  0.8783557046979866
RELU  =  0.87248322147651
GBRELU  =  0.8691275167785235
SoftMax  =  0.8682885906040269
SoftPlus  =  0.8674496644295302
------
```

#Develop a model using Tensor Flow that will predict Loan Default.

3.0.8 RELU

[ ]:
```
WHO = "RELUAMT"

A_theShapeSize = V_train.shape[1]
A_theActivation = tf.keras.activations.relu
A_theLossMetric = tf.keras.losses.MeanSquaredError()
A_theOptimizer = tf.keras.optimizers.Adam()
A_theEpochs = 800

A_theUnits = int( 2*A_theShapeSize  )

A_LAYER_01 = tf.keras.layers.Dense( units=A_theUnits,␣
 ↪activation=A_theActivation, input_dim=A_theShapeSize )
A_LAYER_OUTPUT = tf.keras.layers.Dense( units=1, activation=tf.keras.
 ↪activations.linear )

AMT = tf.keras.Sequential()
AMT.add( A_LAYER_01 )
AMT.add( A_LAYER_OUTPUT )
AMT.compile( loss=A_theLossMetric, optimizer=A_theOptimizer)
AMT.fit( V_train, Z_train[TARGET_A], epochs=A_theEpochs, verbose=False )


TRAIN_AMT = getAmtAccuracyScores( WHO + "_Train", AMT, V_train[GB_amt],␣
 ↪Z_train[TARGET_A] )
TEST_AMT = getAmtAccuracyScores( WHO, AMT, V_test[GB_amt], Z_test[TARGET_A] )
print_Accuracy( WHO + " RMSE ACCURACY", [ TRAIN_AMT, TEST_AMT ] )

RELUAMT_AMT = TEST_AMT.copy()
```

```
RELUAMT RMSE ACCURACY
======
RELUAMT_Train  =  5752.77874922548
RELUAMT  =  5758.149468386336
------
```

###SoftPlus

[ ]:
```
WHO = "SoftPlusAMT"

A_theShapeSize = V_train.shape[1]
A_theActivation = tf.keras.activations.softplus
A_theLossMetric = tf.keras.losses.MeanSquaredError()
A_theOptimizer = tf.keras.optimizers.Adam()
A_theEpochs = 800
```

```
A_theUnits = int( 2*A_theShapeSize  )

A_LAYER_01 = tf.keras.layers.Dense( units=A_theUnits,␣
 ↪activation=A_theActivation, input_dim=A_theShapeSize )
A_LAYER_OUTPUT = tf.keras.layers.Dense( units=1, activation=tf.keras.
 ↪activations.linear )

AMT = tf.keras.Sequential()
AMT.add( A_LAYER_01 )
AMT.add( A_LAYER_OUTPUT )
AMT.compile( loss=A_theLossMetric, optimizer=A_theOptimizer)
AMT.fit( V_train, Z_train[TARGET_A], epochs=A_theEpochs, verbose=False )


TRAIN_AMT = getAmtAccuracyScores( WHO + "_Train", AMT, V_train[GB_amt],␣
 ↪Z_train[TARGET_A] )
TEST_AMT = getAmtAccuracyScores( WHO, AMT, V_test[GB_amt], Z_test[TARGET_A] )
print_Accuracy( WHO + " RMSE ACCURACY", [ TRAIN_AMT, TEST_AMT ] )

SoftPlusAMT_AMT = TEST_AMT.copy()
```

```
SoftPlusAMT RMSE ACCURACY
======
SoftPlusAMT_Train  =  5812.055528991011
SoftPlusAMT  =  5767.211603781986
------
```

###Softmax

[ ]:
```
WHO = "SoftMaxAMT"

A_theShapeSize = V_train.shape[1]
A_theActivation = tf.keras.activations.softmax
A_theLossMetric = tf.keras.losses.MeanSquaredError()
A_theOptimizer = tf.keras.optimizers.Adam()
A_theEpochs = 800

A_theUnits = int( 2*A_theShapeSize  )

A_LAYER_01 = tf.keras.layers.Dense( units=A_theUnits,␣
 ↪activation=A_theActivation, input_dim=A_theShapeSize )
A_LAYER_OUTPUT = tf.keras.layers.Dense( units=1, activation=tf.keras.
 ↪activations.linear )

AMT = tf.keras.Sequential()
```

```
AMT.add( A_LAYER_01 )
AMT.add( A_LAYER_OUTPUT )
AMT.compile( loss=A_theLossMetric, optimizer=A_theOptimizer)
AMT.fit( V_train, Z_train[TARGET_A], epochs=A_theEpochs, verbose=False )


TRAIN_AMT = getAmtAccuracyScores( WHO + "_Train", AMT, V_train[GB_amt],␣
 ↪Z_train[TARGET_A] )
TEST_AMT = getAmtAccuracyScores( WHO, AMT, V_test[GB_amt], Z_test[TARGET_A] )
print_Accuracy( WHO + " RMSE ACCURACY", [ TRAIN_AMT, TEST_AMT ] )

SoftMaxAMT_AMT = TEST_AMT.copy()
```

```
SoftMaxAMT RMSE ACCURACY
======
SoftMaxAMT_Train  =  17100.74551913646
SoftMaxAMT  =  17603.37259575714
------
```

3.0.9 1 Hidden

[ ]:
```
WHO = "hidRELUAMT"

A_theShapeSize = V_train.shape[1]
A_theActivation = tf.keras.activations.relu
A_theLossMetric = tf.keras.losses.MeanSquaredError()
A_theOptimizer = tf.keras.optimizers.Adam()
A_theEpochs = 800

A_theUnits = int( 2*A_theShapeSize  )

A_LAYER_01 = tf.keras.layers.Dense( units=A_theUnits,␣
 ↪activation=A_theActivation, input_dim=A_theShapeSize )
A_LAYER_02 = tf.keras.layers.Dense( units=A_theUnits,␣
 ↪activation=A_theActivation )

A_LAYER_OUTPUT = tf.keras.layers.Dense( units=1, activation=tf.keras.
 ↪activations.linear )

AMT = tf.keras.Sequential()
AMT.add( A_LAYER_01 )
AMT.add( A_LAYER_02 )
AMT.add( A_LAYER_OUTPUT )
AMT.compile( loss=A_theLossMetric, optimizer=A_theOptimizer)
AMT.fit( V_train, Z_train[TARGET_A], epochs=A_theEpochs, verbose=False )
```

```
TRAIN_AMT = getAmtAccuracyScores( WHO + "_Train", AMT, V_train[GB_amt],␣
 ↪Z_train[TARGET_A] )
TEST_AMT = getAmtAccuracyScores( WHO, AMT, V_test[GB_amt], Z_test[TARGET_A] )
print_Accuracy( WHO + " RMSE ACCURACY", [ TRAIN_AMT, TEST_AMT ] )

hidRELUAMT_AMT = TEST_AMT.copy()
```

```
hidRELUAMT RMSE ACCURACY
======
hidRELUAMT_Train  =  5170.246126040626
hidRELUAMT  =  5823.2075875012715
------
```

###2 Hidden

[ ]:
```
WHO = "a2hidRELUAMT"

A_theShapeSize = V_train.shape[1]
A_theActivation = tf.keras.activations.relu
A_theLossMetric = tf.keras.losses.MeanSquaredError()
A_theOptimizer = tf.keras.optimizers.Adam()
A_theEpochs = 800

A_theUnits = int( 2*A_theShapeSize  )

A_LAYER_01 = tf.keras.layers.Dense( units=A_theUnits,␣
 ↪activation=A_theActivation, input_dim=A_theShapeSize )
A_LAYER_02 = tf.keras.layers.Dense( units=A_theUnits,␣
 ↪activation=A_theActivation )
A_LAYER_03 = tf.keras.layers.Dense( units=A_theUnits,␣
 ↪activation=A_theActivation )
A_LAYER_OUTPUT = tf.keras.layers.Dense( units=1, activation=tf.keras.
 ↪activations.linear )

AMT = tf.keras.Sequential()
AMT.add( A_LAYER_01 )
AMT.add( A_LAYER_02 )
AMT.add( A_LAYER_03 )
AMT.add( A_LAYER_OUTPUT )
AMT.compile( loss=A_theLossMetric, optimizer=A_theOptimizer)
AMT.fit( V_train, Z_train[TARGET_A], epochs=A_theEpochs, verbose=False )
```

```
TRAIN_AMT = getAmtAccuracyScores( WHO + "_Train", AMT, V_train[GB_amt],␣
 ↪Z_train[TARGET_A] )
TEST_AMT = getAmtAccuracyScores( WHO, AMT, V_test[GB_amt], Z_test[TARGET_A] )
print_Accuracy( WHO + " RMSE ACCURACY", [ TRAIN_AMT, TEST_AMT ] )

a2hidRELUAMT_AMT = TEST_AMT.copy()
```

```
a2hidRELUAMT RMSE ACCURACY
======
a2hidRELUAMT_Train  =  5136.5372186230425
a2hidRELUAMT  =  5923.830470706929
------
```

###Try using a Dropout Layer

[ ]:
```
WHO = "dropamt"

A_theShapeSize = V_train.shape[1]
A_theActivation = tf.keras.activations.relu
A_theLossMetric = tf.keras.losses.MeanSquaredError()
A_theOptimizer = tf.keras.optimizers.Adam()
A_theEpochs = 800

A_theUnits = int( 2*A_theShapeSize  )

A_LAYER_01 = tf.keras.layers.Dense( units=A_theUnits,␣
 ↪activation=A_theActivation, input_dim=A_theShapeSize )
A_LAYER_DROP = tf.keras.layers.Dropout( 0.2 )
A_LAYER_OUTPUT = tf.keras.layers.Dense( units=1, activation=tf.keras.
 ↪activations.linear )

AMT = tf.keras.Sequential()
AMT.add( A_LAYER_01 )
AMT.add( A_LAYER_DROP )
AMT.add( A_LAYER_OUTPUT )
AMT.compile( loss=A_theLossMetric, optimizer=A_theOptimizer)
AMT.fit( V_train, Z_train[TARGET_A], epochs=A_theEpochs, verbose=False )


TRAIN_AMT = getAmtAccuracyScores( WHO + "_Train", AMT, V_train[GB_amt],␣
 ↪Z_train[TARGET_A] )
TEST_AMT = getAmtAccuracyScores( WHO, AMT, V_test[GB_amt], Z_test[TARGET_A] )
print_Accuracy( WHO + " RMSE ACCURACY", [ TRAIN_AMT, TEST_AMT ] )

dropamt_AMT = TEST_AMT.copy()
```

```
dropamt RMSE ACCURACY
======
dropamt_Train  =  5736.202489668712
dropamt  =  5866.629067952708
------
```

###List the RMSE for the test data set for all of the models created

[ ]:
```
ALL_AMT = [ RELUAMT_AMT, SoftPlusAMT_AMT, SoftMaxAMT_AMT, hidRELUAMT_AMT,␣
 ↪a2hidRELUAMT_AMT, dropamt_AMT, ]
ALL_AMT = sorted( ALL_AMT, key = lambda x: x[1] )
print_Accuracy( "ALL DAMAGE MODEL ACCURACY", ALL_AMT )
```

```
ALL DAMAGE MODEL ACCURACY
======
RELUAMT  =  5758.149468386336
SoftPlusAMT  =  5767.211603781986
hidRELUAMT  =  5823.2075875012715
dropamt  =  5866.629067952708
a2hidRELUAMT  =  5923.830470706929
SoftMaxAMT  =  17603.37259575714
------
```