TEXAS A&M UNIVERSITY

# MID TERM PROJECT - STAT 656


**Submitted by: -**

Krit Gupta (UIN 927001565)

Ashish Jatav (UIN 826006525)

Aparajit Koshal (UIN 326004962)

Neehar Yalamarti (UIN 226008080)

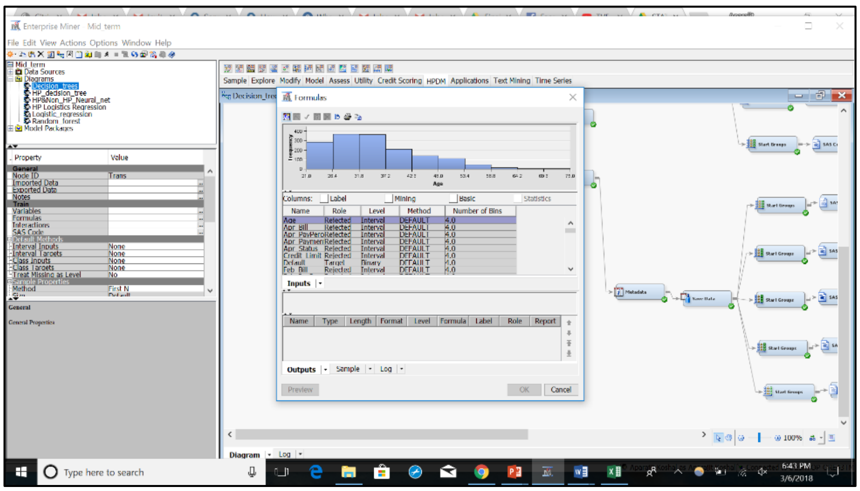
Kshitijraj Sasavade (UIN 726006949)

## Objective

The objective of this project is to determine the risk of defaulting for customers of a major German Credit Bank. This is done by the use of several classification techniques like logistic regression, decision trees, neural networks and random forests. A comparison is draw-n from the models obtained and the best one is selected according to classification metrics.

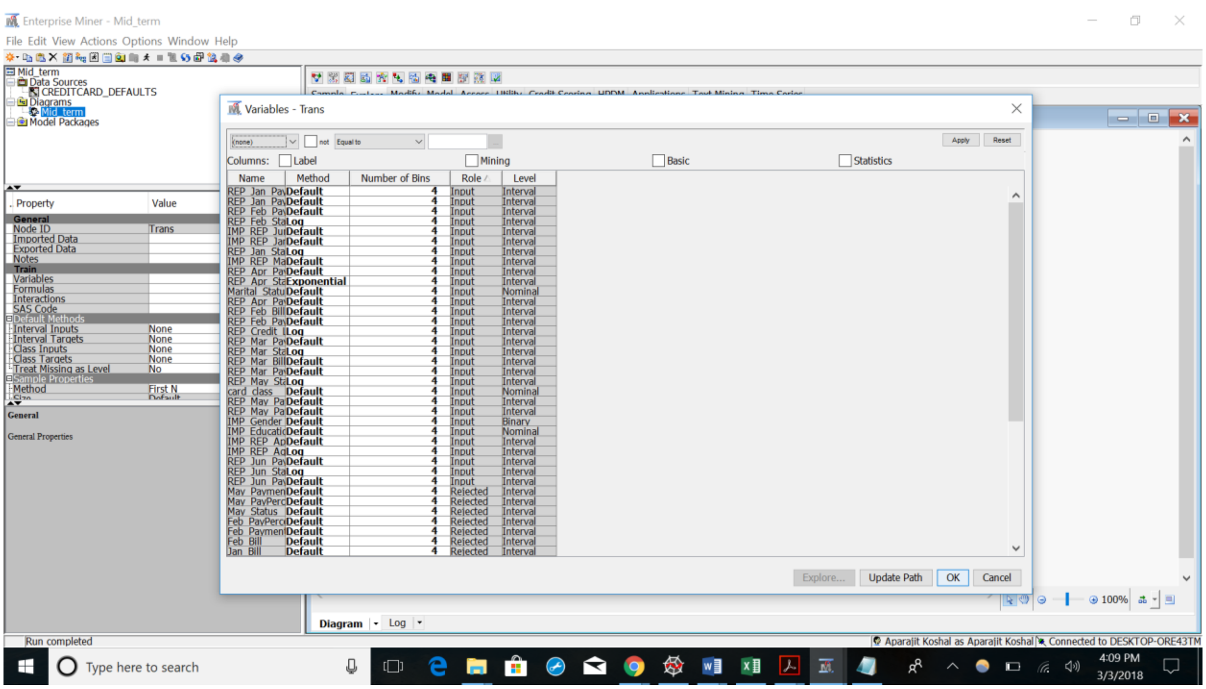
*The metric that is most relevant in this case is sensitivity, as the prediction is for a person who is defaulter and can cause serious loss to the company. Accuracy and F1 score will be our focus as well but the priority of our predictive model will be higher true positive rate.*

Initially exploratory data analysis was done for the parameters and it was observed that some of them have a skewed distribution. For example, for age as shown below:



Other parameters that showed this behavior were:

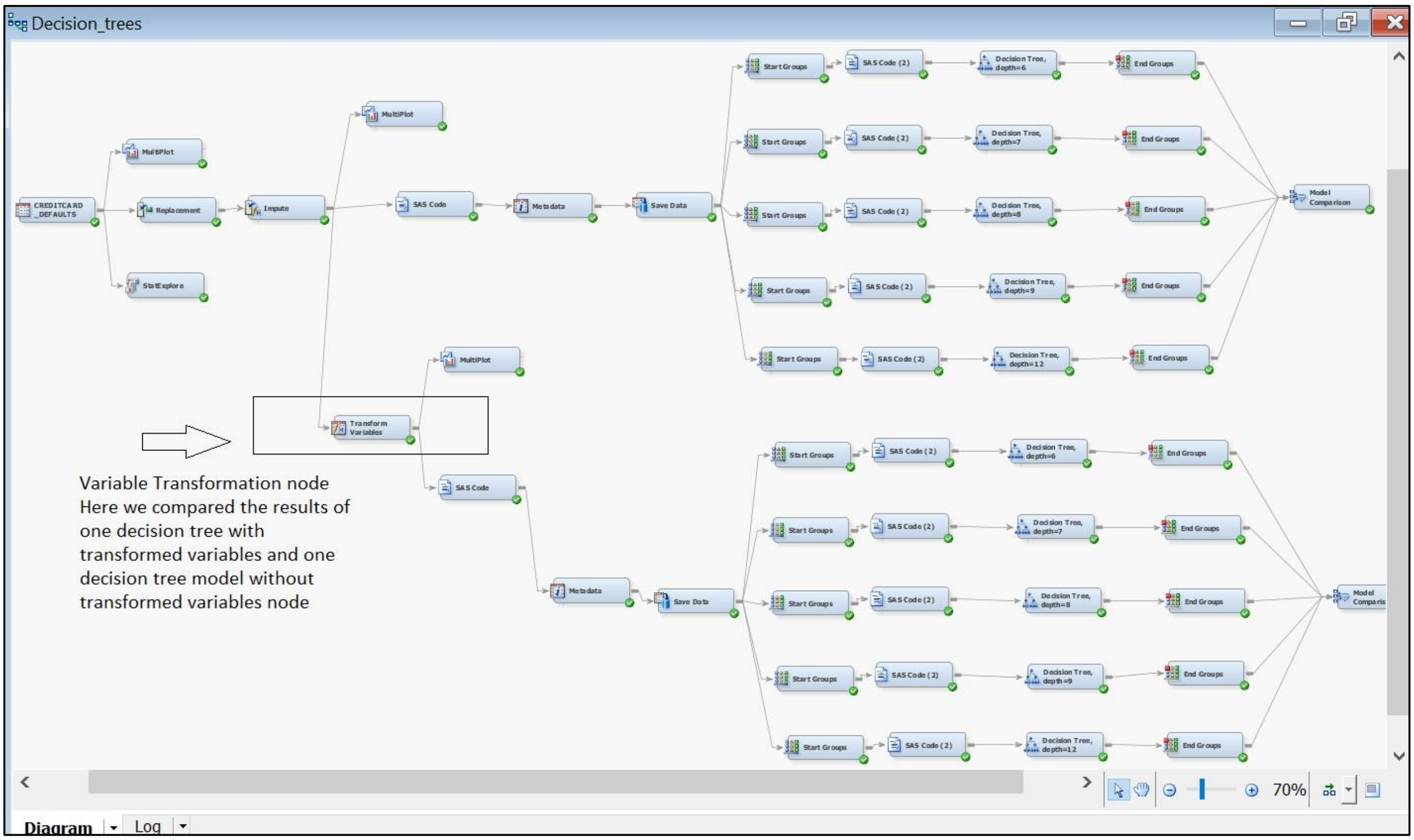
Credit Limit
Age
Jan Status
Feb Status
March Status
April Status
May Status
June Status



Since the distribution for the above-mentioned variables were skewed to the right, log transform was applied to them. After this was done, decision trees were applied to both data sets: with the variables transformed and to the data which was not transformed.

## Decision Trees:

This model (decision trees) compares different result, one model is with Variable Transformation node and one model is without Variable Transformation node.



*Comparing the two outputs:*

Event Classification Table
Model Selection based on Train: Misclassification Rate (_MISC_)

***Decision Tree Model with transform variable node***

| Model Node | Model Description | Data Role | Target Variable | Target Label | FALSEEVENT | FALSENEVENT | TRUEEVENT | TRUENEVENT |
|---|---|---|---|---|---|---|---|---|
| EndGrp | End Groups | TRAIN | Default | | 935 | 2581 | 2273 | 24211 |
| EndGrp3 | End Groups | TRAIN | Default | | 966 | 2581 | 2273 | 24180 |
| EndGrp2 | End Groups | TRAIN | Default | | 980 | 2632 | 2222 | 24166 |
| EndGrp4 | End Groups | TRAIN | Default | | 935 | 2528 | 2326 | 24211 |
| EndGrp5 | End Groups | TRAIN | Default | | 854 | 2483 | 2371 | 24292 |

```
*------------------------------------------------------------*
* Score Output
*------------------------------------------------------------*


*------------------------------------------------------------*
* Report Output
*------------------------------------------------------------*
```

Classification accuracy and senstitvity is equivalent for both the models.

Sensitivity= 48.81%

Event Classification Table
Model Selection based on Train: Misclassification Rate (_MISC_)

| Model Node | Model Description | Data Role | Target Variable | Target Label | FALSEEVENT | FALSENEVENT | TRUEEVENT | TRUENEVENT |
|---|---|---|---|---|---|---|---|---|
| EndGrp6 | End Groups | TRAIN | Default | | 935 | 2581 | 2273 | 24211 |
| EndGrp8 | End Groups | TRAIN | Default | | 966 | 2581 | 2273 | 24180 |
| EndGrp7 | End Groups | TRAIN | Default | | 980 | 2632 | 2222 | 24166 |
| EndGrp9 | End Groups | TRAIN | Default | | 935 | 2528 | 2326 | 24211 |
| EndGrp10 | End Groups | TRAIN | Default | | 854 | 2484 | 2370 | 24292 |

```
*------------------------------------------------------------*
* Score Output
*------------------------------------------------------------*


*------------------------------------------------------------*
* Report Output
*------------------------------------------------------------*
```

***Decision Tree Model without transform variable node***

As observed, there wasn't much difference in the sensitivity. Hence, we decide to avoid transformation of variables for further classification techniques.

Cross validation metrics obtained for the regular decision tree without the transformed variables:

## DECISION TREE (with different depths)

**Depth 6**

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.503006012 | 0.709039548 | 0.883 | 0.5885111372 |
| 0.4661190965 | 0.6837349398 | 0.8783333333 | 0.5543345543 |
| 0.4632034632 | 0.6645962733 | 0.8813333333 | 0.5459183673 |
| 0.4261954262 | 0.6856187291 | 0.8766666667 | 0.5256410256 |
| 0.4313304721 | 0.6836734694 | 0.8806666667 | 0.5289473684 |
| 0.4501992032 | 0.7084639498 | 0.877 | 0.5505481121 |
| 0.4742063492 | 0.7286585366 | 0.882 | 0.5745192308 |
| 0.4573170732 | 0.6859756098 | 0.8766666667 | 0.5487804878 |
| 0.4547368421 | 0.6945337621 | 0.882 | 0.5496183206 |
| 0.4485596708 | 0.6920634921 | 0.8783333333 | 0.5443196005 |

Average Values:
Average Recall: 0.45748736084866704
Average Precision: 0.6936358309864399
Average Accuracy: 0.8795999999999999
Average F1: 0.5511138204649162

**Depth 7**

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.503006012 | 0.7130681818 | 0.8836666667 | 0.5898942421 |
| 0.4722792608 | 0.6948640483 | 0.8806666667 | 0.5623471883 |
| 0.4523809524 | 0.7157534247 | 0.888 | 0.5543766578 |
| 0.4303534304 | 0.6993243243 | 0.879 | 0.5328185328 |
| 0.4527896996 | 0.6850649351 | 0.8826666667 | 0.5452196382 |
| 0.4541832669 | 0.7080745342 | 0.8773333333 | 0.5533980583 |
| 0.496031746 | 0.7246376812 | 0.8836666667 | 0.5889281508 |
| 0.5 | 0.6776859504 | 0.879 | 0.5754385965 |
| 0.4821052632 | 0.6918429003 | 0.884 | 0.5682382134 |
| 0.4362139918 | 0.7090301003 | 0.8796666667 | 0.5401273885 |

Average Values:
Average Recall: 0.4679343623000992

Average Precision: 0.701934608057404
Average Accuracy: 0.8817666666666668
Average F1: 0.5610786666662371

**Depth 8**

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.50501002 | 0.7159090909 | 0.8843333333 | 0.5922444183 |
| 0.476386037 | 0.7051671733 | 0.8826666667 | 0.568627451 |
| 0.4329004329 | 0.7434944238 | 0.8896666667 | 0.5471956224 |
| 0.4365904366 | 0.7023411371 | 0.88 | 0.5384615385 |
| 0.4527896996 | 0.6872964169 | 0.883 | 0.5459249677 |
| 0.4561752988 | 0.7133956386 | 0.8783333333 | 0.5565006075 |
| 0.494047619 | 0.7345132743 | 0.885 | 0.590747331 |
| 0.5040650407 | 0.6775956284 | 0.8793333333 | 0.5780885781 |
| 0.4842105263 | 0.7033639144 | 0.886 | 0.5735660848 |
| 0.4362139918 | 0.7090301003 | 0.8796666667 | 0.5401273885 |

Average Values:
Average Recall: 0.467838910265089340
Average Precision: 0.7092106798103452
Average Accuracy: 0.8827999999999999
Average F1: 0.563148398777270

**Depth 9**

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.5270541082 | 0.7127371274 | 0.886 | 0.6059907834 |
| 0.4845995893 | 0.717325228 | 0.8853333333 | 0.5784313725 |
| 0.4415584416 | 0.7445255474 | 0.8906666667 | 0.5543478261 |
| 0.4407484407 | 0.7066666667 | 0.881 | 0.542893726 |
| 0.491416309 | 0.6775147929 | 0.8846666667 | 0.5696517413 |
| 0.4741035857 | 0.7125748503 | 0.88 | 0.5693779904 |
| 0.5 | 0.7390029326 | 0.8863333333 | 0.5964497041 |
| 0.5020325203 | 0.6918767507 | 0.8816666667 | 0.581861013 |
| 0.4842105263 | 0.7232704403 | 0.889 | 0.580075662 |
| 0.4423868313 | 0.7142857143 | 0.881 | 0.5463786531 |

Average Values:
Average Recall: 0.4788110352432656
Average Precision: 0.7139780050434418
Average Accuracy: 0.8845666666666666
Average F1: 0.572545847201698

**Depth 12**

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.5531062124 | 0.7340425532 | 0.8923333333 | 0.6308571429 |
| 0.4702258727 | 0.7557755776 | 0.8893333333 | 0.5797468354 |
| 0.4653679654 | 0.770609319 | 0.8963333333 | 0.5802968961 |
| 0.4698544699 | 0.7174603175 | 0.8853333333 | 0.567839196 |
| 0.4892703863 | 0.7169811321 | 0.8906666667 | 0.5816326531 |
| 0.4940239044 | 0.7337278107 | 0.8853333333 | 0.5904761905 |
| 0.498015873 | 0.7583081571 | 0.889 | 0.6011976048 |
| 0.487804878 | 0.7185628743 | 0.8846666667 | 0.5811138015 |
| 0.4905263158 | 0.7420382166 | 0.8923333333 | 0.5906210393 |
| 0.462962963 | 0.7097791798 | 0.8823333333 | 0.5603985056 |

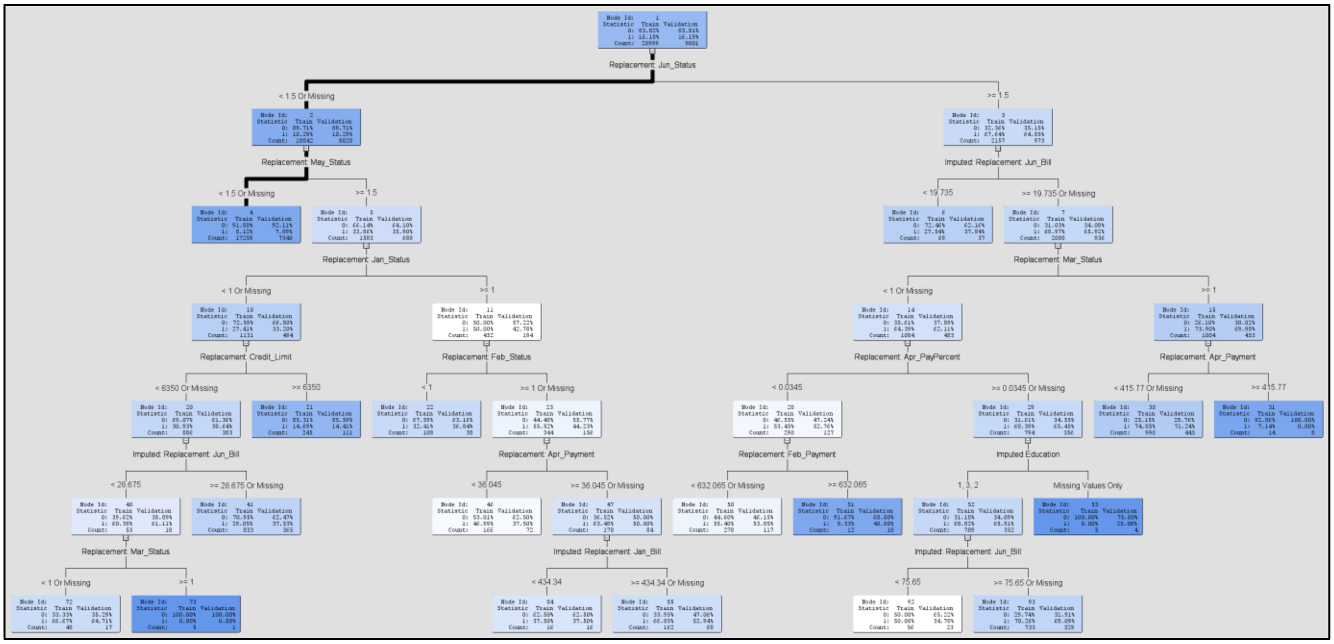
Average Values:
Average Recall: **0.48811588408028783**
Average Precision: **0.7357285137654769**
Average Accuracy: **0.8887666666666668**
Average F1: **0.5864179865041296**

In the regular decision trees, the model with the best F1 score and accuracy is: **12**

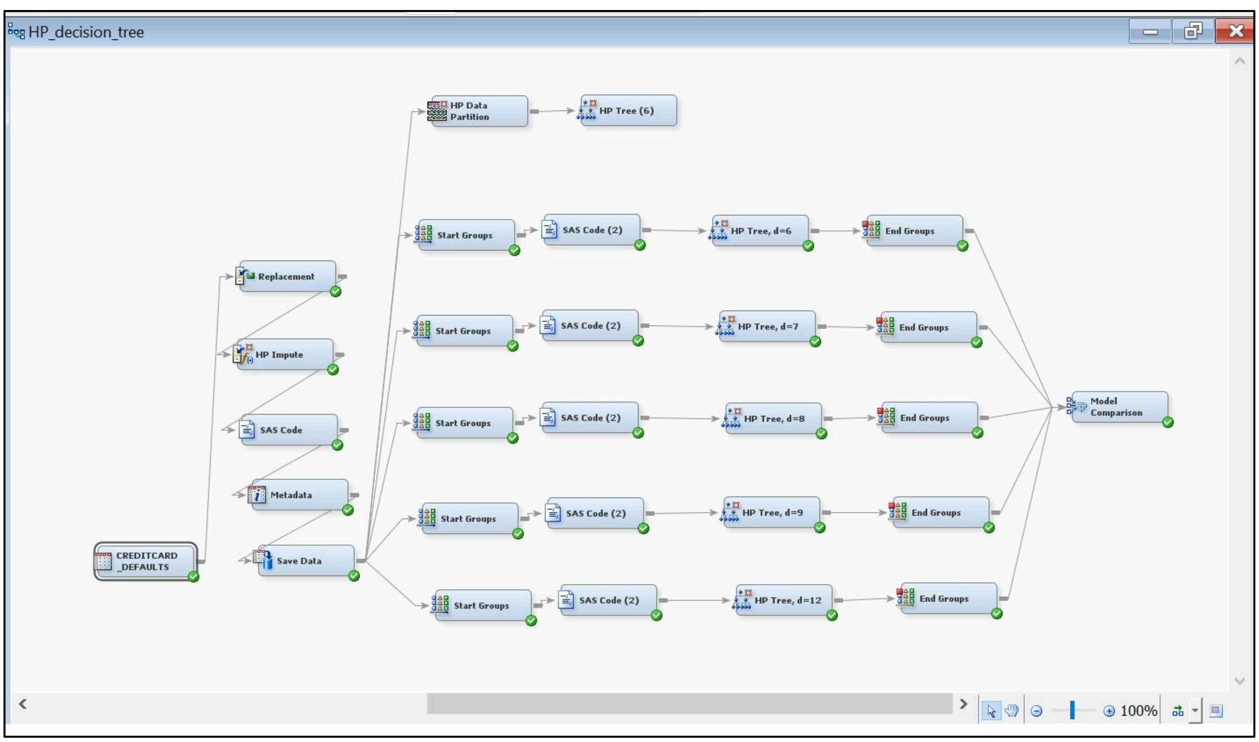
Hence the data is split to train and test and metrics for are obtained:

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.4468085106 | 0.6670081967 | 0.8743472947 | 0.5351418002 |

Tree obtained:



# HP decision tree:



Cross validation metrics obtained:

**HP DECISION TREE**

**Depth = 6**

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.501002004 | 0.7122507123 | 0.8833333333 | 0.5882352941 |
| 0.4599589322 | 0.6607669617 | 0.874 | 0.5423728814 |
| 0.4025974026 | 0.6838235294 | 0.8793333333 | 0.5068119891 |
| 0.41995842 | 0.6412698413 | 0.8693333333 | 0.5075376884 |
| 0.4270386266 | 0.6567656766 | 0.8763333333 | 0.5175552666 |
| 0.4501992032 | 0.6827794562 | 0.873 | 0.5426170468 |
| 0.4742063492 | 0.7029411765 | 0.878 | 0.5663507109 |
| 0.4573170732 | 0.6656804734 | 0.8733333333 | 0.5421686747 |
| 0.44 | 0.6897689769 | 0.88 | 0.5372750643 |
| 0.4341563786 | 0.6940789474 | 0.8773333333 | 0.5341772152 |

Average Values:
Average Recall: 0.4466434389576629
Average Precision: 0.6790125751454725
Average Accuracy: 0.8764
Average F1: **0.5385101831471805**

**Depth = 7**

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.50501002 | 0.6980609418 | 0.8813333333 | 0.5860465116 |
| 0.4579055441 | 0.6757575758 | 0.8763333333 | 0.5458996328 |
| 0.4242424242 | 0.675862069 | 0.88 | 0.5212765957 |
| 0.4241164241 | 0.6276923077 | 0.8673333333 | 0.5062034739 |
| 0.4334763948 | 0.6516129032 | 0.876 | 0.5206185567 |
| 0.4422310757 | 0.6788990826 | 0.8716666667 | 0.5355850422 |
| 0.4543650794 | 0.7089783282 | 0.877 | 0.553808948 |
| 0.4613821138 | 0.6560693642 | 0.872 | 0.5417661098 |
| 0.44 | 0.6897689769 | 0.88 | 0.5372750643 |
| 0.4341563786 | 0.6940789474 | 0.8773333333 | 0.5341772152 |

Average Values:

Average Recall: **0.44768854548808107**
Average Precision: 0.6756780496639605
Average Accuracy: 0.8759
Average F1: 0.5382657150288772

**Depth = 8**

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.50501002 | 0.6980609418 | 0.8813333333 | 0.5860465116 |
| 0.4435318275 | 0.6878980892 | 0.877 | 0.5393258427 |
| 0.4199134199 | 0.6830985915 | 0.8806666667 | 0.5201072386 |
| 0.4324324324 | 0.6380368098 | 0.8696666667 | 0.5154894672 |
| 0.4377682403 | 0.6601941748 | 0.8776666667 | 0.5264516129 |
| 0.4282868526 | 0.6847133758 | 0.8713333333 | 0.5269607843 |
| 0.4523809524 | 0.7058823529 | 0.8763333333 | 0.5513905683 |
| 0.4349593496 | 0.6584615385 | 0.8703333333 | 0.5238678091 |
| 0.44 | 0.6897689769 | 0.88 | 0.5372750643 |
| 0.4176954733 | 0.719858156 | 0.8793333333 | 0.5286458333 |

Average Values:
Average Recall: 0.44119785680597995
Average Precision: 0.682597300724771
Average Accuracy: 0.8763666666666665
Average F1: 0.5355560732287155

**Depth = 9**

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.50501002 | 0.6980609418 | 0.8813333333 | 0.5860465116 |
| 0.4435318275 | 0.6878980892 | 0.877 | 0.5393258427 |
| 0.4177489177 | 0.6942446043 | 0.882 | 0.5216216216 |
| 0.4241164241 | 0.6455696203 | 0.8703333333 | 0.5119196989 |
| 0.4034334764 | 0.6596491228 | 0.875 | 0.500665779 |
| 0.4282868526 | 0.6847133758 | 0.8713333333 | 0.5269607843 |
| 0.4623015873 | 0.7082066869 | 0.8776666667 | 0.5594237695 |
| 0.4532520325 | 0.6539589443 | 0.871 | 0.5354141657 |
| 0.44 | 0.6897689769 | 0.88 | 0.5372750643 |
| 0.4176954733 | 0.719858156 | 0.8793333333 | 0.5286458333 |

Average Values:
Average Recall: 0.43953766114782555
Average Precision: **0.6841928518310813**
**A**verage Accuracy: **0.8765000000000001**
Average F1: 0.5347299070866788

**Depth = 12**

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.50501002 | 0.6980609418 | 0.8813333333 | 0.5860465116 |
| 0.4435318275 | 0.6900958466 | 0.8773333333 | 0.54 |
| 0.4523809524 | 0.6656050955 | 0.8806666667 | 0.5386597938 |
| 0.4054054054 | 0.6478405316 | 0.8693333333 | 0.4987212276 |
| 0.4034334764 | 0.6596491228 | 0.875 | 0.500665779 |
| 0.4282868526 | 0.6847133758 | 0.8713333333 | 0.5269607843 |
| 0.4761904762 | 0.6936416185 | 0.8766666667 | 0.5647058824 |
| 0.4593495935 | 0.6569767442 | 0.872 | 0.5406698565 |
| 0.44 | 0.6897689769 | 0.88 | 0.5372750643 |
| 0.4279835391 | 0.7074829932 | 0.8786666667 | 0.5333333333 |

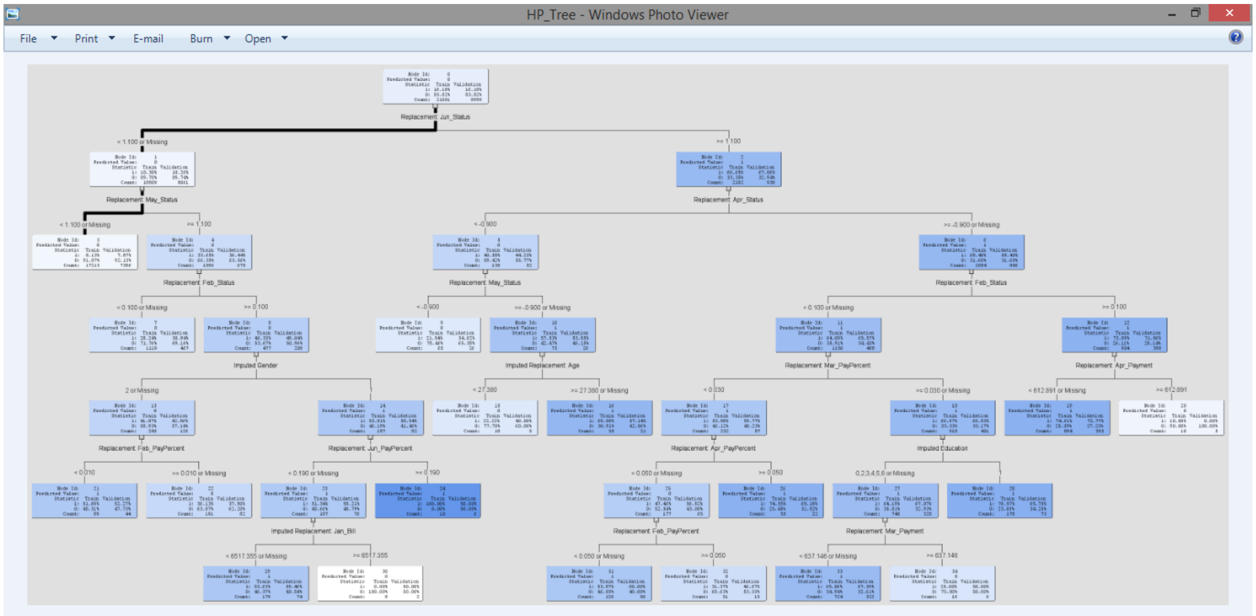
Average Values:
Average Recall: 0.4441572143107392
Average Precision: 0.6793835246957807
Average Accuracy: 0.8762333333333334
Average F1: 0.5367038232751891

As observed, the best model obtained was for **depth 6**. This is different from regular decision trees as the imputation for HP trees is done by mean or mode while for regular trees imputation is done by trees.

The testing metrics for backward regression obtained upon splitting to train and test:

| Recall | Precision | Accuracy | F1 | |
|---|---|---|---|---|
| 0.4656593407 | 0.6746268657 | 0.8772085787 | 0.5509955303 | |

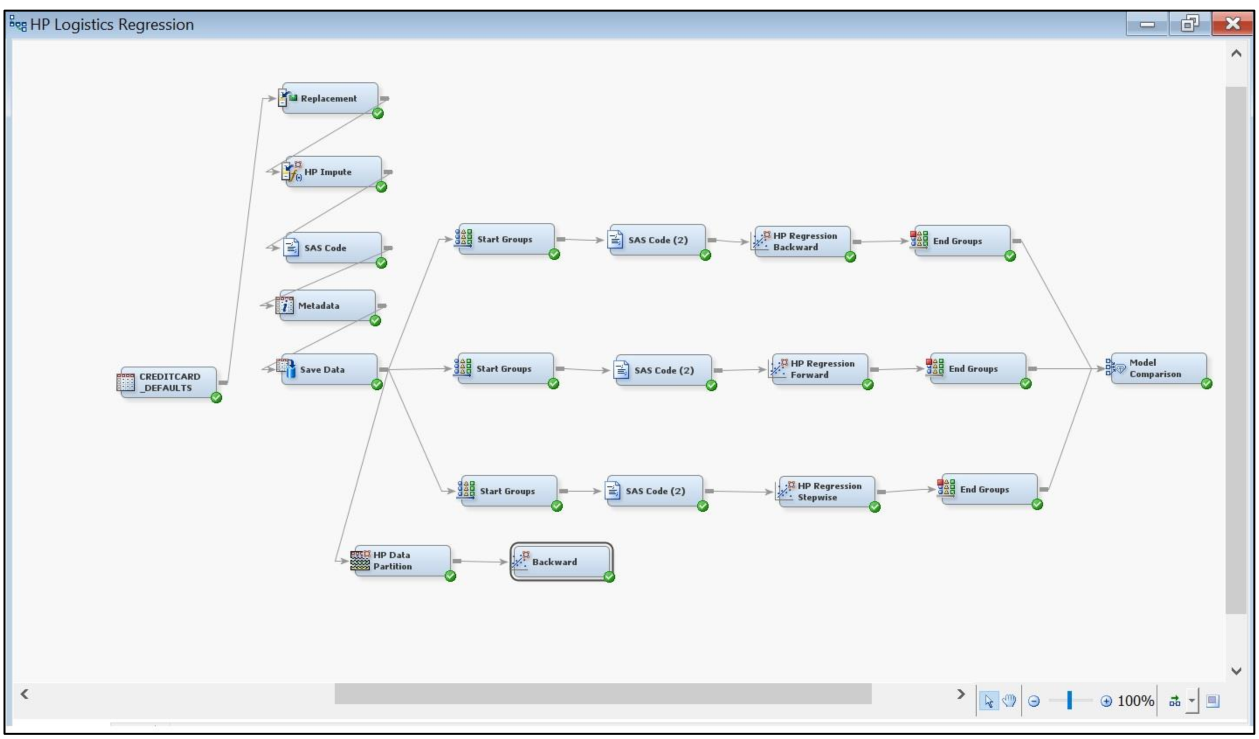
The tree obtained:

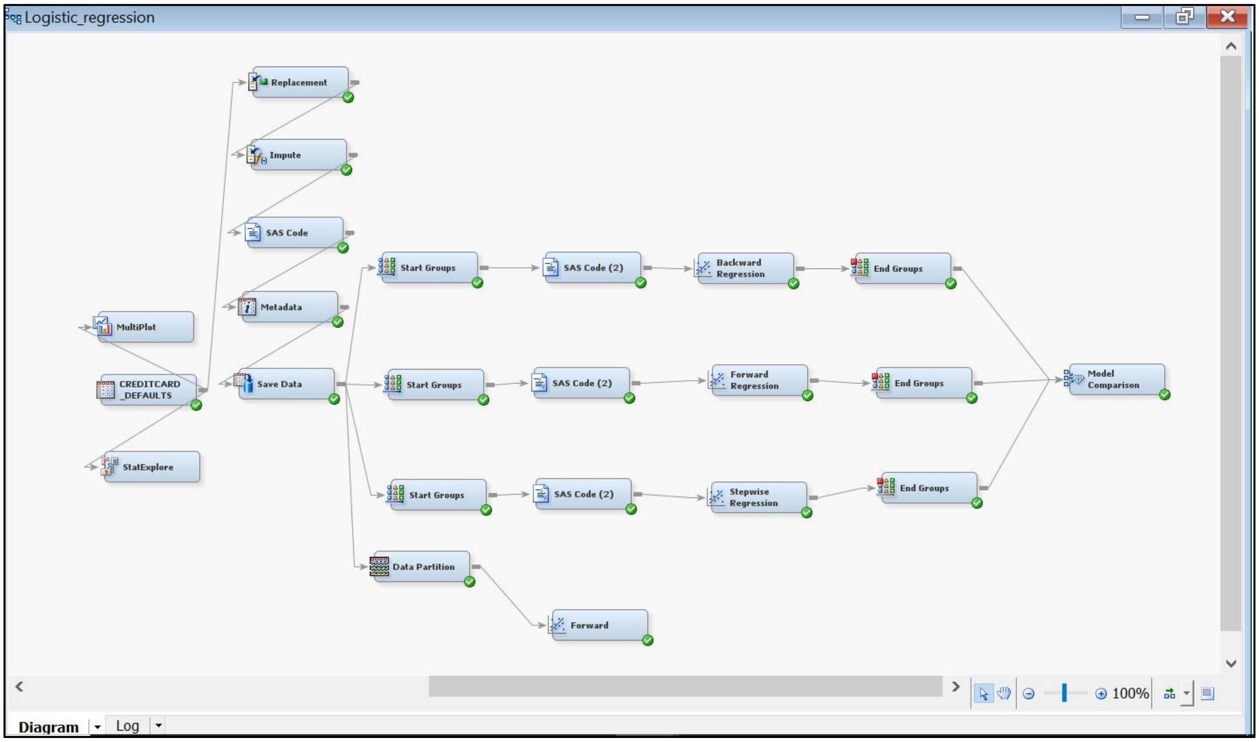



# HP Logistic Regression and Regular logistic regression:

The only difference between them is the use of HP nodes for imputation and logistic regression.

SAS Diagram: -






Cross validation metrics obtained for Non HP:

## LOGISTIC REGRESSION

### BACKWARD REGRESSION - Non HP

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.2945891784 | 0.7033492823 | 0.862 | 0.4152542373 |
| 0.2628336756 | 0.6530612245 | 0.8576666667 | 0.3748169839 |
| 0.2792207792 | 0.6515151515 | 0.866 | 0.3909090909 |
| 0.2848232848 | 0.6431924883 | 0.86 | 0.3948126801 |
| 0.2725321888 | 0.6479591837 | 0.864 | 0.3836858006 |
| 0.2988047809 | 0.6912442396 | 0.8603333333 | 0.4172461752 |
| 0.2936507937 | 0.635193133 | 0.853 | 0.4016282225 |
| 0.2825203252 | 0.681372549 | 0.8606666667 | 0.3994252874 |
| 0.2926315789 | 0.6915422886 | 0.8673333333 | 0.4112426036 |
| 0.2716049383 | 0.6947368421 | 0.8626666667 | 0.3905325444 |

Average Values:
Average Recall: 0.2833211523756174
Average Precision: 0.669316638259861
Average Accuracy: 0.8613666666666667
Average F1: 0.39795536258637665

**FORWARD REGRESSION - Non HP**

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.2945891784 | 0.7033492823 | 0.862 | 0.4152542373 |
| 0.2628336756 | 0.6530612245 | 0.8576666667 | 0.3748169839 |
| 0.2792207792 | 0.6515151515 | 0.866 | 0.3909090909 |
| 0.2848232848 | 0.6431924883 | 0.86 | 0.3948126801 |
| 0.2725321888 | 0.6479591837 | 0.864 | 0.3836858006 |
| 0.2988047809 | 0.6912442396 | 0.8603333333 | 0.4172461752 |
| 0.2936507937 | 0.635193133 | 0.853 | 0.4016282225 |
| 0.2825203252 | 0.681372549 | 0.8606666667 | 0.3994252874 |
| 0.2947368421 | 0.6930693069 | 0.8676666667 | 0.4135893648 |
| 0.2716049383 | 0.6947368421 | 0.8626666667 | 0.3905325444 |

Average Values:
Average Recall: **0.2835316786914069**
Average Precision**: 0.6694693400972088**
Average Accuracy: **0.8614**
Average F1: **0.3981900387158375**

**STEPWISE REGRESSION - Non HP**

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.2945891784 | 0.7033492823 | 0.862 | 0.4152542373 |
| 0.2628336756 | 0.6530612245 | 0.8576666667 | 0.3748169839 |
| 0.2792207792 | 0.6515151515 | 0.866 | 0.3909090909 |
| 0.2848232848 | 0.6431924883 | 0.86 | 0.3948126801 |
| 0.2725321888 | 0.6479591837 | 0.864 | 0.3836858006 |
| 0.2988047809 | 0.6912442396 | 0.8603333333 | 0.4172461752 |
| 0.2936507937 | 0.635193133 | 0.853 | 0.4016282225 |
| 0.2825203252 | 0.681372549 | 0.8606666667 | 0.3994252874 |
| 0.2947368421 | 0.6930693069 | 0.8676666667 | 0.4135893648 |
| 0.2716049383 | 0.6947368421 | 0.8626666667 | 0.3905325444 |

Average Values:
Average Recall: 0.2833211523756174
Average Precision: 0.669316638259861
Average Accuracy: 0.8613666666666667
Average F1: 0.39795536258637665

The best model obtained from Non- HP regression was of **Forward Regression** based on F1 score and accuracy.

The testing metrics for forward regression obtained upon splitting to train and test:

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.2910089224 | 0.6366366366 | 0.8583490723 | 0.3994347621 |

## HP Logistic Regression:

**BACKWARD REGRESSION – HP**

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.2965931864 | 0.7047619048 | 0.8623333333 | 0.4174894217 |
| 0.2648870637 | 0.6581632653 | 0.8583333333 | 0.3777452416 |
| 0.2813852814 | 0.6565656566 | 0.8666666667 | 0.3939393939 |
| 0.2869022869 | 0.6478873239 | 0.8606666667 | 0.3976945245 |
| 0.2682403433 | 0.641025641 | 0.863 | 0.378214826 |
| 0.296812749 | 0.6898148148 | 0.86 | 0.4150417827 |
| 0.2956349206 | 0.6422413793 | 0.854 | 0.4048913043 |
| 0.2886178862 | 0.6826923077 | 0.8613333333 | 0.4057142857 |
| 0.2947368421 | 0.6965174129 | 0.868 | 0.4142011834 |
| 0.2674897119 | 0.6914893617 | 0.862 | 0.3857566766 |

Average Values:
Average Recall: **0.284130027152017**
Average Precision: 0.6711159068057906
Average Accuracy: 0.8616333333333334
Average F1: **0.3990688640539977**

**FORWARD REGRESSION – HP**

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.2945891784 | 0.7033492823 | 0.862 | 0.4152542373 |
| 0.2628336756 | 0.6564102564 | 0.858 | 0.3753665689 |
| 0.2813852814 | 0.6565656566 | 0.8666666667 | 0.3939393939 |
| 0.2869022869 | 0.6478873239 | 0.8606666667 | 0.3976945245 |
| 0.2682403433 | 0.641025641 | 0.863 | 0.378214826 |
| 0.296812749 | 0.6898148148 | 0.86 | 0.4150417827 |
| 0.2936507937 | 0.6379310345 | 0.8533333333 | 0.402173913 |
| 0.2865853659 | 0.6811594203 | 0.861 | 0.4034334764 |
| 0.2905263158 | 0.696969697 | 0.8676666667 | 0.4101040119 |
| 0.2674897119 | 0.6914893617 | 0.862 | 0.3857566766 |

Average Values:
Average Recall: 0.2829015701788669
Average Precision: 0.670260248850112
Average Accuracy: 0.8614333333333335
Average F1: 0.3976979411272413

**STEPWISE REGRESSION – HP**

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.2945891784 | 0.7033492823 | 0.862 | 0.4152542373 |
| 0.2628336756 | 0.6564102564 | 0.858 | 0.3753665689 |
| 0.2813852814 | 0.6565656566 | 0.8666666667 | 0.3939393939 |
| 0.2723492723 | 0.6616161616 | 0.861 | 0.3858615611 |
| 0.2703862661 | 0.6331658291 | 0.8623333333 | 0.3789473684 |
| 0.2928286853 | 0.713592233 | 0.862 | 0.4152542373 |
| 0.2936507937 | 0.649122807 | 0.8546666667 | 0.4043715847 |
| 0.2926829268 | 0.6923076923 | 0.8626666667 | 0.4114285714 |
| 0.2926315789 | 0.695 | 0.8676666667 | 0.4118518519 |
| 0.2674897119 | 0.6914893617 | 0.862 | 0.3857566766 |

Average Values:
Average Recall: 0.282082737037092
Average Precision: **0.6752619280071527**
Average Accuracy: **0.8619**
Average F1: 0.39780320515087075

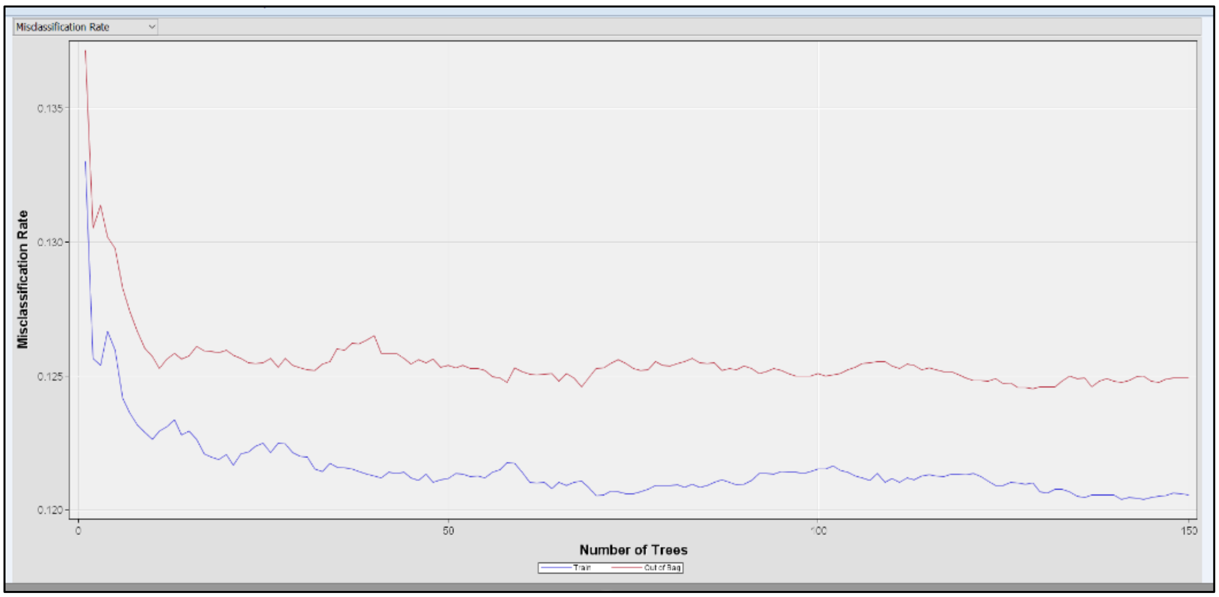
For HP regression, the best model was decided based on F1 score and **Backward Regression** was obtained.

The testing metrics for backward regression obtained upon splitting to train and test:

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.2850274725 | 0.6916666667 | 0.8637626403 | 0.4036964981 |

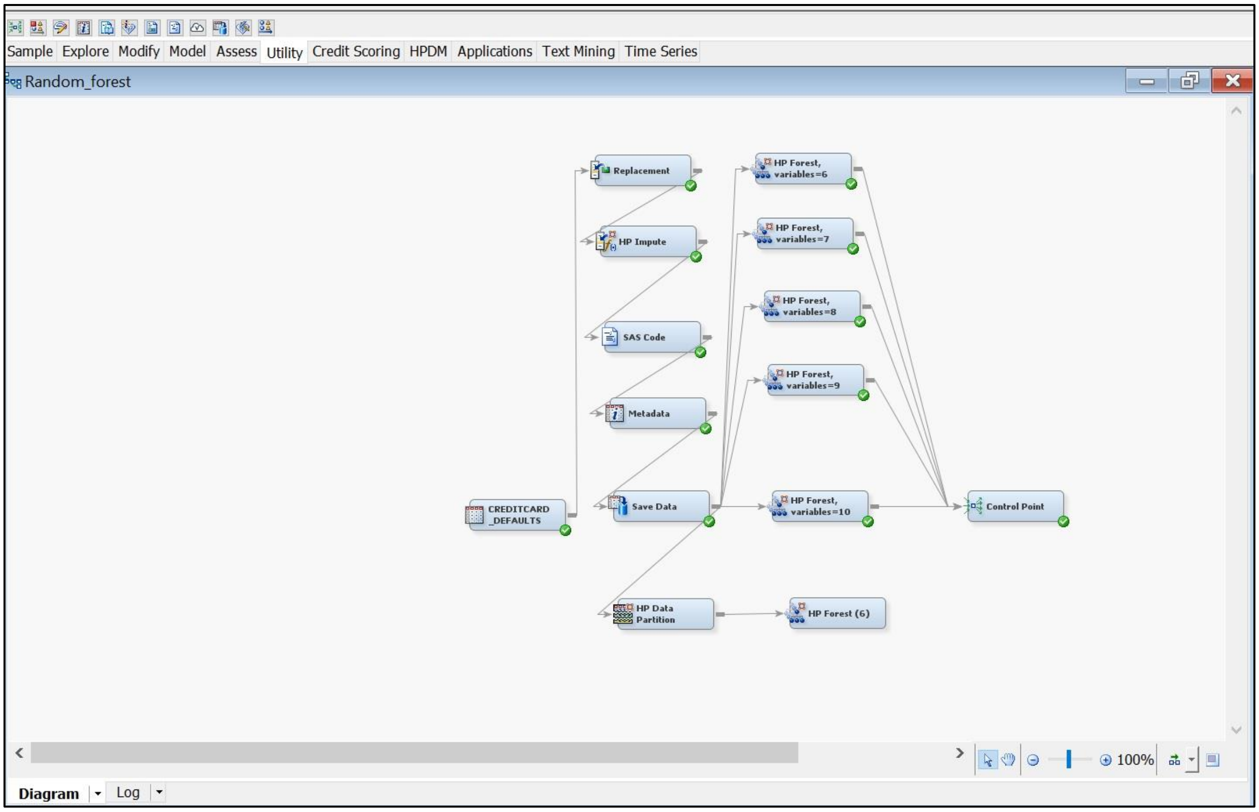
## Random Forest:

In random forests, initially the number of trees was iterated and was found that the misclassification error rate was constant above 40 trees. Hence we take 100 trees. Below is the figure that shows this:




The depth of the tree was also iterated over and the sensitivity compared, it was observed that the model with the best depth of 20 gave a higher sensitivity compared to the other depths, while the accuracy was similar.  Hence we iterate over different number of parameters with depth of 20.

SAS Diagram: -




Cross Validation metrics obtained:

**RANDOM FOREST**

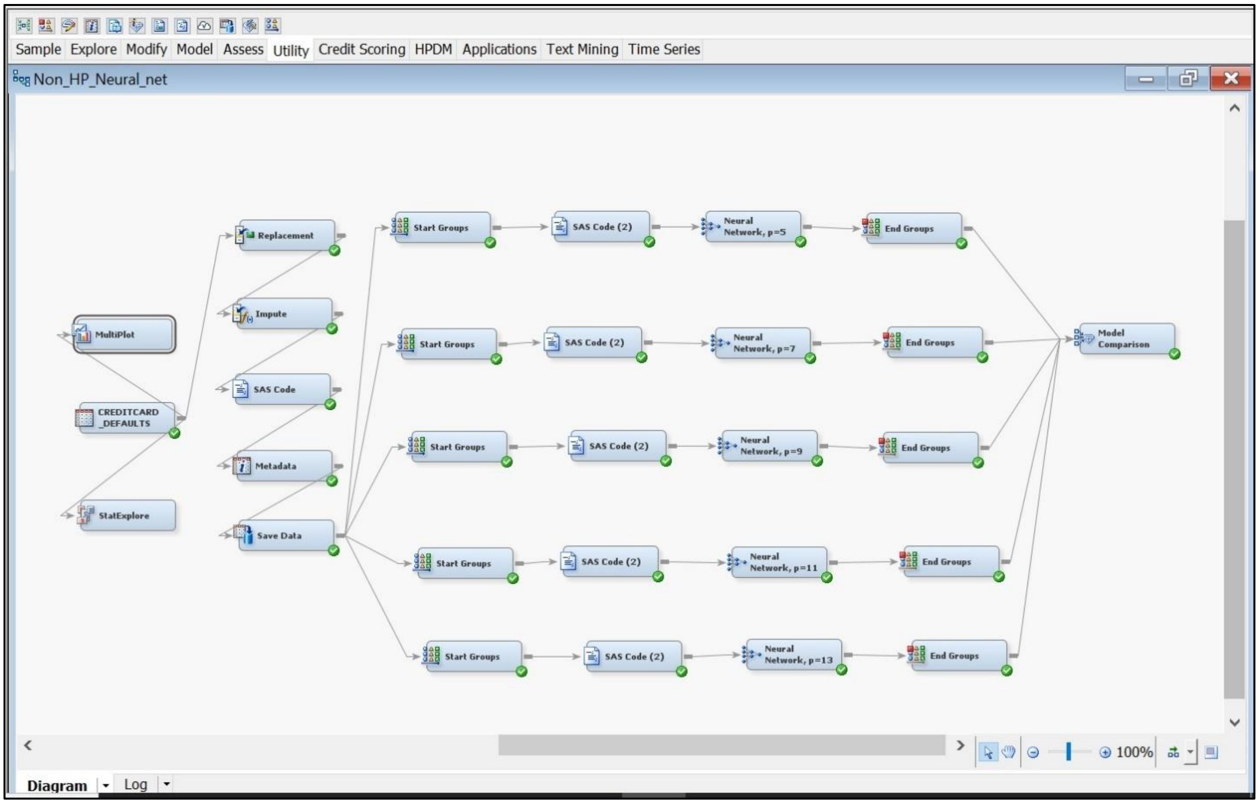
| Recall | Precision | Accuracy | F1 | |
|---|---|---|---|---|
| **0.4653893696** | **0.7000309885** | **0.8812333333** | **0.5590892216** | m = 6 |
| 0.4590028842 | 0.6910669975 | 0.8792666667 | 0.5516216885 | m = 7 |
| 0.4622991347 | 0.6992832658 | 0.8808333333 | 0.5566166439 | m = 8 |
| 0.4587968686 | 0.6946350593 | 0.8798 | 0.5526054591 | m = 9 |
| 0.4590028842 | 0.6997487437 | 0.8806 | 0.5543667579 | m = 10 |

The best model obtained from random forest was for **6 parameters.**

The testing metrics for backward regression obtained upon splitting to train and test:

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.4423076923 | 0.6743455497 | 0.8752083565 | 0.5342181667 |

## Neural Network:




Cross Validation metrics obtained :

### NEURAL NETWORK

### # Perceptron -- 5

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.4809619238 | 0.6976744186 | 0.879 | 0.5693950178 |
| 0.4599589322 | 0.6666666667 | 0.875 | 0.5443499392 |
| 0.4112554113 | 0.6785714286 | 0.8793333333 | 0.5121293801 |
| 0.395010395 | 0.6312292359 | 0.866 | 0.4859335038 |
| 0.4098712446 | 0.6324503311 | 0.8713333333 | 0.4973958333 |
| 0.438247012 | 0.6707317073 | 0.87 | 0.5301204819 |
| 0.4226190476 | 0.6677115987 | 0.8676666667 | 0.517618469 |
| 0.4491869919 | 0.6368876081 | 0.8676666667 | 0.52681764 |
| 0.4505263158 | 0.6881028939 | 0.8806666667 | 0.5445292621 |
| 0.4197530864 | 0.6754966887 | 0.8733333333 | 0.5177664975 |

Average Values:
Average Recall: 0.4337390360637272
Average Precision: **0.664552257761369**
Average Accuracy: **0.873**
Average F1: 0.5246056024803437

**# Perceptron -- 7**

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.4809619238 | 0.6818181818 | 0.8763333333 | 0.5640423032 |
| 0.4435318275 | 0.6605504587 | 0.8726666667 | 0.5307125307 |
| 0.4134199134 | 0.6474576271 | 0.875 | 0.5046235139 |
| 0.41995842 | 0.6412698413 | 0.8693333333 | 0.5075376884 |
| 0.4077253219 | 0.6418918919 | 0.8726666667 | 0.498687664 |
| 0.4262948207 | 0.6564417178 | 0.8666666667 | 0.5169082126 |
| 0.4603174603 | 0.6843657817 | 0.8736666667 | 0.5504151839 |
| 0.4430894309 | 0.646884273 | 0.869 | 0.5259348613 |
| 0.4610526316 | 0.6780185759 | 0.88 | 0.5488721805 |
| 0.4279835391 | 0.6731391586 | 0.8736666667 | 0.5232704403 |

Average Values:
Average Recall: 0.43843352892323406
Average Precision: 0.6611837507740959
Average Accuracy: 0.8728999999999999
Average F1: 0.5271004578648892

**# Perceptron -- 9**

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.4809619238 | 0.6976744186 | 0.879 | 0.5693950178 |
| 0.4599589322 | 0.6666666667 | 0.875 | 0.5443499392 |
| 0.4112554113 | 0.6785714286 | 0.8793333333 | 0.5121293801 |
| 0.395010395 | 0.6312292359 | 0.866 | 0.4859335038 |
| 0.4098712446 | 0.6324503311 | 0.8713333333 | 0.4973958333 |
| 0.438247012 | 0.6707317073 | 0.87 | 0.5301204819 |
| 0.4226190476 | 0.6677115987 | 0.8676666667 | 0.517618469 |
| 0.4491869919 | 0.6368876081 | 0.8676666667 | 0.52681764 |
| 0.4505263158 | 0.6881028939 | 0.8806666667 | 0.5445292621 |

Average Values:
Average Recall: 0.3917637274217519
Average Precision: 0.5970025888871968
Average Accuracy: 0.7856666666666666
Average F1: 0.4728289527341508

**# Perceptron -- 11**

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.4849699399 | 0.6875 | 0.8776666667 | 0.5687426557 |
| 0.4168377823 | 0.6590909091 | 0.8703333333 | 0.5106918239 |
| 0.3874458874 | 0.6729323308 | 0.8766666667 | 0.4917582418 |
| 0.4095634096 | 0.6334405145 | 0.8673333333 | 0.4974747475 |
| 0.4206008584 | 0.6426229508 | 0.8736666667 | 0.5084306096 |
| 0.422310757 | 0.6523076923 | 0.8656666667 | 0.5126964933 |
| 0.4444444444 | 0.6871165644 | 0.8726666667 | 0.5397590361 |
| 0.4613821138 | 0.6579710145 | 0.8723333333 | 0.5424133811 |
| 0.4547368421 | 0.6728971963 | 0.8786666667 | 0.5427135678 |
| 0.4156378601 | 0.6688741722 | 0.872 | 0.5126903553 |

Average Values:
Average Recall: 0.431792989502428
Average Precision: 0.663475334487184
Average Accuracy: 0.8727
Average F1: 0.52273709122157

**# Perceptron -- 13**

| Recall | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.4869739479 | 0.6675824176 | 0.8743333333 | 0.5631517961 |
| 0.4496919918 | 0.6347826087 | 0.8686666667 | 0.5264423077 |
| 0.4220779221 | 0.6456953642 | 0.8753333333 | 0.5104712042 |
| 0.4345114345 | 0.6276276276 | 0.868 | 0.5135135135 |
| 0.4034334764 | 0.5968253968 | 0.865 | 0.4814340589 |
| 0.4322709163 | 0.6656441718 | 0.8686666667 | 0.5241545894 |
| 0.4682539683 | 0.6430517711 | 0.867 | 0.5419058553 |
| 0.4552845528 | 0.64 | 0.8686666667 | 0.5320665083 |
| 0.4378947368 | 0.6887417219 | 0.8796666667 | 0.5353925354 |
| 0.4403292181 | 0.6369047619 | 0.8686666667 | 0.5206812652 |

Average Values:
Average Recall: **0.4430722165049705**
Average Precision: 0.6446855841624879
Average Accuracy: 0.8704000000000001
Average F1: **0.5249213633976965**

As observed, the best model was obtained with **Perceptrons = 13 with one hidden layer**

The testing metrics for backward regression obtained upon splitting to train and test:

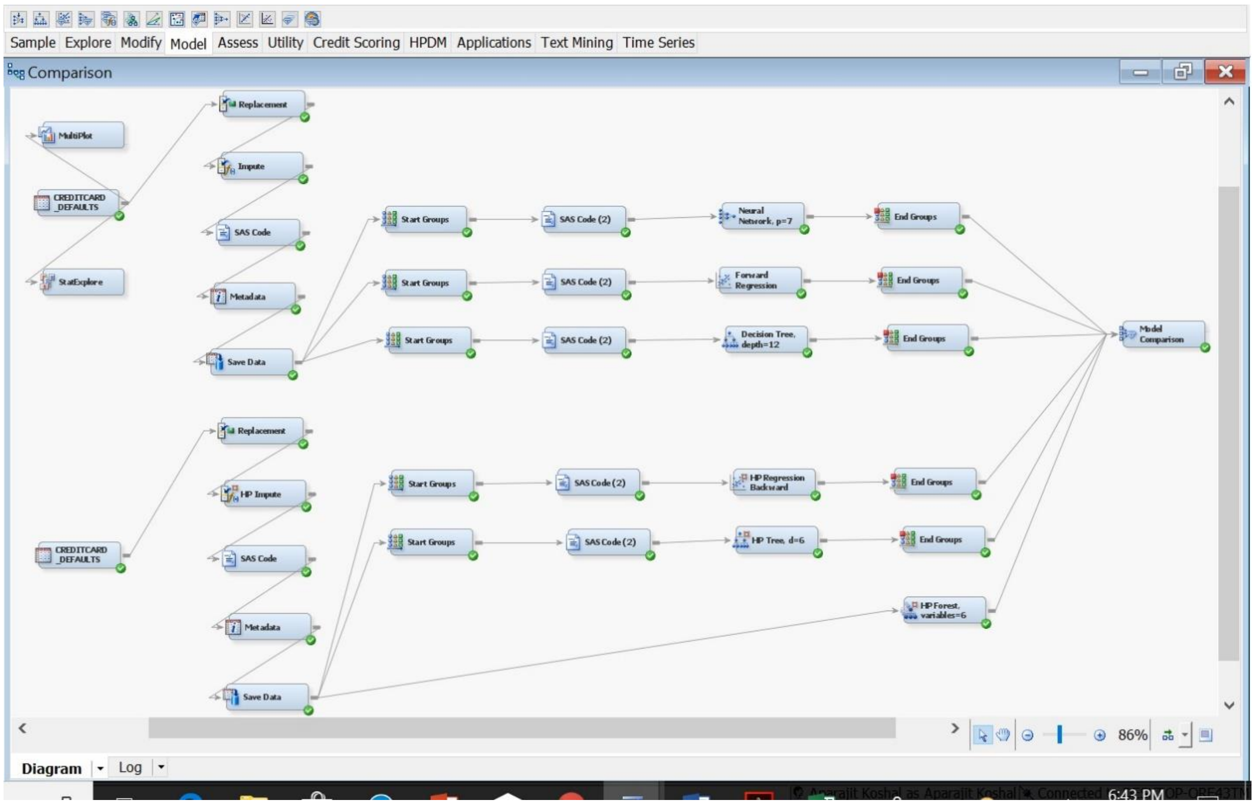
| Sensititity (Recall) | Precision | Accuracy | F1 |
|---|---|---|---|
| 0.440631434 | 0.624514 | 0.8665704 | 0.5167 |

## Final comparison between different algorithms: -

| Models | Sensititity (Recall) | Precision | Accuracy | F1 |
|---|---|---|---|---|
| Neural Network | 0.4430 | 0.6446 | 0.8740 | 0.5167 |
| Random Forest | 0.4653 | 0.7000 | 0.8812 | 0.5590 |
| **Decision Tree** | **0.4881** | **0.7357** | **0.8887** | **0.5864** |
| HP Decision Tree | 0.4466 | 0.6790 | 0.8764 | 0.5385 |
| HP Logistic Regression | 0.2841 | 0.6711 | 0.8616 | 0.3990 |
| Logistic Regression | 0.2835 | 0.6694 | 0.8614 | 0.3981 |

As can be seen above, decision tree model is the winner amongst all scores. We also carried out this analysis on SAS (attached a screenshot of the diagram). With higher sensitivity and highest accuracy Decision Trees is our best model according to SAS.
The output from SAS cannot be compared to Python because **data preprocessing in Python** carried out by us is very different than SAS.

Hence output from both the software cannot be compared explicitly.




Python Solution is as follows: -

Please note that decision tree diagram has been attached at the very end.

# Python_Solution

March 8, 2018

In [3]: data.head()

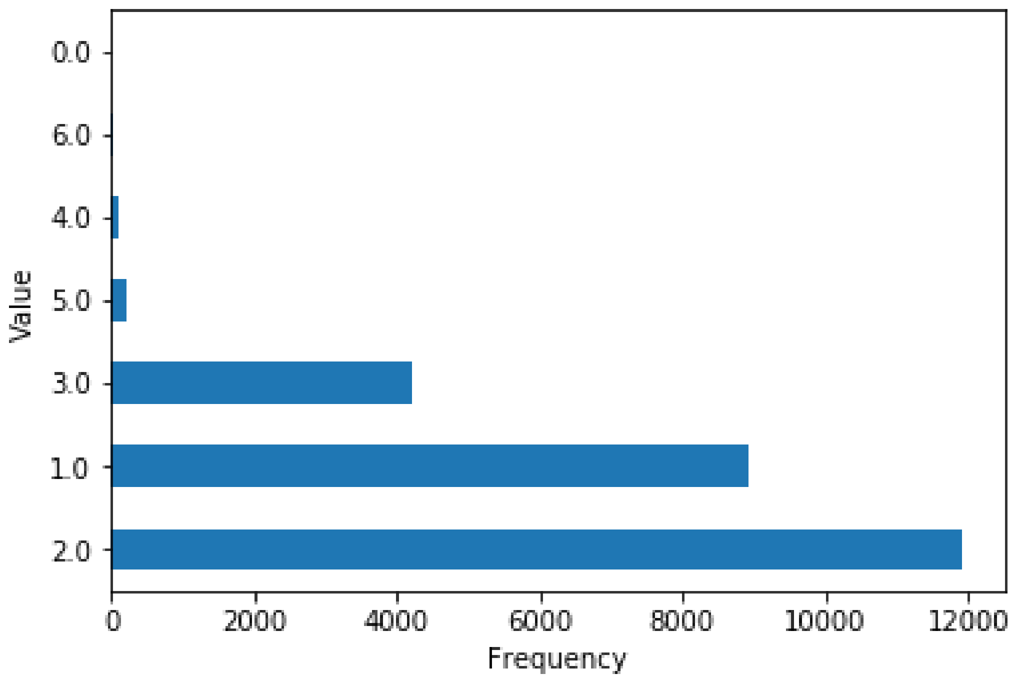
```
Out[3]:    Customer  Default  card_class  Gender  Education  Marital_Status   Age  \
        0         1        1           1     2.0        2.0               1  24.0
        1         2        0           2     2.0        2.0               2  26.0
        2         3        0           2     2.0        2.0               2  34.0
        3         4        0           1     2.0        2.0               1  37.0
        4         5        0           1     1.0        2.0               1  57.0

           Credit_Limit  Jun_Status  May_Status  Apr_Status  Mar_Status  Feb_Status  \
        0           700           2           2          -1          -1          -2
        1          4100          -1           2           0           0           0
        2          3100           0           0           0           0           0
        3          1700           0           0           0           0           0
        4          1700          -1           0          -1           0           0

           Jan_Status  Jun_Bill  May_Bill  Apr_Bill  Mar_Bill  Feb_Bill  Jan_Bill  \
        0          -2    133.82    106.09     23.56      0.00      0.00      0.00
        1           2     91.72     59.00     91.72    111.90    118.16    111.53
        2           0    999.97    479.72    463.72    490.12    511.22    531.78
        3           0   1607.06   1649.57   1685.75    968.34    990.40   1010.51
        4           0    294.70    193.91   1225.56    716.15    654.79    654.28

           Jun_Payment  May_Payment  Apr_Payment  Mar_Payment  Feb_Payment  \
        0         0.00        23.56         0.00         0.00         0.00
        1         0.00        34.20        34.20        34.20         0.00
        2        51.92        51.30        34.20        34.20        34.20
        3        68.40        69.05        41.04        37.62        36.56
        4        68.40      1254.49       342.00       307.80        23.56

           Jan_Payment  Jun_PayPercent  May_PayPercent  Apr_PayPercent  \
        0         0.00          0.0000          0.2221          0.0000
        1        68.40          0.0000          0.5797          0.3729
        2       171.00          0.0519          0.1069          0.0738
        3        34.20          0.0426          0.0419          0.0243
        4        23.22          0.2321          1.0000          0.2791

           Mar_PayPercent  Feb_PayPercent  Jan_PayPercent
```

| | | | |
|---|---|---|---|
| 0 | 1.0000 | 1.0000 | 1.0000 |
| 1 | 0.3056 | 0.0000 | 0.6133 |
| 2 | 0.0698 | 0.0669 | 0.3216 |
| 3 | 0.0388 | 0.0369 | 0.0338 |
| 4 | 0.4298 | 0.0360 | 0.0355 |

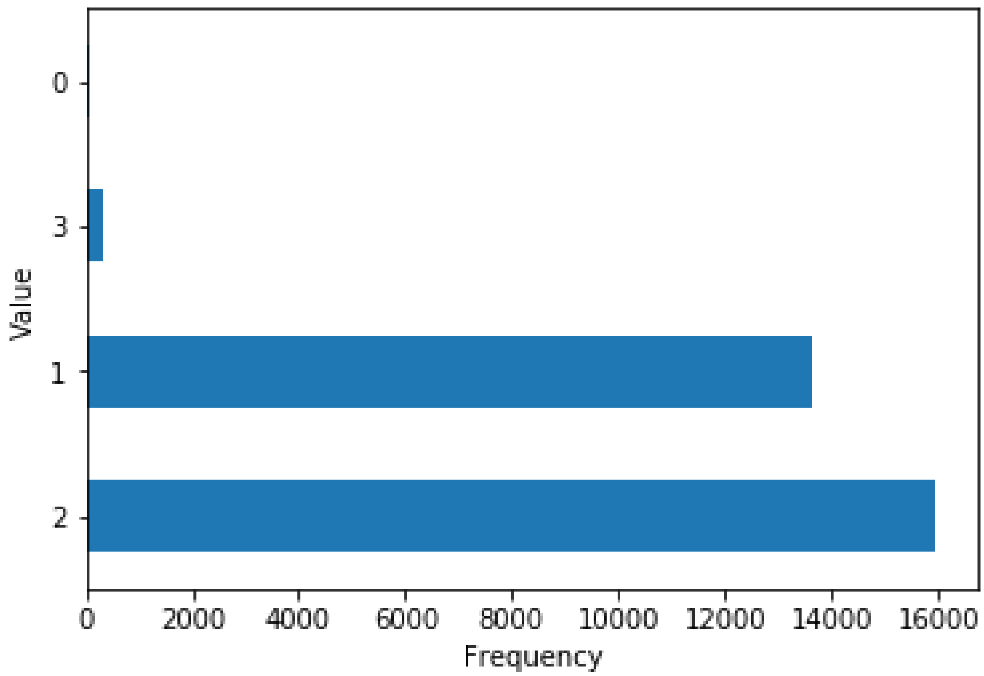
# 1 Data Pre-Processing

We will explore the data first and will try to modify the variables that make more sense in the dataset. Let's first look at the bar graphs of variables 'Education' and 'Marital_Status'.

```
In [12]: barplot('Education')
```




```
In [13]: barplot('Marital_Status')
```




Bar graphs reveal that some classes in columns 'Education' and 'Marital_Status' occur far more less frequently than its counterparts. So, we can bin these less frequent observations into a new value.

1. Observations having value 0,5,6 under Education column can be clubbed together into the value 4.
2. Observations having value 0,3 under 'Marital_Status'column can be clubbed together into the value 3.

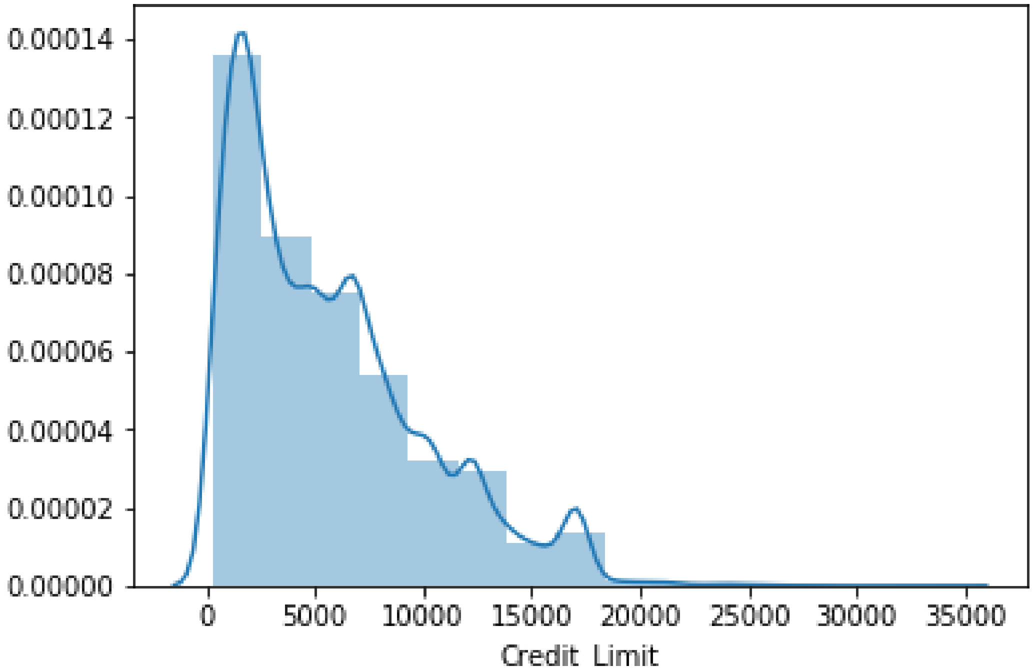
Binning reduces the final number of columns in the dataframe without much loss of information.

The 'Age' column is similarly binned into 6 factors as follows:
Age>=20 and Age<30 = 1
Age>=30 and Age<40 = 2
Age>=40 and Age<50 = 3
Age>=50 and Age<60 = 4
Age>=60 and Age<70 = 5
Age>=70 and Age<81 = 6

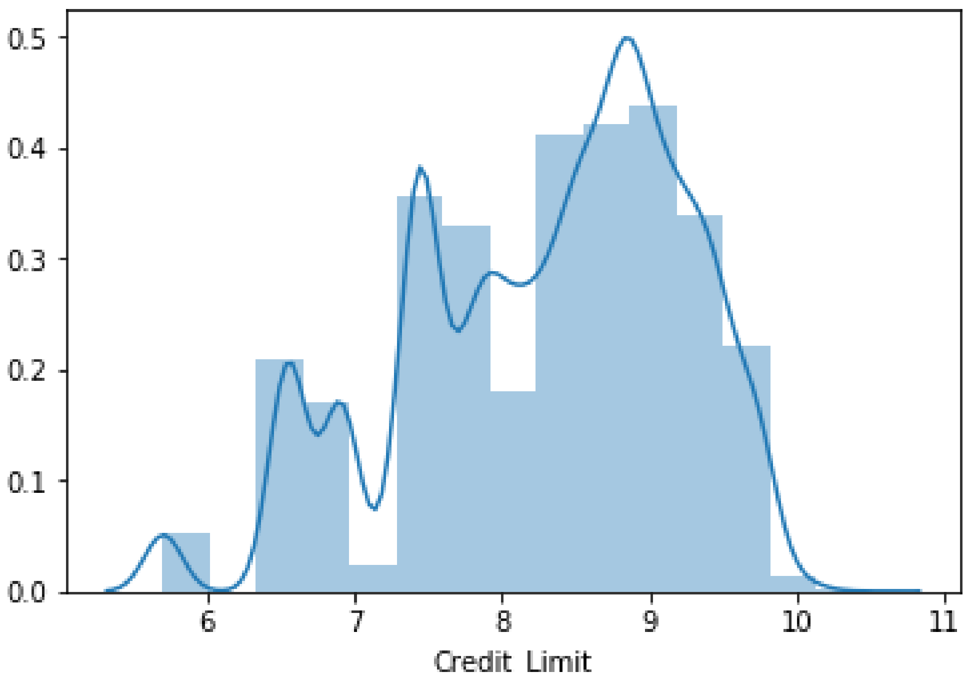
The column 'Credit_Limit' can be seen is skewed towards the right. To handle this skewness the variable is transformed using a log transformation, which makes it evenly distributed.

The histogram that are plotted below are good visualiztions of the original skewness and normalization after transformation.

```
In [19]: sns.distplot(data['Credit_Limit'], bins=15);
```




```
In [20]: sns.distplot(np.log(data['Credit_Limit']+1), bins=15);  ## After applying Log
         transformation
```



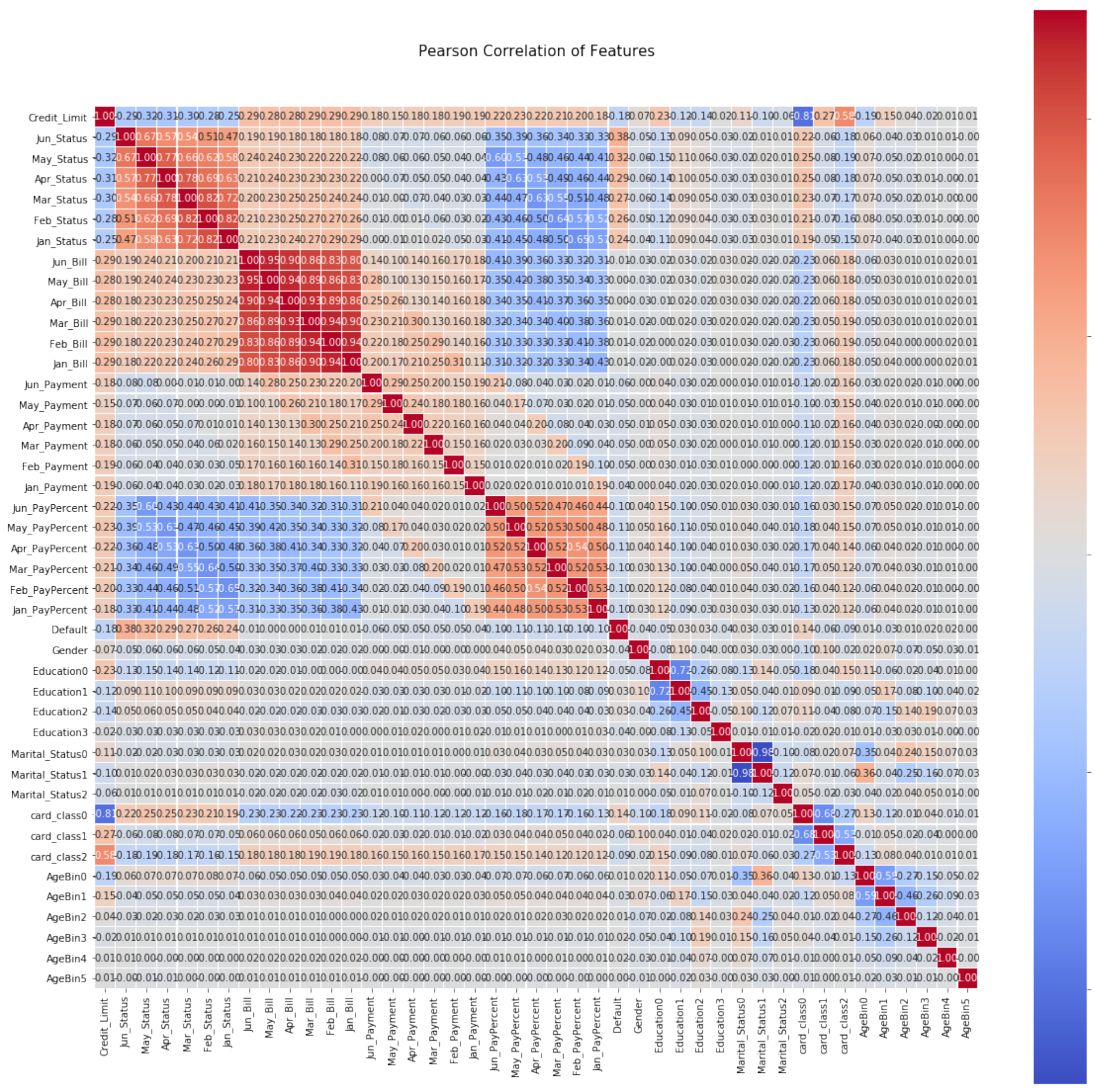
The following step replaces, imputes and performs one-hot encoding using the ReplaceImputeEncode class.

Also, log transformation is applied on the 'Credit_Limit' column to deal with its skewness.

Now to improve the quality of prediction, we have to deal with columns which have a high correlation amongst themselves.

To check which all columns have high correlation we plot the Pearson correlation matrix.

```
In [28]: CorrelationPlot(encoded_df)
```



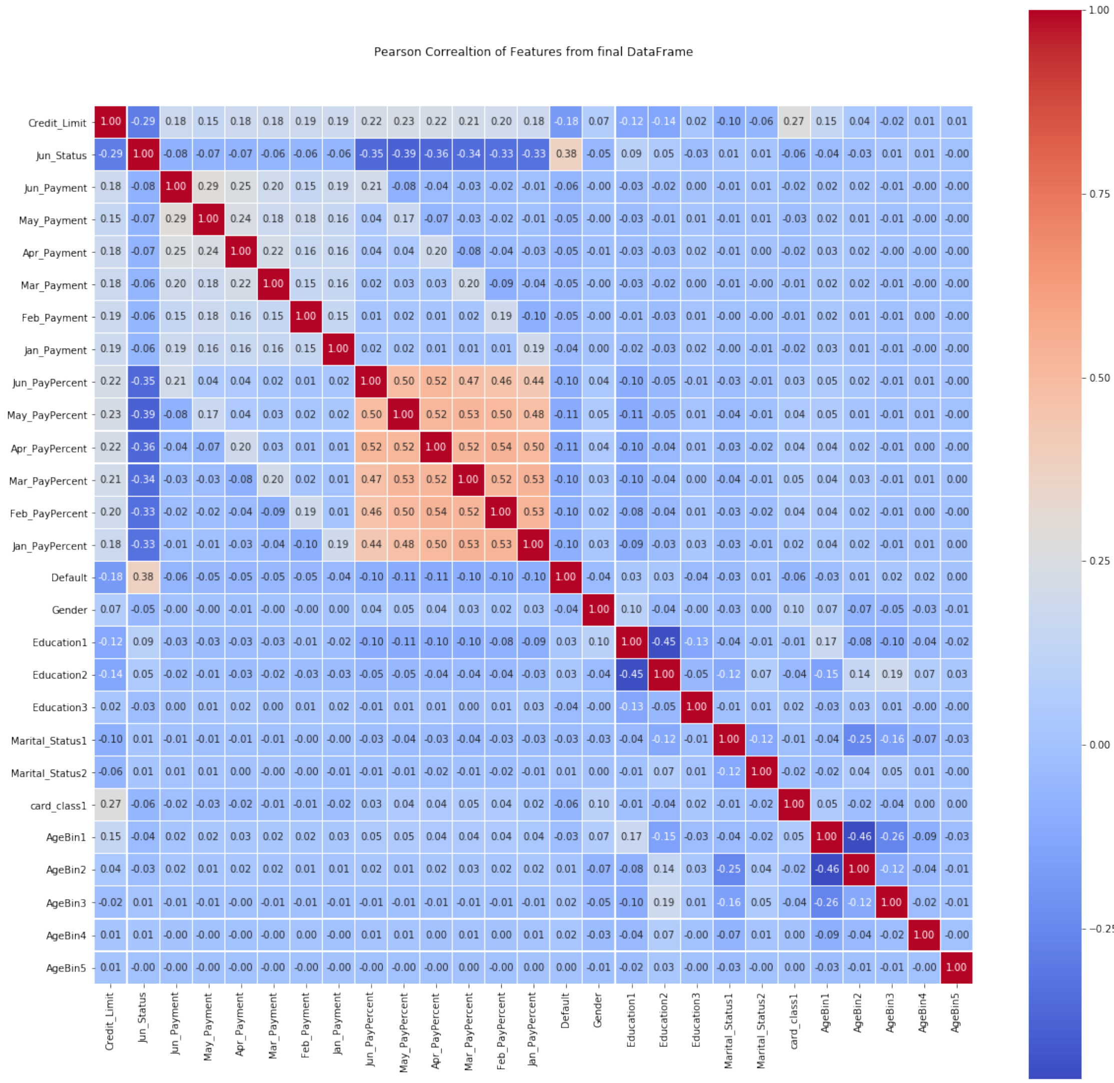

Looking at the correlation between the variabls using the correlation matrix, we can see that following variables seem to be correlated to each other :

1. card_class0 and card_class2 with Credit_Limit
2. Marital_Status0 with Marital_Status1
3. May_Status, Apr_Status, Mar_Status, Feb_Status, Jan_Status with Jun_Status, May_Status, Apr_Status, Mar_Status, Feb_Status respectively.

4. Education0 with Education1 and Education2
5. AgeBin0 with AgeBin1
6. Every month Bill is related with the last month's bill.

So, we can safely drop the above variables without affected the accuracy of the predictive models.

```
In [29]: CorrelationPlot(encoded_df1)
         plt.title('Pearson Correaltion of Features from final DataFrame')
         plt.show();
```




## 2 Prediction Model Building

Now, we will first perform cross validation on Logistic Regression, Random Forest, Decision Tree and Neural Networks to optimize the hyperparameters.

Since, the target variable in our dataset is unbalanced with 5000 default cases as opposed to 25000 non-defaulters, we will use class_weight='Balanced' in Logistic Regression, Random Forest and Decision Tree and this can be seen in the cross validated result of these three models below.

We will cross-validate Logistic Regression using two different solvers i.e 'sag' and 'newton-cg' and see which solver reach the local minima first.

In Random Forest, we will cross-validate using different number of tree and maximum number of features.

In Decision tree, we will identify the best max_depth.

And in Neural networks, we will cross-validate on different number of hidden layers and perceptrons.

```
In [41]: ## Splitting data into 70% training and 30% validation
         X_train, X_validate, y_train, y_validate = train_test_split(X,y,test_size = 0.3,
         random_state=7)
```

### 2.0.1 1. Logistic Regression

```
In [31]: ## Cross-validation on Logistic Regression for two different Class weights:
         LR()
```

```
Class Weight:  None
accuracy... 0.8595    0.0047
recall... 0.2429    0.0342
precision... 0.6886    0.0410
f1... 0.3576    0.0368
Class Weight:  balanced
accuracy... 0.7595    0.0251
recall... 0.6790    0.0370
precision... 0.3705    0.0347
f1... 0.4786    0.0327
```

```
Out[31]: ('Best class weight: ', 'balanced')
```

```
In [21]: ## Validating on the test data
         print("\nTraining Data\nRandom Selection of 70% of Original Data")
         logreg.display_binary_split_metrics(lgr_train, X_train, y_train, \
                                             X_validate, y_validate)
```

```
Training Data
Random Selection of 70% of Original Data


Model Metrics...       Training     Validation
Observations...          21000          9000
Coefficients...             27            27
DF Error...          20973           8973
Mean Absolute Error...        0.3775        0.3827
Avg Squared Error...        0.1813        0.1864
Accuracy...         0.7578        0.7540
Precision...         0.3704        0.3536
Recall (Sensitivity)...        0.6890        0.6711
F1-score...         0.4818        0.4631
MISC (Misclassification)...      24.2%          24.6%
     class 0...       22.9%          23.0%
     class 1...       31.1%          32.9%
```

```
Training
Confusion Matrix  Class 0   Class 1
Class 0...     13550      4019
Class 1...      1067      2364


Validation
Confusion Matrix  Class 0   Class 1
Class 0...      5831      1746
Class 1...       468       955
```

### 2.0.2 2. Random Forest Classifier

In [38]:
```
## Cross-validation on Random Forest for two different trees and Max number of features
RF()
```

```
Number of Trees:  10  Max_features:  auto
Metric...  Mean    Std. Dev.
accuracy... 0.8672    0.0065
recall... 0.3441    0.0382
precision... 0.6784    0.0477
f1... 0.4550    0.0354

Number of Trees:  10  Max_features:  0.3
Metric...  Mean    Std. Dev.
accuracy... 0.8701    0.0075
recall... 0.3616    0.0296
precision... 0.6895    0.0506
f1... 0.4737    0.0323

Number of Trees:  10  Max_features:  0.5
Metric...  Mean    Std. Dev.
accuracy... 0.8690    0.0064
recall... 0.3657    0.0204
precision... 0.6787    0.0481
f1... 0.4745    0.0213

Number of Trees:  10  Max_features:  0.7
Metric...  Mean    Std. Dev.
accuracy... 0.8711    0.0072
recall... 0.3778    0.0214
precision... 0.6870    0.0504
f1... 0.4869    0.0247

Number of Trees:  15  Max_features:  auto
Metric...  Mean    Std. Dev.
accuracy... 0.8715    0.0073
recall... 0.3976    0.0366
precision... 0.6778    0.0499
f1... 0.4997    0.0315

Number of Trees:  15  Max_features:  0.3
Metric...  Mean    Std. Dev.
accuracy... 0.8730    0.0074
recall... 0.4079    0.0271
```

```
precision... 0.6812    0.0468
f1... 0.5095    0.0277

Number of Trees:  15  Max_features:  0.5
Metric...  Mean    Std. Dev.
accuracy... 0.8726    0.0071
recall... 0.4122    0.0259
precision... 0.6764    0.0464
f1... 0.5113    0.0241

Number of Trees:  15  Max_features:  0.7
Metric...  Mean    Std. Dev.
accuracy... 0.8729    0.0071
recall... 0.4203    0.0257
precision... 0.6735    0.0431
f1... 0.5169    0.0257

Number of Trees:  20  Max_features:  auto
Metric...  Mean    Std. Dev.
accuracy... 0.8728    0.0061
recall... 0.3797    0.0363
precision... 0.6992    0.0460
f1... 0.4906    0.0306

Number of Trees:  20  Max_features:  0.3
Metric...  Mean    Std. Dev.
accuracy... 0.8745    0.0071
recall... 0.3914    0.0262
precision... 0.7032    0.0486
f1... 0.5022    0.0271

Number of Trees:  20  Max_features:  0.5
Metric...  Mean    Std. Dev.
accuracy... 0.8729    0.0071
recall... 0.3906    0.0223
precision... 0.6917    0.0477
f1... 0.4987    0.0251

Number of Trees:  20  Max_features:  0.7
Metric...  Mean    Std. Dev.
accuracy... 0.8728    0.0067
recall... 0.3997    0.0229
precision... 0.6853    0.0447
f1... 0.5042    0.0234

Best based on F1-Score
Best Number of Estimators (trees) =  15
Best Maximum Features =  0.7
```

In [45]:
```python
## Validating on the test data
print("\nTraining Data\nRandom Selection of 70% of Original Data")
DecisionTree.display_binary_split_metrics(rfc_train, X_train, y_train, \
                                          X_validate, y_validate)
DecisionTree.display_importance(rfc, encoded_df1.columns)
```

```
Training Data
Random Selection of 70% of Original Data
```

| Model Metrics... | Training | Validation |
|---|---|---|
| Observations... | 21000 | 9000 |
| Features... | 26 | 26 |
| Maximum Tree Depth... | 10 | 10 |
| Minimum Leaf Size... | 1 | 1 |
| Minimum split Size... | 2 | 2 |
| Mean Absolute Error... | 0.2372 | 0.2759 |
| Avg Squared Error... | 0.0925 | 0.1265 |
| Accuracy... | 0.9001 | 0.8533 |
| Precision... | 0.6693 | 0.5333 |
| Recall (Sensitivity)... | 0.7686 | 0.5791 |
| F1-score... | 0.7155 | 0.5553 |
| MISC (Misclassification)... | 10.0% | 14.7% |
| class 0... | 7.4% | 9.5% |
| class 1... | 23.1% | 42.1% |

Training

| Confusion Matrix | Class 0 | Class 1 |
|---|---|---|
| Class 0... | 16266 | 1303 |
| Class 1... | 794 | 2637 |

Validation

| Confusion Matrix | Class 0 | Class 1 |
|---|---|---|
| Class 0... | 6856 | 721 |
| Class 1... | 599 | 824 |

| FEATURE... | IMPORTANCE |
|---|---|
| Jun_Status... | 0.2559 |
| Credit_Limit... | 0.0749 |
| May_Payment... | 0.0591 |
| Jun_PayPercent.. | 0.0591 |
| Apr_Payment... | 0.0551 |
| Jun_Payment... | 0.0535 |
| May_PayPercent.. | 0.0518 |
| Apr_PayPercent.. | 0.0502 |
| Jan_Payment... | 0.0498 |
| Mar_Payment... | 0.0489 |
| Feb_Payment... | 0.0472 |
| Mar_PayPercent.. | 0.0446 |
| Feb_PayPercent.. | 0.0394 |
| Jan_PayPercent.. | 0.0376 |
| Education3... | 0.0118 |
| Default... | 0.0098 |
| card_class1... | 0.0094 |
| Gender... | 0.0093 |
| Education1... | 0.0074 |
| AgeBin1... | 0.0074 |
| AgeBin2... | 0.0060 |
| Marital_Status2. | 0.0049 |
| AgeBin3... | 0.0030 |
| Marital_Status1. | 0.0019 |
| Education2... | 0.0014 |
| AgeBin4... | 0.0005 |

### 2.0.3 3. Decision Tree Classifier

```
In [48]: ## Cross-validation on Decision Tree with different max_depth
         DTC()
```

```
For max_depth= 5
Metric...  Mean    Std. Dev.
accuracy... 0.8734    0.0072
recall... 0.4312    0.0348
precision... 0.6713    0.0460
f1... 0.5238    0.0285
For max_depth= 6
Metric...  Mean    Std. Dev.
accuracy... 0.8734    0.0078
recall... 0.4357    0.0253
precision... 0.6684    0.0430
f1... 0.5268    0.0265
For max_depth= 7
Metric...  Mean    Std. Dev.
accuracy... 0.8720    0.0071
recall... 0.4353    0.0326
precision... 0.6599    0.0379
f1... 0.5238    0.0300
For max_depth= 8
Metric...  Mean    Std. Dev.
accuracy... 0.8718    0.0067
recall... 0.4407    0.0322
precision... 0.6552    0.0368
f1... 0.5266    0.0266
For max_depth= 10
Metric...  Mean    Std. Dev.
accuracy... 0.8643    0.0085
recall... 0.4499    0.0322
precision... 0.6132    0.0428
f1... 0.5170    0.0269
For max_depth= 12
Metric...  Mean    Std. Dev.
accuracy... 0.8555    0.0097
recall... 0.4434    0.0421
precision... 0.5715    0.0430
f1... 0.4965    0.0339
For max_depth= 15
Metric...  Mean    Std. Dev.
accuracy... 0.8440    0.0101
recall... 0.4333    0.0411
precision... 0.5220    0.0427
f1... 0.4731    0.0329
For max_depth= 20
Metric...  Mean    Std. Dev.
accuracy... 0.8329    0.0092
recall... 0.4207    0.0288
precision... 0.4894    0.0343
f1... 0.4532    0.0187
For max_depth= 25
Metric...  Mean    Std. Dev.
accuracy... 0.8327    0.0102
recall... 0.4219    0.0258
precision... 0.4827    0.0344
f1... 0.4457    0.0248
```

```
Max depth for the Decision Tree is 6
```

```
In [50]: ## Validating on the test data
         print("\nTable of the metrics for 70/30 split")
         DecisionTree.display_binary_split_metrics(dtc_train, X_train,y_train,X_validate,
         y_validate)
```

```
Table of the metrics for 70/30 split


Model Metrics...       Training     Validation
Observations...          21000           9000
Features...            26              26
Maximum Tree Depth...              6              6
Minimum Leaf Size...             5              5
Minimum split Size...              5              5
Mean Absolute Error...         0.3211         0.3320
Avg Squared Error...         0.1506         0.1619
Accuracy...         0.8242         0.8048
Precision...         0.4738         0.4223
Recall (Sensitivity)...         0.6843         0.6374
F1-score...         0.5599         0.5080
MISC (Misclassification)...      17.6%          19.5%
     class 0...      14.8%          16.4%
     class 1...      31.6%          36.3%


Training
Confusion Matrix  Class 0   Class 1
Class 0...     14961      2608
Class 1...      1083      2348


Validation
Confusion Matrix  Class 0   Class 1
Class 0...      6336      1241
Class 1...       516       907
```

```
In [ ]: dtc_graph()
```

Decision tree graph is attached at the end

### 2.0.4 Neural Networks

```
In [52]: ## Cross validation on Neural Networks with different number of hidden layers and
         perceptrons.

         NN()
```

```
Network:  3
Metric...  Mean    Std. Dev.
Metric...  Mean    Std. Dev.
accuracy... 0.8593    0.0113
recall... 0.2974    0.1481
precision... 0.6547    0.0648
```

```
f1... 0.3836    0.1408

Network:  11
Metric...  Mean    Std. Dev.
Metric...  Mean    Std. Dev.
accuracy... 0.8690    0.0088
recall... 0.3999    0.1027
precision... 0.6602    0.0308
f1... 0.4870    0.1022

Network:  (5, 4)
Metric...  Mean    Std. Dev.
Metric...  Mean    Std. Dev.
accuracy... 0.8690    0.0110
recall... 0.4131    0.0939
precision... 0.6474    0.0464
f1... 0.4981    0.0902

Network:  (6, 5)
Metric...  Mean    Std. Dev.
Metric...  Mean    Std. Dev.
accuracy... 0.8701    0.0097
recall... 0.4194    0.0929
precision... 0.6584    0.0436
f1... 0.5040    0.0852

Network:  (7, 6)
Metric...  Mean    Std. Dev.
Metric...  Mean    Std. Dev.
accuracy... 0.8670    0.0123
recall... 0.4193    0.1310
precision... 0.6420    0.0508
f1... 0.4884    0.1389
Best neural network configuration is : (6, 5)
```

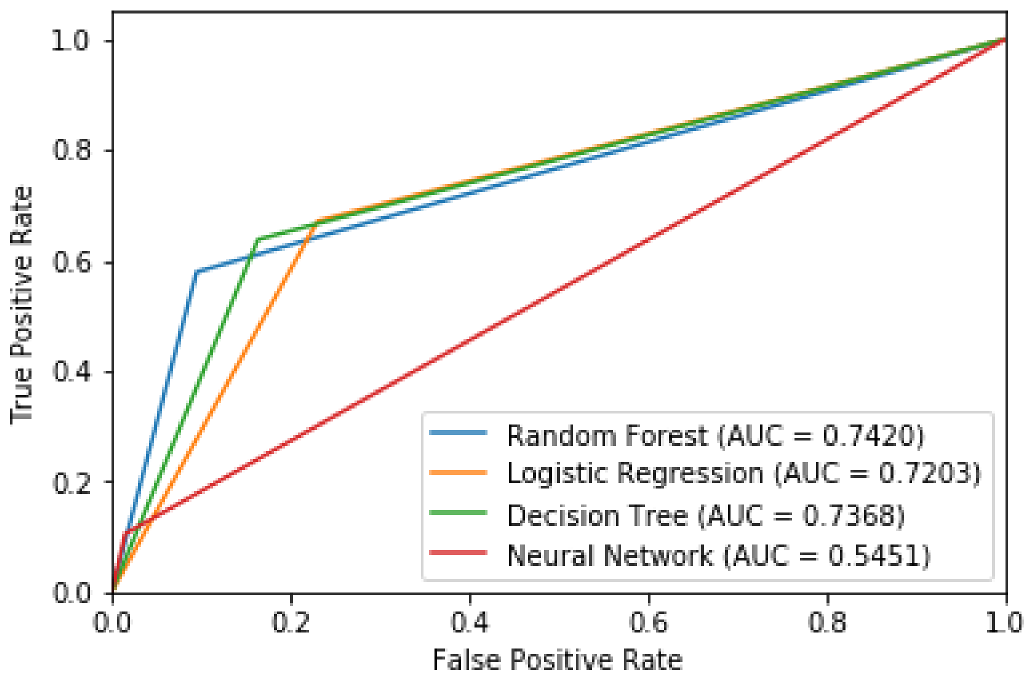
In [55]:
```
## Validating on the test data.
print("\nTable of the metrics for 70/30 split")
NeuralNetwork.display_binary_split_metrics(bestfnn, X_train, y_train,\
                                           X_validate, y_validate)
```

```
Table of the metrics for 70/30 split


Model Metrics...       Training     Validation
Observations...          21000           9000
Features...             26             26
Number of Layers...             2              2
Number of Outputs...             1              1
Number of Weights...            203            203
Activation Function...       logistic       logistic
Mean Absolute Error...         0.2487         0.2496
Avg Squared Error...         0.1243         0.1257
Accuracy...         0.8488         0.8458
Precision...         0.6861         0.5660
Recall (Sensitivity)...         0.1370         0.1054
F1-score...         0.2284         0.1777
MISC (Misclassification)...      15.1%          15.4%
     class 0...       1.2%           1.5%
```

```
    class 1...      86.3%          89.5%


Training
Confusion Matrix  Class 0   Class 1
Class 0...     17354       215
Class 1...      2961       470


Validation
Confusion Matrix  Class 0   Class 1
Class 0...      7462       115
Class 1...      1273       150
```

#### 2.0.5 ROC Curves

```
In [67]: roc_curves()
```




From the ROC curve above, we can see that Random Forest has the highest Area under the curve followed by Decision tree an logistic regression.

#### 2.0.6 RESULTS:

We predicted on 4 different models i.e. Logistic Regression, Random Forest, Decision trees and Neural Networks.

1. Even though area under the curve for Random Forest and Decision trees are higher than Logistic Regression, Logistic Regression has higher sensitivity than both of the models at 67% as compared to 58% and 63%.

2. It would seem logical for banks to correctly identify the number of defaulters as opposed to just reducing the misclassification rate. Even a slight decrease in accuracy to improve the sensitivity is warranted.
3. In our predictions, Neural Networks performed poorly in correctly identifying the number of true defaulters.
4. Decision trees and Logistic regression are both close in accuracy and sensitivity, it would depend on the cost associated with correctly identifying the true defaulters, so there is no best model in our case.

```python
# -*- coding: utf-8 -*-


## Loading libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import scipy
%matplotlib inline
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_validate
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score
from Class_replace_impute_encode import ReplaceImputeEncode
from Class_regression import logreg
from Class_tree import DecisionTree
from Class_FNN import NeuralNetwork
from sklearn.metrics import roc_curve, auc
from Class_FNN import NeuralNetwork
from sklearn import tree
from sklearn.tree import export_graphviz
from pydotplus import graph_from_dot_data
import graphviz

## Reading data
data= pd.read_excel("CreditCard_Defaults.xlsx")


## defining a function to create barplots
def barplot(column):
    data[column].value_counts().plot(kind='barh')
    plt.xlabel('Frequency')
    plt.ylabel('Value')
    plt.show()

barplot('Education')
barplot('Marital_Status')


## Modifying Education variable
low = (data['Education'] == 5) | (data['Education'] == 6) | (data['Education'] == 0)
data.loc[low, 'Education'] = 4
data['Education'].value_counts()

## Modifying Marital_Status variable
data.loc[data['Marital_Status']== 0, 'Marital_Status'] = 3
data['Marital_Status'].value_counts()


## Creating Bins in Age variable
data['AgeBin'] = 0 #creates a column of 0
data.loc[((data['Age'] > 20) & (data['Age'] < 30)) , 'AgeBin'] = 1
```

```python
data.loc[((data['Age'] >= 30) & (data['Age'] < 40)) , 'AgeBin'] = 2
data.loc[((data['Age'] >= 40) & (data['Age'] < 50)) , 'AgeBin'] = 3
data.loc[((data['Age'] >= 50) & (data['Age'] < 60)) , 'AgeBin'] = 4
data.loc[((data['Age'] >= 60) & (data['Age'] < 70)) , 'AgeBin'] = 5
data.loc[((data['Age'] >= 70) & (data['Age'] < 81)) , 'AgeBin'] = 6


## Dropping Age variable
data.drop('Age', axis=1, inplace=True)

sns.distplot(data['Credit_Limit'], bins=15);
sns.distplot(np.log(data['Credit_Limit']+1), bins=15);  ## After applying Log transformation


## Creating a function to create correlation heatmaps
def CorrelationPlot (df):
    colormap = plt.cm.RdBu
    plt.figure(figsize=(20,20))
    plt.title('Pearson Correlation of Features', y=1.05, size=15)
    sns.heatmap(df.corr(),linewidths=0.1,vmax=1.0,
                square=True, cmap = "coolwarm", fmt = ".2f",linecolor='white', annot=True)

attribute_map = {
    'Default':[1,(0,1),[0,0]],
    'Gender':[1,(1,2),[0,0]],
    'Education':[2,(1,2,3,4),[0,0]],
    'Marital_Status':[2,(1,2,3),[0,0]],
    'card_class':[2,(1,2,3),[0,0]],
    'AgeBin':[2,(1,2,3,4,5,6),[0,0]],
    'Credit_Limit':[0,(100,80000),[0,0]],
    'Jun_Status':[0,(-2,8),[0,0]],
    'May_Status':[0,(-2,8),[0,0]],
    'Apr_Status':[0,(-2,8),[0,0]],
    'Mar_Status':[0,(-2,8),[0,0]],
    'Feb_Status':[0,(-2,8),[0,0]],
    'Jan_Status':[0,(-2,8),[0,0]],
    'Jun_Bill':[0,(-12000,32000),[0,0]],
    'May_Bill':[0,(-12000,32000),[0,0]],
    'Apr_Bill':[0,(-12000,32000),[0,0]],
    'Mar_Bill':[0,(-12000,32000),[0,0]],
    'Feb_Bill':[0,(-12000,32000),[0,0]],
    'Jan_Bill':[0,(-12000,32000),[0,0]],
    'Jun_Payment':[0,(0,60000),[0,0]],
    'May_Payment':[0,(0,60000),[0,0]],
    'Apr_Payment':[0,(0,60000),[0,0]],
    'Mar_Payment':[0,(0,60000),[0,0]],
    'Feb_Payment':[0,(0,60000),[0,0]],
    'Jan_Payment':[0,(0,60000),[0,0]],
    'Jun_PayPercent':[0,(0,1),[0,0]],
    'May_PayPercent':[0,(0,1),[0,0]],
    'Apr_PayPercent':[0,(0,1),[0,0]],
    'Mar_PayPercent':[0,(0,1),[0,0]],
    'Feb_PayPercent':[0,(0,1),[0,0]],
    'Jan_PayPercent':[0,(0,1),[0,0]]}


rie = ReplaceImputeEncode(data_map=attribute_map, display=True,
```

```
                                nominal_encoding='one-hot', drop=False)
encoded_df = rie.fit_transform(data)

encoded_df['Credit_Limit']=np.log(encoded_df['Credit_Limit']+1)

CorrelationPlot(encoded_df)




encoded_df1= encoded_df.drop(['Jun_Bill', 'May_Bill', 'Apr_Bill', 'Mar_Bill',
                              'Feb_Bill', 'Jan_Bill','May_Status','Apr_Status',
                              'Mar_Status','Feb_Status','Jan_Status',
                              'card_class0', 'card_class2', 'Marital_Status0',
                              'Education0', 'AgeBin0'], axis=1).copy()


CorrelationPlot(encoded_df1)
plt.title('Pearson Correaltion of Features from final DataFrame')
plt.show()


y = np.asarray(encoded_df1['Default'])
# Drop the target, 'object'.  Axis=1 indicates the drop is for a column.
X = np.asarray(encoded_df1.drop('Default', axis=1))

##Splitting data
X_train, X_validate, y_train, y_validate = train_test_split(X,y,test_size = 0.3, random_state=7)
score_list = ['accuracy', 'recall', 'precision', 'f1']

## Cross-validation on Logistic Regression
class_weight=['None', 'balanced']
max_f1 = 0
for i in class_weight:
    print("Class Weight: ", i)
    lgr = LogisticRegression(class_weight=i, solver='newton-cg', max_iter=10000)

    logreg_scores = cross_validate(lgr, X, np.ravel(y), scoring=score_list, \
                                    return_train_score=False, cv=10)
    for s in score_list:
        var = "test_"+s
        mean = logreg_scores[var].mean()
        std  = logreg_scores[var].std()
        print("{:.<13s}{:>7.4f}{:>10.4f}".format(s, mean, std))
    if mean > max_f1:
        max_f1 = mean
        best_class_weight    = i
print("Best class weight: " ,best_class_weight)

lgr_train = LogisticRegression(class_weight='balanced',solver='newton-cg',
                               max_iter=10000).fit(X_train,y_train)

## Validating on the test data
print("\nTraining Data\nRandom Selection of 70% of Original Data")
logreg.display_binary_split_metrics(lgr_train, X_train, y_train, \
                                    X_validate, y_validate)
```

```
varlist = ['Default']
X = encoded_df1.drop(varlist, axis=1).copy()
y = encoded_df1[varlist]
np_y = np.ravel(y)

## Cross-validation on Random Forest
estimators_list   = [10, 15, 20]
max_features_list = ['auto', 0.3, 0.5, 0.7]
score_list = ['accuracy', 'recall', 'precision', 'f1']
max_f1 = 0
for e in estimators_list:
    for f in max_features_list:
        print("\nNumber of Trees: ", e, " Max_features: ", f)
        rfc = RandomForestClassifier(n_estimators=e, criterion="gini", \
                    max_depth=None, min_samples_split=2, \
                            min_samples_leaf=1, max_features=f, \
                            n_jobs=1, bootstrap=True, random_state=12345, class_weight='balanced')
        rfc= rfc.fit(X, np_y)
        scores = cross_validate(rfc, X, np_y, scoring=score_list, \
                                        return_train_score=False, cv=10)

        print("{:.<13s}{:>6s}{:>13s}".format("Metric", "Mean", "Std. Dev."))
        for s in score_list:
            var = "test_"+s
            mean = scores[var].mean()
            std  = scores[var].std()
            print("{:.<13s}{:>7.4f}{:>10.4f}".format(s, mean, std))
        if mean > max_f1:
            max_f1 = mean
            best_estimator    = e
            best_max_features = f

print("\nBest based on F1-Score")
print("Best Number of Estimators (trees) = ", best_estimator)
print("Best Maximum Features = ", best_max_features)

rfc_train = RandomForestClassifier(n_estimators=best_estimator, criterion="gini", \
                    max_depth=10, min_samples_split=2, \
                    min_samples_leaf=1, max_features= best_max_features,\
                    n_jobs=1, bootstrap=True, random_state=12345,
                    class_weight='balanced').fit(X_train, y_train)

## Validating on the test data
print("\nTraining Data\nRandom Selection of 70% of Original Data")
DecisionTree.display_binary_split_metrics(rfc_train, X_train, y_train, \
                                          X_validate, y_validate)
DecisionTree.display_importance(rfc, encoded_df1.columns)


## Cross validation on Decision Trees
max_depth=[5,6,7,8,10,12,15,20,25]
for i in max_depth:
    dtc = DecisionTreeClassifier(criterion='gini', max_depth=i, \
    min_samples_split=5, min_samples_leaf=5)
    dtc = dtc.fit(X,y)
    score_list = ['accuracy', 'recall', 'precision', 'f1']
    mean_score = []
```

```python
    std_score = []
    print("For max_depth=",i)
    print("{:.<13s}{:>6s}{:>13s}".format("Metric", "Mean", "Std. Dev."))
    for s in score_list:
        dtc_10 = cross_val_score(dtc, X, y, scoring=s, cv=10)
        mean = dtc_10.mean()
        std = dtc_10.std()
        mean_score.append(mean)
        std_score.append(std)
        print("{:.<13s}{:>7.4f}{:>10.4f}".format(s, mean, std))
print("Max depth for the Decision Tree is 6")

dtc_train = DecisionTreeClassifier(criterion='gini', max_depth=6, \
                                   min_samples_split=5, min_samples_leaf=5,
                                   class_weight='balanced').fit(X_train, y_train)

print("\nTable of the metrics for 70/30 split")
DecisionTree.display_binary_split_metrics(dtc_train, X_train,y_train,X_validate, y_validate)


def dtc_graph():
    dot_data = tree.export_graphviz(dtc_train, out_file=None,
    feature_names=list(X.columns),
    class_names=['0','1'],
    filled=True, rounded=True,
    special_characters=True)
    graph = graphviz.Source(dot_data)
    return(graph)

dtc_graph()



## Cross-validation on Neural Networks
network_list = [(3), (11), (5,4), (6,5), (7,6)]
# Scoring for Interval Prediction Neural Networks
max_f1_fnn=0.0
for nn in network_list:
    print("\nNetwork: ", nn)
    fnn = MLPClassifier(hidden_layer_sizes=nn, activation='logistic', \
                        solver='lbfgs', max_iter=1000, random_state=12345)
    fnn = fnn.fit(X,np_y)
    score_list = ['accuracy', 'recall', 'precision', 'f1']
    mean_score = []
    std_score  = []
    print("{:.<13s}{:>6s}{:>13s}".format("Metric", "Mean", "Std. Dev."))
    scores = cross_validate(fnn, X, np_y, scoring=score_list, \
                                    return_train_score=False, cv=10)

    print("{:.<13s}{:>6s}{:>13s}".format("Metric", "Mean", "Std. Dev."))
    for s in score_list:
        var = "test_"+s
        mean = scores[var].mean()
        std  = scores[var].std()
        print("{:.<13s}{:>7.4f}{:>10.4f}".format(s, mean, std))
    if mean > max_f1_fnn:
        max_f1_fnn = mean
```

```
        best_nn=nn
print('Best neural network configuration is :',best_nn)


## Validating on test data
bestfnn = MLPClassifier(hidden_layer_sizes=best_nn, activation='logistic', \
                    solver='lbfgs', max_iter=1000, random_state=12345).fit(X_train, y_train)

NeuralNetwork.display_metrics(bestfnn, X_train, y_train,\
                             X_validate, y_validate)



def roc_curves():
    fpr, tpr, _ = roc_curve(y_validate, rfc_train.predict(X_validate))
    AUC  = auc(fpr, tpr)
    plt.plot(fpr, tpr, label='Random Forest (AUC = %0.4f)' % AUC)
    fpr, tpr, _ = roc_curve(y_validate, lgr_train.predict(X_validate))
    AUC  = auc(fpr, tpr)
    plt.plot(fpr, tpr, label='Logistic Regression (AUC = %0.4f)' % AUC)
    fpr, tpr, _ = roc_curve(y_validate, dtc_train.predict(X_validate))
    AUC  = auc(fpr, tpr)
    plt.plot(fpr, tpr, label='Decision Tree (AUC = %0.4f)' % AUC)
    fpr, tpr, _ = roc_curve(y_validate, bestfnn.predict(X_validate))
    AUC  = auc(fpr, tpr)
    plt.plot(fpr, tpr, label='Neural Network (AUC = %0.4f)' % AUC)
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.legend()

roc_curves()
```