Aim Shams University

Faculty of Engineering ICHEP

Computer Engineering and Software Systems

CSE:382 Data Mining &

Business Intelligence

Course Project

Prepared by:

Sohayla Ihab Hamed [19P7343]

Youssef Mahmoud Massoud [18P8814]

Salma Ihab Abdelmawgoud [19P8794]

Yasmin Haitham Abdelmoaty [18P3102]

Youssef Hany Onsy [18P1789]

CSE:382 Data Mining & Business Intelligence

Major Task

Loan prediction is a common data mining problem which most retail banks solve on a daily basis. It’s the process of predicting who deserves to receive a given loan and at what rate based on certain characteristics of the borrower, be it an individual or a company. Those characteristics are mined for their use in a risk assessment process to determine the amount of risks the lender (the bank) will be incurring when loaning the particular individual/company. For example, some banks can model their interest rate for lending based on how much risk their model assumes a certain individual pose, and thereby require higher interest rates for those who pose higher risks of default and vice versa.

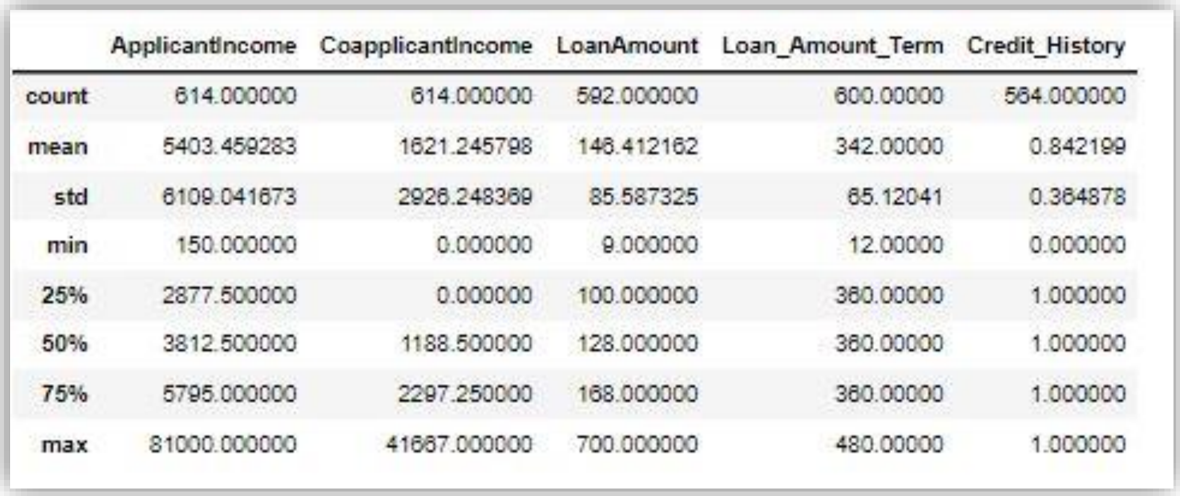
In our project, we use data mining techniques to analyze and predict whether a certain individual can be allowed to take a loan from our bank or whether said individual shall be denied the loan. This will be based on a set of features like marital status, education, employments and other features within the restrictions of the dataset we were provided. We will apply various data mining techniques to achieve our required goal where we end up with a classification model that we will train and later test.

DATA CLEANING

We begin with a set of preprocessing methods used to preprocess our given dataset, the first of which is performing a data cleaning step.

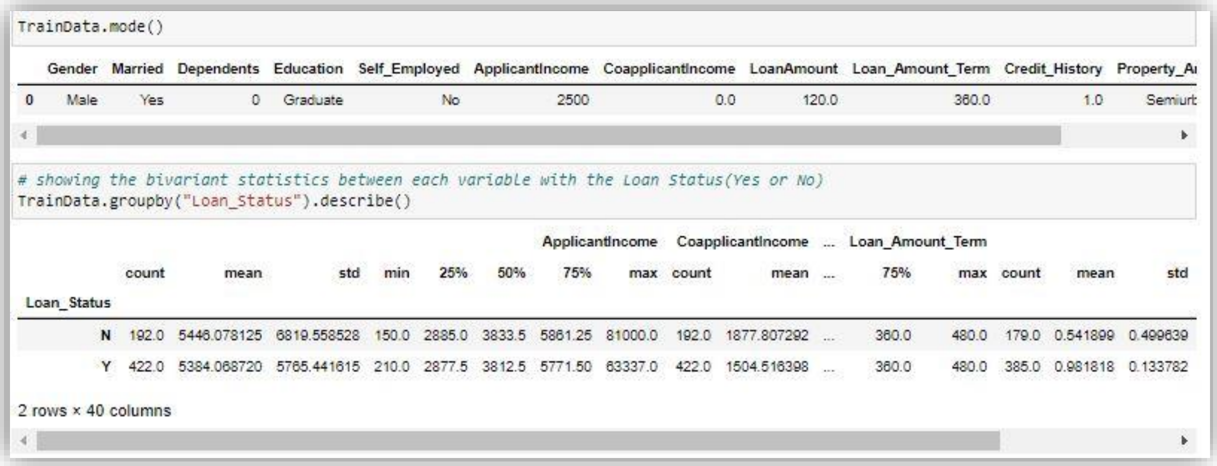
Handle Missing Data

We noticed that some of the training data points in the dataset lack information, and so we begin our data cleaning phase by handling the missing Data. Below we Show the univariant statistics for numerical variables.



	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History
count	614.000000	614.000000	592.000000	600.00000	564.000000
mean	5403.459283	1621.245798	146.412162	342.00000	0.842199
std	6109.041673	2926.248369	85.587325	65.12041	0.364878
min	150.000000	0.000000	9.000000	12.00000	0.000000
25%	2877.500000	0.000000	100.000000	360.00000	1.000000
50%	3812.500000	1188.500000	128.000000	360.00000	1.000000
75%	5795.000000	2297.250000	168.000000	360.00000	1.000000
max	81000.000000	41667.000000	700.000000	480.00000	1.000000

Next, we Show the Bivariant statistics between each variable with the Loan Status.



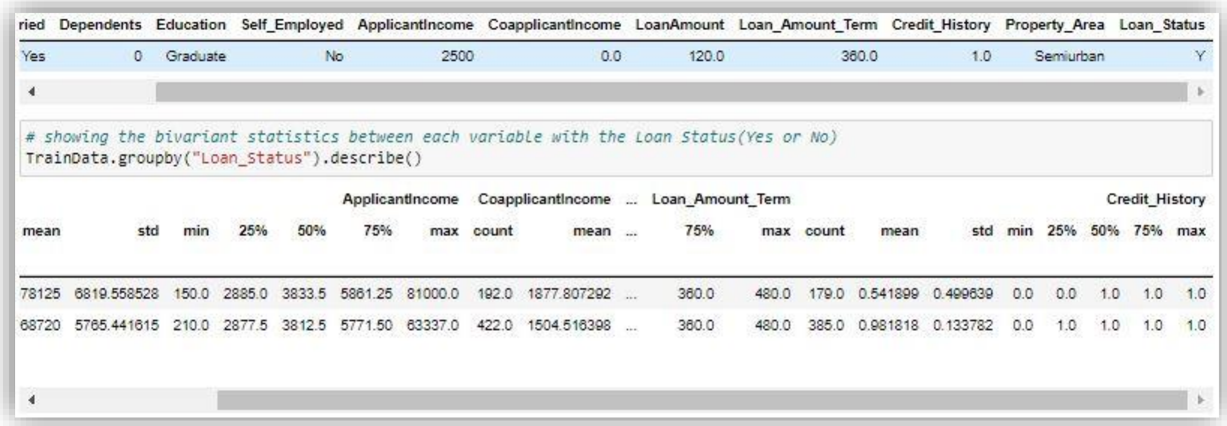
```
TrainData.mode()
```

	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History	Property_Ai
0	Male	Yes	0	Graduate	No	2500	0.0	120.0	360.0	1.0	Semiurt

```
# showing the bivariant statistics between each variable with the Loan Status(Yes or No)
TrainData.groupby("Loan_Status").describe()
```

								ApplicantIncome		CoapplicantIncome	...		Loan_Amount_Term			
	count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max	count	mean	std
Loan_Status																
N	192.0	5446.078125	6819.558528	150.0	2885.0	3833.5	5861.25	81000.0	192.0	1877.807292	...	360.0	480.0	179.0	0.541899	0.499639
Y	422.0	5384.068720	5765.441615	210.0	2877.5	3812.5	5771.50	63337.0	422.0	1504.516398	...	360.0	480.0	385.0	0.981818	0.133782

2 rows × 40 columns

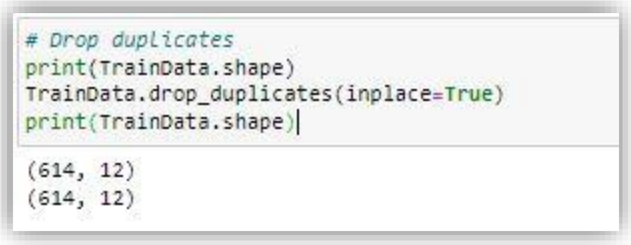


```
ried  Dependents  Education  Self_Employed  ApplicantIncome  CoapplicantIncome  LoanAmount  Loan_Amount_Term  Credit_History  Property_Area  Loan_Status
Yes            0   Graduate             No             2500                0.0       120.0             360.0             1.0      Semiurban            Y
```

```python
# showing the bivariant statistics between each variable with the Loan Status(Yes or No)
TrainData.groupby("Loan_Status").describe()
```

```
                                            ApplicantIncome  CoapplicantIncome  ...  Loan_Amount_Term                                       Credit_History
mean          std    min    25%     50%      75%       max   count        mean  ...        75%    max  count      mean       std  min  25%  50%  75%  max

78125  6819.558528  150.0  2885.0  3833.5  5861.25  81000.0  192.0  1877.807292  ...      360.0  480.0  179.0  0.541899  0.499639  0.0  0.0  1.0  1.0  1.0
68720  5765.441615  210.0  2877.5  3812.5  5771.50  63337.0  422.0  1504.516398  ...      360.0  480.0  385.0  0.981818  0.133782  0.0  1.0  1.0  1.0  1.0
```

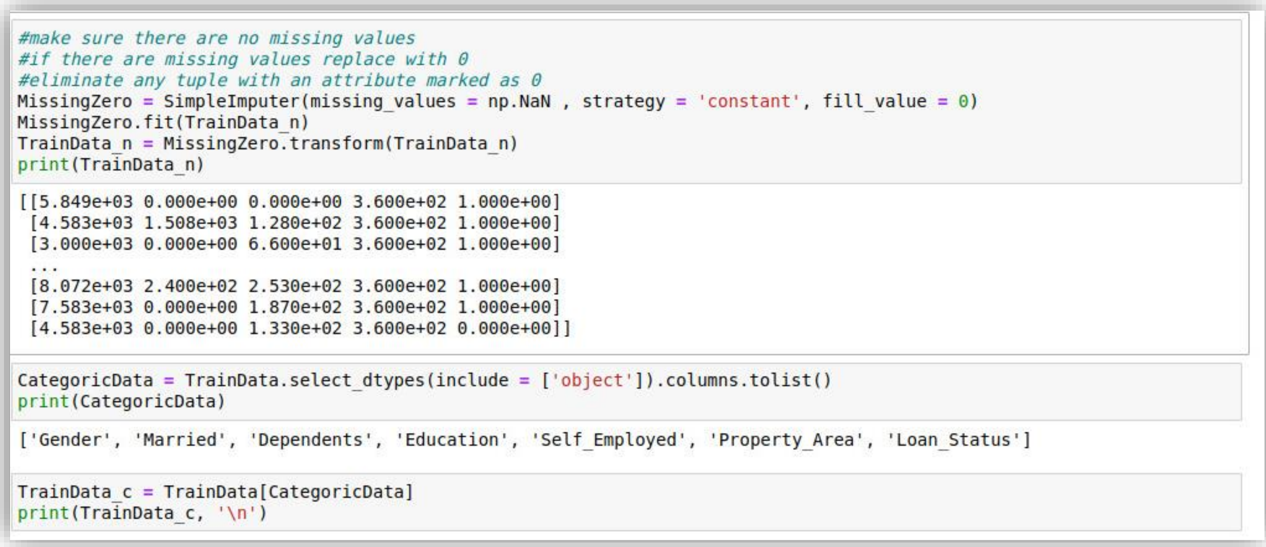
We identify missing data of both types, numerical data as well as categorical ones, but first we drop duplicate values as shown below.



```python
# Drop duplicates
print(TrainData.shape)
TrainData.drop_duplicates(inplace=True)
print(TrainData.shape)
```

```
(614, 12)
(614, 12)
```

Now, we try to figure out what kind of missing values are there. Missing values would be represented as “0”, “Not Applicable”, “NA”, “None”, “Null”, or “INF” all of which would identify that a particular value is missing.



```
#make sure there are no missing values
#if there are missing values replace with 0
#eliminate any tuple with an attribute marked as 0
MissingZero = SimpleImputer(missing_values = np.NaN , strategy = 'constant', fill_value = 0)
MissingZero.fit(TrainData_n)
TrainData_n = MissingZero.transform(TrainData_n)
print(TrainData_n)
```

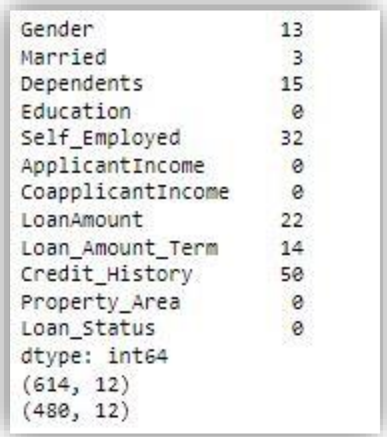
```
[[5.849e+03 0.000e+00 0.000e+00 3.600e+02 1.000e+00]
 [4.583e+03 1.508e+03 1.280e+02 3.600e+02 1.000e+00]
 [3.000e+03 0.000e+00 6.600e+01 3.600e+02 1.000e+00]
 ...
 [8.072e+03 2.400e+02 2.530e+02 3.600e+02 1.000e+00]
 [7.583e+03 0.000e+00 1.870e+02 3.600e+02 1.000e+00]
 [4.583e+03 0.000e+00 1.330e+02 3.600e+02 0.000e+00]]
```

```
CategoricData = TrainData.select_dtypes(include = ['object']).columns.tolist()
print(CategoricData)
```

```
['Gender', 'Married', 'Dependents', 'Education', 'Self_Employed', 'Property_Area', 'Loan_Status']
```

```
TrainData_c = TrainData[CategoricData]
print(TrainData_c, '\n')
```

One Approach would be removing tuples. We do this only if the dataset is large enough, and the tuple has multiple missing values.



```
Gender               13
Married               3
Dependents           15
Education             0
Self_Employed        32
ApplicantIncome       0
CoapplicantIncome     0
LoanAmount           22
Loan_Amount_Term     14
Credit_History       50
Property_Area         0
Loan_Status           0
dtype: int64
(614, 12)
(480, 12)
```

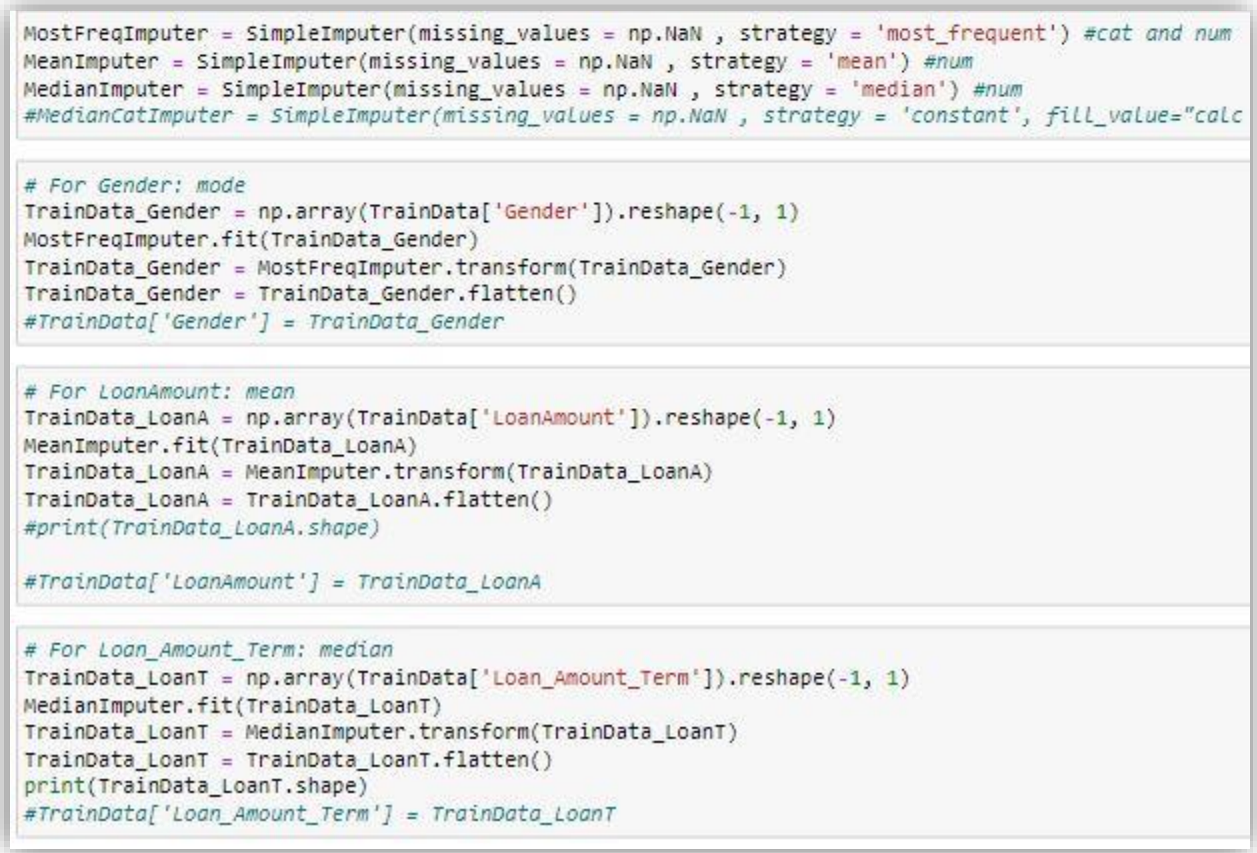
Another Approach would be using our conclusions from the data visualization we conducted earlier to decide for each feature how to fill those missing values.

We identified that for the Gender feature we can use the most_frequent (male) value since there is a huge variance between males and females.

On the other hand, for the Married feature: we can drop the 3 tuples which contain those missing values, as their effect over a total of 614 tuples won't be significant.

For the features; Dependents, Self Employed and Credit_History, we can use K nearest neighbor to estimate the proper values based on the closest value each datapoint is found to be, or put in other way, we fill the missing value by finding the k records closest to the record with missing values then chose the closest value to fill in the missing data.

For the LoanAmount feature which seems to be normally distributed, as the mean=342, median=360, mode=360, we can impute with the mean. Similarly, for the Loan_Amount_Term feature we cal also impute with the median based on a similar observation.



```
MostFreqImputer = SimpleImputer(missing_values = np.NaN , strategy = 'most_frequent') #cat and num
MeanImputer = SimpleImputer(missing_values = np.NaN , strategy = 'mean') #num
MedianImputer = SimpleImputer(missing_values = np.NaN , strategy = 'median') #num
#MedianCatImputer = SimpleImputer(missing_values = np.NaN , strategy = 'constant', fill_value="calc
```

```
# For Gender: mode
TrainData_Gender = np.array(TrainData['Gender']).reshape(-1, 1)
MostFreqImputer.fit(TrainData_Gender)
TrainData_Gender = MostFreqImputer.transform(TrainData_Gender)
TrainData_Gender = TrainData_Gender.flatten()
#TrainData['Gender'] = TrainData_Gender
```

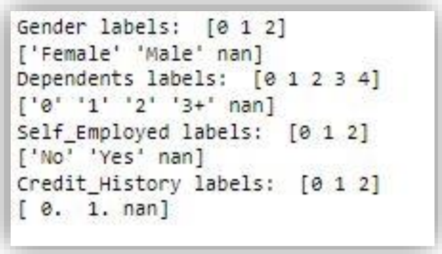
```
# For LoanAmount: mean
TrainData_LoanA = np.array(TrainData['LoanAmount']).reshape(-1, 1)
MeanImputer.fit(TrainData_LoanA)
TrainData_LoanA = MeanImputer.transform(TrainData_LoanA)
TrainData_LoanA = TrainData_LoanA.flatten()
#print(TrainData_LoanA.shape)

#TrainData['LoanAmount'] = TrainData_LoanA
```

```
# For Loan_Amount_Term: median
TrainData_LoanT = np.array(TrainData['Loan_Amount_Term']).reshape(-1, 1)
MedianImputer.fit(TrainData_LoanT)
TrainData_LoanT = MedianImputer.transform(TrainData_LoanT)
TrainData_LoanT = TrainData_LoanT.flatten()
print(TrainData_LoanT.shape)
#TrainData['Loan_Amount_Term'] = TrainData_LoanT
```

K-Nearest Neighbor

The k-nearest neighbors' algorithm, also known as KNN or k-NN, is a non-parametric, supervised learning classifier, which uses proximity to make classifications or predictions about the grouping of an individual data point. While it can be used for either regression or classification problems, it is typically used as a classification algorithm, working off the assumption that similar points can be found near one another. For classification problems, a class label is assigned on the basis of a majority vote i.e., the label that is most frequently represented around a given data point is used.

Below we show the results of implementing KNN to estimate the proper values by finding the k records closest to the record with missing values and choosing the closest value to fill in the missing data for Dependents, Self Employed and Credit_History Features.



```
Gender labels:  [0 1 2]
['Female' 'Male' nan]
Dependents labels:  [0 1 2 3 4]
['0' '1' '2' '3+' nan]
Self_Employed labels:  [0 1 2]
['No' 'Yes' nan]
Credit_History labels:  [0 1 2]
[ 0.  1. nan]
```

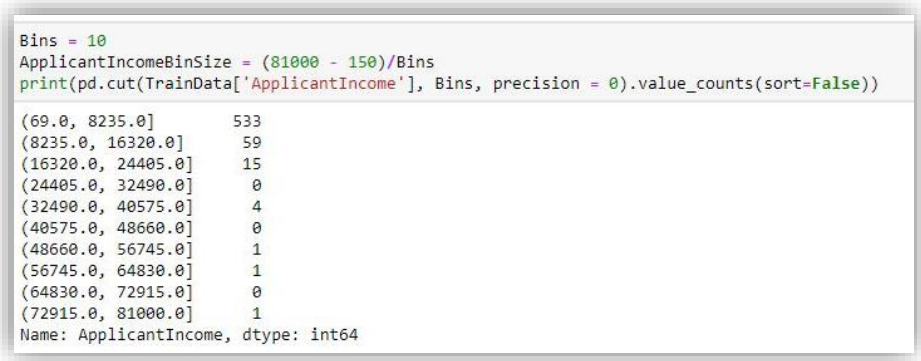
Handle Noisy Data

Binning By pd.cut

Since we are dealing with continuous numeric data, it's more efficient to bin the data into multiple bins for further analysis. There are several different terms for binning including bucketing, discrete binning, discretization or quantization. Pandas supports these approaches using the cut and qcut functions.

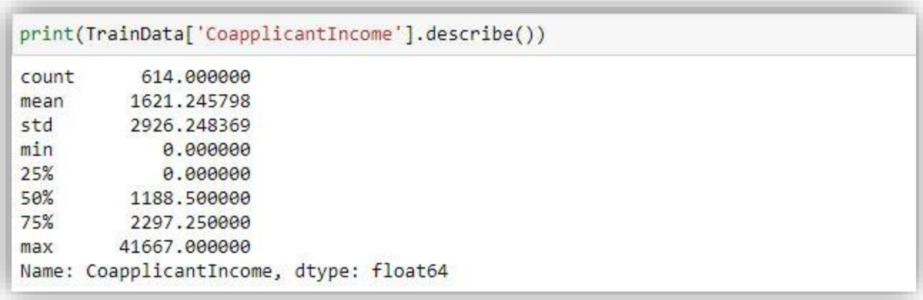
The pandas cut binning function is used to specifically define the bin edges. However, notice that there is no guarantee for the item's distribution in each bin. In fact, with a pandas cut we can define bins in such a way that no items are included in a bin or nearly all items are in a single bin.

as shown below we perform a binning through a pandas cut by the following features; Applicant's and Co-applicant's incomes as well as the Loan Amount, and we group the values into 10 bins as shown below.



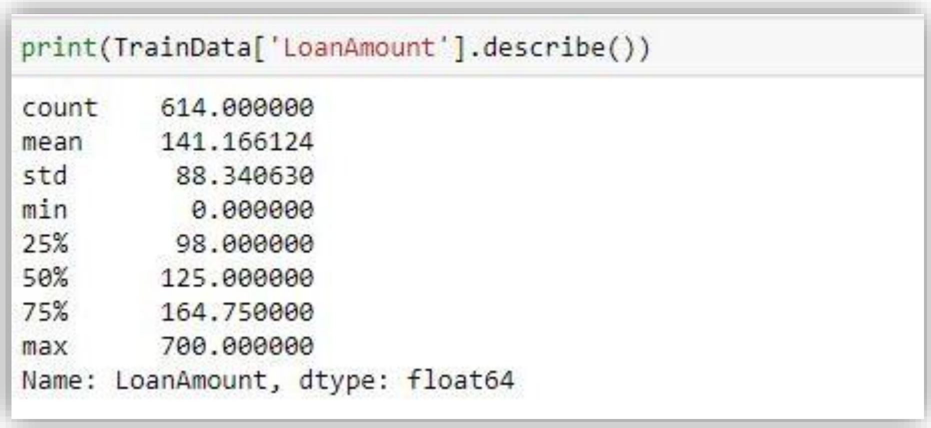
```
Bins = 10
ApplicantIncomeBinSize = (81000 - 150)/Bins
print(pd.cut(TrainData['ApplicantIncome'], Bins, precision = 0).value_counts(sort=False))
```

```
(69.0, 8235.0]        533
(8235.0, 16320.0]      59
(16320.0, 24405.0]     15
(24405.0, 32490.0]      0
(32490.0, 40575.0]      4
(40575.0, 48660.0]      0
(48660.0, 56745.0]      1
(56745.0, 64830.0]      1
(64830.0, 72915.0]      0
(72915.0, 81000.0]      1
Name: ApplicantIncome, dtype: int64
```



```
print(TrainData['CoapplicantIncome'].describe())
```

```
count      614.000000
mean      1621.245798
std       2926.248369
min          0.000000
25%          0.000000
50%       1188.500000
75%       2297.250000
max      41667.000000
Name: CoapplicantIncome, dtype: float64
```

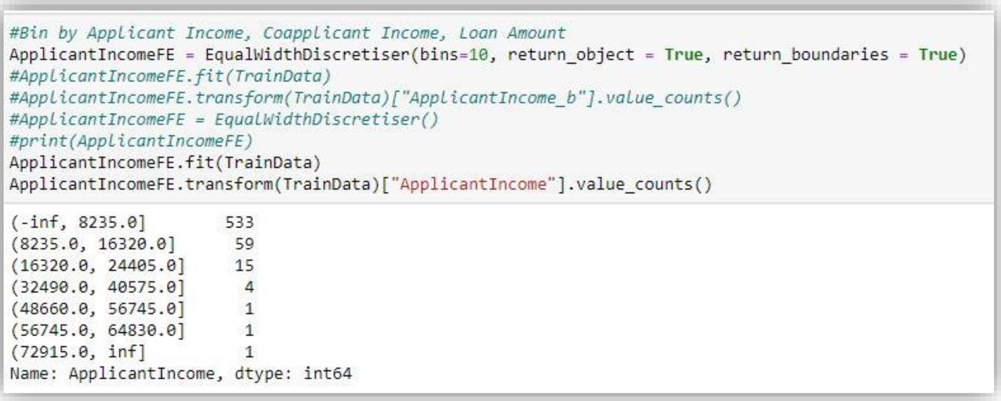


```
print(TrainData['LoanAmount'].describe())
```

```
count    614.000000
mean     141.166124
std       88.340630
min        0.000000
25%       98.000000
50%      125.000000
75%      164.750000
max      700.000000
Name: LoanAmount, dtype: float64
```

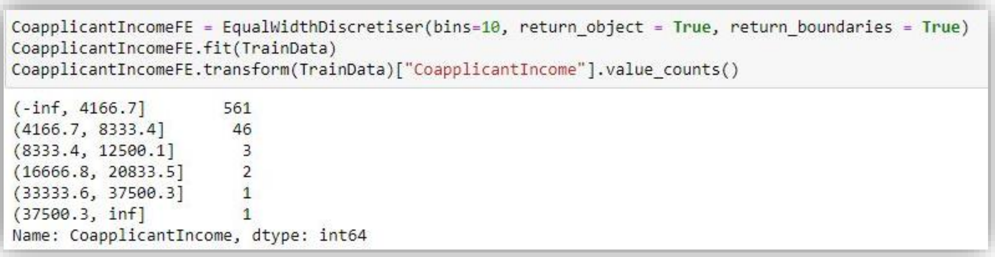
Binning By feature Engine

Next, we perform another Binning procedure however this time we perform it via Feature engineering, also known as feature binning. Like any binning procedure, the process of feature binning, is used for the transformation of a continuous or numerical variable into a categorical feature and we use it to identify missing values or outliers. Here we use an equal width binning algorithm where we divide the continuous variable into 10 categories having bins or range of the same width. As was the case previously, we perform the binning by all three features; Applicant's and Co-applicant's incomes and the Loan Amount as shown below.



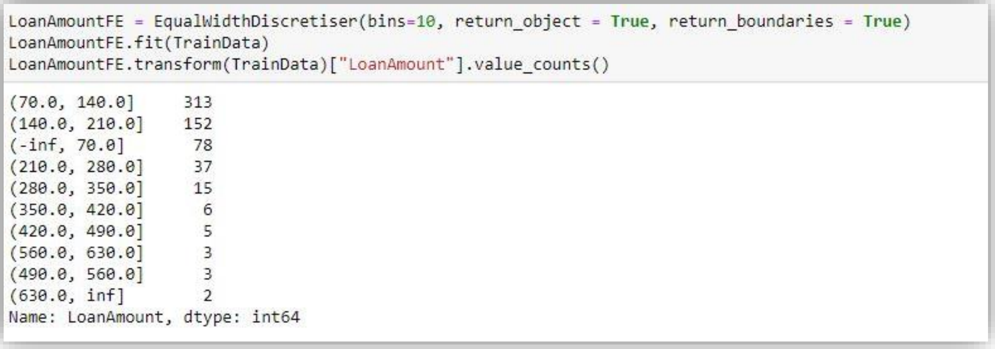
```
#Bin by Applicant Income, Coapplicant Income, Loan Amount
ApplicantIncomeFE = EqualWidthDiscretiser(bins=10, return_object = True, return_boundaries = True)
#ApplicantIncomeFE.fit(TrainData)
#ApplicantIncomeFE.transform(TrainData)["ApplicantIncome_b"].value_counts()
#ApplicantIncomeFE = EqualWidthDiscretiser()
#print(ApplicantIncomeFE)
ApplicantIncomeFE.fit(TrainData)
ApplicantIncomeFE.transform(TrainData)["ApplicantIncome"].value_counts()
```

```
(-inf, 8235.0]        533
(8235.0, 16320.0]      59
(16320.0, 24405.0]     15
(32490.0, 40575.0]      4
(48660.0, 56745.0]      1
(56745.0, 64830.0]      1
(72915.0, inf]          1
Name: ApplicantIncome, dtype: int64
```



```
CoapplicantIncomeFE = EqualWidthDiscretiser(bins=10, return_object = True, return_boundaries = True)
CoapplicantIncomeFE.fit(TrainData)
CoapplicantIncomeFE.transform(TrainData)["CoapplicantIncome"].value_counts()
```

```
(-inf, 4166.7]        561
(4166.7, 8333.4]       46
(8333.4, 12500.1]       3
(16666.8, 20833.5]      2
(33333.6, 37500.3]      1
(37500.3, inf]          1
Name: CoapplicantIncome, dtype: int64
```

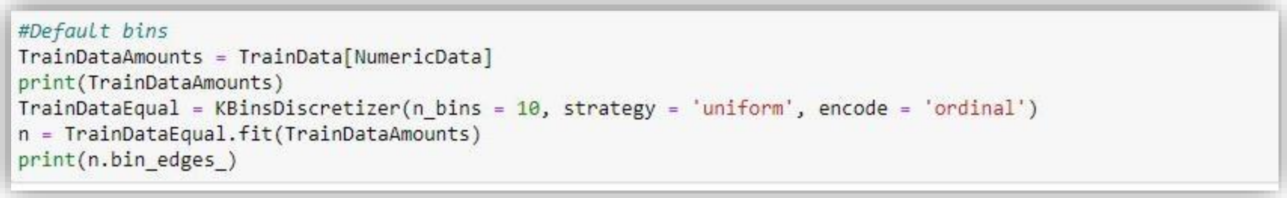


```
LoanAmountFE = EqualWidthDiscretiser(bins=10, return_object = True, return_boundaries = True)
LoanAmountFE.fit(TrainData)
LoanAmountFE.transform(TrainData)["LoanAmount"].value_counts()
```

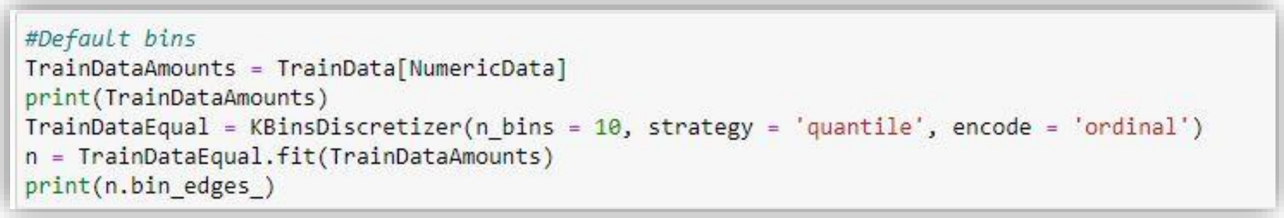
```
(70.0, 140.0]     313
(140.0, 210.0]    152
(-inf, 70.0]       78
(210.0, 280.0]     37
(280.0, 350.0]     15
(350.0, 420.0]      6
(420.0, 490.0]      5
(560.0, 630.0]      3
(490.0, 560.0]      3
(630.0, inf]        2
Name: LoanAmount, dtype: int64
```

Binning By K-Bins-Discretizer

We perform yet another binning procedure using sklearn's KbinsDiscritizer. KBinsDiscretizer implements different binning strategies, which can be selected with the strategy parameter. The 'uniform' strategy uses constant-width bins. The 'quantile' strategy uses the quantiles values to have equally populated bins in each feature. The k-means strategy defines bins based on a k-means clustering procedure performed on each feature independently. Below we show two strategies through which we applied this binning technique, uniform and quantile.



```
#Default bins
TrainDataAmounts = TrainData[NumericData]
print(TrainDataAmounts)
TrainDataEqual = KBinsDiscretizer(n_bins = 10, strategy = 'uniform', encode = 'ordinal')
n = TrainDataEqual.fit(TrainDataAmounts)
print(n.bin_edges_)
```

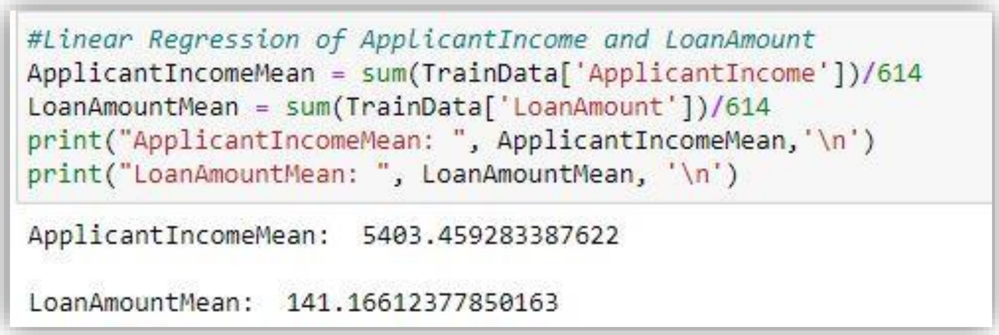


```
#Default bins
TrainDataAmounts = TrainData[NumericData]
print(TrainDataAmounts)
TrainDataEqual = KBinsDiscretizer(n_bins = 10, strategy = 'quantile', encode = 'ordinal')
n = TrainDataEqual.fit(TrainDataAmounts)
print(n.bin_edges_)
```

We discretize the feature and one-hot encode the transformed data. Note that if the bins are not reasonably wide, there would appear to be a substantially increased risk of overfitting, so the discretizer parameters should usually be tuned under cross validation. A linear regression model becomes much more flexible when we perform the previous binning process.

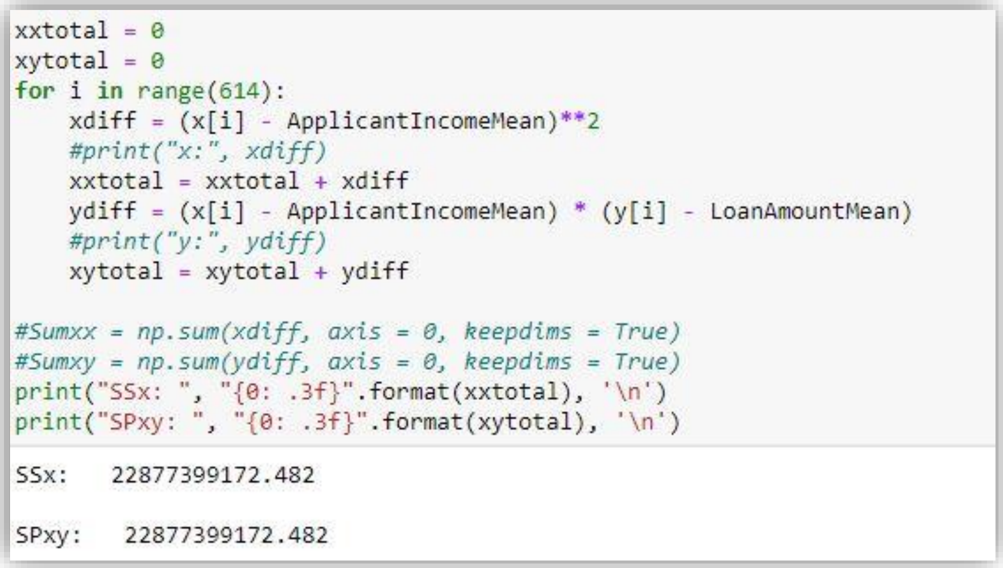
Single Linear regression

Next, we perform a linear regression between two features, the Applicant income and the Loan Amount. We first compute the means. Then we compute the sum of square (SSx) of X and the sum of product (SPxy) for X and Y, such that X and Y both refer to the Applicant's income. We then compute the slope and intercepts of the regression line, then show the results of the prediction all shown below.



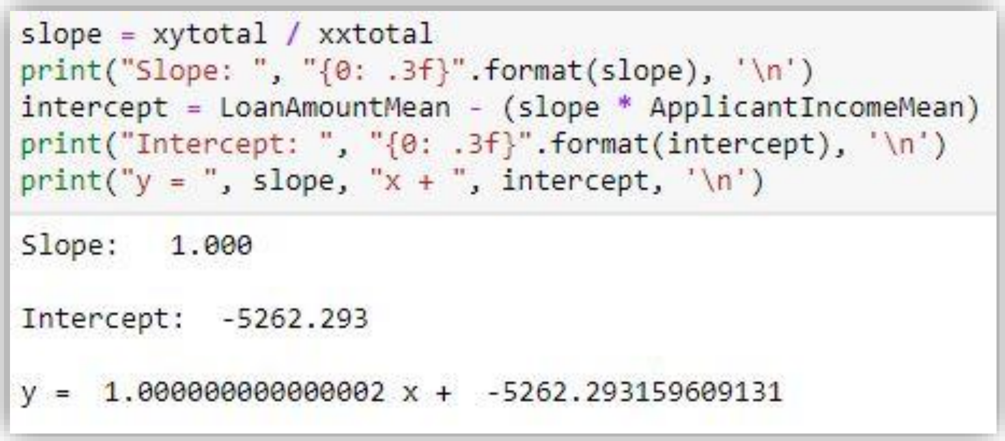
```
#Linear Regression of ApplicantIncome and LoanAmount
ApplicantIncomeMean = sum(TrainData['ApplicantIncome'])/614
LoanAmountMean = sum(TrainData['LoanAmount'])/614
print("ApplicantIncomeMean: ", ApplicantIncomeMean,'\n')
print("LoanAmountMean: ", LoanAmountMean, '\n')
```

```
ApplicantIncomeMean:  5403.459283387622

LoanAmountMean:  141.16612377850163
```



```
xxtotal = 0
xytotal = 0
for i in range(614):
    xdiff = (x[i] - ApplicantIncomeMean)**2
    #print("x:", xdiff)
    xxtotal = xxtotal + xdiff
    ydiff = (x[i] - ApplicantIncomeMean) * (y[i] - LoanAmountMean)
    #print("y:", ydiff)
    xytotal = xytotal + ydiff

#Sumxx = np.sum(xdiff, axis = 0, keepdims = True)
#Sumxy = np.sum(ydiff, axis = 0, keepdims = True)
print("SSx: ", "{0: .3f}".format(xxtotal), '\n')
print("SPxy: ", "{0: .3f}".format(xytotal), '\n')
```

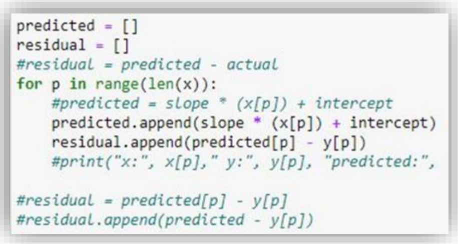
```
SSx:   22877399172.482

SPxy:   22877399172.482
```



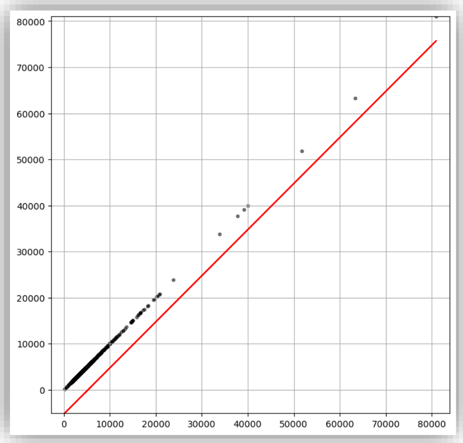
```
slope = xytotal / xxtotal
print("Slope: ", "{0: .3f}".format(slope), '\n')
intercept = LoanAmountMean - (slope * ApplicantIncomeMean)
print("Intercept: ", "{0: .3f}".format(intercept), '\n')
print("y = ", slope, "x + ", intercept, '\n')
```

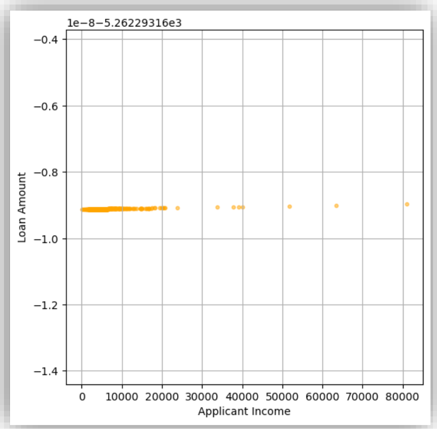
```
Slope:   1.000

Intercept:  -5262.293

y =  1.0000000000000002 x +  -5262.293159609131
```

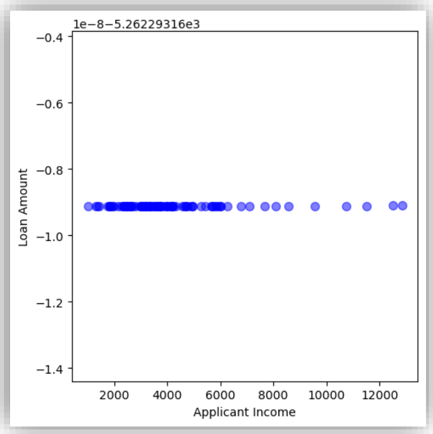
Next, we compute the standardized residual, and plot the relationship between the Applicant Income and the loan Amount, where we predict the loan amount based on the applicant's income. We then show the results by sampling points from the plot.



```
predicted = []
residual = []
#residual = predicted - actual
for p in range(len(x)):
    #predicted = slope * (x[p]) + intercept
    predicted.append(slope * (x[p]) + intercept)
    residual.append(predicted[p] - y[p])
    #print("x:", x[p]," y:", y[p], "predicted:",

#residual = predicted[p] - y[p]
#residual.append(predicted - y[p])
```



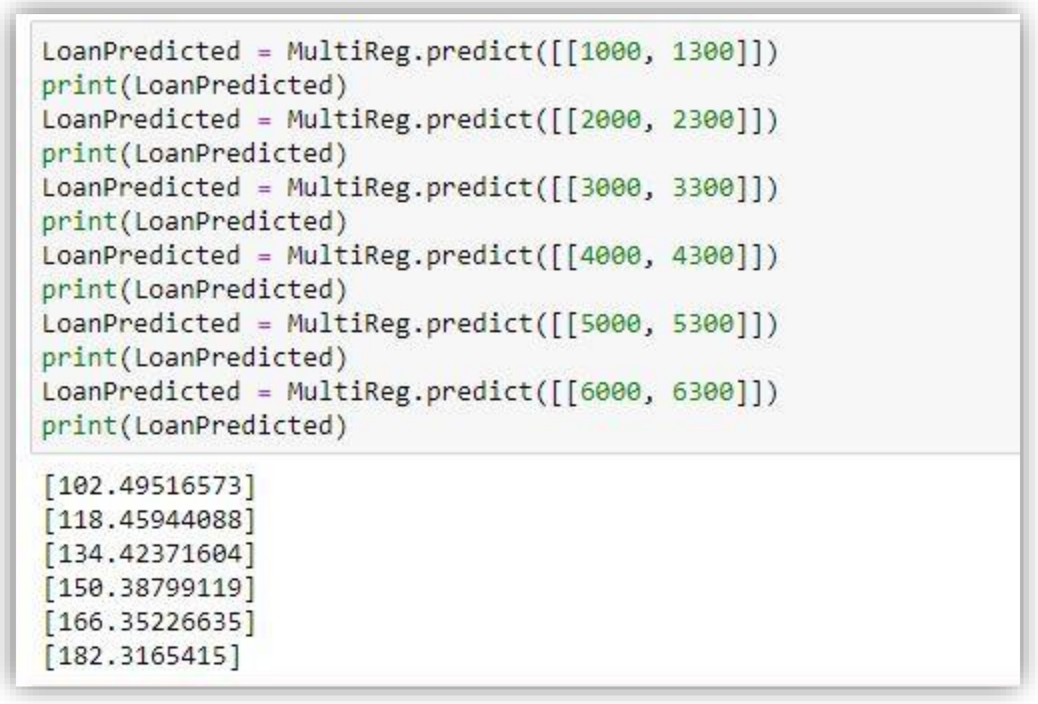




As shown above, it seems like the Applicant's income barely determines the loan amount, the line is almost parallel to the x-axis.

Multilinear regression: Dependents and property and Income Vs Loan Amount

Next, we perform a multilinear regression, where we use both the applicant's and Co-applicant's incomes to predict the Loan amount, as shown below.

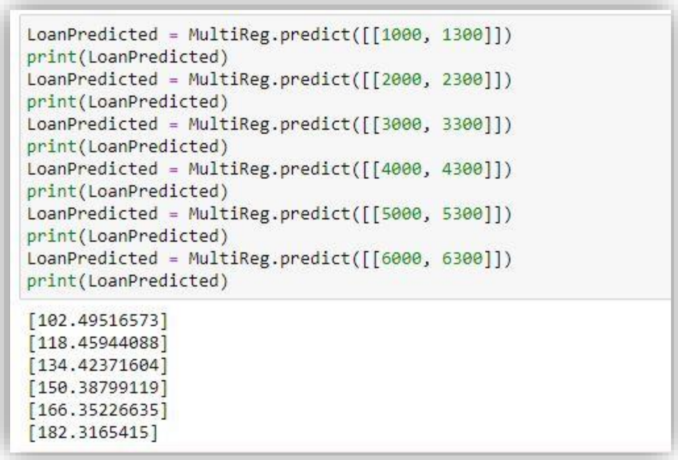


```
LoanPredicted = MultiReg.predict([[1000, 1300]])
print(LoanPredicted)
LoanPredicted = MultiReg.predict([[2000, 2300]])
print(LoanPredicted)
LoanPredicted = MultiReg.predict([[3000, 3300]])
print(LoanPredicted)
LoanPredicted = MultiReg.predict([[4000, 4300]])
print(LoanPredicted)
LoanPredicted = MultiReg.predict([[5000, 5300]])
print(LoanPredicted)
LoanPredicted = MultiReg.predict([[6000, 6300]])
print(LoanPredicted)
```

```
[102.49516573]
[118.45944088]
[134.42371604]
[150.38799119]
[166.35226635]
[182.3165415]
```

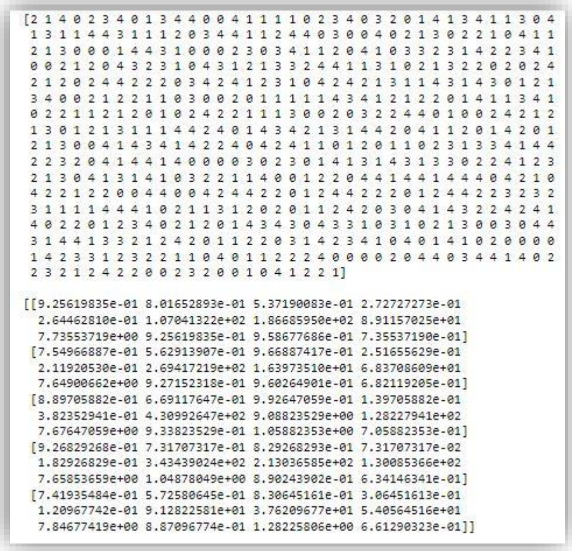
Clustering by K-Means

On this stage we scan for the set of clusters formed by the datapoints as another mean to detect outliers. We use a k-mean clustering algorithm to cluster the datapoints across 5 clusters based on Gender, marital status, Dependents, Education, self-employment, applicant income, co-applicant's income, loan amount term, credit history, property area, and loan status. We end up with the clusters shown below. Then we plot the data to visualize the cluster as shown below.

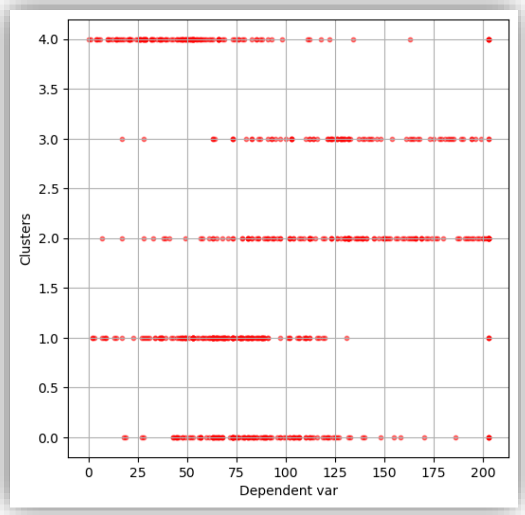


```
LoanPredicted = MultiReg.predict([[1000, 1300]])
print(LoanPredicted)
LoanPredicted = MultiReg.predict([[2000, 2300]])
print(LoanPredicted)
LoanPredicted = MultiReg.predict([[3000, 3300]])
print(LoanPredicted)
LoanPredicted = MultiReg.predict([[4000, 4300]])
print(LoanPredicted)
LoanPredicted = MultiReg.predict([[5000, 5300]])
print(LoanPredicted)
LoanPredicted = MultiReg.predict([[6000, 6300]])
print(LoanPredicted)
```

```
[102.49516573]
[118.45944088]
[134.42371604]
[150.38799119]
[166.35226635]
[182.3165415]
```



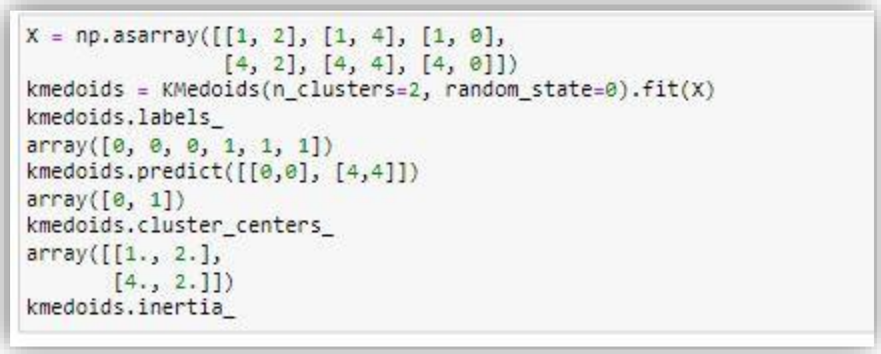
```
[2 1 4 0 2 3 4 0 1 3 4 4 0 0 4 1 1 1 1 0 2 3 4 0 3 2 0 1 4 1 3 4 1 1 3 0 4
 1 3 1 1 4 4 3 1 1 1 2 0 3 4 4 1 1 2 4 4 0 3 0 0 4 0 2 1 3 0 2 2 1 0 4 1 1
 2 1 3 0 0 0 1 4 4 3 1 0 0 0 2 3 0 3 4 1 1 2 0 4 1 0 3 3 2 3 1 4 2 2 3 4 1
 0 0 2 1 2 0 4 3 2 3 1 0 4 3 1 2 1 3 3 2 4 4 1 1 3 1 0 2 1 3 2 2 0 2 0 2 4
 2 1 2 0 2 4 4 2 2 2 0 3 4 2 4 1 2 3 1 0 4 2 4 2 1 3 1 1 4 3 1 4 3 0 1 2 1
 3 4 0 0 2 1 2 2 1 1 0 3 0 0 2 0 1 1 1 1 1 4 3 4 1 2 1 2 2 0 1 4 1 1 3 4 1
 0 2 2 1 1 2 1 2 0 1 0 2 4 2 2 1 1 1 3 0 0 2 0 3 2 2 4 4 0 1 0 0 2 4 2 1 2
 1 3 0 1 2 1 3 1 1 1 4 4 2 4 0 1 4 3 4 2 1 3 1 4 4 2 0 4 1 1 2 0 1 4 2 0 1
 2 1 3 0 0 4 1 4 3 4 1 4 2 2 4 0 4 2 4 1 1 0 1 2 0 1 1 0 2 3 1 3 3 4 1 4 4
 2 2 3 2 0 4 1 4 4 1 4 0 0 0 0 3 0 2 3 0 1 4 1 3 1 4 3 1 3 3 0 2 2 4 1 2 3
 2 1 3 0 4 1 3 1 4 1 0 3 2 2 1 1 4 0 0 1 2 2 0 4 4 1 4 4 1 4 4 4 0 4 2 1 0
 4 2 2 1 2 2 0 0 4 4 0 0 4 2 4 4 2 2 0 1 2 4 4 2 2 2 0 1 2 4 4 2 2 3 2 3 2
 3 1 1 1 1 4 4 4 1 0 2 1 1 3 1 2 0 2 0 1 1 2 4 2 0 3 0 4 1 4 3 2 2 4 2 4 1
 4 0 2 2 0 1 2 3 4 0 2 1 2 0 1 4 3 4 3 0 4 3 3 1 0 3 1 0 2 1 3 0 0 3 0 4 4
 3 1 4 4 1 3 3 2 1 2 4 2 0 1 1 2 2 0 3 1 4 2 3 4 1 0 4 0 1 4 1 0 2 0 0 0 0
 1 4 2 3 3 1 2 3 2 2 1 1 0 4 0 1 1 2 2 2 4 0 0 0 0 2 0 4 4 0 3 4 4 1 4 0 2
 2 3 2 1 2 4 2 2 0 0 2 3 2 0 0 1 0 4 1 2 2 1]

[[9.25619835e-01 8.01652893e-01 5.37190083e-01 2.72727273e-01
  2.64462810e-01 1.07041322e+02 1.86685950e+02 8.91157025e+01
  7.73553719e+00 9.25619835e-01 9.58677686e-01 7.35537190e-01]
 [7.54966887e-01 5.62913907e-01 9.66887417e-01 2.51655629e-01
  2.11920530e-01 2.69417219e+02 1.63973510e+01 6.83708609e+01
  7.64900662e+00 9.27152318e-01 9.60264901e-01 6.82119205e-01]
 [8.89705882e-01 6.69117647e-01 9.92647059e-01 1.39705882e-01
  3.82352941e-01 4.30992647e+02 9.08823529e+00 1.28227941e+02
  7.67647059e+00 9.33823529e-01 1.05882353e+00 7.05882353e-01]
 [9.26829268e-01 7.31707317e-01 8.29268293e-01 7.31707317e-02
  1.82926829e-01 3.43439024e+02 2.13036585e+02 1.30085366e+02
  7.65853659e+00 1.04878049e+00 8.90243902e-01 6.34146341e-01]
 [7.41935484e-01 5.72580645e-01 8.30645161e-01 3.06451613e-01
  1.20967742e-01 9.12822581e+01 3.76209677e+01 5.40564516e+01
  7.84677419e+00 8.87096774e-01 1.28225806e+00 6.61290323e-01]]
```



Clustering by K-medoids

Another clustering technique we can use is the K-medoid clustering algorithm, which is an improvised version of the K-Means algorithm mainly designed to deal with outlier data sensitivity. A Medoid is a point in the cluster from which the sum of distances to other data points is minimal. Instead of centroids as reference points in K-Means algorithms, the K-Medoids algorithm takes a Medoid as a reference point.

As we did previously, we also scan for the set of clusters formed by the datapoints to detect the outliers. This time however we use the k-medoids clustering algorithm to cluster the datapoints across 5 clusters from 0 to 4.

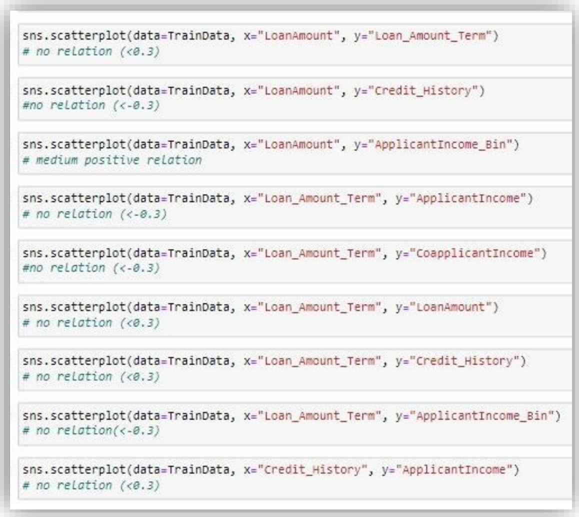


```
X = np.asarray([[1, 2], [1, 4], [1, 0],
                [4, 2], [4, 4], [4, 0]])
kmedoids = KMedoids(n_clusters=2, random_state=0).fit(X)
kmedoids.labels_
array([0, 0, 0, 1, 1, 1])
kmedoids.predict([[0,0], [4,4]])
array([0, 1])
kmedoids.cluster_centers_
array([[1., 2.],
       [4., 2.]])
kmedoids.inertia_
```

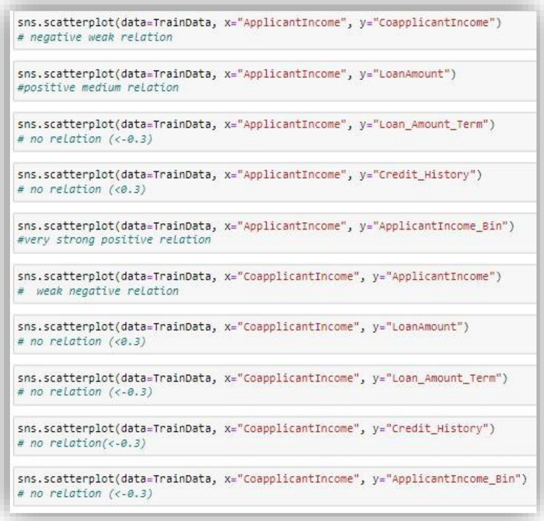
The correlations and their visualization

Next, we want to check if there are any corrections between each variable/feature pair to ensure that we eliminate any redundant feature as a means of reducing the dimensionality of the data. If a pair of features happen to have a strong correlation between them we eliminate one of the features as it happens to be redundant since only one of the two features would be sufficient to convey the information stored in the datapoints. The goal is to maximize efficiency and accuracy and this happens when we maximize the information we capture and use in training our model to maximize the accuracy while minimizing the redundant features and data to maximize efficiency.

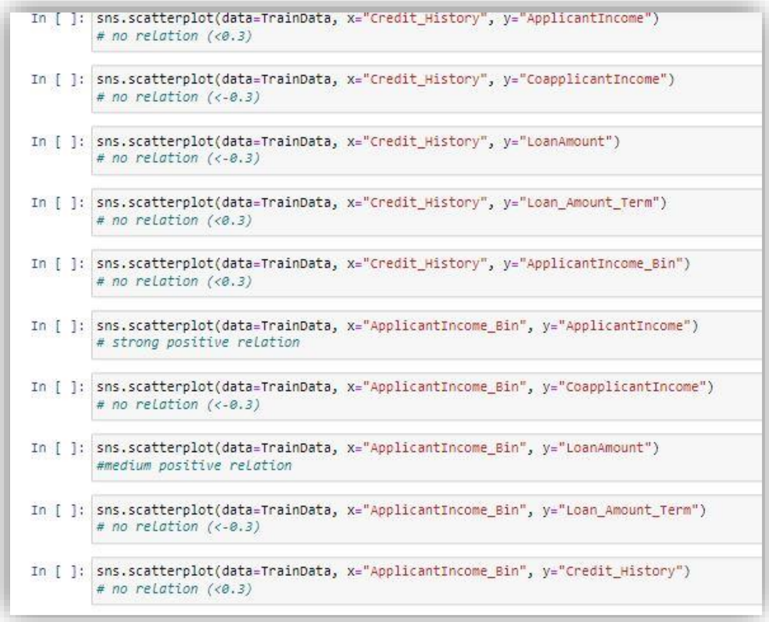
Below we show the correlation results for all variable pairs and we notice that we find medium positive correlations between the Applicant's income and Loan Amount, and a strong positive correlation between the Applicant's income and Applicant Income Bin, a medium positive correlation between the Loan Amount and Applicant Income Bin, a weak positive one between Loan Amount and Co-applicant Income and negative weak correlation between the Applicant's income and the Co-Applicant's Income. We find no relations between the rest of the variable pair combinations as they result in a value lower than what is considered statistically significant < +/-3, as shown below.



```
sns.scatterplot(data=TrainData, x="LoanAmount", y="Loan_Amount_Term")
# no relation (<0.3)

sns.scatterplot(data=TrainData, x="LoanAmount", y="Credit_History")
#no relation (<-0.3)

sns.scatterplot(data=TrainData, x="LoanAmount", y="ApplicantIncome_Bin")
# medium positive relation

sns.scatterplot(data=TrainData, x="Loan_Amount_Term", y="ApplicantIncome")
# no relation (<-0.3)

sns.scatterplot(data=TrainData, x="Loan_Amount_Term", y="CoapplicantIncome")
#no relation (<-0.3)

sns.scatterplot(data=TrainData, x="Loan_Amount_Term", y="LoanAmount")
# no relation (<0.3)

sns.scatterplot(data=TrainData, x="Loan_Amount_Term", y="Credit_History")
# no relation (<0.3)

sns.scatterplot(data=TrainData, x="Loan_Amount_Term", y="ApplicantIncome_Bin")
# no relation(<-0.3)

sns.scatterplot(data=TrainData, x="Credit_History", y="ApplicantIncome")
# no relation (<0.3)
```



```
sns.scatterplot(data=TrainData, x="ApplicantIncome", y="CoapplicantIncome")
# negative weak relation

sns.scatterplot(data=TrainData, x="ApplicantIncome", y="LoanAmount")
#positive medium relation

sns.scatterplot(data=TrainData, x="ApplicantIncome", y="Loan_Amount_Term")
# no relation (<-0.3)

sns.scatterplot(data=TrainData, x="ApplicantIncome", y="Credit_History")
# no relation (<0.3)

sns.scatterplot(data=TrainData, x="ApplicantIncome", y="ApplicantIncome_Bin")
#very strong positive relation

sns.scatterplot(data=TrainData, x="CoapplicantIncome", y="ApplicantIncome")
#  weak negative relation

sns.scatterplot(data=TrainData, x="CoapplicantIncome", y="LoanAmount")
# no relation (<0.3)

sns.scatterplot(data=TrainData, x="CoapplicantIncome", y="Loan_Amount_Term")
# no relation (<-0.3)

sns.scatterplot(data=TrainData, x="CoapplicantIncome", y="Credit_History")
# no relation(<-0.3)

sns.scatterplot(data=TrainData, x="CoapplicantIncome", y="ApplicantIncome_Bin")
# no relation (<-0.3)
```



```
In [ ]: sns.scatterplot(data=TrainData, x="Credit_History", y="ApplicantIncome")
        # no relation (<0.3)

In [ ]: sns.scatterplot(data=TrainData, x="Credit_History", y="CoapplicantIncome")
        # no relation (<-0.3)

In [ ]: sns.scatterplot(data=TrainData, x="Credit_History", y="LoanAmount")
        # no relation (<-0.3)

In [ ]: sns.scatterplot(data=TrainData, x="Credit_History", y="Loan_Amount_Term")
        # no relation (<0.3)

In [ ]: sns.scatterplot(data=TrainData, x="Credit_History", y="ApplicantIncome_Bin")
        # no relation (<0.3)

In [ ]: sns.scatterplot(data=TrainData, x="ApplicantIncome_Bin", y="ApplicantIncome")
        # strong positive relation

In [ ]: sns.scatterplot(data=TrainData, x="ApplicantIncome_Bin", y="CoapplicantIncome")
        # no relation (<-0.3)

In [ ]: sns.scatterplot(data=TrainData, x="ApplicantIncome_Bin", y="LoanAmount")
        #medium positive relation

In [ ]: sns.scatterplot(data=TrainData, x="ApplicantIncome_Bin", y="Loan_Amount_Term")
        # no relation (<-0.3)

In [ ]: sns.scatterplot(data=TrainData, x="ApplicantIncome_Bin", y="Credit_History")
        # no relation (<0.3)
```