

**ANL 252**

**Python for Data Analytics**

**End-of-Course Assessment (ECA)**

**July 2022**

**Submitted by:**

|  |  |
| --- | --- |
| **Name** | **PI No.** |
| **Vivian Wee Si Min** | **E2010380** |

**Tutorial Group: ­­­­­­­­­­­­ T 09**

**Instructor’s Name: Prof. Kumar Munish**

**Submission Date: 05/09/2022**

**Question 1**

|  |  |
| --- | --- |
| **Categorical Variables** | **Numeric Variables** |
| ID | LIMIT |
| GENDER | BALANCE |
| EDUCATION | INCOME |
| MARITAL | AGE |
| S(n) | B(n) |
| RATING | R(n) |

**Question 2**

Importing relevant libraries:

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

%matplotlib inline

import statsmodels.api as sm

from statsmodels.formula.api import ols

import sklearn

from sklearn.metrics import mean\_squared\_error , r2\_score

from sklearn.model\_selection import train\_test\_split

Read CSV file:

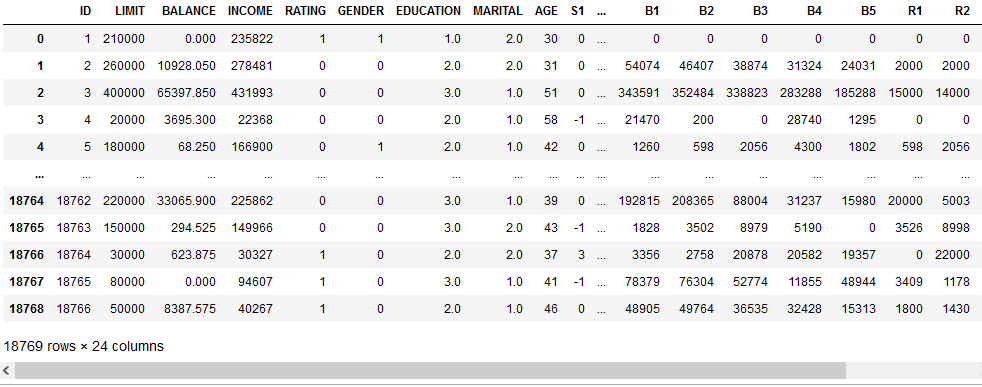
#read csv file

df = pd.read\_csv(r'C:\Users\vivia\Desktop\ANL252\ECA\ECA\_data.csv')

df

print(df.shape)

Output shows dataframe df:



Dataframe has 18769 rows and 24 columns:

**1st pre-processing task: Removing duplicates**

#remove duplicates

df = df[~df.duplicated()]

df.shape



Output shape shows 18766 rows, meaning 3 duplicate records dropped

**2nd pre-processing task: Remove missing values**

Checking the number of rows with missing values:

df.isnull().values.sum()

Output:



Since the number of rows with missing values are small, it is okay to drop them as it will not impact the dataset a lot.

#remove rows with any missing values

df = df.dropna(how='any')

df.shape

Output:



**3rd pre-processing task: Remove ambiguous values**

Remove negative age and ages of 199

#Remove impossible values for AGE

drop\_age = df[(df['AGE']==-1) | (df['AGE']==199)].index

drop\_age = list(drop\_age)

df = df.drop(drop\_age,axis=0).reset\_index(drop=True)

Remove dollar sign and commas for column R3 so that we can convert to integer datatype later on.

#remove '$' & ',' sign from column R3

df['R3'] = df['R3'].str.replace('[$,]','')

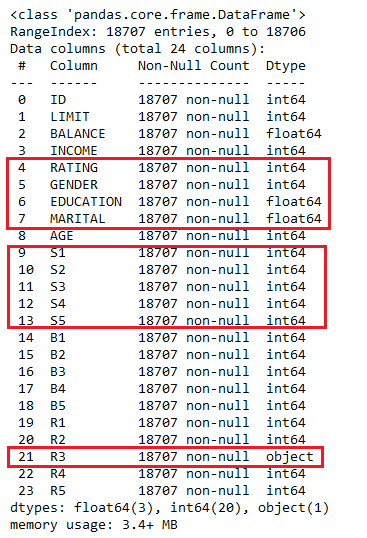
df.shape

Output:



**4th pre-processing task: Changing appropriate datatypes**

The data types of those variables highlighted were discovered to be not appropriate, as such, these needs to be changed. The datatypes are all either integers or floats, and it needs to be changed to a string so for the dummy variable in the later parts.



#Change EDUCATION & MARITAL from float to int to remove decimal places

df[['EDUCATION','MARITAL']] = df[['EDUCATION','MARITAL']].astype(int)

#Create a list of data types to update

update\_dtypes = [str,str,str,str,int,str,str,str,str,str]

#create a list of variables where the datatype needs to be changed

update\_vars = ['RATING','GENDER','EDUCATION','MARITAL',

'R3','S1','S2','S3','S4','S5']

#create dictionary with variable as keys and datatype as values

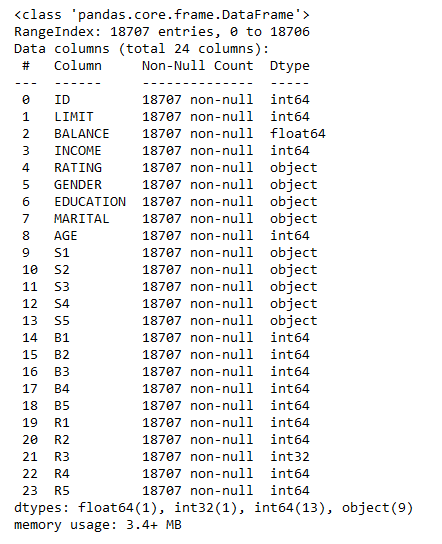
update = dict(zip(update\_vars,update\_dtypes))

#Update datatypes for dataframe

df = df.astype(update)

df.info()

Output:



Datatypes successfully changed for further processing.

**Extra pre-processing task: Removing outliers**

#Select columns that have numeric variables only

cols = df.select\_dtypes('number').columns

#Create a subset dataframe for numerical columns

df\_sub = df.loc[:, cols]

#Remove outliers that are outside 3 standard deviations from mean for all columns;

lim = np.abs((df\_sub - df\_sub.mean()) / df\_sub.std(ddof=0)) < 3

#Replace outliers with NAN in main dataframe

df.loc[:, cols] = df\_sub.where(lim, np.nan)

#Drop null values

df.dropna(inplace=True)

df.shape

Output:



The above code first selects the columns that holds numeric variables in the dataframe. These columns were then used to create a subset of the dataframe, where all values are subjected to filtering based on 3 standard deviations from its mean. The outliers were converted to missing values (NAN) in the main dataframe, where it was subsequently dropped. As such, outliers outside 3 standard deviations are removed. Outliers are removed so that it would not skew the data causing bias when analysing the data in later stages.

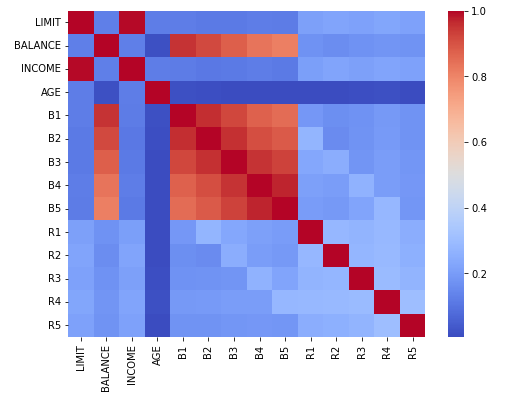
**Question 3**

**1st Visualization: Heatmap**

fig = plt.figure(figsize=(8,6))

sns.heatmap(df.corr(),cmap='coolwarm');

Output:



The correlation heatmap above shows the correlations between all numeric variables. Based on the heatmap above, we can see that there was a strong positive correlation between the variables ‘INCOME’ and ‘LIMIT’. This suggests that income is a significant factor in determining a high credit limit.

Another interesting insight is that there is a low correlation between ‘AGE’ and ‘INCOME’ and between ‘AGE’ and ‘LIMIT’. Usually, one would expect that the older a person gets, the higher income they would have and thus resulting in a lower credit limit. However, the data shows otherwise. As such, the age range of customers may not be a significant factor in the credit business.

Lastly, there was weak correlation between ‘INCOME’ and ‘BALANCE’, this suggests that the earning power of an individual does not determine the balance they would have which ultimately affects their credit ratings. This proves that, even a high-income earner may have poor payment patterns which results in bad ratings. Therefore, the credit facility should investigate other factors to predict defaults.

**2nd Visualization: Pie charts**

fig = plt.figure(figsize=(12,8))

ax1 = fig.add\_subplot(221)

df['GENDER'].value\_counts().plot(kind='pie', autopct='%.1f%%', startangle=90,

labels=['Female','Male'],colors=['hotpink','royalblue'],

wedgeprops={'edgecolor':'k'},

textprops={'fontweight':'bold'});

plt.axis('equal')

ax1.set\_ylabel('GENDER',size=15,

fontdict={'weight':'bold','color':'navy'})

ax2 = fig.add\_subplot(222)

df['RATING'].value\_counts().plot(kind='pie', autopct='%.1f%%', startangle=90,

labels=['Good','Bad'],colors=['limegreen','tomato'],

wedgeprops={'edgecolor':'k'},

textprops={'fontweight':'bold'});

plt.axis('equal')

ax2.set\_ylabel('RATING',size=15,

fontdict={'weight':'bold','color':'navy'})

ax3 = fig.add\_subplot(223)

df['EDUCATION'].value\_counts().plot(kind='pie', autopct='%.1f%%', startangle=90, labels=['Tertiary','Postgraduate','Highschool','Others'], colors=['darkorange','y','gold','lightyellow'], wedgeprops={'edgecolor':'k'},textprops={'fontweight':'bold'});

plt.axis('equal')

ax3.set\_ylabel('EDUCATION',size=15,

fontdict={'weight':'bold','color':'navy'})

ax4 = fig.add\_subplot(224)

df['MARITAL'].value\_counts().plot(kind='pie', autopct='%.1f%%', startangle=90,labels =['Married','Single','Others'],

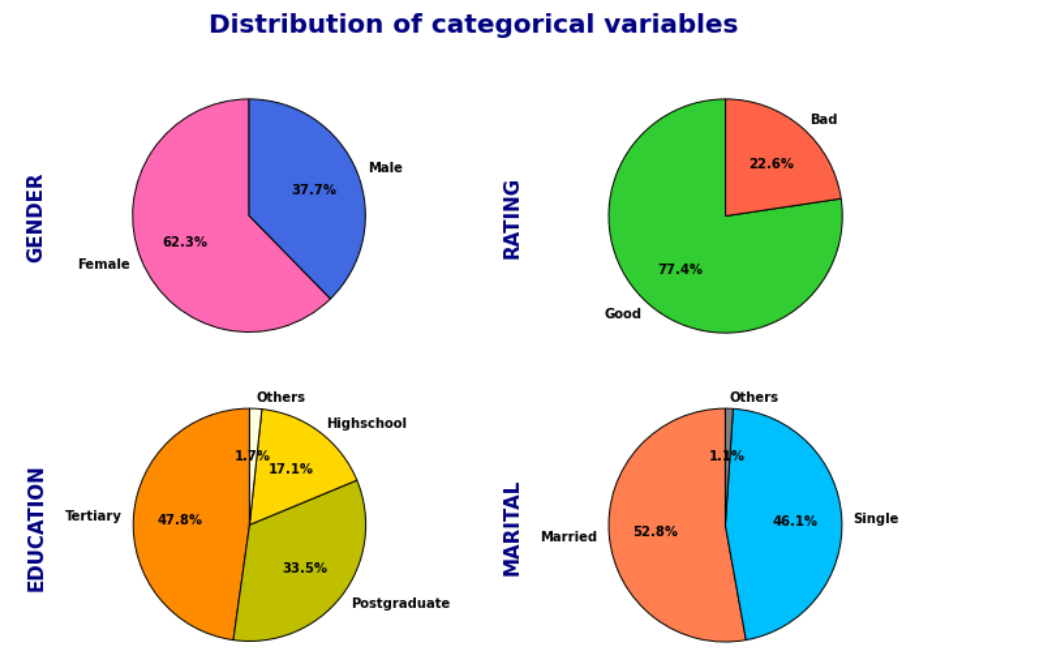
colors=['coral','deepskyblue','grey'], wedgeprops={'edgecolor':'k'},textprops={'fontweight':'bold'});

plt.axis('equal')

ax4.set\_ylabel('MARITAL',size=15, fontdict={'weight':'bold','color':'navy'})

fig.suptitle('Distribution of categorical variables',size=20,fontweight='bold',color='navy');

Output:



The above pie charts show a distribution of categorical variables such as Gender, Rating, Education and Marital. It shows the demographics of people in this dataset, which can be used to determine the profiles of a typical customer that frequents the credit facility.

From the pie charts, we can see that most of the users are female (62.3%) as compared to males (37.7%). Most of the customers have either a tertiary (47.8%) or postgraduate (33.5%) level of education. In addition, the credit facility has an equal split of customers that are single (46.1%) and married (52.8%). Lastly, we can also see that most of the customers here have a good credit rating of 77.4%, however about 22.6% of them have a bad credit rating.

**3rd Visualization: Stacked bar chart**

#subset payment statuses

df\_status = df.loc[:,'S1':'S5']

#get proportion of payment statuses

def get\_proportions(A):

#Group all S(n) > 0 as delayed payments, then count

x\_delay = df\_status[A][(df\_status[A]!='0') & (df\_status[A]!='-1')].count()

#Count prompt payments

num\_prompt\_payment = df\_status[A][df\_status[A]=='-1'].count()

#Count minimum payments

num\_min\_payment = df\_status[A][df\_status[A]=='0'].count()

#Total number of payments

total\_num = x\_delay + num\_prompt\_payment + num\_min\_payment

#Get proportion of each payment status

proportion = [num\_prompt\_payment,

num\_min\_payment,x\_delay]/total\_num

return proportion

#Assign proportions to each rows

x = df\_status.columns

y = []

for i in x:

y.append(get\_proportions(i))

#create dataframe for plot

z = pd.DataFrame(y,columns=

['prompt payment','minimum payment','late payment'],index=(['S1','S2','S3','S4','S5']))

z.plot.bar(stacked=True)

plt.ylabel('Proportions',weight='bold')

plt.xlabel('Repayment status',weight='bold')

plt.xticks(rotation=0)

plt.legend(bbox\_to\_anchor=(1,0.6))

#To place proportions on stacked graph

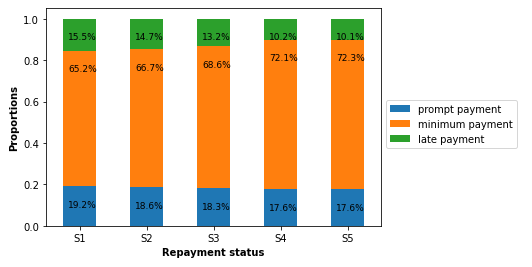
for n, x in enumerate([\*z.index.values]):

for (proportion, y\_loc) in zip(z.loc[x],z.loc[x].cumsum()):

plt.text(x=n - 0.17,y=y\_loc - 0.1, s=f'{np.round(proportion \* 100, 1)}%',

color="black",fontsize=9);

Output:



In this chart, all the payment statuses for S(n) = x were classified as delayed or late payment. One prominent insight from this chart is that the late payment status has increased gradually by approximately 5% over the previous 5 months. Next, the proportion of customers with prompt payment status has improved marginally by about 2% over the previous 5 months. Lastly, the proportion of customers who made minimum payment has decreased by approximately 7% as well. However, this could be mainly attributed to the increase in late payment customers and the slight increase in customers that make prompt payment.

This suggests that there was a significant conversion from minimum paying customers to late paying customers, meaning this should be the segment of customers to focus on to improve collection rates. In addition, this could also suggest an internal issue with the facility’s credit policy , where perhaps there was no incentive to make prompt payment or lacking significant penalties when there were any late payments.

**4th Visualization: Violin plot**

#Setting order of marital status to show

order=['0','1','2']

#setting up violin plot

sns.set\_style('whitegrid')

ax = sns.catplot(x='RATING',y='AGE',hue='EDUCATION',

row='MARITAL',data=df,kind='violin',height=8,aspect=2.2,

row\_order=order)

fig = ax.fig

#Set overall title

fig.suptitle(

"Distribution of age to good and bad ratings",

y=1.05,x=0.45,size=25,fontweight='bold')

#Set title of subplot

a0 = fig.axes[0].set\_title('MARITAL status = Others',size=20)

a1 = fig.axes[1].set\_title('MARITAL status = Single',size=20)

a2 = fig.axes[2].set\_title('MARITAL status = Married',size=20)

#Customise overall axes

ax.set\_xlabels('RATING',size=30)

ax.set\_ylabels('AGE',size=30)

ax.set\_xticklabels(['Bad','Good'],size=25)

ax.set\_yticklabels(size=25)

#Customise legend

ax.\_legend.set\_title('EDUCATION')

plt.setp(ax.\_legend.get\_title(), fontsize=25)

new\_labels = ['Postgraduate','Tertiary','High school','Others']

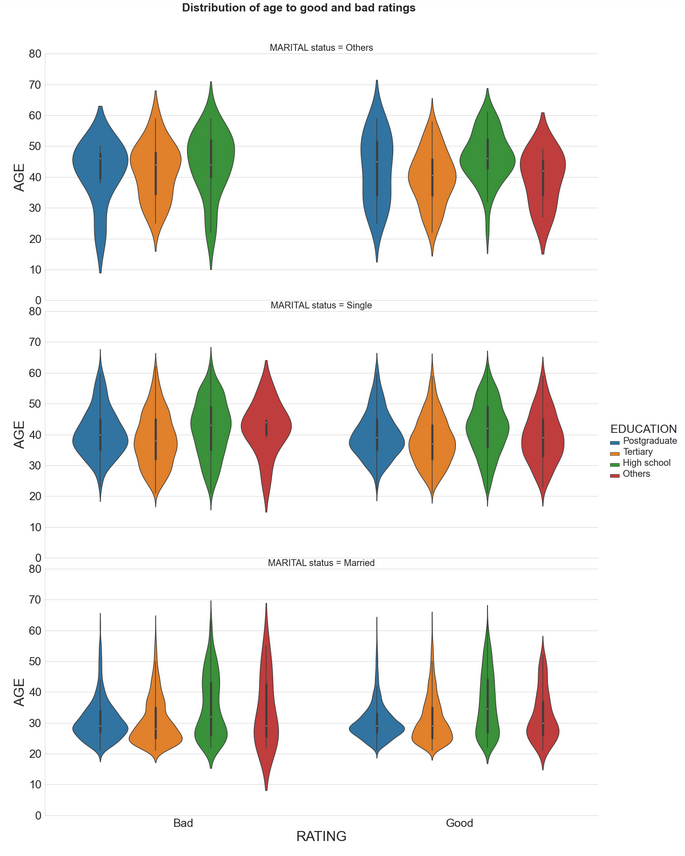
for t, l in zip(ax.\_legend.texts, new\_labels):

t.set\_text(l)

plt.setp(ax.\_legend.get\_texts(), fontsize=20)

plt.setp(ax.\_legend,bbox\_to\_anchor=(1.1,0.5));

Output:



The violin plot above shows the distribution of age between customers with good and bad ratings, separated by marital status and education. The width of the plot shows the highest frequency of age, while the lines in the violin plot represents a boxplot with their respective quartiles. The plot could be used to generalise traits of a typical customer in accordance with their ratings.

Firstly, the distribution of customer’s age with good and bad ratings seems to increase from the clusters of late 20s to late 30s to late 40s for postgraduates from marital status of Married to Others. While comparing the distribution between ratings, there seems to be no significant difference between Single and Married individuals. However, for marital status of Others, most customers with bad ratings are in their late 40s while customers with good ratings are more evenly spread across age groups. This suggest that customers who are widowed, divorced, or separated with a postgraduate education that are in their 40s are more likely to have a bad credit rating based on the given data set.

Next, comparing the distribution of age for different education levels for single customers. The age distribution of single customers is concentrated between their 30s to 40s for postgraduates, tertiary and high school individuals, which was similar between good and bad ratings. However, for those that are single and has received other education, a high proportion of customers with bad ratings are in their mid-40s while a high proportion of customers with good ratings are in their mid-30s. This suggests that customers who are single that has other education, generally has a bad rating when they are in their mid-40s.

**5th Visualization: Scatter plot**

#copy original dataframe

df\_ilr = df.copy()

#create new column for income-limit ratio

df\_ilr['INCOME/LIMIT'] = df['INCOME']/df['LIMIT']

#Set plot style

sns.set\_style("darkgrid")

#Set up scatterplot

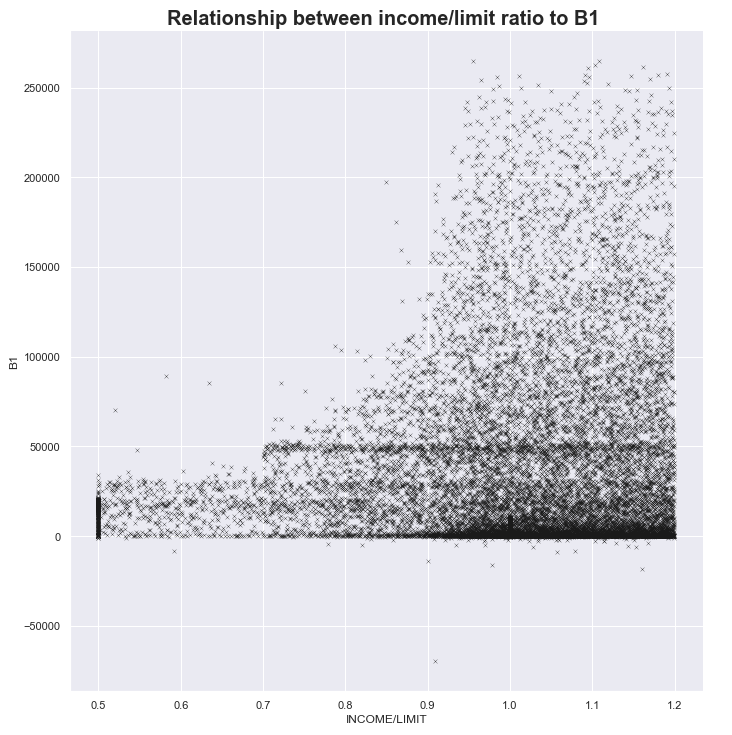
ax1 = sns.relplot(x='INCOME/LIMIT',y='B1',data=df\_ilr,

s=15,color='k',alpha=1,marker='x',height=10)

#Set title

plt.title('Relationship between income/limit ratio to B1 ',size=20,fontweight='bold');

Output:



The code creates a new column from the dataframe by dividing income with limit, resulting in an income-limit ratio, where it represents the income for every credit limit given to some individual.

In this scatterplot, it clearly shows a non-linear relationship between income-limit ratio and B1. As income-limit ratio increases, B1 will likely increase exponentially on average. This suggests that when limits are higher than what an individual earn, they are more likely to make payment such that balances are low. On the flipside, customers with lower limits relative to their income, tend to not make payment such that their balances are high. Hence, perhaps the credit facility could look into their policy on limits as it directly affects the customers repayment rates.

**Question 4**

df = df.reset\_index()

#use min max normalization technique

scaler = sklearn.preprocessing.MinMaxScaler()

#create a copy of original dataframe

df\_minmax = df.copy()

#get numeric columns

num\_cols = df\_minmax.select\_dtypes('number').columns

#get categorical columns

cat\_cols = df\_minmax.select\_dtypes('object').columns

#scale numeric columns

df\_minmax = pd.DataFrame(scaler.fit\_transform(df[num\_cols[1:]]),columns=df[num\_cols[1:]].columns)

#Add categorical columns to normalised dataframe

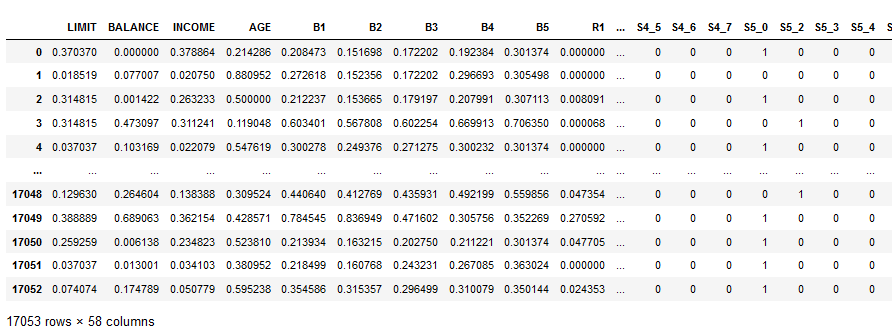
df\_minmax[cat\_cols] = df[cat\_cols]

#Generate dummy variables

df\_minmax = pd.get\_dummies(df\_minmax,drop\_first=True)

df\_minmax

Output:



The data was separated into numeric variables using cols as defined previously. Subsequently, the numeric variables were normalized to let values sit in a common scale where it ranges from 0 to 1, now, each predictor will affect the response variable equally depending on their coefficients. In this case, the variable age lies on a different scale as the other variables: income, balance etc, hence they are normalized. The categorical variables were left untouched and later joined back into the normalized dataframe to form an updated dataframe. Next, a set of dummy variables were generated from the categorical variables. This is to facilitate the fitting of linear regression model later.

In this part, we will be using the statsmodel package from python.

import statsmodels.api as sm

from statsmodels.formula.api import ols

import sklearn

from sklearn.metrics import mean\_squared\_error , r2\_score

from sklearn.model\_selection import train\_test\_split

#create string for regression later

df\_ols = df\_minmax.drop('B1',axis=1)

x = df\_ols.columns

seperator = '+'

y = seperator.join(x)

z = 'B1 ~ ' + y

print(z)

In order to input the response and predictor variables using the Ordinary Least Squared (OLS) model in statsmodel, it is required to input this as a string format to ‘response variable ~ predictor 1 + predictor 2 …’ etc. As such, a string was created based on the column names extracted from the normalized dataframe.

Output:



#Split data into 80% training set, 20% testing set with seed=123

dftrain,dftest = train\_test\_split(df\_norm\_1,test\_size=0.2,random\_state=123)

In order to test the linear regression model later on, we reserve 20% of the records from the dataframe as a test set, while the remaining 80% of records can be used to train the regression model. Random\_state was set to 123 to ensure replicability.

#Fit ols model to training set

mlr = ols(z,data=dftrain).fit()

#Use fitted model to predict B1 using test data

ypred = mlr.predict(dftest)

#print mean squared error of train model

print('MSE =',mlr.mse\_resid)

#print r-squared of test model

print('Rsq =', r2\_score(dftest.B1, ypred))

#print root mean squared error of test model

print('RMSPE =', np.sqrt(mean\_squared\_error(dftest.B1, ypred)))

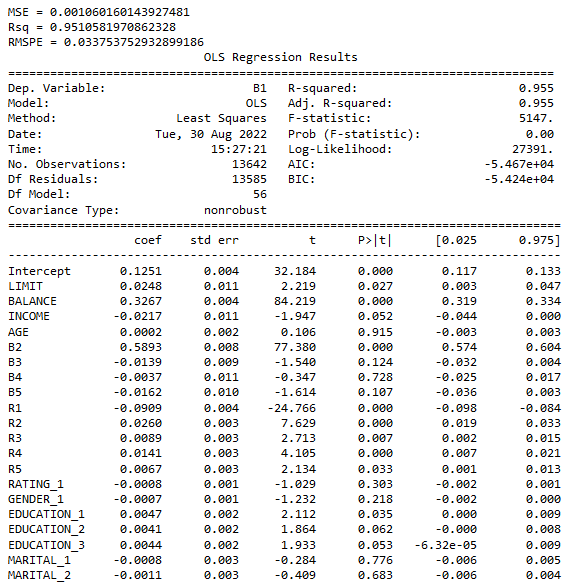
#print summary of train model

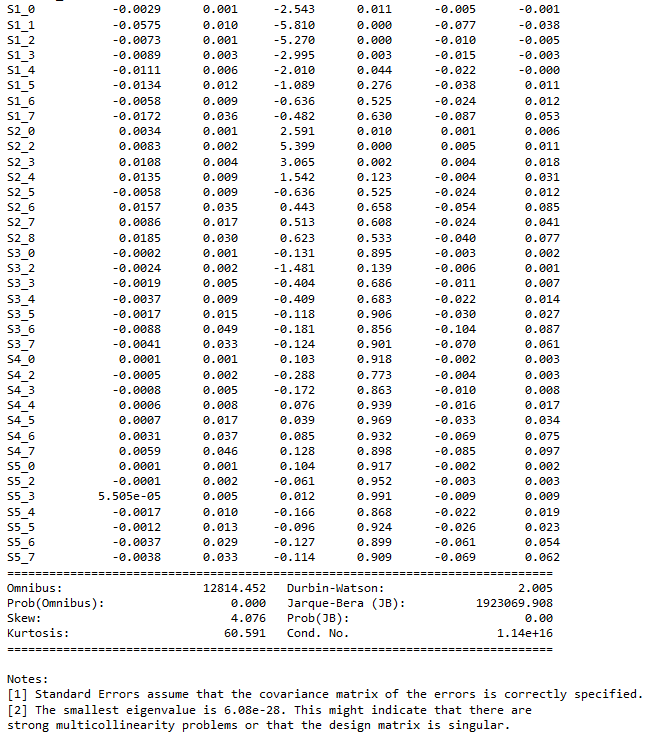
print(mlr.summary())

The model was fitted with the training data, where it includes all the predictor variables. The fitted model was assigned to the variable ‘mlr’. Next, the fitted model was used to predict the response variable ‘B1’ using the test data. To evaluate the model, we print out the mean squared error of the train model, the r-squared of the regression on the test data, and the root mean squared error of the regression on the test data.

Lastly, the summary of the linear regression was printed out.

Output:





Based on the evaluation metrics, it seems that the model explains most of the variability of B1, as R-squared is about 0.955 and the errors are small. However, it seems like there are too many predictors in this model. In order to optimize and simplify the model for business use, we are going to remove any insignificant predictors using the t-test statistics, where we set a significance level of 0.05.

response='B1'

alpha = 0.05

#create function to remove insignificant predictors for each iteration

def backward\_selection(df, response, alpha=0.05):

#get all predictor variables

predictors = df.columns.drop(response)

model = []

#Cannot have 0 predictors

while len(predictors)>0:

#set up string for ols function

seperator = '+'

var\_string = seperator.join(predictors)

ols\_string = 'B1 ~ ' + var\_string

#fit ols model based on current predictors

mlr = ols(ols\_string,dftrain).fit()

#create a list of the sets of predictors for each iteration

model.append(mlr.model.exog\_names)

#Get the maximum p-value for the current set of predictors

pmax = mlr.pvalues.iloc[1:].max()

#remove predictors that is above the significance level

if pmax > alpha:

predictors = predictors.drop(mlr.pvalues.iloc[1:].idxmax())

else:

break

#return the optimal predictors

return model[-1]

#get optimal predictors

new\_predictors = backward\_selection(dftrain, response, alpha)

The code above is a user defined function to progressively remove insignificant predictors from the model. First, we get all the predictor variables from the dataframe column and assign it as ‘predictors’. Subsequently, a while loop was used when the number of predictors is more than 0 as the minimum number of predictors is 1. For each iteration, a new set of string (updated predictors) will be updated and fed into the linear regression model, where the training data set is used. Next, the attributes of the fitted model like the names of the remaining predictor variables and its corresponding p-value were extracted and compared against the significance level (alpha). If the variable with the highest p-value exceeds the significance level, it will be removed from the ‘predictors’ array where the loop repeats. Once all insignificant variables are removed, the loop breaks and returns the remaining predictor variables as an array.

#exclude intercept term

new\_predictors = new\_predictors[1:]

#set up string for ols function

seperator = '+'

var\_string = seperator.join(new\_predictors)

ols\_string = 'B1 ~ ' + var\_string

print(ols\_string)

Output:



We update the significant variables into a string for the new OLS model.

#fit model with optimal predictors

mlr\_new = ols(ols\_string,data=dftrain).fit()

#Use updated fitted model to predict B1 using test data

ypred\_new = mlr\_new.predict(dftest)

#print mean squared errors of residuals for train model

print('MSE =',mlr.mse\_resid)

#print r-squared of test model

print('Rsq =', r2\_score(dftest.B1, ypred\_new))

#print root mean squared errors for test model

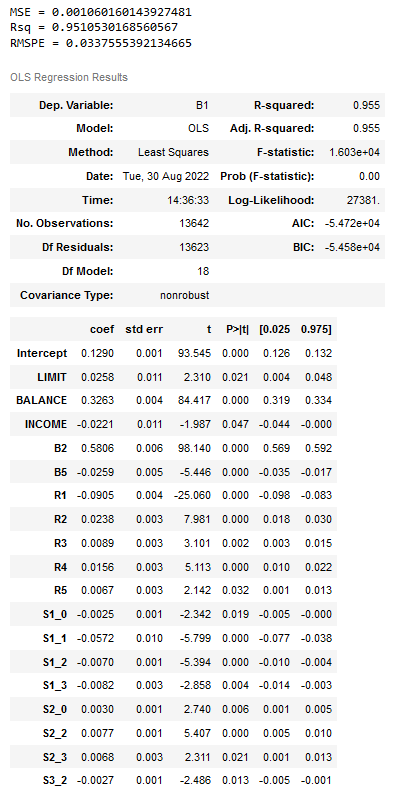
print('RMSPE =', np.sqrt(mean\_squared\_error(dftest.B1, ypred\_new)))

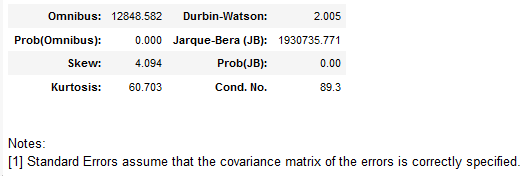
#print summary for optimal model

mlr\_new.summary()

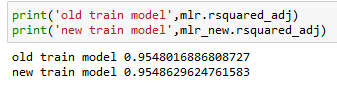
The significant variables are then fitted into the OLS model, which was assigned as ‘mlr\_new’. The outputs of MSE, R-squared and RMSE are once again generated.

Output:





Adjusted R-squared:



Based on the new model, we can see that all the p-value of corresponding variables are less than the significance value. As such, this will be the set of predictors used to predict B1. The new model seems to have only improved marginally since R-squared, MSE & RMSE remained similar to the previous model.

**Question 5**

#get predictor names for new model

var\_names = mlr\_new.model.exog\_names

#get coefficients from of new model

coefficients = (round(mlr\_new.params,4))

#convert coefficients to string

l = []

for i in coefficients:

l.append(str(i))

#combine coefficients with predictor as string

test = []

for i,x in enumerate(var\_names):

test.append('(' + l[i]+' '+ x +')')

Regression\_eqn = 'B1 = ' + seperator.join(test)

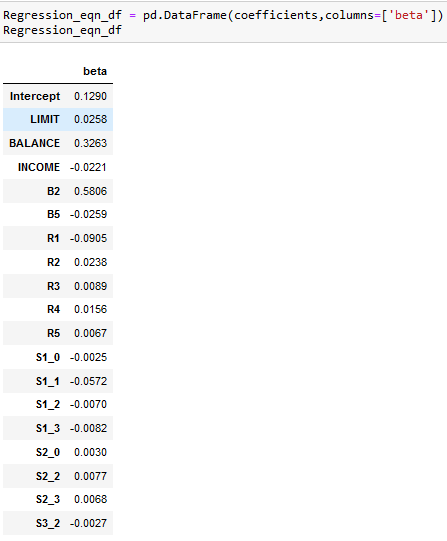
#print out the regression equation as string format

print(Regression\_eqn)

Output:



Another view:



Since the data was normalised using the min max method, where a normalised variable = (value – min.value)/(max.value-min.value). Therefore, when combined with the coefficients, it could be understood as a linear multiplier to the variable where (y-value = (coefficient\*(max.value-min.value)+min.value). For example, looking at the variable ‘LIMIT’, while keeping all other variables constant, for every normalised unit increase in ‘LIMIT’, the normalised value of B1 will increase by 0.0258 on average.

However, to properly interpret the regression equation, we can convert the values back to its unnormalized form. Below are the equations to do that:

For example:

The equation is B1norm = β1\*LIMITnorm + β2\*BALANCEnorm + …. + β18\*S3\_2norm

Considering the normalised LIMIT term, while keeping all other variables constant. When LIMITnorm increases by 1 unit, B1norm will change by a certain amount.

The change is thus:

B1 norm+1 – B1norm = β1\*(LIMITnorm + 1) - β1\*LIMITnorm

B1 norm+1 – B1norm = β1

Let B1norm correspond to B11 and B1norm+1 correspond to B12;

Then B1norm = (B11 – B1min)/(B1max – B1min) ; thus,

(B12 – B1min)/(B1max – B1min) - (B11 – B1min)/(B1max – B1min) = β1

(B12 – B11)/(B1max – B1min) = β1

ΔB1 = β1(B1max – B1min)

Hence, generalising the above for all predictor variables, for every unit change in the normalised predictor, the dollar change in B1 is the predictor’s coefficient multiply by the range of B1.

Rewriting the above into codes:

#Get maximum and minimum values for B1

maxi = df['B1'].max()

mini = df['B1'].min()

#range of for B1

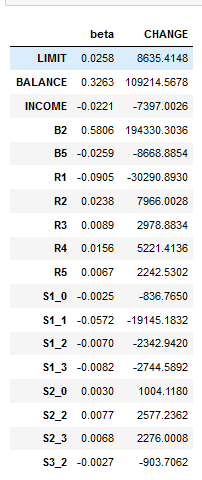
B1\_range = maxi – mini

#change in B1 per unit change in normalised predictor

Regression\_eqn\_df['CHANGE'] = Regression\_eqn\_df['beta']\*B1\_range

Regression\_eqn\_df.drop('Intercept',axis=0)

Output:



Insights

The initial model had all 58 predictors included in it. However, through backwards selection, the new model now includes only 18 predictors. It is interesting to see that certain notable variables like age, gender, education, and ratings are considered insignificant in predicting the previous month’s balances. In other words, they are irrelevant in determining the B1.

As mentioned above, based on the R-squared value, MSE and RMSPE , the predicting accuracy of the new model had only improved marginally as compared to the old model. However, a significant improvement is the reduction of features, where the regression is less complicated. Furthermore, the information required to accurately predict B1 is fewer.

Next, based on the coefficients, we can see that the predictors like BALANCE, B2, R2 to R5 etc has a positive relationship with B1 while the others have a negative relationship with B1. As expected, the higher the BALANCE, the higher the B1 is, due to the possibility of a unpaid balance rolling over to the current month. Also, the higher the INCOME, the lower the balance since customers with more disposable income can afford to repay comfortably.

Out of all the predictors, B2 has the most positive effect on B1, while R1 has the most negative effect on B1. This suggests that a customer with a high balance two months ago, will likely have a high balance last month, which will likely to have been rolled over from the previous month (B2) . On the other hand, the balances in the previous month will rightfully decrease when the customer has made repayment on the same month.

It was interesting to observe that the repayment R2 to R5 was inversely related to B1. This suggests that when a customer repaid more debt in prior months leading to B1, it results in the customer having a higher billable balance B1, this could be due to their expenditure and cashflow patterns where higher repayment leads to lower cash on hand, thus being unable to repay fully, which leads to higher balances.

It was observed that a few variables of payment statuses were significant to the model, the payment status of the previous month, where minimum payment and delayed payments interestingly contributed to the reduction in B1. This could suggest that customers who have such payment statuses in the same month usually has a smaller billable amount to work with.

It is important to note that the amount of change as stated in the CHANGE column varies according to the degree of change for the numeric predictors. Whereas for categorical predictors such as payment statuses, the change in B1 stays fixed as the value of statuses is either 0 or 1. This shows that the numeric predictors can have a greater effect on B1 as compared to the categorical predictors since the numeric predictors can have cases where it exceeds 1.

**References**

1. Forrest-Baldini, J. (2020, May 23). Create your own coefficient plot function in Python. Medium. Retrieved August 31, 2022, from https://medium.com/analytics-vidhya/create-your-own-coefficient-plot-function-in-python-aadb9fe27a77
2. Introduction - statsmodels. statsmodels. (n.d.). Retrieved August 31, 2022, from <https://www.statsmodels.org/dev/index.html>
3. 3.3. Metrics and scoring: Quantifying the quality of predictions. scikitlearn. (n.d.). Retrieved September 1, 2022, from <https://scikit-learn.org/stable/modules/model_evaluation.html>
4. Kumar, B. (2021, December 26). Matplotlib Pie Chart tutorial. Python Guides. Retrieved September 3, 2022, from <https://pythonguides.com/matplotlib-pie-chart/>
5. Real Python. (2022, May 16). Linear regression in python. Real Python. Retrieved September 2, 2022, from https://realpython.com/linear-regression-in-python/
6. Sklearn.preprocessing.MinMaxScaler. scikitlearn. (n.d.). Retrieved September 1, 2022, from <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html#sklearn.preprocessing.MinMaxScaler>
7. Seaborn.catplot. seaborn. (n.d.). Retrieved September 3, 2022, from <https://seaborn.pydata.org/generated/seaborn.catplot.html>
8. Seaborn.relplot. seaborn. (n.d.). Retrieved September 4, 2022, from <https://seaborn.pydata.org/generated/seaborn.relplot.html>
9. Seaborn.violinplot. seaborn.violinplot. (n.d.). Retrieved September 4, 2022, from <https://seaborn.pydata.org/generated/seaborn.violinplot.html>