I. Introduction

a. The Problem

Big data is a rapidly evolving area for business intelligence. For businesses to be successful it has become crucial to collect vast quantities of data, which can be used to keep a competitive advantage by tailoring and improving goods and services for their customer base. In addition, businesses can mitigate risks and optimize sales. Machine learning (ML) techniques are a useful tool to assist business intelligence as it uses sophisticated algorithms to analyse multivariate data using a high-level programming language such as python, to assist decision making. Using as programming language is necessary as it is difficult for humans to manually carry out these algorithms as well as being expensive and time costly. However, due to the rapid growth of big data and the constant innovation of data collection techniques the original frameworks to handle big data such as MapReduce are becoming obsolete. This is where Spark an open source computing technology, can be implemented. Despite Spark being able to use different programming languages, python will be used for this investigation. Python is a simple to use effective programming language, which has visual and real time analytic properties which make it an appropriate choice for this research task. PySpark is a collaboration between Apache Spark and Python (Edureka, 2020)

This project is a data science investigation carried out in PySpark solving a classification problem. The study is used to assess whether running a logistic regression in PySpark can successfully predict credit card defaults for a Taiwan credit card issuer.

b. Previous research

There has been limited published literature using this data set. However, the results found on the website Kaggle are all close to 77% accuracy (Kaggle, 2018, 2019). This will be the benchmark where the model generated in this study will be compared.

c. Data set

The data set for this investigation was taken from the UCI Machine learning repository. It contains information on credit card users from Taiwan in 2005 (Machine Learning Repository, 2016).

d. Aims

The default credit card client's data set has had extensive data analytical projects carried out on it using a variety of ML methods. To demonstrate my skill set and to differentiate this research to the existing body of research carried out using this data set. The research question is:

- Can logistic regression using under-sampling successfully predict whether a client will or will not default on their credit card?

In addition to this aim, the relationship between the features and the target variable will be explored using visualizations.

The visualizations were carried out using matplotlib and seaborn libraries in python.

II. Implementation

1. Background

1.1 Spark

Apache Spark is a general distributed processing engine. It is intended for fast, large scale data processing over a cluster of computer systems. Its primary purpose was to replace MapReduce as the preferred big data processing platform; however, it has arguably surpassed this as it overcomes many of MapReduce short falls. It is the most actively developed open source project for big data (Databricks). Some of Sparks key features are; it's in memory computations which overcome the latency issue associated with MapReduce to allow Spark to be as fast as it is, the general purpose platform which allows it to be applied to different types of data analytical tasks , its versatility in processing capacity which is enabled through its horizontal scalability capability and fault tolerance attributes. At a high-level Spark consists of two core parts the head node and the workers. The code, which is wrote by the programmer, is ran from the head node using the spark drivers. The code is then executed from the workers, in typical practice there is one worker per processing core.

1.2 PySpark

PySpark allows programmers to use Apache Spark through using python. Apache Spark is originally written in Scalar programming language. However, due to the Py4j library, Spark can support python. This allows programmers to interact with RDDS and data frames using the python programming language.

1.3 Spark Context

Spark context is the main entry point to any spark functionality. A Java virtual machine (JVM) is launched and creates a Java Spark Context through initiating a Spark context in PySpark. A Spark Context is initiated by running a Spark application. Using the Py4J library, initiating a Spark Context in PySpark will automatically create a JVM and a Java Spark Context. This enables PySpark to carry out any Spark functionality (Tutorials Point).

1.4 RDD

Resilient Distributed Data sets (RDD) is a fundamental data structure in Spark. An RDD is partitioned across machines logically in a cluster automatically by Spark and parallelizes the operations performed on them. They are generated in two primary ways. The first being a transformation of an existing RDD. The second being data imported as an RDD from an external data set. The first method utilises transformations such as filter, reduce and map which result in a new RDD being created. The second form of RDD operation, is an RDD action which yields a result rather than a new RDD. RDDs are resilient as they contain the ability to rebuild a data set in case of a node failure, this attribute assists in the fault tolerance feature of Apache Spark.

1.5 Data Frame

A PySpark data frame is a tabular storage method for data. It provides a useful tool for the programmer to visually see the data in a readable format. It also allows the programmer to carry out operations on the data set.

1.6 Spark MLib

Spark Mlib is the Spark machine learning library. It can distribute statistical and machine learning algorithms over a cluster of machines. Sparks Mlib is nine times faster than the Apache mahute Hadoop disk version (Simplilearn, 2017).

2.1 Data Set

Structured Data was used in this investigation

The data set used in this study contained information regarding thirty thousand customers from a Taiwan-based credit card issuer. There are 24 attributes and one target variable. The target variable is a binomial classification on whether the client defaulted acknowledged by a 1 or did no default acknowledged by a 0 on their credit card. The other attributes are described in *Table 1.*

Table 1 – Table of Attributes

Name of Attribute	Description	Data Type
ID	Identification number assigned to each client	Integer
LIMIT_BAL	Amount of credit given in NT dollars. Includes individual and family/supplementary credit.	Continuous
SEX	Gender (1=Male, 2 = Female)	Integer
Education	1 = Graduate school, 2 = University, 3 = High School, 4 = Others, 5 = Unknown, 6 = Unknown	Integer
MARRIAGE	Marital status (1 = Married, 2 = single, 3 = others	Integer
AGE	Age of client in years	Continuous
PAY_0	History of payments made per month from April to September 2005. Repayment status in September (2005) -1 = pay duly, 1 = payment delay for one month, 2 = payment delay for two months, 3 = payment delay for three months, 4 = payment delay for four months, 5 = payment delay for five months, 6 = payment delay for six months, 7 = payment delay for seven months, 8 = payment delay for eight months, 9 = payment delay for nine months and above. -2 = Unknown,	Integer

	0 = Unknown	
PAY_2	Repayment status in August (2005) (scale same as above)	Integer
PAY_3	Repayment status in July (2005) (scale same as above)	Integer
PAY_4	Repayment status in June (2005) (scale same as above)	Integer
PAY_5	Repayment status in May (2005) (scale same as above)	Integer
PAY_6	Repayment status in April (2005) (scale same as above)	Integer
BILL_AMT1	Amount of bill statement in September (2005) (NT dollar)	Continuous
BILL_AMT2	Amount of bill statement in August (2005) (NT dollar)	Continuous
BILL_AMT3	Amount of bill statement in July (2005) (NT dollar)	Continuous
BILL_AMT4	Amount of bill statement in June (2005) (NT dollar)	Continuous
BILL_AMT5	Amount of bill statement in May (2005) (NT dollar)	Continuous
BILL_AMT6	Amount of bill statement in April (2005) (NT dollar)	Continuous
PAY_AMT1	Amount of previous payment in September (2005) (NT dollar)	Continuous
PAY_AMT2	Amount of previous payment in August (2005) (NT dollar)	Continuous
PAY_AMT3	Amount of previous payment in July (2005) (NT dollar)	Continuous
PAY_AMT4	Amount of previous payment in June (2005) (NT dollar)	Continuous
PAY_AMT5	Amount of previous payment in May (2005) (NT dollar)	Continuous
PAY_AMT6	Amount of previous payment in April (2005) (NT dollar)	Continuous

(Machine Learning Repository, 2016)

2.2 Installing Apache Spark and PySpark Local Modes

Apache Spark and Pyspark Local Mode was downloaded and installed onto a virtual Machine.

Virtual Machine Specification:

- RAM – 8GB DDR4

- Processor – 2 cores Intel i5-9600K
- Hard Disk – 50GB
- Operating system – Ubuntu version 20.04.2 LTS

Spark Specification:

- 3.2.1 was used

Python Specification:

- version – 3.8.5

Step 1 – Check Java is installed and version. To check the java version, use the following command:

java – version

If Java is not installed, then the following command should be used:

sudo apt install

Step 2 – Download and unzip file into the Desktop directory. To unzip the file the following command should be used:

tar -xvzf spark-3.2.1-bin-hadoop3.1.1.tgz

Step 3 – Check the version of Python. To check the python version the following command should be used:

Python3 -V

If Python is not downloaded it can be downloaded using pip install. See the following code:

pip3 install python3

Step 4 – Update .bashrc file to include environmental path variables for Spark and PySpark. The following code to set up Spark and PySpark environment in the bashrc file is as follows:

export SPARK_HOME=”/usr/local/spark/spark-3.2.1-bin-hadoop3.1.1”

export PATH=”${PATH}:$[SPARK_HOME}/bin”

export PYSPARK_PYTHON=”/usr/binpython3”

export PYSPARK_DRIVER_PYTHON=”/usr/bin/python”

Step 5 – open Spark shell and PySpark to check it is installed correctly. The following command to open the spark shell and PySpark is as follows:

spark-shell

pyspark

Step 6 – After testing PySpark, Jupyter notebook can be used to interact with PySpark. To install Jupyter notebook the following command should be used:

pip3 install jupyter

Step 7 – To use Jupyter with PySpark the follow commands should be added to the .bashrc file:

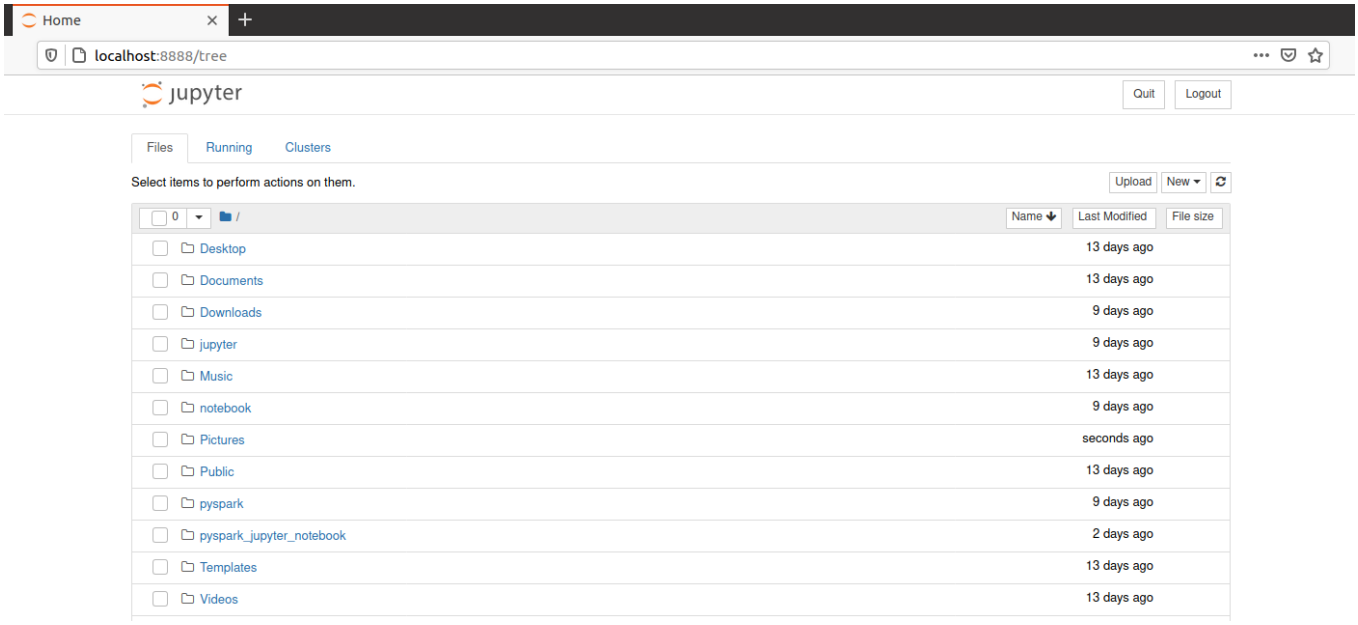
export PYSPARK_DRIVER_PYTHON=Jupyter

export PYSPARK_DRIVER_PYTHON_OPTS='notebook'

export PYSPARK_PYTHON=python3

Step 8 – To load PySpark, the following command should be used:

pyspark



2.3 Pre-processing the Dataset

Before any analysis could commence, the data set had to be cleaned. Data sets must be cleaned as in their raw forms it is unlikely that the data set is suitably for analysis. Data cleaning can involve checking and dealing with missing values, checking and dealing with incorrect data types and removing unwanted columns. This is not an exhaustive list, as there is a variety of ways in which data sets are cleaned.

The data set in this study contained missing values and unknown categories. These had to be dealt with before any data analysis could be started.

2.3.1 Loading the Data

The data set can either be loaded through and RDD or a data frame. For this investigation the data was loaded through an RDD first and then converted to a data frame after a series of RDD operations.

The data was loaded through an RDD using the following code:

rdd = sc.textFile('default of credit card clients.csv')

Results:

```
rdd = sc.textFile('default of credit card clients.csv')

rdd.take(3)
```

```
Out[5]: [',X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,Y',
 'ID,LIMIT_BAL,SEX,EDUCATION,MARRIAGE,AGE,PAY_0,PAY_2,PAY_3,PAY_4,PAY_5,PAY_6,BILL_AMT1,BILL_AMT2,BILL_AMT3,BILL_AMT4,BILL_AMT5,BILL_AMT6,PAY_AMT1,PAY_AMT2,PAY_AMT3,PAY_AMT4,PAY_AMT5,PAY_AMT6,default payment next month',
 '1,20000,2,2,1,24,2,2,-1,-1,-2,-2,3913,3102,689,0,0,0,0,689,0,0,0,0,1']
```

Code 1.1: Load data through RDD

2.3.2 RDD Operations

2.3.2.1 Filter function

Before the data was converted to a data frame. A transformation was performed using the .filter function. The top two rows which contained ambiguous column names were filtered. The following code was used to filter the first row:

rdd_head = rdd.first()

rdd1 = rdd.filter(lambda line:line!=rdd_head)

The next line of code filtered the second row:

head = rdd1.first()

rdd2 = rdd1.filter(lambda line:line!=head)

rdd2.first()

Results:

```
In [6]: rdd_head = rdd.first()
        rdd1 = rdd.filter(lambda line:line!=rdd_head)

In [7]: head = rdd1.first()
        rdd2 = rdd1.filter(lambda line:line!=head)
        rdd2.first()
Out[7]: '1,20000,2,2,1,24,2,2,-1,-1,-2,-2,3913,3102,689,0,0,0,0,689,0,0,0,0,1'
```

Code 2.1: Filtered RDD

2.3.2.2 Map and toDF function

Before the RDD was changed into a data frame the map function was used to split the data into separate elements using the comma string. In the same line of code, the columns were renamed and the RDD was converted to a data frame. The following code demonstrates this:

rdd2.map(lambda line:line.split(',')).map(lambda line: Row(ID = line[0], … ,Default = line[24])).toDF()

Results:

```
In [3]: #split by comma and change column names
from pyspark.sql import Row
df = rdd2.map(lambda line:line.split(',')).map(lambda line: Row(ID = line[0],
                                Limit_Balance= line[1],
                                Sex = line[2],
                                Education = line[3],
                                Marraige = line[4],
                                Age = line[5],
                                Repayment_sept = line[6],
                                Repayment_Aug = line[7],
                                Repayment_July = line[8],
                                Repayment_June = line[9],
                                Repayment_may = line[10],
                                Repayment_Apr = line[11],
                                Bill_stat_Sept = line[12],
                                Bill_stat_Aug = line[13],
                                Bill_stat_July = line[14],
                                Bill_stat_June = line[15],
                                Bill_stat_may = line[16],
                                Bill_stat_Apr = line[17],
                                Paid_AMT_Sept = line[18],
                                Paid_AMT_Aug = line[19],
                                Paid_AMT_July = line[20],
                                Paid_AMT_June = line[21],
                                Paid_AMT_may = line[22],
                                Paid_AMT_Apr = line[23],
                                Default = line[24]
                                )).toDF()
df.show()
```

```
+---+-------------+---+---------+--------+---+--------------+-------------+--------------+--------------+---------
----+-------------+--------------+-------------+-------------+-------------+--------+
-----+------------+-------------+-------------+------------+------------+-------+
| ID|Limit_Balance|Sex|Education|Marraige|Age|Repayment_sept|Repayment_Aug|Repayment_July|Repayment_June|Repayment
_may|Repayment_Apr|Bill_stat_Sept|Bill_stat_Aug|Bill_stat_July|Bill_stat_June|Bill_stat_may|Bill_stat_Apr|Paid_AMT
_Sept|Paid_AMT_Aug|Paid_AMT_July|Paid_AMT_June|Paid_AMT_may|Paid_AMT_Apr|Default|
+---+-------------+---+---------+--------+---+--------------+-------------+--------------+--------------+---------
----+-------------+--------------+-------------+-------------+-------------+--------+
-----+------------+-------------+-------------+------------+------------+-------+
|  1|        20000|  2|        2|       1| 24|             2|            2|            -1|            -1|
-2|           -2|          3913|         3102|          689|            0|            0|            0|
0|          689|             0|            0|           0|            0|      1|
|  2|       120000|  2|        2|       2| 26|            -1|            2|             0|             0|
```

Code 2.2: Map and toDF functions

2.3.3 Data frame operations in PySpark

2.3.3.1 Visualizing the data frame

To see the data frame in a more readable format, it can be shown as a Pandas data frame. This is useful for programmers who are familiar with Pandas. The following code can be used:

df.toPandas()

Results:

```
df.toPandas()
```

Out[5]:

	ID	Limit_Balance	Sex	Education	Marraige	Age	Repayment_sept	Repayment_Aug	Repayment_July	Repayment_June	...	Bill_stat_June	Bill_st
0	1	20000	2	2	1	24	2	2	-1	-1	...	0	
1	2	120000	2	2	2	26	-1	2	0	0	...	3272	
2	3	90000	2	2	2	34	0	0	0	0	...	14331	
3	4	50000	2	2	1	37	0	0	0	0	...	28314	
4	5	50000	1	2	1	57	-1	0	-1	0	...	20940	
...	...	...	...	...	...	...	...	...	...	...	...	...	
29995	29996	220000	1	3	1	39	0	0	0	0	...	88004	
29996	29997	150000	1	3	2	43	-1	-1	-1	-1	...	8979	
29997	29998	30000	1	2	2	37	4	3	2	-1	...	20878	
29998	29999	80000	1	3	1	41	1	-1	0	0	...	52774	
29999	30000	50000	1	2	1	46	0	0	0	0	...	36535	

30000 rows × 25 columns

2.3.3.2 Dropping columns

The ID column was removed because these numbers were randomly assigned to each client and therefore do not represent any meaning. The following code can be used:

df = df.drop('ID')

Results:

```
In [6]: #drop ID column
        df = df.drop('ID')
        df.columns

Out[6]: ['Limit_Balance',
         'Sex',
         'Education',
         'Marraige',
         'Age',
         'Repayment_sept',
         'Repayment_Aug',
         'Repayment_July',
         'Repayment_June',
         'Repayment_may',
         'Repayment_Apr',
         'Bill_stat_Sept',
         'Bill_stat_Aug',
         'Bill_stat_July',
         'Bill_stat_June',
         'Bill_stat_may',
         'Bill_stat_Apr',
         'Paid_AMT_Sept',
         'Paid_AMT_Aug',
         'Paid_AMT_July',
         'Paid_AMT_June',
         'Paid_AMT_may',
         'Paid_AMT_Apr',
         'Default']
```

Code 3.1: Drop ID column

2.3.3.2 Data Types

The code is too long from this point to write out in word. Screen shots will be used to show the implementation of the code.

The data types had to be converted from strings to floats and integers. The categorical features were converted to integers and the continuous features were converted to floats. The following code was used:

```
In [7]: #convert data types to appropriate formats
        from pyspark.sql.types import *
        from pyspark.sql.functions import *

        df = df.withColumn('Limit_Balance', df['Limit_Balance'].cast(FloatType()))\
            .withColumn('Sex', df['Sex'].cast(IntegerType()))\
            .withColumn('Education', df['Education'].cast(IntegerType()))\
            .withColumn('Age', df['Age'].cast(IntegerType()))\
            .withColumn('Marraige', df['Marraige'].cast(IntegerType()))\
            .withColumn('Repayment_sept', df['Repayment_sept'].cast(IntegerType()))\
            .withColumn('Repayment_Aug', df['Repayment_Aug'].cast(IntegerType()))\
            .withColumn('Repayment_July', df['Repayment_July'].cast(IntegerType()))\
            .withColumn('Repayment_June', df['Repayment_June'].cast(IntegerType()))\
            .withColumn('Repayment_may', df['Repayment_may'].cast(IntegerType()))\
            .withColumn('Repayment_Apr', df['Repayment_Apr'].cast(IntegerType()))\
            .withColumn('Bill_stat_Sept', df['Bill_stat_Sept'].cast(FloatType()))\
            .withColumn('Bill_stat_Aug', df['Bill_stat_Aug'].cast(FloatType()))\
            .withColumn('Bill_stat_July', df['Bill_stat_July'].cast(FloatType()))\
            .withColumn('Bill_stat_June', df['Bill_stat_June'].cast(FloatType()))\
            .withColumn('Bill_stat_may', df['Bill_stat_may'].cast(FloatType()))\
            .withColumn('Bill_stat_Apr', df['Bill_stat_Apr'].cast(FloatType()))\
            .withColumn('Paid_AMT_Sept', df['Paid_AMT_Sept'].cast(FloatType()))\
            .withColumn('Paid_AMT_Aug', df['Paid_AMT_Aug'].cast(FloatType()))\
            .withColumn('Paid_AMT_July', df['Paid_AMT_July'].cast(FloatType()))\
            .withColumn('Paid_AMT_June', df['Paid_AMT_June'].cast(FloatType()))\
            .withColumn('Paid_AMT_may', df['Paid_AMT_may'].cast(FloatType()))\
            .withColumn('Paid_AMT_Apr', df['Paid_AMT_Apr'].cast(FloatType()))\
            .withColumn('Default', df['Default'].cast(IntegerType()))

        #Show data types
        df.dtypes

Out[7]: [('Limit_Balance', 'float'),
         ('Sex', 'int'),
         ('Education', 'int'),
         ('Marraige', 'int'),
         ('Age', 'int'),
         ('Repayment_sept', 'int'),
         ('Repayment_Aug', 'int'),
         ('Repayment_July', 'int'),
         ('Repayment_June', 'int'),
         ('Repayment_may', 'int'),
```

Code 3.2: Change data types

2.3.3.3 Unique Values

The unique values for each column was checked using a function. The values were put into separate PySpark data frames to make them easier to read. The following code illustrates this:

```
In [18]: #Check unique values
         def checkvalues():
             for i in df.columns:
                 o = df.select(i).distinct()
                 o.show()


         print(checkvalues())
```

Code 3.3: Check unique values

Results:

See Appendix 2

The columns Marriage, Education, and Repayment_Apr – Sept had unknown variables. The unknown values for each feature was as follows:

Marriage

- 0

Repayment_Apr – Sept

- 0
- -2

Education

- 5
- 6
- 0

To work out how to deal with these values the count for each value was taken using the following code:

```
In [10]: #How many rows do the 5,6,0s in the Education column take up.
         df.filter(df['Education'] == 5).count() + \
         df.filter(df['Education'] == 6).count() + \
         df.filter(df['Education'] == 0).count()

Out[10]: 345

In [11]: #percentage of unknown variables in education
         345/30000*100

Out[11]: 1.15

In [12]: #How many rows do the 0s in the Marraige column takeup.
         df.filter(df['Marraige'] == 0).count()

Out[12]: 54

In [13]: #percentage of unknown variables in Marraige
         54/30000*100

Out[13]: 0.18

In [14]: #function to see how many unknown variables are in repayment columns
         def unknownvariables(values):
             for i in df.columns[5:11]:
                 c = df.filter(df[i]==values).count()
                 print(i, 'number of' ,values, 'is', c)
                 if c >=0:
                     u = c/30000*100
                     print('Percentage of unknown variables for', i, 'is', u)
             return()

         print(unknownvariables(-2))
         print(unknownvariables(0))
```

Code 3.3: value counts and percentages for each unknown value

Results:

```
print(unknownvariables(-2))
print(unknownvariables(0))
```

```
Repayment_sept number of -2 is 2759
Percentage of unknown variables for Repayment_sept is 9.19666666666666
7
Repayment_Aug number of -2 is 3782
Percentage of unknown variables for Repayment_Aug is 12.60666666666666
6
Repayment_July number of -2 is 4085
Percentage of unknown variables for Repayment_July is 13.6166666666666
65
Repayment_June number of -2 is 4348
Percentage of unknown variables for Repayment_June is 14.4933333333333
32
Repayment_may number of -2 is 4546
Percentage of unknown variables for Repayment_may is 15.1533333333333
2
Repayment_Apr number of -2 is 4895
Percentage of unknown variables for Repayment_Apr is 16.3166666666666
6
()
Repayment_sept number of 0 is 14737
Percentage of unknown variables for Repayment_sept is 49.1233333333333
35
Repayment_Aug number of 0 is 15730
Percentage of unknown variables for Repayment_Aug is 52.43333333333333
Repayment_July number of 0 is 15764
Percentage of unknown variables for Repayment_July is 52.5466666666666
7
Repayment_June number of 0 is 16455
Percentage of unknown variables for Repayment_June is 54.85
Repayment_may number of 0 is 16947
Percentage of unknown variables for Repayment_may is 56.48999999999999
5
Repayment_Apr number of 0 is 16286
Percentage of unknown variables for Repayment_Apr is 54.28666666666667
()
```

2.3.3.5 Missing values/ Unknown Variables

There are three main solutions to dealing with unknown values. The first is, treat them as missing values and use an imputation method, deletion method or to delete the entire column. The second method is to learn the true meaning of the unknown values by contacting the original data collectors or finally ignore the unknown values.

For this study deletion and imputation will be used for the marriage and education columns. The repayment status columns will not be used for the visualizations; however, they will be used in the logistic regression models. Justifications for these decisions are written below.

The percentage values are less than 5% in the Marriage and Education column therefore these rows can be treated as missing values and removed as it will be of little consequence to the results (Dong and Peng, 2013). The missing values in these features can also be imputed using the most frequent value. This is because the values being imputed are integers.

The unknown values in the repayment columns made up over 50% of the total number of rows and thus could not be removed without substantially reducing the amount data to work with. Also, the meaning of the values could not be found online. Therefore, all the repayment columns were removed for the visualizations as it is impossible to infer results from unknown values because they carry no meaning. However, these columns could still be used for the logistic regression model. This is because, the values are unknown and not missing, which means they are unlikely to be random and thus can still be used to decipher a probability of a client defaulting or not via the logistic regression model.

The following code is used for the deletion method by dropping the unknown variables from the Marriage and Education columns:

```
In [28]: #drop unknowns from marraige and education
         df = df.filter(df.Education !=0)
         df = df.filter(df.Education !=5)
         df = df.filter(df.Education !=6)
         df = df.filter(df.Marraige !=0)
```

Code 3.4: Drop unwanted values

The following code is used to implement the most frequent imputation method for the unknown variables from the marriage and education columns:

```
In [15]: #impute missing values using most frequent
from sklearn.impute import SimpleImputer
import numpy as np

df = df.withColumn('Education', regexp_replace('Education', '5', '0'))
df = df.withColumn('Education', regexp_replace('Education', '6', '0'))
newdf = df.withColumn('Education', regexp_replace('Education', '0', 'np.nan'))
newdf = newdf.withColumn('Marraige', regexp_replace('Marraige' , '0', 'np.nan'))
imp_mean = SimpleImputer(missing_values='np.nan', strategy='most_frequent')
imp_mean.fit(newdf.toPandas())
imputed_df = imp_mean.transform(newdf.toPandas())

#convert numpy ndarray to pandas dataframe to
#convert it to pyspark dataframe
import pandas as pd

df_impute = pd.DataFrame(imputed_df, columns = ['Limit_Balance',
                                'Sex',
                                'Education',
                                'Marraige',
                                'Age',
                                'Repayment_sept',
                                'Repayment_Aug',
                                'Repayment_July',
                                'Repayment_June',
                                'Repayment_may',
                                'Repayment_Apr',
                                'Bill_stat_Sept',
                                'Bill_stat_Aug',
                                'Bill_stat_July',
                                'Bill_stat_June',
                                'Bill_stat_may',
                                'Bill_stat_Apr',
                                'Paid_AMT_Sept',
                                'Paid_AMT_Aug',
                                'Paid_AMT_July',
                                'Paid_AMT_June',
                                'Paid_AMT_may',
                                'Paid_AMT_Apr',
                                'Default'])

df_2 = spark.createDataFrame(df_impute)
print('Education variables' ,df_2.toPandas()['Education'].unique())
print('Marraige variables' ,df_2.toPandas()['Marraige'].unique())
```

Code 3.5: Impute missing values

Results:

```
Education variables ['2' '1' '3' '4']
Marraige variables ['1' '2' '3']
```

The following code removes the repayment columns:

```
In [17]: #drop repayment columns (too many unexplained rows)

aa = df.columns[5:11]
for i in aa:
    df = df.drop(i)

df.columns
```

Code 3.6: Drop unwanted columns

Results:

```
Out[17]: ['Limit_Balance',
          'Sex',
          'Education',
          'Marraige',
          'Age',
          'Bill_stat_Sept',
          'Bill_stat_Aug',
          'Bill_stat_July',
          'Bill_stat_June',
          'Bill_stat_may',
          'Bill_stat_Apr',
          'Paid_AMT_Sept',
          'Paid_AMT_Aug',
          'Paid_AMT_July',
          'Paid_AMT_June',
          'Paid_AMT_may',
          'Paid_AMT_Apr',
          'Default']
```

3. Analysis

3.1 Univariate analysis:

For the rest of the investigation the results are collected using the data set where the deletion method was used, not the imputation method.

The descriptive statistics including count, mean, standard deviation and minimum and maximum values was collected for each column and printed into separate tables. If the information was not split into separate tables the column names would run over to the next line, making it difficult to see the information. Therefore, to make it easier to see the descriptive information, separate tables were produced using a function. The following code was used:

```
[71]: #summary table of each column

def descriptives(columns):
    for i in columns:
        df.describe(i).show()

print(descriptives(df.columns))
```

Code 4.1: summary tables

Results:

```
+-------+------------------+
|summary|     Limit_Balance|
+-------+------------------+
|  count|             29601|
|   mean|167550.54491402316|
| stddev|129944.02095269752|
|    min|           10000.0|
|    max|         1000000.0|
+-------+------------------+
```

```
+-------+------------------+
|summary|               Sex|
+-------+------------------+
|  count|             29601|
|   mean| 1.603189081449951|
| stddev|0.4892444171231663|
|    min|                 1|
|    max|                 2|
+-------+------------------+
```

```
+-------+------------------+
|summary|         Education|
+-------+------------------+
|  count|             29601|
|   mean|1.8154792067835546|
| stddev|0.7103992840785517|
|    min|                 1|
|    max|                 4|
+-------+------------------+
```

```
+-------+------------------+
|summary|          Marraige|
+-------+------------------+
|  count|             29601|
|   mean|1.5554542076281206|
| stddev| 0.518092327098095|
|    min|                 1|
|    max|                 3|
+-------+------------------+
```

```
+-------+------------------+
|summary|    Bill_stat_Sept|
+-------+------------------+
|  count|             29601|
|   mean|50957.432012432015|
| stddev| 73370.24240377411|
|    min|         -165580.0|
|    max|          964511.0|
+-------+------------------+
```

summary	Age
count	29601
mean	35.46407215972433
stddev	9.213243333797596
min	21
max	79

summary	Bill_stat_Aug
count	29601
mean	48942.18955440695
stddev	70923.98515063159
min	-69777.0
max	983931.0

summary	Bill_stat_July
count	29601
mean	46803.20327015979
stddev	69123.892106309
min	-157264.0
max	1664089.0

summary	Bill_stat_June
count	29601
mean	43122.55420424986
stddev	64196.383913109705
min	-170000.0
max	891586.0

summary	Bill_stat_may
count	29601
mean	40235.54518428432
stddev	60699.34488357182
min	-81334.0
max	927171.0

summary	Bill_stat_Apr
count	29601
mean	38858.449815884596
stddev	59519.893042828335
min	-339603.0
max	961664.0

summary	Paid_AMT_Sept
count	29601
mean	5649.560318908145
stddev	16568.26494144611
min	0.0
max	873552.0

summary	Paid_AMT_Aug
count	29601
mean	5894.788385527516
stddev	23089.193621490787
min	0.0
max	1684259.0

summary	Paid_AMT_July
count	29601
mean	5198.415898111551
stddev	17580.914805811994
min	0.0
max	896040.0

summary	Paid_AMT_June
count	29601
mean	4828.659268267964
stddev	15711.05799234724
min	0.0
max	621000.0

summary	Paid_AMT_may
count	29601
mean	4795.0327353805615
stddev	15244.217154322845
min	0.0
max	426529.0

summary	Paid_AMT_Apr
count	29601
mean	5181.326374108983
stddev	17657.2607390526
min	0.0
max	528666.0

summary	Default
count	29601
mean	0.2231343535691362
stddev	0.4163547406844325
min	0
max	1

Plots were produced using matplotlib to illustrate the counts of each categorical feature investigated in this study. The counts of the ages in the age feature was also plotted. Pie charts and Bar charts were used.

The percentage of clients who did default and did not default was plotted on a pie chart using the following code:

```
In [98]: import numpy as np
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.axis('equal')
cat = ['Yes Default', 'No Default']
clients = [6605, 22996]
ax.pie(clients, labels = cat,autopct='%1.2f%%')
plt.title ('Client distribution amongst default classes')
plt.show()
```

Code 4.1: Pie chart plot

Results:

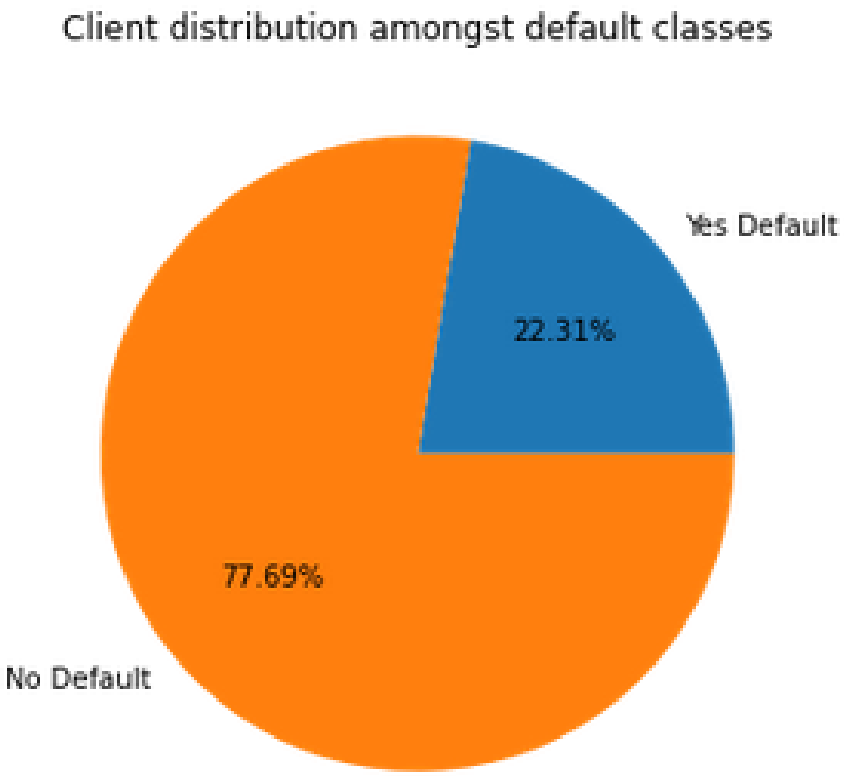


Image 1: Pie chart to show percentage of clients who did default and did not default

The following code was used to plot the frequency tables of each categorical feature and age:

In [22]:
```
#distribution plot by count for features, sex, age, education and marraige
import matplotlib.pyplot as plt
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

def plotstats(columns):
    plt.figure()
    for i in columns:
        u = df.toPandas()[i].unique()
        if len(u)==2:
            T = df.groupBy(i).count().toPandas()
            plt.figure()

            x = T[i]
            y = T["count"]
            plt.bar(x,y , color=['orange','blue'])
            plt.xticks([2,1],['Female', 'Male'])
            plt.xlabel(i)
            plt.ylabel("Count")
            plt.title("Male and Female count")
            plt.show()
        if len(u)==4:
            T = df.groupBy(i).count().toPandas()
            plt.figure()

            x = T[i]
            y = T["count"]
            plt.bar(x,y)
            plt.xticks([0,1,2,3],['High School', 'Graduate School','Others', 'University school'])
            plt.xlabel(i)
            plt.ylabel("Count")
            plt.title("Education class count")
            plt.show()
        if len(u)>10:
            T = df.groupBy(i).count().toPandas()
            plt.figure()
            x = T[i]
            y = T["count"]
            plt.bar(x,y)
            plt.xlabel(i)
            plt.ylabel("Count")
            plt.title("Age frequency count")
            plt.show()
        if len(u)==3:
            T = df.groupBy(i).count().toPandas()
            plt.figure()
            x = T[i]
            y = T["count"]
            plt.bar(x,y)
            plt.xticks([1,2,3],['Married', 'Single','Others'])
            plt.xlabel(i)
            plt.ylabel("Count")
            plt.title("Marraige class count")
            plt.show()

print(plotstats(df.columns[1:5]))
```

Code 4.2: Bar chart plots

Plots:

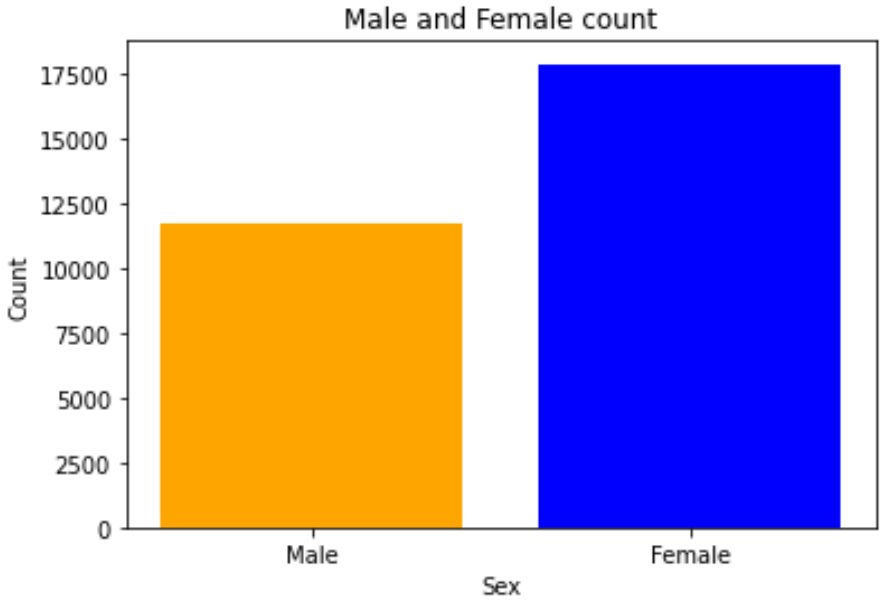


Image 2.1: Bar plot of sex count

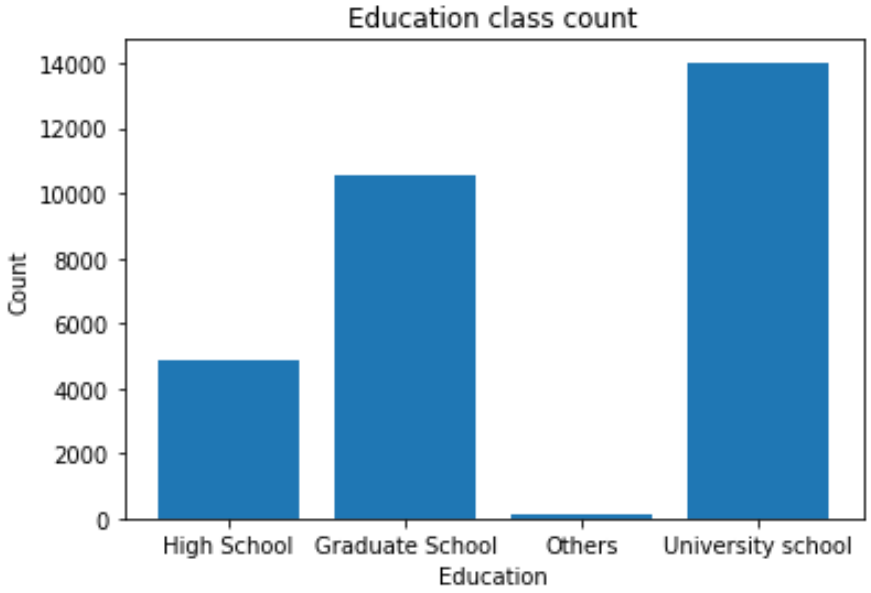


Image 2.2: Bar plot of education count

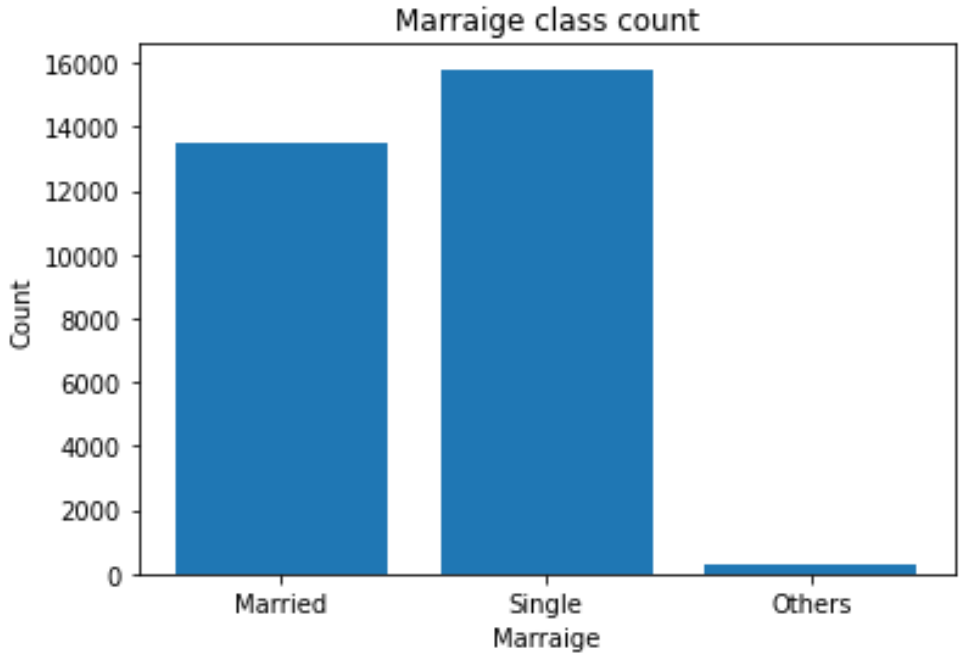


Image 2.3: Bar plot of relationship count

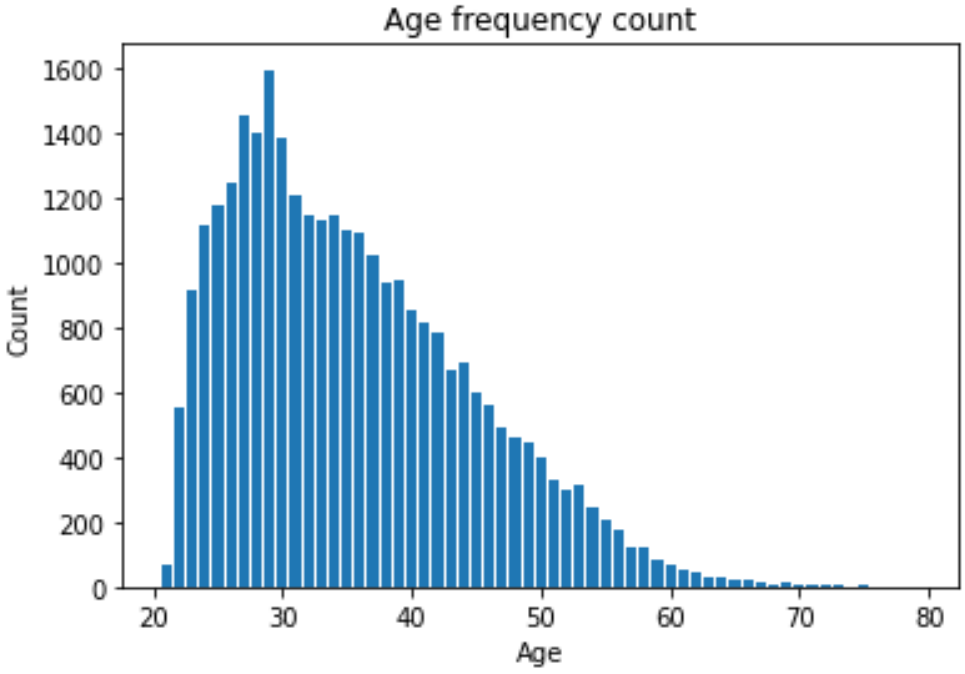


Image 2.4: Bar plot of age frequency

Matplotlib was used to produce box plots which identify the median and outliers in the features Bill_AMT_Apr- Sept and Paid_stat_Apr - Sept . The following code was used:

```
In [24]: #Box plot with the Bill satement columns
         plt.figure()
         data=df['Bill_Stat_sept', 'Bill_stat_Aug','Bill_stat_July',
                 'Bill_stat_June',
                 'Bill_stat_Apr', 'Bill_stat_may'].toPandas()
         fig1, ax1 = plt.subplots()
         ax1.boxplot(data)
         plt.title('''Box plot comparing median, upper quartile and lower quatile
                   for bill statments from April to Septmeber''')
         plt.xticks([1,2,3,4,5,6], ['September', 'August','July',
                 'June',
                 'April', 'May'], rotation = 90)
         plt.xlabel('Month')
         plt.ylabel('Bill statment amount in NT dollar')
         plt.show()
```

Code 4.3: Box plots

Results:

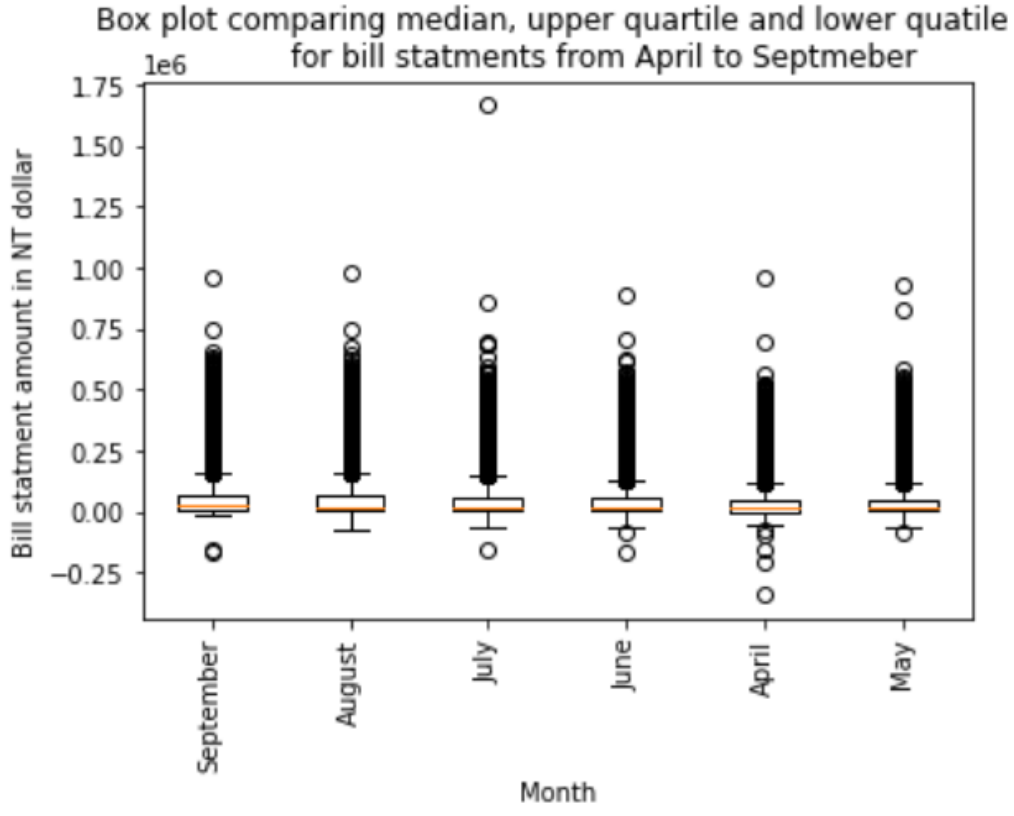


Image 3.1: Box plot for bill statements per month from April to September

Code:

```
In [25]: #box plot for paid amount columns
plt.figure()
data=df['Paid_AMT_Sept',
 'Paid_AMT_Aug',
 'Paid_AMT_July',
 'Paid_AMT_June',
 'Paid_AMT_may',
 'Paid_AMT_Apr'].toPandas()
fig1, ax1 = plt.subplots()
ax1.boxplot(data)
plt.title('''Box plot comparing median, upper quartile and lower quatile
          for paid amounts per month from April to Septmeber''')
plt.xlabel('Month')
plt.ylabel('Paid amount in NT dollar')
plt.xticks([1,2,3,4,5,6], ['September', 'August','July',
        'June',
        'April', 'May'], rotation = 90)
plt.show()
```

Code 4.4: Box plots

Results:

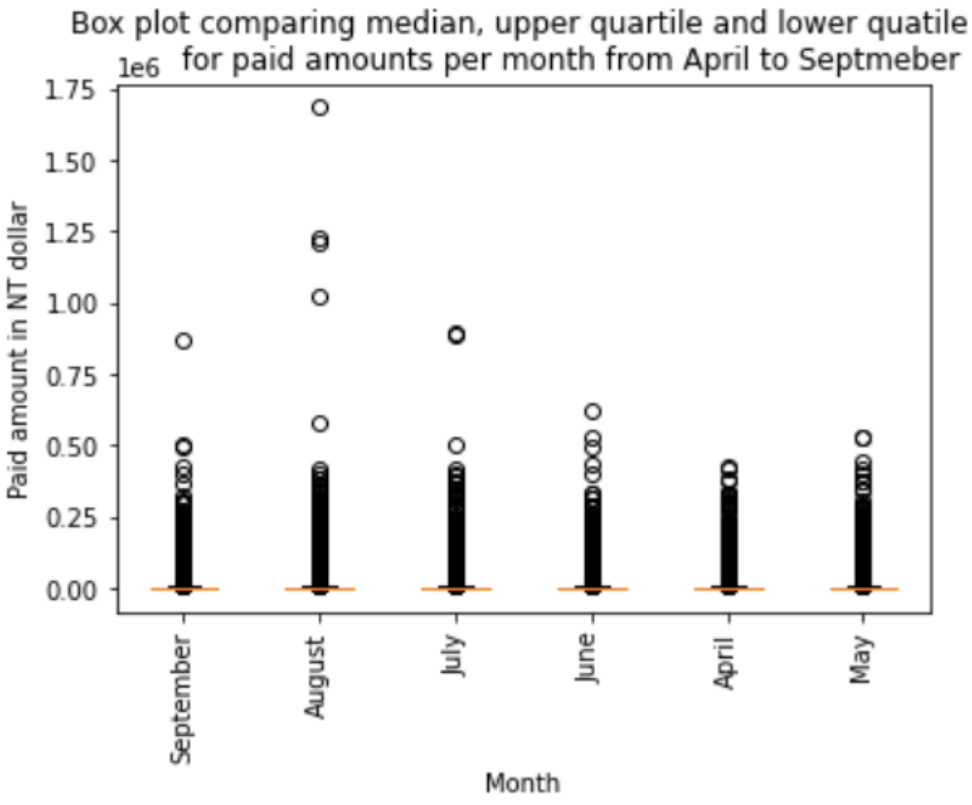


Image 3.2: Box plot for paid amounts per month from April to September

A box plot was also produced for Limit Balance. The following code was used:

```
In [38]: #box plot for limit balance
         plt.figure()
         data=df.toPandas()['Limit_Balance']
         fig1, ax1 = plt.subplots()
         ax1.boxplot(data)
         plt.title('''Box plot to show the mean, upper quartile and lower quatile of the
                    Limit Balance feature''')
         plt.ylabel('Limit Balance in NT dollar')
         plt.xticks([1],['Limit Balance'])
         plt.show()
```

Code 4.5: Box plot

Results:

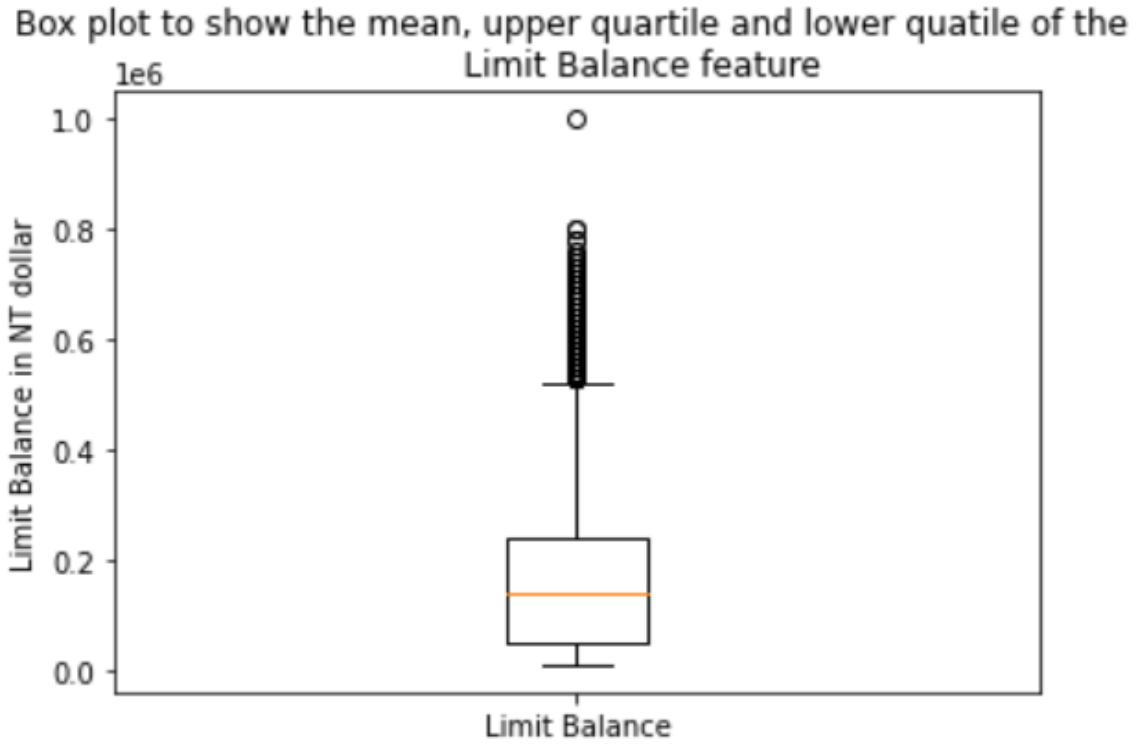


Image 4: Box plot for Limit Balance.

3.2 Multivariate analysis

Stacked bar charts comparing credit card defaults with marriage, sex, and education categorical features.

The following code builds the plots in matplotlib:

In [38]:
```
#grouped barplot of male and female count that defaulted

df_i = df_ci.groupBy('Sex','Default').count().toPandas()

plt.figure()
df_i.pivot(index='Default', columns = 'Sex', values = 'count').plot(kind='bar')
plt.xticks([0,1],['No Default', 'Yes Default'])
plt.xlabel("Default")
plt.ylabel("Count")
plt.title("Male and Female Default count")
labels = ('Female', 'Male')
plt.legend(labels)
plt.show()


df_i = df_ci.groupBy('Marraige','Default').count().toPandas()

plt.figure()
df_i.pivot(index='Default', columns = 'Marraige', values = 'count').plot(kind='bar')
plt.xticks([0,1],['No Default', 'Yes Default'])
plt.xlabel("Default")
plt.ylabel("Count")
plt.title("Marraige Default class count")
labels = ('Married', 'Single', 'Others')
plt.legend(labels)
plt.show()


df_i = df_ci.groupBy('Education','Default').count().toPandas()

plt.figure()
df_i.pivot(index='Default', columns = 'Education', values = 'count').plot(kind='bar')
plt.xticks([0,1],['No Default', 'Yes Default'])
plt.xlabel("Default")
plt.ylabel("Count")
plt.title("Education Default class count")
labels = ('Graduate school', 'University', 'High school', 'Others')
plt.legend(labels)
plt.show()
```

Code 5.1: Grouped bar charts

Results:

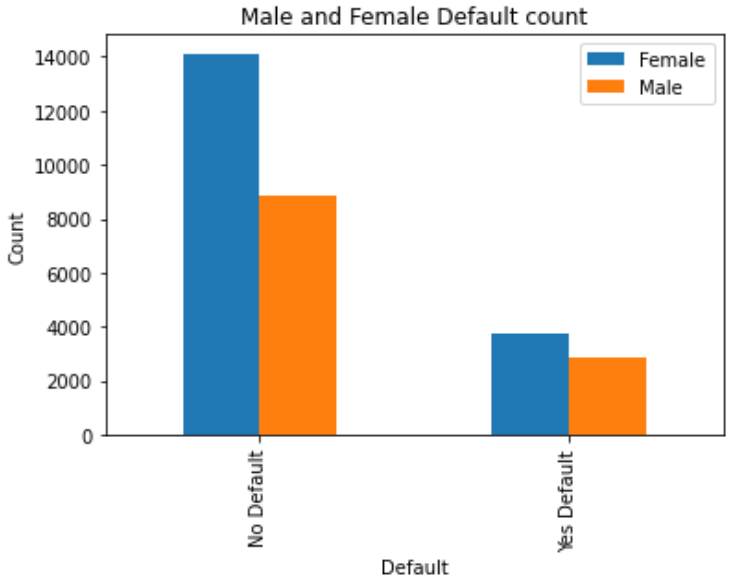


Image 5.1: Grouped bar plot comparing sex count with default class

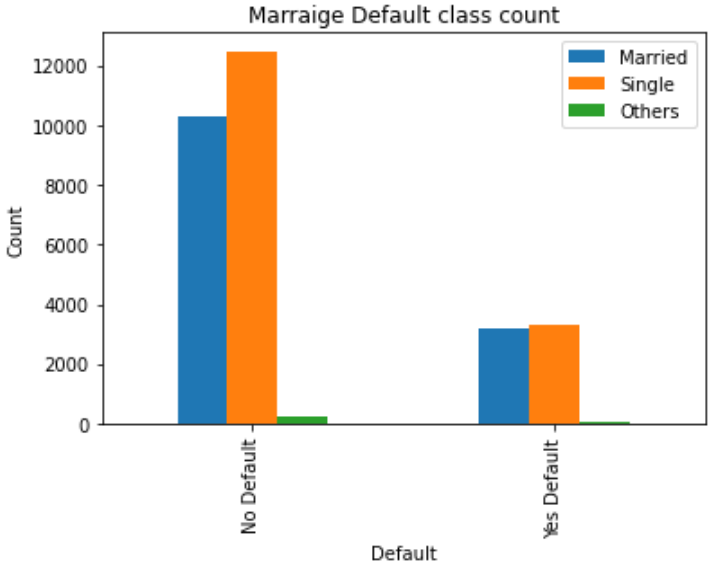


Image 5.2: Grouped bar plot comparing marriage count with default class

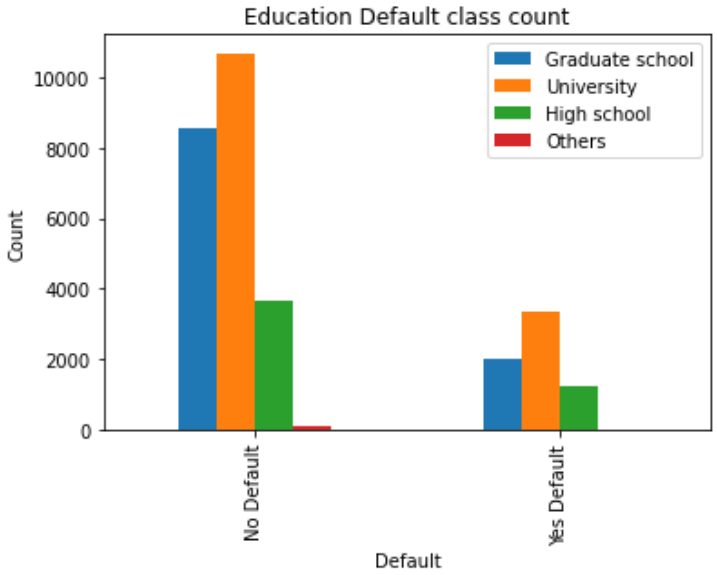


Image 5.3: Grouped bar plot comparing education count with default class

The percentages of each category population who did and did not default were plotted. Using the following code:

In [43]:

```
#Bar plot for proportion of Sex in each default class
df5 = spark.createDataFrame([
                             ('Male', 0, 75.64277200749191),
                             ('Male', 1, 24.357227992508086),
                             ('Female', 0, 79.03108373004761),
                             ('Female', 1, 20.968916269952395)],
                            ['Sex','Default','percentage'])




plt.figure()
df5=df5.toPandas()
df5.pivot(index='Default', columns = 'Sex', values = 'percentage').plot(kind='bar')
plt.xticks([0,1],['No Default', 'Yes Default'])
plt.xlabel("Default")
plt.ylabel("percentage")
plt.title("Male and Female Default percentage")
plt.show()


#Bar plot for proportion of marraige in each default class


df5 = spark.createDataFrame([
                             ('Single', 0, 78.93837783120334),
                             ('Single', 1, 21.06162216879666),
                             ('Married', 0, 76.3152036803443),
                             ('Married', 1, 23.68479631965571),
                             ('Other',   1, 26.41509433962264),
                             ('Other',    0,  73.58490566037736)],
                            ['Marriage','Default','percentage'])

plt.figure()
df5=df5.toPandas()
df5.pivot(index='Default', columns = 'Marriage', values = 'percentage').plot(kind='bar')
plt.xticks([0,1],['No Default', 'Yes Default'])
plt.xlabel("Default")
plt.ylabel("percentage")
plt.title("'The percentage of each Marriage catagory population who did default and did not default")
plt.show()

#Bar plot for proportion of education in each default class

df5 = spark.createDataFrame([
                             ('University', 0, 76.26212207644039),
                             ('University', 1, 23.737877923559612),
                             ('Graduate School', 0, 80.75796238540781),
                             ('Graduate School', 1, 19.242037614592192),
                             ('High School',   0, 74.69731171762774),
                             ('High School',    1,  25.302688282372255),
                             ('Other',   0, 94.3089430894309),
                             ('Other',    1,  5.691056910569106)],
                            ['Education','Default','percentage'])

plt.figure()
df5=df5.toPandas()
df5.pivot(index='Default', columns = 'Education', values = 'percentage').plot(kind='bar')
plt.xticks([0,1],['No Default', 'Yes Default'])
plt.xlabel("Default")
plt.ylabel("percentage")
plt.title("'The percentage of each Education catagory population who did default and did not default")
plt.show()
```

Code 5.2: Grouped bar charts

To see how the percentages were calculated please see appendix 3.

Results:

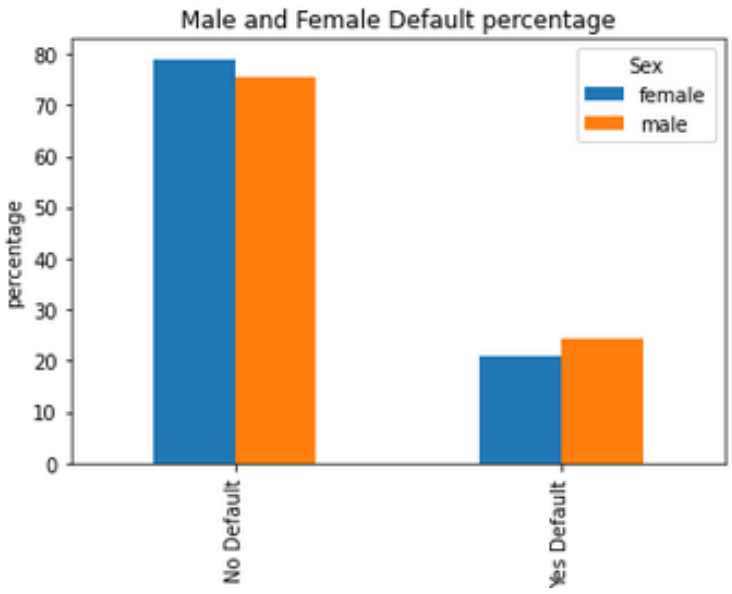


Image 6: bar plot to show percentage proportion of each sex category who defaulted

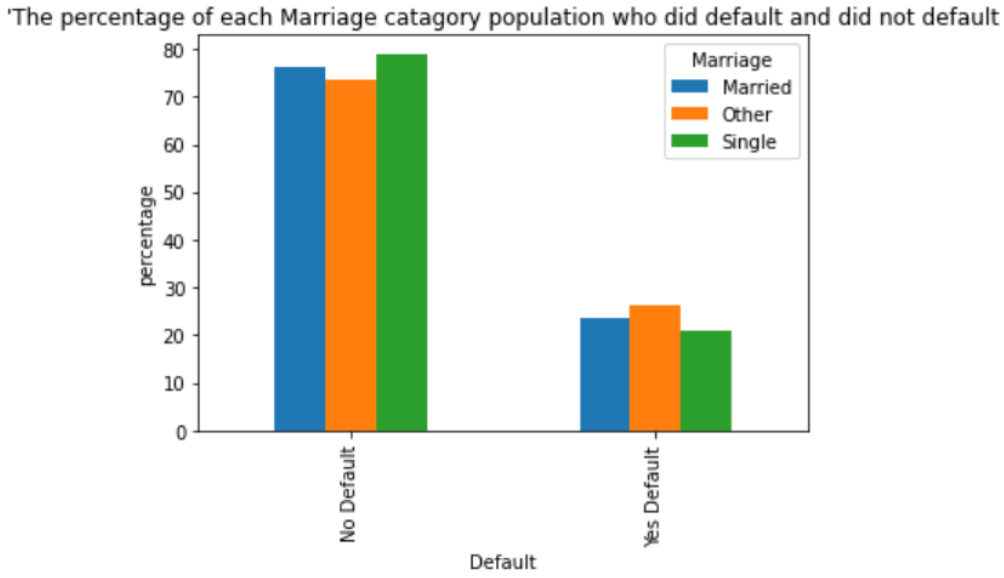


Image 6.1: bar plot to show percentage proportion of each marriage category who defaulted

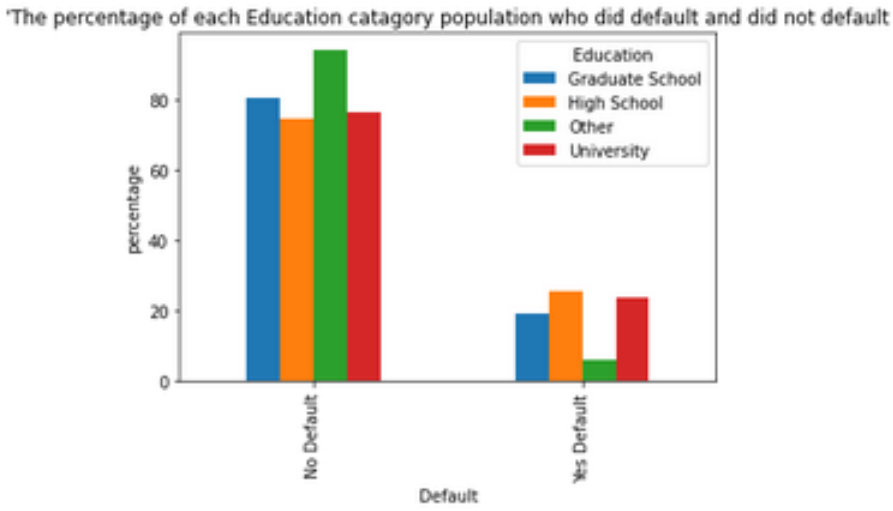


Image 6.2: Bar plot to show percentage proportion of each Education class who defaulted

To compare the continuous data with each default class the mean average of each continuous feature was calculated. See the following code:

```
In [*]: df_HB = df2.groupBy('Bill_stat_Sept', 'Bill_stat_Aug','Bill_stat_July',
                'Bill_stat_June',
                'Bill_stat_Apr', 'Bill_stat_may'
                            ,'Paid_AMT_Sept',
         'Paid_AMT_Aug',
         'Paid_AMT_July',
         'Paid_AMT_June',
         'Paid_AMT_may',
         'Paid_AMT_Apr','Limit_Balance','Default').count()

        df_HB
        def average(columns):
            for i in columns:
                df_HB.groupBy("Default").mean(i).show()

        print(average(df_HB.columns[0:13]))
```

Code 5.3: calculating mean values per column

The results from the tables produced from the previous code was then inputted to form a new data frame for each continuous feature. This code was then used to produce line plots and a horizontal bar plot. See the following code:

In [162]:
```
df5 = spark.createDataFrame([
                              ('April', 0, 39877.026526259666),
                              ('April', 1, 40221.0662314918),
                              ('May', 0, 41317.0564738292),
                              ('May', 1, 41550.823913389584),
                              ('June', 0, 44392.99844485915),
                              ('June', 1, 44134.89921986945),
                              ('July', 0, 48331.45778903404),
                              ('July', 1, 44134.89921986945),
                              ('August', 0, 50538.47134097574),
                              ('August', 1, 49540.24677599109),
                              ('September', 0, 52803.34261974585),
                              ('September', 1, 50765.21827734437)],
                             ['Month','Default','mean'])
df5.show()


aa = df5.filter(df5['Default']==1).toPandas()
ab = df5.filter(df5['Default']==0).toPandas()


# multiple line plots
plt.plot( 'Month', 'mean', data=aa, marker='o', markerfacecolor='blue', markersize=12, color='skyblue', linewidth=4)
plt.plot( 'Month', 'mean', data=ab, marker='o', markerfacecolor='Red', markersize=12, color='Green', linewidth=4)

plt.xlabel("Month")
plt.ylabel("Mean")
# show legend
plt.legend(['Yes Default', 'No Default'])

plt.title("Mean bill statment amount for each default class per month")
# show graph
plt.show()


df5.plot.line()
```

In [157]:
```
df5 = spark.createDataFrame([
                              ('April', 0, 5812.764373944726),
                              ('April', 1, 3588.167011622353),
                              ('May', 0, 5367.004976450724),
                              ('May', 1, 3364.693679350422),
                              ('June', 0, 5423.739136230339),
                              ('June', 1, 3319.806877885687),
                              ('July', 0, 5852.874477917),
                              ('July', 1, 3524.4588441331),
                              ('August', 0, 6759.728339109571),
                              ('August', 1, 3556.935679032001),
                              ('September', 0, 6442.003821203235),
                              ('September', 1, 3538.7648463620444)],
                             ['Month','Default','mean'])

aa = df5.filter(df5['Default']==1).toPandas()
ab = df5.filter(df5['Default']==0).toPandas()



# multiple line plots
plt.plot( 'Month', 'mean', data=aa, marker='o', markerfacecolor='blue', markersize=12, color='skyblue', linewidth=4)
plt.plot( 'Month', 'mean', data=ab, marker='o', markerfacecolor='Red', markersize=12, color='Green', linewidth=4)

plt.xlabel("Month")
plt.ylabel("Mean")
# show legend
plt.legend(['Yes Default', 'No Default'])

plt.title("Mean paid statment amount for each default class per month")
# show graph
plt.show()
```

In [52]:
```

x = ['Deafult', 'No Default']
y = [126253.6137667304, 178002.31131656148]

plt.xlabel("Limit Balance amount")
plt.ylabel("Default classes")
plt.title("Horizontal bar plot to show the difference in mean Limit Balance for each class of default")
plt.barh(x,y)
plt.show()
```

```
In [92]: import pyspark.sql.functions as f
         from pyspark.sql.window import Window

         cc = df.groupBy("Age","Default").count().show()

         cc

         uu = df.filter(df['Default']==1).groupBy("Age", 'Default').mean('Age').toPandas()

         uu = spark.createDataFrame(uu)
         uu.show()
         ii = df.filter(df['Default']==0).groupBy("Age", 'Default').mean('Age').toPandas()


         ii = spark.createDataFrame(ii)
         ii.show()


         ii.groupBy("Default").mean('Age').show()
         uu.groupBy("Default").mean('Age').show()

         ll = spark.createDataFrame([
                                     ('Age', 1, 47.075471698113205),
                                     ('Age', 0, 48.55357142857143)],
                                    ['Age','Default','Mean'])

         plt.figure()
         ll = ll.toPandas()
         ll.pivot(index='Default', columns = 'Age', values = 'Mean').plot(kind='bar')
         plt.xticks([0,1],['No Default', 'Yes Default'])
         plt.xlabel("Default")
         plt.ylabel("Mean Age")
         plt.title("Mean Age per Default Class")
         plt.show()
```

Code 5.4: Line plots, bar plots and horizontal bar plots

To see how the mean values were calculated please see appendix 4.

Results:

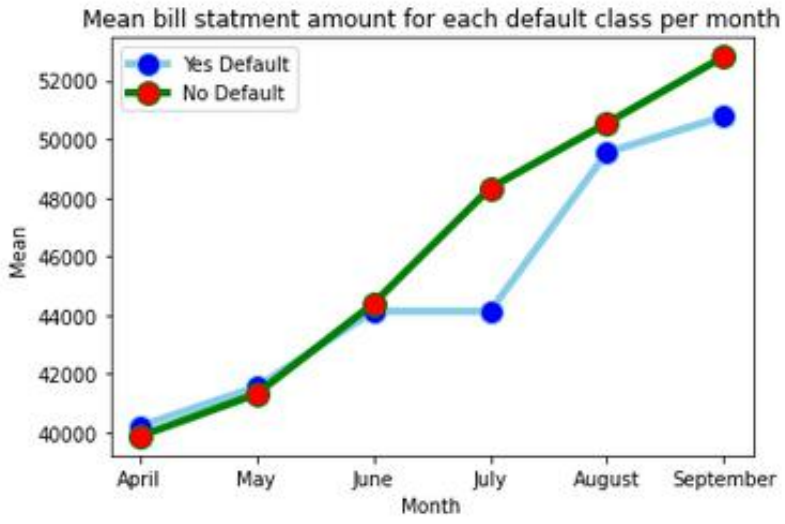


Image 7: Line plot comparing mean bill amount per month between each default class

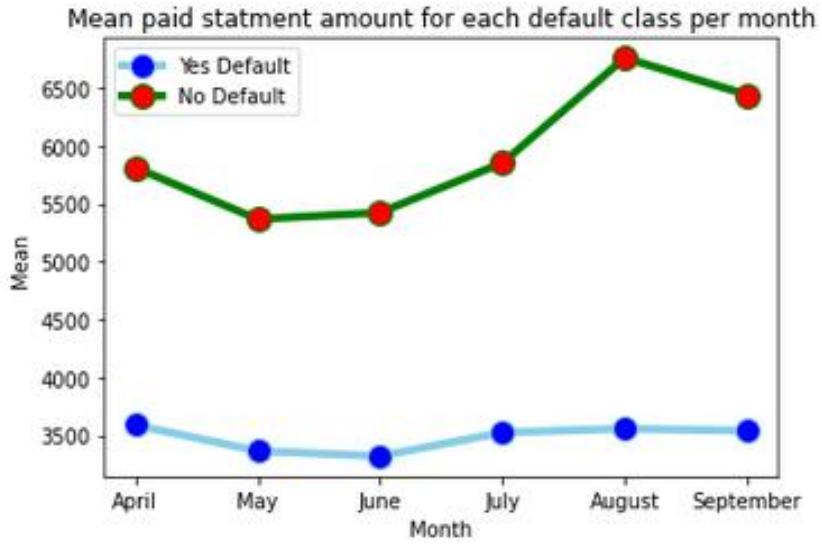


Image 7.1: Line plot comparing mean paid amount per month between each default class

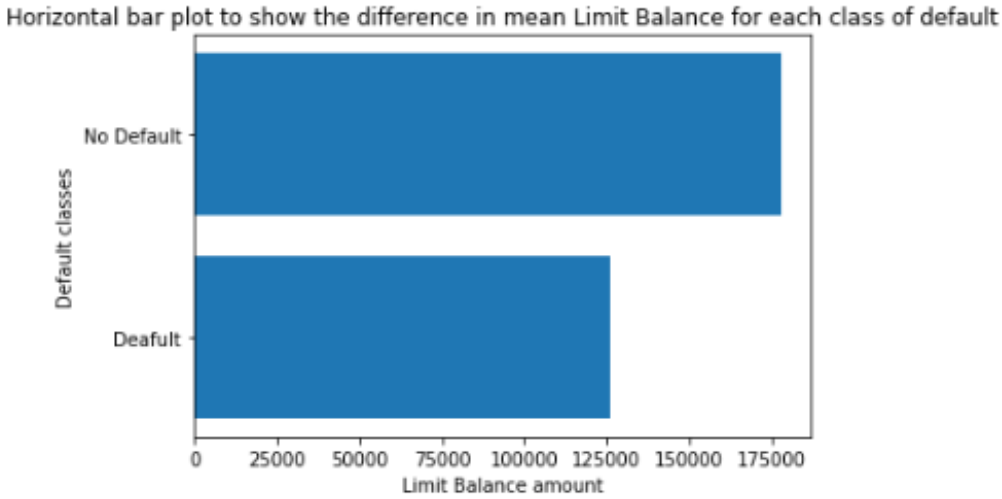


Image 7.2: Horizontal bar plot comparing mean limit balance between each default class

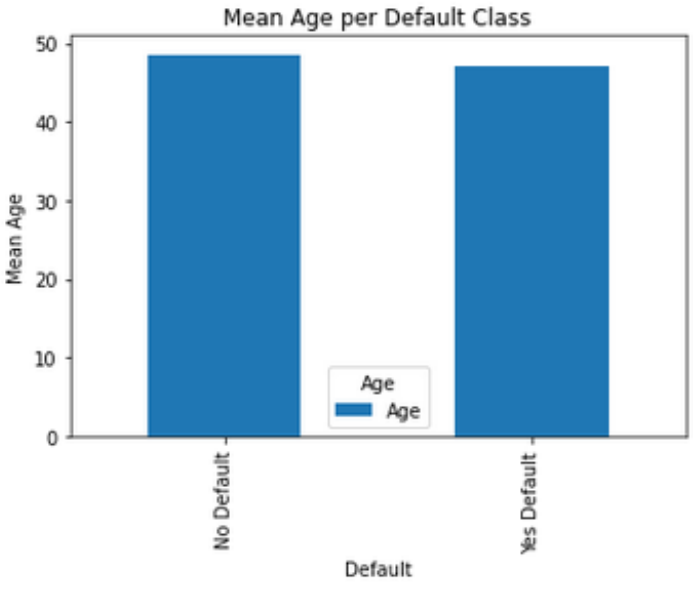


Image 7.3: bar plot comparing the mean age between each default class

3.3 Multicollinearity issues

Before the logistic regression model could be implemented, multicollinearity issues had to be addressed. Judging by the name of the columns, there is a potential for multicollinearity problems to exist between the monthly payment columns (Paid_AMT_Apr – sept) and the monthly bill amount columns (Bill_stat_April – Sept). The following code was used to identify any multicollinearity issues:

```
In [37]: from pyspark.mllib.stat import Statistics
         import pandas as pd

         #correlation matrix
         corr_data = df.select(df.columns)

         colnames = corr_data.columns
         features = corr_data.rdd.map(lambda row: row[0:])
         corrmatrix=Statistics.corr(features, method="pearson")
         corrdf1 = pd.DataFrame(corrmatrix)
         corrdf1.index, corrdf1.columns = colnames, colnames

         print(corrdf1.to_string())

         #easier to view
         corrdf1
```

Code 6.1: Correlation matrix

Results:

```
                Limit_Balance       Sex  Education  Marraige       Age
Bill_stat_Sept  Bill_stat_Aug  Bill_stat_July  Bill_stat_June  Bill_st
at_may  Bill_stat_Apr  Paid_AMT_Sept  Paid_AMT_Aug  Paid_AMT_July  Pai
d_AMT_June  Paid_AMT_may  Paid_AMT_Apr   Default
Limit_Balance        1.000000 -0.024817  -0.244039 -0.109756  0.144214
0.283695       0.276673        0.281738        0.293442       0.295316
0.289701       0.195666       0.177669       0.210523       0.203263
0.217251      0.219676 -0.154357
Sex                 -0.024817  1.000000  -0.012016  0.030073  0.091439
0.034193       0.031562        0.025359        0.022611       0.017774
0.017685      -0.000381       0.001413       0.008779       0.001966
0.001823      0.001971  0.039815
Education           -0.244039 -0.012016   1.000000 -0.147977  0.187425
0.005120       0.001250       -0.003499       -0.014428      -0.018165
-0.015190      -0.045972      -0.038628       -0.051966      -0.043094
-0.049469     -0.053670  0.049087
Marraige            -0.109756  0.030073  -0.147977  1.000000 -0.418284
-0.024971      -0.022505       -0.026171       -0.023902      -0.02614
8      -0.022198       -0.005120      -0.008058       -0.002921       -0.0
13957     -0.002882      -0.006024 -0.026903
Age                 0.144214  0.091439   0.187425 -0.418284  1.000000
0.054704       0.052385        0.051839        0.049839       0.048323
0.046712       0.025440       0.022402       0.029478       0.021567
0.021511      0.019139  0.014424
Bill_stat_Sept       0.283695  0.034193   0.005120 -0.024971  0.054704
1.000000       0.951255        0.891886        0.861238       0.831406
0.804834       0.140489       0.098947       0.156813       0.157643
0.165041      0.175798 -0.019303
Bill_stat_Aug        0.276673  0.031562   0.001250 -0.022505  0.052385
0.951255       1.000000        0.927801        0.893418       0.861465
0.833846       0.280903       0.100613       0.151052       0.146479
0.155457      0.170718 -0.013710
Bill_stat_July       0.281738  0.025359  -0.003499 -0.026171  0.051839
0.891886       0.927801        1.000000        0.925094       0.885669
0.855571       0.244564       0.318039       0.131112       0.142468
0.177549      0.179539 -0.013494
Bill_stat_June       0.293442  0.022611  -0.014428 -0.023902  0.049839
0.861238       0.893418        0.925094        1.000000       0.940458
0.902348       0.233241       0.208175       0.300270       0.129010
0.160036      0.175067 -0.009474
Bill_stat_may        0.295316  0.017774  -0.018165 -0.026148  0.048323
0.831406       0.861465        0.885669        0.940458       1.000000
0.947211       0.218717       0.181988       0.252535       0.293204
0.140896      0.162081 -0.006226
Bill_stat_Apr        0.289701  0.017685  -0.015190 -0.022198  0.046712
0.804834       0.833846        0.855571        0.902348       0.947211
1.000000       0.202036       0.173825       0.234608       0.250144
0.307504      0.115525 -0.005339
```

To present the data in a more readable format the table was printed as a Pandas data frame.

Results:

57]:

Bill_stat_Sept	Bill_stat_Aug	Bill_stat_July	Bill_stat_June	Bill_stat_may	Bill_stat_Apr	Paid_AMT_Sept	Paid_AMT_Aug	Paid_AMT_July	Paid_AMT_June	Pa
0.283695	0.276673	0.281738	0.293442	0.295316	0.289701	0.195666	0.177669	0.210523	0.203263	
0.034193	0.031562	0.025359	0.022611	0.017774	0.017685	-0.000381	0.001413	0.008779	0.001966	
-0.024971	-0.022505	-0.026171	-0.023902	-0.026148	-0.022198	-0.005120	-0.008058	-0.002921	-0.013957	
0.054704	0.052385	0.051839	0.049839	0.048323	0.046712	0.025440	0.022402	0.029478	0.021567	
1.000000	0.951255	0.891886	0.861238	0.831406	0.804834	0.140489	0.098947	0.156813	0.157643	
0.951255	1.000000	0.927801	0.893418	0.861465	0.833846	0.280903	0.100613	0.151052	0.146479	
0.891886	0.927801	1.000000	0.925094	0.885669	0.855571	0.244564	0.318039	0.131112	0.142468	
0.861238	0.893418	0.925094	1.000000	0.940458	0.902348	0.233241	0.208175	0.300270	0.129010	
0.831406	0.861465	0.885669	0.940458	1.000000	0.947211	0.218717	0.181988	0.252535	0.293204	
0.804834	0.833846	0.855571	0.902348	0.947211	1.000000	0.202036	0.173825	0.234608	0.250144	
0.140489	0.280903	0.244564	0.233241	0.218717	0.202036	1.000000	0.286741	0.253683	0.200094	
0.098947	0.100613	0.318039	0.208175	0.181988	0.173825	0.286741	1.000000	0.246086	0.179612	
0.156813	0.151052	0.131112	0.300270	0.252535	0.234608	0.253683	0.246086	1.000000	0.215711	
0.157643	0.146479	0.142468	0.129010	0.293204	0.250144	0.200094	0.179612	0.215711	1.000000	
0.165041	0.155457	0.177549	0.160036	0.140896	0.307504	0.149749	0.182296	0.160705	0.151510	
0.175798	0.170718	0.179539	0.175067	0.162081	0.115525	0.186283	0.157779	0.160548	0.157001	
-0.019303	-0.013710	-0.013494	-0.009474	-0.006226	-0.005339	-0.073881	-0.058307	-0.056288	-0.057012	
0.940238	0.961307	0.962125	0.964906	0.952450	0.930016	0.230766	0.187803	0.210808	0.192118	

There is a high correlation between the features, bill statements (Bill_stat_Apr – Sept) with each other, as can be seen in the red square. Multicollinearity needs to be addressed because it can weaken the statistical power of the regression model because the precision of the estimate coefficients is reduced. Additionally, the p-values which are used to determine which independent variables are significant might not be as accurate.

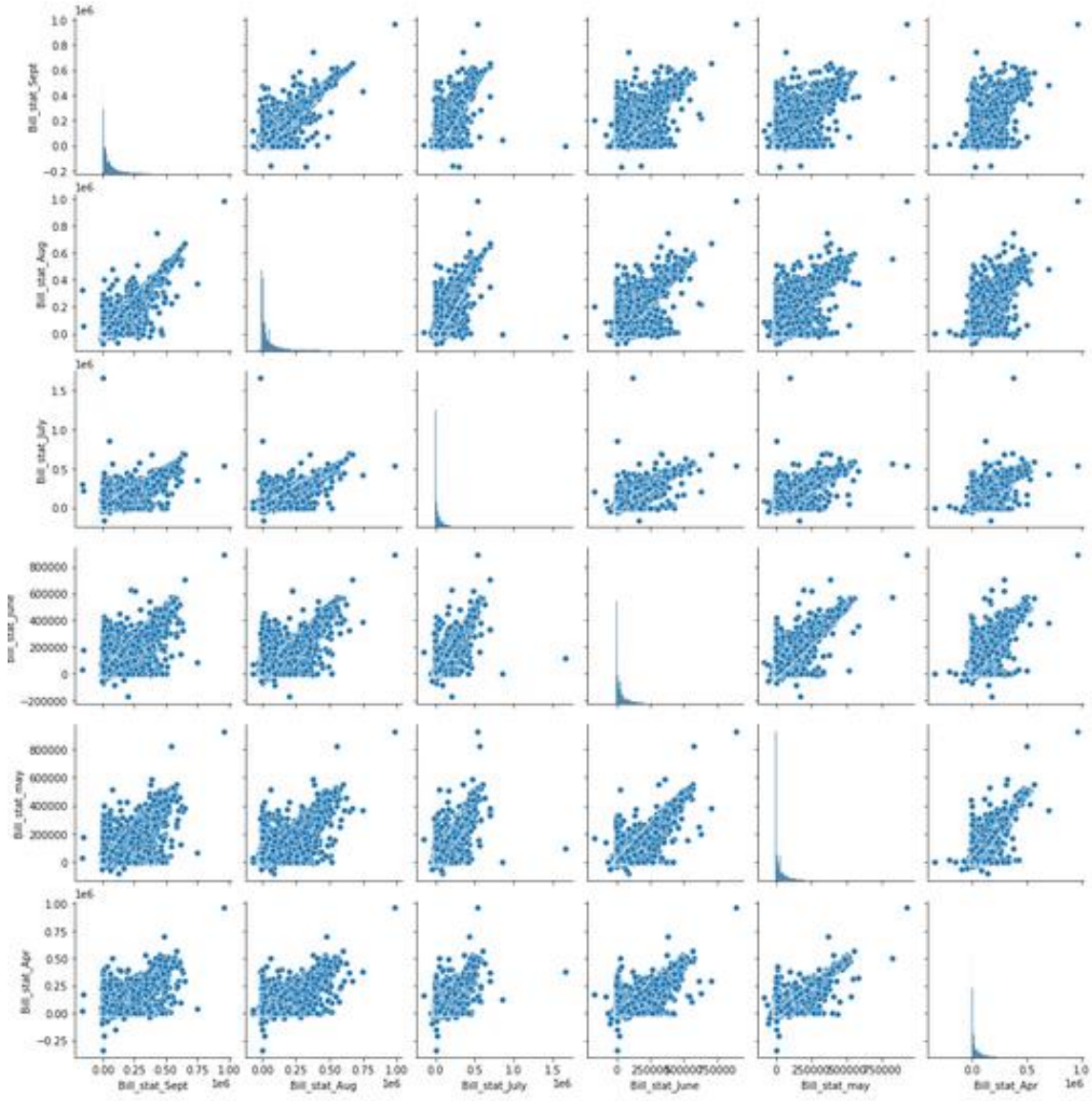


Image 8.1: scatter matrix to show the relationship between the bill statement features.

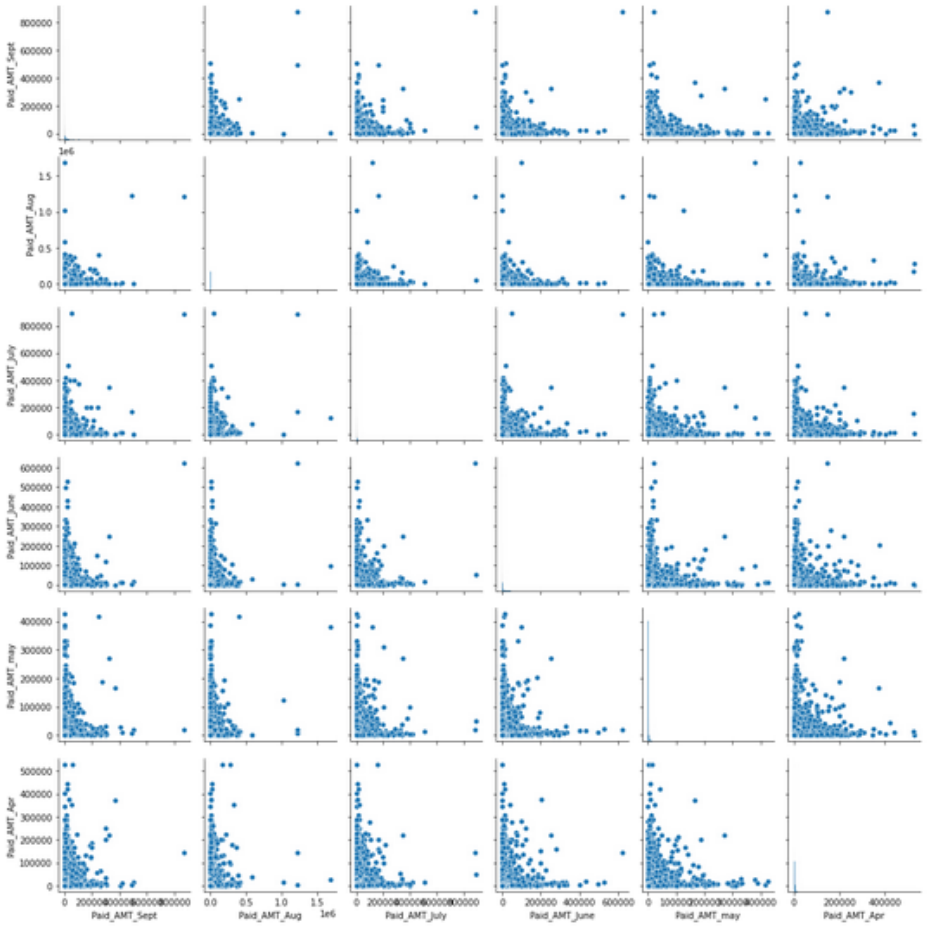


Image 8.2: scatter matrix to show the relationship between the paid amount features.

As can be seen by the visualizations as well as the correlation table, only the bill statement features need to be dealt with to rectify the multicollinearity issue. This was because only the bill statement features had a strong positive correlation with each other. To solve this a mean bill amount column was created and will be used to represent the 6 features. The following code creates the mean bill column and drops the bill statement April- September columns:

```
In [49]:
#add mean bill
from pyspark.sql.functions import col, lit
df = df.withColumn("Mean_Bill",((col('Bill_stat_Sept') +
                col('Bill_stat_Aug')+
                col('Bill_stat_July')+ col('Bill_stat_June')+
                col('Bill_stat_may')+
                col('Bill_stat_Apr'))/lit(6)).alias("mean"))

#Drop bill statment columns
df  = df.drop('Bill_stat_Sept','Bill_stat_Aug','Bill_stat_July','Bill_stat_June',
              'Bill_stat_may', 'Bill_stat_Apr')

df.columns
```

Code 6.2: Creating new mean bill plot

Results:

```
Out[71]: ['Limit_Balance',
          'Sex',
          'Education',
          'Marraige',
          'Age',
          'Paid_AMT_Sept',
          'Paid_AMT_Aug',
          'Paid_AMT_July',
          'Paid_AMT_June',
          'Paid_AMT_may',
          'Paid_AMT_Apr',
          'Default',
          'Mean_Bill']
```

3.4 Class imbalance issues

There was a class imbalance in the target variables. Far more clients did not default than did default. The following code shows the class imbalance:

In [81]:
```
#check for class imbalance
import pyspark.sql.functions as f
from pyspark.sql.window import Window

df.groupby('Default').count().sort("Default",ascending=True).show()

df_ci = spark.createDataFrame([(0,22996),
                               (1, 6605)],
                              ['Default','count'])

df_ci = df_ci.withColumn('percent', f.col('count')/f.sum('count').over(Window.partitionBy()))
df_ci.orderBy('percent', ascending=False).show()
```

Code 7.1: Percentage table for each default class count

Results:

```
+-------+-----+
|Default|count|
+-------+-----+
|      0|22996|
|      1| 6605|
+-------+-----+

+-------+-----+------------------+
|Default|count|           percent|
+-------+-----+------------------+
|      0|22996|0.7768656464308639|
|      1| 6605|0.2231343535691362|
+-------+-----+------------------+
```

The percentage column informs us that only 22% of the clients did default on their cards. This indicates that the default classes are imbalanced as the ratio between defaulting and not defaulting is 1:5. This could adversely affect the accuracy scores from the logistic regression for the multivariate analysis because, the algorithm will favour to predict the most common class. As we are trying to predict the minority class this is problematic.

Under sampling was used. Under sampling is when the majority class is reduced to a similar number as the minority class. However, before under sampling could be undertaken one hot encoding and vector assemble had to be used. One hot encoding is used to prevent the machine learning algorithm weighting the numerical values as higher than their true representation. For example, the number 1 in the education feature represents graduate school and 2 represents university. The algorithm would recognise 2 being double that of 1. However, this does not make sense as university is not twice the value of graduate school. Therefore, to separate these values into discrete categories, the values are converted into 1s and 0s and put into a vector. Vector assembling is when all the features are put together in a vector ready for the machine learning algorithm.

The following code was used to carry out one hot encoding:

```
In [60]: from pyspark.ml.feature import OneHotEncoder
         #One hot encoding

         encoder = OneHotEncoder(inputCols=["Sex", "Education" ,"Marraige","Repayment_sept", "Repayment_Aug",
                                            "Repayment_July", "Repayment_June", "Repayment_may", "Repayment_Apr"],
                                 outputCols=["Sex_V", "Edu_V", "Marraige_V", "RepaymentS_V", "RepaymentA_V",
                                             "RepaymentJULV", "RepaymentJUNV", "RepaymentmV", "Repayment_ApV"])
         model = encoder.fit(U)
         encoded = model.transform(U)
```

Code 7.2: One hot encoder

The following code was used for vector assembling:

```
In [61]: from pyspark.ml.linalg import Vectors
         from pyspark.ml.feature import VectorAssembler

         #vector assembling

         assembler = VectorAssembler( inputCols=['Limit_Balance',
                                  'Age',
                                  'Mean_Bill',
                                  'Paid_AMT_Sept',
                                  'Paid_AMT_Aug',
                                  'Paid_AMT_July',
                                  'Paid_AMT_June',
                                  'Paid_AMT_may',
                                  'Paid_AMT_Apr',
                                  'RepaymentS_V',
                                  'Marraige_V',
                                  'RepaymentJULV',
                                  'RepaymentmV',
                                  'Edu_V',
                                  'RepaymentA_V',
                                  'RepaymentJUNV',
                                  'Sex_V',
                                  'Repayment_ApV'],
                                 outputCol = "features")

         output = assembler.transform(encoded)
         output.select("features", "Default").show(truncate=False)
```

Code 7.3: Vector assemble

The following code was used to carry out under sampling:

```
In [62]: Pysparkdf = output.select("features", "Default")
```

```
In [105]: from pyspark.sql.functions import col, explode, array, lit
          major_df = Pysparkdf.filter(col("Default") == 0)
          minor_df = Pysparkdf.filter(col("Default") == 1)
          ratio = int(major_df.count()/minor_df.count())
          print("ratio: {}".format(ratio))
```

```
In [108]: sampled_majority_df = major_df.sample(False, 1/ratio)
          combined_df_2 = sampled_majority_df.unionAll(minor_df)
          combined_df_2.show()
```

```
In [109]: combined_df_2.groupby('Default').count().show()
```

Code 7.4: Undersampling

Results:

ratio: 3

```
+--------------------+-------+
|            features|Default|
+--------------------+-------+
|[90000.0,34.0,151...|      0|
|(17,[0,1,3,4,5,6,...|      0|
|[320000.0,49.0,10...|      0|
|(17,[0,1,10,12,15...|      0|
|[130000.0,39.0,30...|      0|
|[50000.0,30.0,130...|      0|
|[100000.0,32.0,30...|      0|
|(17,[0,1,2,3,5,6,...|      0|
|[20000.0,24.0,131...|      0|
|[310000.0,49.0,78...|      0|
|[180000.0,34.0,85...|      0|
|(17,[0,1,4,6,10,1...|      0|
|[400000.0,29.0,17...|      0|
|(17,[0,1,3,4,5,6,...|      0|
|[200000.0,32.0,58...|      0|
|(17,[0,1,3,5,10,1...|      0|
|[340000.0,32.0,57...|      0|
|[50000.0,25.0,175...|      0|
|[300000.0,45.0,29...|      0|
|[470000.0,33.0,64...|      0|
+--------------------+-------+
only showing top 20 rows
```

```
+-------+-----+
|Default|count|
+-------+-----+
|      1| 6605|
|      0| 7666|
+-------+-----+
```

3.5 Logistic regression

Logistic regression is a method of machine learning. It is used to solve classification problems and builds upon the foundation of linear regression. Logistic regression was used for this investigation because it is an algorithm that can be used to predict a binomial response value. In addition to this it

can handle both discrete and continuous data types to predict a target variable. Logistic regression works by determining the probability that the outcome is either the target variable or not. If that probability is over 50% that the prediction is the target variable, then it will predict it as such. If the probability is less than 50% then it will predict it as not the target variable.

In this study, the data set was split; 70% training and 30% testing. Following the division of data, the train data set was used to train the logistic regression model. This trained model was then used to predict the target variables on the test data set.

Logistic regression was performed using the following features:

- Limit Balance
- Age
- Repayment status April – September
- Paid amount April – September
- Marriage
- Education
- Sex
- Mean Bill

The model was run on both datasets, with and without under sampling, to compare whether under sampling improved the accuracy scores. The following code is the implementation of the logistic regression model using the under-sampled dataset.

```
In [64]: from pyspark.ml.classification import LogisticRegression
         #Logistic regression model
         #with undersampling
         train, test = combined_df_2.randomSplit([0.7, 0.3], seed = 2018)
         lr = LogisticRegression(featuresCol = 'features', labelCol = 'Default', maxIter=10)
         lrModel_3 = lr.fit(train)
         predictions = lrModel_3.transform(test)

         accuracy = predictions.filter(predictions.Default == predictions.prediction).count() / float(predictions.count()
         print("Accuracy : ",accuracy)

         Accuracy :  0.7216494845360825
```

Code 8.1: Logistic regression with under-sampling

Logistic regression was performed on the data set without under-sampling using the following code:

```
In [65]: from pyspark.ml.classification import LogisticRegression
         #Logistic regression model
         #without undersampling
         train, test = Pysparkdf.randomSplit([0.7, 0.3], seed = 2018)
         lr = LogisticRegression(featuresCol = 'features', labelCol = 'Default', maxIter=10)
         lrModel_3 = lr.fit(train)
         predictions = lrModel_3.transform(test)

         accuracy = predictions.filter(predictions.Default == predictions.prediction).count() / float(predictions.count())
         print("Accuracy : ",accuracy)

         Accuracy :  0.813149949681315
```

Code 8.2: Logistic regression without under-sampling

III. Discussion

The following discussions are derived from the visualizations and machine learning algorithms used in this investigation. The key results will be discussed in their respected categories, univariate analysis, and multivariate analysis.

Univariate analysis:

The results firstly inform us that most people do not default on their credit card (image 1). During the univariate analysis the counts of each categorical variable and age in each feature were compared. Notably image 2.4 showed that there were no clients who were aged under 20, this could be because younger people are less likely to have a fixed source of income and credit cards are not usually approved to people unless they have a reliable source of income. In addition to this, the distribution plot of ages was skewed to the left stating that more young people possessed a credit card than older people. Interestingly most of the clients attended some form of further education as illustrated by the image 2.2. When analysing the continuous data, box plots were used. Image 3.1 found that the bill statement for the month July contained the highest value outlier. Stating that the largest bill statement was issued during the month of July. However, the most paid was not during July, unexpectedly, Image 3.2 found that the most paid in a single month was August. Additionally, all the box plots identified more outliers above the median value compared to below the median value for all continuous features analysed.

Multivariate analysis:

Image 5.1, 5.2 and 5.3 showed the counts of each category who did and did not default. However, as the distribution of clients was not evenly spread amongst each category, this was a poor representation of the proportions of each category who defaulted and did not default. For example, in each image 5.1, 5.2 and 5.3 the categorical variables, Female, Single and University, all had the highest number of clients who defaulted. However, this was expected as they all also had the highest counts of clients. To fix this, the proportions of those who did and did not default was calculated as a percentage and plotted. The proportion of males who defaulted was more than the proportion of females (image 6). More clients who identified as other defaulted as a proportion compared to the marriage and single categories (image 6.1). Also, High school clients had the highest proportion of individuals who defaulted compared to the other educational institutions examined (Image 6.2).

The multivariate analysis also analysed the differences between the continuous variables with defaulting on a credit card. There is a discrepancy between the mean bill amount per month and mean paid amount per month. Image 7 and 7.1 show the difference in scales. The mean bill amount is approximately 8-9 times larger than the mean paid amount, suggesting the average client was paying substantially less than what they owed. However, on credit cards, there is usually a minimum owed amount which can be calculated from 5% (mint, 2019) of the outstanding balance, thus explains the reason for the discrepancies between the amount owed and paid. Image 7.1 found that the mean amount paid was lower for people who defaulted whereas Image 7 depicts that the mean bill amount for each default class is very similar. Therefore, people who defaulted were most likely underpaying on their minimum payments. Also, the mean bill grew in an upward trajectory as the months went on whereas the paid amounts per month only grew for the no default class. Thus, the proportion paid of the bill was reducing as the months went on, the most, by the clients of those who defaulted. The upward trajectory growth for the mean bill, could be due to the interest accumulated from not paying the full bill amount off. Mean limit balance found that people who defaulted on their credit card had

a lower limit balance on average compared to people who did not default (Image 7.2). The mean age is similar across both default classes (Image 7.3).

During the multivariate analysis image 8.1 and 8.2 show the linear correlations between the features bill amounts per months and payment amounts per month. As mentioned in the multicollinearity section of the multivariate analysis the relationships between the bill amounts per month were highly correlated with each other. This is represented using image 8.1. Image 8.2 found that the paid amounts per month had a weak positive correlation with each other.

The aim of this investigation was to identify if logistic regression could predict credit card defaults and whether using under sampling to remove the class imbalance would improve these results. The accuracy scores show that under sampling did not improve the accuracy of the model. The model with under sampling scored an accuracy score of 72% whereas the accuracy score of the model without under sampling was 81%. The score of 81% is similar with the findings from Kaggle, as mentioned in the introduction, so were to be expected. However, this study converted the bill amount (April – September) columns into a single mean column. Therefore, this study has successfully reduced the number of features used, subsequently reducing the processing power required, whilst maintaining a high accuracy score.

The low accuracy score of 72% for the under sampled data set could be attributed to there not being enough instances in the data set.

IV. Performance of the programme

As mention prior, the logistic regression model using PySpark is efficient and effect in predicting credit card defaults. However, the programme suffered from a series of drawbacks. There was a prolonged time delay of a few minutes before the programme had finished running. The results from the logistic regression were not cross validated and finally some segments of code could have been converted into functions.

As the logistic regression model only used 19 features it is unlikely the time delay is caused due to the feature quantity. Therefore, the time delay suffered can be attributed to two likely causes. The first being the virtual machine being used. The specification for the virtual machine may not have been able to run the programme as effectively as another machine. The second reason is the format of the code. Many segments of code could have been reformatted into functions, however, due to the time restrictions these functions could not be tested.

Cross validation is a useful tool to validate the results. It works by running the ML model over several times using different split points for the test and train dataset each time. It then can be used to calculate the average score, or just to produce the best score. Without, cross validation it limits the reliability of the study as the accuracy results may be due to a fortunate split of the data and may not reflect the normal accuracy scores. However, as the scores produced in this paper, closely coincide with previous findings then this paper does still attain some reliability. Furthermore, to include cross validation would have resulted in a further time delay for the programme to execute the logistic regression model as multiple models would have been ran.

V. Conclusions

A range of features can be used to predict whether a client will default or not on their credit card. Male, high school, and other (relationship) classes had the highest proportion of clients who defaulted

on their credit cards. Those who defaulted on average paid less per month then those clients who did not default, whilst the bill for both clients remained the same.

Overall, this study suffered from a large quantity of unknown variables. However, the study concludes that logistic regression can be used to predict credit card defaults and under sampling did not improve this model.

VI. References

Databricks *Apache Spark* [online] < https://databricks.com/spark/about> [21 March 2021]

Dong, Y. and Peng, C.J. (2013) 'Principled missing data methods for researchers' *Springerplus* [online] available from < https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3701793/> [15th February 2021]

Edureka (2020) *PySpark Programming – Integrating Speed With Simplicity* [online] available from < https://www.edureka.co/blog/pyspark-programming/#:~:text=PySpark%20is%20the%20collaboration%20of,%2C%20high%2Dlevel%20programming%20language> [23 March 2021]

Kaggle (2017) *Defaults of Credit Card Clients Dataset* [online] available from < https://www.kaggle.com/uciml/default-of-credit-card-clients-dataset/discussion> [16 March 2021]

Kaggle (2018) *Credit Card Default Prediction & Analysis* [online] available from <https://www.kaggle.com/ainslie/credit-card-default-prediction-analysis> [20 March 2021]

Kaggle (2019) *Prediction of Credit Card Default* [online] available from < https://www.kaggle.com/selener/prediction-of-credit-card-default > [17 March 2021]

Machine Learning Repository (2016) *Default of credit card clients Data Set* [online] <https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients> [23 March 2021]

mint (2019) *What is minimum amount due on credit cards?* [online] available from < https://www.livemint.com/money/personal-finance/what-is-minimum-amount-due-on-credit-cards-1551619428699.html> [16 March 2021]

Ogundur (2020) *Logistic Regression with Pyspark* [online] < https://medium.com/swlh/logistic-regression-with-pyspark-60295d41221>[25th March 2021]

Tutorials Point *PySpark – SparkContext* [online] < https://www.tutorialspoint.com/pyspark/pyspark_sparkcontext.htm > [14 March 2021]

Youtube (2017) *Spark Tutorial For Beginners | Big Data Spark Tutorial | Apache Spark Tutorial | Simplilearn* [online] < Spark Tutorial For Beginners | Big Data Spark Tutorial | Apache Spark Tutorial | Simplilearn - YouTube> [22 March 2021]

VII. Appendices:

Appendix 1

Code for the pre-processing the data section.

In [1]:
```
from pyspark import SparkContext
#creat sparkContext
sc = SparkContext.getOrCreate()
#initiating spark session
from pyspark.sql import SparkSession
spark = SparkSession \
    .builder \
    .appName("Logistic Regression Model") \
    .config("spark.execute.memory", "1gb") \
    .getOrCreate()

#Load data through RDD
rdd = sc.textFile('default of credit card clients.csv')

rdd.take(3)
```

Out[1]:
```
[',X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,
X20,X21,X22,X23,Y',
 'ID,LIMIT_BAL,SEX,EDUCATION,MARRIAGE,AGE,PAY_0,PAY_2,PAY_3,PAY_4,PAY_
5,PAY_6,BILL_AMT1,BILL_AMT2,BILL_AMT3,BILL_AMT4,BILL_AMT5,BILL_AMT6,PA
Y_AMT1,PAY_AMT2,PAY_AMT3,PAY_AMT4,PAY_AMT5,PAY_AMT6,default payment ne
xt month',
 '1,20000,2,2,1,24,2,2,-1,-1,-2,-2,3913,3102,689,0,0,0,0,689,0,0,0,0,1
']
```

In [2]:
```
#Filter first two rows
rdd_head = rdd.first()
rdd1 = rdd.filter(lambda line:line!=rdd_head)

head = rdd1.first()
rdd2 = rdd1.filter(lambda line:line!=head)
rdd2.first()
```

Out[2]:
```
'1,20000,2,2,1,24,2,2,-1,-1,-2,-2,3913,3102,689,0,0,0,0,689,0,0,0,0,1'
```

In [2]:
```
#Filter first two rows
rdd_head = rdd.first()
rdd1 = rdd.filter(lambda line:line!=rdd_head)

head = rdd1.first()
rdd2 = rdd1.filter(lambda line:line!=head)
rdd2.first()
```

Out[2]:
```
'1,20000,2,2,1,24,2,2,-1,-1,-2,-2,3913,3102,689,0,0,0,0,689,0,0,0,0,1'
```

In [3]:
```
#split by comma and change column names
from pyspark.sql import Row
df = rdd2.map(lambda line:line.split(',')).map(lambda line: Row(ID = line[0],
                                Limit_Balance= line[1],
                                Sex = line[2],
                                Education = line[3],
                                Marraige = line[4],
                                Age = line[5],
                                Repayment_sept = line[6],
                                Repayment_Aug = line[7],
                                Repayment_July = line[8],
                                Repayment_June = line[9],
                                Repayment_may = line[10],
                                Repayment_Apr = line[11],
                                Bill_stat_Sept = line[12],
                                Bill_stat_Aug = line[13],
                                Bill_stat_July = line[14],
                                Bill_stat_June = line[15],
                                Bill_stat_may = line[16],
                                Bill_stat_Apr = line[17],
                                Paid_AMT_Sept = line[18],
                                Paid_AMT_Aug = line[19],
                                Paid_AMT_July = line[20],
                                Paid_AMT_June = line[21],
                                Paid_AMT_may = line[22],
                                Paid_AMT_Apr = line[23],
                                Default = line[24]
                                )).toDF()
df.show()
```

```
+---+-------------+---+---------+--------+---+--------------+-------------+--------------+--------------+---------
----+-------------+--------------+-------------+--------------+--------------+--------------+--------+---------
-----+------------+-------------+-------------+------------+------------+-------+
| ID|Limit_Balance|Sex|Education|Marraige|Age|Repayment_sept|Repayment_Aug|Repayment_July|Repayment_June|Repayment
_may|Repayment_Apr|Bill_stat_Sept|Bill_stat_Aug|Bill_stat_July|Bill_stat_June|Bill_stat_may|Bill_stat_Apr|Paid_AMT
_Sept|Paid_AMT_Aug|Paid_AMT_July|Paid_AMT_June|Paid_AMT_may|Paid_AMT_Apr|Default|
+---+-------------+---+---------+--------+---+--------------+-------------+--------------+--------------+---------
----+-------------+--------------+-------------+--------------+--------------+--------------+--------+---------
-----+------------+-------------+-------------+------------+------------+-------+
|  1|        20000|  2|        2|       1| 24|             2|            2|            -1|            -1|
-2|           -2|          3913|         3102|           689|             0|             0|             0|
```

In [4]:
```
#to see data frame easier
df.toPandas()
```

Out[4]:

	ID	Limit_Balance	Sex	Education	Marraige	Age	Repayment_sept	Repayment_Aug
0	1	20000	2	2	1	24	2	2
1	2	120000	2	2	2	26	-1	2
2	3	90000	2	2	2	34	0	0
3	4	50000	2	2	1	37	0	0
4	5	50000	1	2	1	57	-1	0
...	...	...	...	...	...	...	...	...
29995	29996	220000	1	3	1	39	0	0
29996	29997	150000	1	3	2	43	-1	-1
29997	29998	30000	1	2	2	37	4	3
29998	29999	80000	1	3	1	41	1	-1
29999	30000	50000	1	2	1	46	0	0

30000 rows × 25 columns

In [6]:
```
#discriptive statistics of dataframe
df.summary().show()
```

```
+-------+-----------------+------------------+------------------+----
--------------+------------------+-----------------+------------------
+--------------------+------------------+--------------------+--------
----------+------------------+-----------------+-----------------+----
-------------+------------------+-----------------+-----------------+-
-----------------+------------------+------------------+---------------
---+------------------+------------------+------------------+
|summary|               ID|     Limit_Balance|               Sex|
Education|          Marraige|               Age|    Repayment_sept|
Repayment_Aug|    Repayment_July|      Repayment_June|     Repayment_m
ay|     Repayment_Apr|   Bill_stat_Sept|    Bill_stat_Aug|   Bill_stat
_July|    Bill_stat_June|    Bill_stat_may|    Bill_stat_Apr|     Paid
_AMT_Sept|      Paid_AMT_Aug|    Paid_AMT_July|     Paid_AMT_June|
Paid_AMT_may|      Paid_AMT_Apr|             Default|
+-------+-----------------+------------------+------------------+----
--------------+------------------+-----------------+------------------
+--------------------+------------------+--------------------+--------
----------+------------------+-----------------+-----------------+----
-------------+------------------+-----------------+-----------------+-
-----------------+------------------+------------------+---------------
---+------------------+------------------+------------------+
|  count|            30000|             30000|              30000|
30000|             30000|            30000|            30000|
30000|             30000|               30000|            30000|
30000|            30000|            30000|           30000|
30000|            30000|            30000|             30000|
30000|            30000|             30000|             30000|
30000|              30000|
|   mean|          15000.5|167484.32266666667| 1.6037333333333332|1.85
31333333333333|1.5518666666666667|           35.4855|           -0.016
7|-0.13376666666666667|          -0.1662|-0.22066666666666668|
-0.2662|           -0.2911|       51223.3309|49179.07516666667|
47013.1548| 43262.94896666666|40311.40096666667|       38871.7604|
5663.5805|         5921.1635|        5225.6815| 4826.076866666666| 479
9.387633333334| 5215.502566666667|             0.2212|
| stddev|8660.398374208891| 129747.6615672022|0.4891291609026045|0.79
03486597207295|0.5219696006132476|9.217904068090172|1.123801527997335
2|  1.1971859730345535|1.1968675684465764|   1.169138622402338|1.13318
74060027476|1.1499876256079036|73635.86057552951|71173.76878252844|693
49.38742703671|64332.856133916306|60797.15577026491|59554.10753674586|
16563.280354025763|23040.870402057244|17606.96146980309|15666.15974403
2013|15278.305679144803|17777.465775435325|0.41506180569093293|
|    min|                1|             10000|                  1|
0|                0|              21|               -1|
-1|               -1|                  -1|               -1|
```

In [7]:
```
#drop ID column
df = df.drop('ID')
df.columns
```

Out[7]:
```
['Limit_Balance',
 'Sex',
 'Education',
 'Marraige',
 'Age',
 'Repayment_sept',
 'Repayment_Aug',
 'Repayment_July',
 'Repayment_June',
 'Repayment_may',
 'Repayment_Apr',
 'Bill_stat_Sept',
 'Bill_stat_Aug',
 'Bill_stat_July',
 'Bill_stat_June',
 'Bill_stat_may',
 'Bill_stat_Apr',
 'Paid_AMT_Sept',
 'Paid_AMT_Aug',
 'Paid_AMT_July',
 'Paid_AMT_June',
 'Paid_AMT_may',
 'Paid_AMT_Apr',
 'Default']
```

In [8]:
```
#convert data types to appropriate formats
from pyspark.sql.types import *
from pyspark.sql.functions import *

df = df.withColumn('Limit_Balance', df['Limit_Balance'].cast(FloatType()))\
    .withColumn('Sex', df['Sex'].cast(IntegerType()))\
    .withColumn('Education', df['Education'].cast(IntegerType()))\
    .withColumn('Age', df['Age'].cast(IntegerType()))\
    .withColumn('Marraige', df['Marraige'].cast(IntegerType()))\
    .withColumn('Repayment_sept', df['Repayment_sept'].cast(IntegerType()))\
    .withColumn('Repayment_Aug', df['Repayment_Aug'].cast(IntegerType()))\
    .withColumn('Repayment_July', df['Repayment_July'].cast(IntegerType()))\
    .withColumn('Repayment_June', df['Repayment_June'].cast(IntegerType()))\
    .withColumn('Repayment_may', df['Repayment_may'].cast(IntegerType()))\
    .withColumn('Repayment_Apr', df['Repayment_Apr'].cast(IntegerType()))\
    .withColumn('Bill_stat_Sept', df['Bill_stat_Sept'].cast(FloatType()))\
    .withColumn('Bill_stat_Aug', df['Bill_stat_Aug'].cast(FloatType()))\
    .withColumn('Bill_stat_July', df['Bill_stat_July'].cast(FloatType()))\
    .withColumn('Bill_stat_June', df['Bill_stat_June'].cast(FloatType()))\
    .withColumn('Bill_stat_may', df['Bill_stat_may'].cast(FloatType()))\
    .withColumn('Bill_stat_Apr', df['Bill_stat_Apr'].cast(FloatType()))\
    .withColumn('Paid_AMT_Sept', df['Paid_AMT_Sept'].cast(FloatType()))\
    .withColumn('Paid_AMT_Aug', df['Paid_AMT_Aug'].cast(FloatType()))\
    .withColumn('Paid_AMT_July', df['Paid_AMT_July'].cast(FloatType()))\
    .withColumn('Paid_AMT_June', df['Paid_AMT_June'].cast(FloatType()))\
    .withColumn('Paid_AMT_may', df['Paid_AMT_may'].cast(FloatType()))\
    .withColumn('Paid_AMT_Apr', df['Paid_AMT_Apr'].cast(FloatType()))\
    .withColumn('Default', df['Default'].cast(IntegerType()))

#Show data types
df.dtypes
```

Out[8]:
```
[('Limit_Balance', 'float'),
 ('Sex', 'int'),
 ('Education', 'int'),
 ('Marraige', 'int'),
 ('Age', 'int'),
 ('Repayment_sept', 'int'),
 ('Repayment_Aug', 'int'),
 ('Repayment_July', 'int'),
 ('Repayment_June', 'int'),
 ('Repayment_may', 'int'),
 ('Repayment_Apr', 'int'),
 ('Bill_stat_Sept', 'float'),
 ('Bill_stat_Aug', 'float'),
 ('Bill_stat_July', 'float'),
 ('Bill_stat_June', 'float'),
 ('Bill_stat_may', 'float'),
 ('Bill_stat_Apr', 'float'),
 ('Paid_AMT_Sept', 'float'),
 ('Paid_AMT_Aug', 'float'),
```

In [18]:
```
#Check unique values
def checkvalues():
    for i in df.columns:
        o = df.select(i).distinct()
        o.show()


print(checkvalues())
```

- Repayment columns both 0s and -2 are unknown
- Education columns 5,6,0 are unknown
- Marraiage columsn 0 are unknown

In [ ]:
```
#How many rows do the 5,6,0s in the Education column take up.
df.filter(df['Education'] == 5).count() + \
df.filter(df['Education'] == 6).count() + \
df.filter(df['Education'] == 0).count()
```

In [ ]:
```
#percentage of unknown variables in education
345/30000*100
```

In [ ]:
```
#How many rows do the 0s in the Marraige column takeup.
df.filter(df['Marraige'] == 0).count()
```

In [ ]:
```
#percentage of unknown variables in Marraige
54/30000*100
```

In [ ]:
```
#function to see how many unknown variables are in repayment columns
def unknownvariables(values):
    for i in df.columns[5:11]:
        c = df.filter(df[i]==values).count()
        print(i, 'number of' ,values, 'is', c)
        if c >=0:
            u = c/30000*100
            print('Percentage of unknown variables for', i, 'is', u)
    return()

print(unknownvariables(-2))
print(unknownvariables(0))
```

- The percentage of unknown variables in marraige and education is less than 5% and therefore can be removed and treated as missing values
- The percentage for unknow varaiables in the repayment columns are over 50% and therefore cannot be removed. However, they also cannot be used for visualizations

In [15]:

```
#impute missing values using most frequent
from sklearn.impute import SimpleImputer
import numpy as np

df = df.withColumn('Education', regexp_replace('Education', '5', '0'))
df = df.withColumn('Education', regexp_replace('Education', '6', '0'))
newdf = df.withColumn('Education', regexp_replace('Education', '0', 'np.nan'))
newdf = newdf.withColumn('Marraige', regexp_replace('Marraige' , '0', 'np.nan'))
imp_mean = SimpleImputer(missing_values='np.nan', strategy='most_frequent')
imp_mean.fit(newdf.toPandas())
imputed_df = imp_mean.transform(newdf.toPandas())

#convert numpy ndarray to pandas dataframe to
#convert it to pyspark dataframe
import pandas as pd

df_impute = pd.DataFrame(imputed_df, columns = ['Limit_Balance',
                                'Sex',
                                'Education',
                                'Marraige',
                                'Age',
                                'Repayment_sept',
                                'Repayment_Aug',
                                'Repayment_July',
                                'Repayment_June',
                                'Repayment_may',
                                'Repayment_Apr',
                                'Bill_stat_Sept',
                                'Bill_stat_Aug',
                                'Bill_stat_July',
                                'Bill_stat_June',
                                'Bill_stat_may',
                                'Bill_stat_Apr',
                                'Paid_AMT_Sept',
                                'Paid_AMT_Aug',
                                'Paid_AMT_July',
                                'Paid_AMT_June',
                                'Paid_AMT_may',
                                'Paid_AMT_Apr',
                                'Default'])

df_2 = spark.createDataFrame(df_impute)
print('Education variables' ,df_2.toPandas()['Education'].unique())
print('Marraige variables' ,df_2.toPandas()['Marraige'].unique())
```

In [16]:
```
#keep copy of original data set with all original columns
U = df
df_2 = df
#drop repayment columns (too many unexplained rows)

aa = df.columns[5:11]
for i in aa:
    df = df.drop(i)

df.columns
```

Out[16]:
```
['Limit_Balance',
 'Sex',
 'Education',
 'Marraige',
 'Age',
 'Bill_stat_Sept',
 'Bill_stat_Aug',
 'Bill_stat_July',
 'Bill_stat_June',
 'Bill_stat_may',
 'Bill_stat_Apr',
 'Paid_AMT_Sept',
 'Paid_AMT_Aug',
 'Paid_AMT_July',
 'Paid_AMT_June',
 'Paid_AMT_may',
 'Paid_AMT_Apr',
 'Default']
```

Summary tables

```
In [32]: #first way to produce a summary table

         df.describe().show()
```

```
5198.415898111551|4828.659268267964|4795.032753503615|5181.326374108983|0.2231343535691362|
| stddev|129944.02095269752|0.4892444171231663|0.7103992840785517| 0.518092327098095|9.213243333797596|16568.26494144611|23089.193621490787|17580.914805811994|15711.05799234724|15244.217154322845| 17657.2607390526|0.4163547406844325|
|    min|           10000.0|                 1|                 1|                 1|                 1|              21|              0.0|              0.0|                0.0|              0.0|               0.0|               0.0|               0.0|
|    max|         1000000.0|                 2|                 4|                 3|              79|         873552.0|          1684259.0|           896040.0|         621000.0|           426529.0|          528666.0|                 1|
+-------+------------------+------------------+------------------+------------------+----------------+-----------------+-------------------+--------------------+-----------------+-------------------+-------------------+------------------+
```

```
In [33]: #second way to produce a summary table is to produce a seperate summary
         #table of each column


         def descriptives(columns):
             for i in columns:
                 df.describe(i).show()

         print(descriptives(df.columns))
```

```
+-------+------------------+
|summary|     Limit_Balance|
+-------+------------------+
|  count|             29601|
|   mean|167550.54491402316|
| stddev|129944.02095269752|
|    min|           10000.0|
|    max|         1000000.0|
+-------+------------------+
```

Appendix 2:

Results from code:

In [18]:
```
#Check unique values
def checkvalues():
    for i in df.columns:
        o = df.select(i).distinct()
        o.show()


print(checkvalues())
```

```
+-------------+
|Limit_Balance|
+-------------+
|     670000.0|
|     440000.0|
|     650000.0|
|     580000.0|
|     550000.0|
|     100000.0|
|     690000.0|
|     130000.0|
|     470000.0|
|     520000.0|
|     310000.0|
|     290000.0|
|     450000.0|
|     370000.0|
|     330000.0|
|      20000.0|
|     280000.0|
|     230000.0|
|     170000.0|
|     260000.0|
+-------------+
only showing top 20 rows
```

Sex
1
2

Education
1
6
3
5
4
2
0

Marraige
1
3
2
0

```
+---+
|Age|
+---+
| 31|
| 65|
| 53|
| 34|
| 28|
| 26|
| 27|
| 44|
| 22|
| 47|
| 52|
| 40|
| 57|
| 54|
| 48|
| 64|
| 41|
| 43|
| 37|
| 61|
+---+
only showing top 20 rows
```

```
+--------------+
|Repayment_sept|
+--------------+
|            -1|
|             1|
|             6|
|             3|
|             5|
|             4|
|             8|
|             7|
|            -2|
|             2|
|             0|
+--------------+
```

Repayment_Aug
-1
1
6
3
5
4
8
7
-2
2
0

Repayment_July
-1
1
6
3
5
4
8
7
-2
2
0

Repayment_June
-1
1
6
3
5
4
8
7
-2
2
0

Repayment_may
-1
6
3
5
4
8
7
-2
2
0

Repayment_Apr
-1
6
3
5
4
8
7
-2
2
0

```
+--------------+
|Bill_stat_Sept|
+--------------+
|      172180.0|
|      141315.0|
|       10237.0|
|        3675.0|
|      270276.0|
|       20272.0|
|      964511.0|
|      499231.0|
|        9317.0|
|        5925.0|
|      276758.0|
|       10415.0|
|       64800.0|
|      123160.0|
|       13159.0|
|       28440.0|
|      315816.0|
|       18460.0|
|       50244.0|
|         -64.0|
+--------------+
only showing top 20 rows

+-------------+
|Bill_stat_Aug|
+-------------+
|      68187.0|
|      25736.0|
|     239025.0|
|      64683.0|
|      83081.0|
|      32903.0|
|      25956.0|
|       7702.0|
|     268508.0|
|     301521.0|
|      51013.0|
|       3238.0|
|      56144.0|
|     104189.0|
|       9995.0|
|      45888.0|
|       4446.0|
|      49061.0|
|      83620.0|
|        -64.0|
+-------------+
only showing top 20 rows
```

```
+--------------+
|Bill_stat_July|
+--------------+
|       95768.0|
|       13268.0|
|        6631.0|
|       31119.0|
|        5758.0|
|       44031.0|
|       70912.0|
|        7702.0|
|      127237.0|
|      106434.0|
|      246011.0|
|         -64.0|
|      109836.0|
|       65348.0|
|       82532.0|
|      146410.0|
|        5750.0|
|       50100.0|
|       24689.0|
|       18621.0|
+--------------+
only showing top 20 rows

+--------------+
|Bill_stat_June|
+--------------+
|       61467.0|
|        4618.0|
|        3238.0|
|      104734.0|
|      181866.0|
|        9995.0|
|        2098.0|
|       46836.0|
|      143828.0|
|       27834.0|
|        9531.0|
|       71370.0|
|      196099.0|
|       61578.0|
|        7509.0|
|        9169.0|
|       27762.0|
|       50244.0|
|       13836.0|
|       80284.0|
+--------------+
only showing top 20 rows
```

```
+-------------+
|Bill_stat_may|
+-------------+
|      13836.0|
|      49061.0|
|      12805.0|
|       9567.0|
|       8984.0|
|      98404.0|
|      18559.0|
|      55501.0|
|      40601.0|
|      20469.0|
|      13268.0|
|      50687.0|
|       5758.0|
|      19052.0|
|      11033.0|
|     139432.0|
|      49728.0|
|      30283.0|
|      11378.0|
|       5750.0|
+-------------+
only showing top 20 rows

+-------------+
|Bill_stat_Apr|
+-------------+
|       2077.0|
|      83988.0|
|     100124.0|
|      52588.0|
|      67069.0|
|      21235.0|
|       9995.0|
|      18285.0|
|      29065.0|
|     145071.0|
|       6801.0|
|       8559.0|
|      20750.0|
|       5925.0|
|      74794.0|
|      18298.0|
|      19665.0|
|      41039.0|
|     352736.0|
|       -184.0|
+-------------+
```

```
+------------+
|Paid_AMT_Aug|
+------------+
|     15029.0|
|      7408.0|
|      1033.0|
|      3675.0|
|      2679.0|
|      5758.0|
|      1575.0|
|      2098.0|
|      7464.0|
|     18285.0|
|     26373.0|
|      5343.0|
|     13999.0|
|      2077.0|
|      4271.0|
|     20469.0|
|      3451.0|
|      6130.0|
|      2976.0|
|      3599.0|
+------------+
only showing top 20 rows

+-------------+
|Paid_AMT_July|
+-------------+
|       1033.0|
|       3238.0|
|       9995.0|
|       4446.0|
|     104734.0|
|       7509.0|
|       2679.0|
|       5750.0|
|       5039.0|
|       3675.0|
|      26454.0|
|       1575.0|
|        714.0|
|       2077.0|
|       5758.0|
|       2098.0|
|       3451.0|
|       7464.0|
|      97874.0|
|       8451.0|
+-------------+
only showing top 20 rows
```

```
+-------------+
|Paid_AMT_June|
+-------------+
|       3238.0|
|       9567.0|
|       1033.0|
|       5758.0|
|      65349.0|
|        714.0|
|       6130.0|
|       6471.0|
|      30070.0|
|      11480.0|
|       5039.0|
|       7509.0|
|      18559.0|
|       2098.0|
|       2976.0|
|       5138.0|
|      15029.0|
|       2679.0|
|       5925.0|
|       3675.0|
+-------------+
only showing top 20 rows

+------------+
|Paid_AMT_may|
+------------+
|      6833.0|
|      2318.0|
|     19665.0|
|     57229.0|
|      6801.0|
|      7464.0|
|      1033.0|
|      5138.0|
|     15962.0|
|       714.0|
|      2976.0|
|      5039.0|
|     12714.0|
|      4271.0|
|      8984.0|
|      2098.0|
|     32528.0|
|     11378.0|
|     10358.0|
|     12636.0|
+------------+
```

```
+------------+
|Paid_AMT_Apr|
+------------+
|     33764.0|
|      5750.0|
|      3238.0|
|      4271.0|
|      1033.0|
|      5039.0|
|      6130.0|
|      7509.0|
|      5138.0|
|      2679.0|
|      3675.0|
|      9567.0|
|      8984.0|
|      2976.0|
|      2077.0|
|      2098.0|
|      3599.0|
|     37984.0|
|     12714.0|
|       886.0|
+------------+
only showing top 20 rows

+-------+
|Default|
+-------+
|      1|
|      0|
+-------+
```

Appendix 3:

Univariate analysis

The following code follows on from Appendix 1.

In [22]:
```
#distribution plot by count for features, sex, age, education and marraige
import matplotlib.pyplot as plt
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

def plotstats(columns):
    plt.figure()
    for i in columns:
        u = df.toPandas()[i].unique()
        if len(u)==2:
            T = df.groupBy(i).count().toPandas()
            plt.figure()

            x = T[i]
            y = T["count"]
            plt.bar(x,y , color=['orange','blue'])
            plt.xticks([2,1],['Female', 'Male'])
            plt.xlabel(i)
            plt.ylabel("Count")
            plt.title("Male and Female count")
            plt.show()
        if len(u)==4:
            T = df.groupBy(i).count().toPandas()
            plt.figure()

            x = T[i]
            y = T["count"]
            plt.bar(x,y)
            plt.xticks([0,1,2,3],['High School', 'Graduate School','Others', 'University school'])
            plt.xlabel(i)
            plt.ylabel("Count")
            plt.title("Education class count")
            plt.show()
        if len(u)>10:
            T = df.groupBy(i).count().toPandas()
            plt.figure()
            x = T[i]
            y = T["count"]
            plt.bar(x,y)
            plt.xlabel(i)
            plt.ylabel("Count")
            plt.title("Age frequency count")
            plt.show()
        if len(u)==3:
            T = df.groupBy(i).count().toPandas()
            plt.figure()
            x = T[i]
            y = T["count"]
            plt.bar(x,y)
            plt.xticks([1,2,3],['Married', 'Single','Others'])
            plt.xlabel(i)
            plt.ylabel("Count")
            plt.title("Marraige class count")
            plt.show()

print(plotstats(df.columns[1:5]))
```

In [24]:
```
#Box plot with the Bill satement columns
plt.figure()
data=df['Bill_Stat_sept', 'Bill_stat_Aug','Bill_stat_July',
        'Bill_stat_June',
        'Bill_stat_Apr', 'Bill_stat_may'].toPandas()
fig1, ax1 = plt.subplots()
ax1.boxplot(data)
plt.title('''Box plot comparing median, upper quartile and lower quatile
          for bill statments from April to Septmeber''')
plt.xticks([1,2,3,4,5,6], ['September', 'August','July',
        'June',
        'April', 'May'], rotation = 90)
plt.xlabel('Month')
plt.ylabel('Bill statment amount in NT dollar')
plt.show()
```

In [25]:
```
#box plot for paid amount columns
plt.figure()
data=df['Paid_AMT_Sept',
 'Paid_AMT_Aug',
 'Paid_AMT_July',
 'Paid_AMT_June',
 'Paid_AMT_may',
 'Paid_AMT_Apr'].toPandas()
fig1, ax1 = plt.subplots()
ax1.boxplot(data)
plt.title('''Box plot comparing median, upper quartile and lower quatile
          for paid amounts per month from April to Septmeber''')
plt.xlabel('Month')
plt.ylabel('Paid amount in NT dollar')
plt.xticks([1,2,3,4,5,6], ['September', 'August','July',
        'June',
        'April', 'May'], rotation = 90)
plt.show()
```

In [38]:
```
#box plot for limit balance
plt.figure()
data=df.toPandas()['Limit_Balance']
fig1, ax1 = plt.subplots()
ax1.boxplot(data)
plt.title('''Box plot to show the mean, upper quartile and lower quatile of the
           Limit Balance feature''')
plt.ylabel('Limit Balance in NT dollar')
plt.xticks([1],['Limit Balance'])
plt.show()
```

Appendix 4:

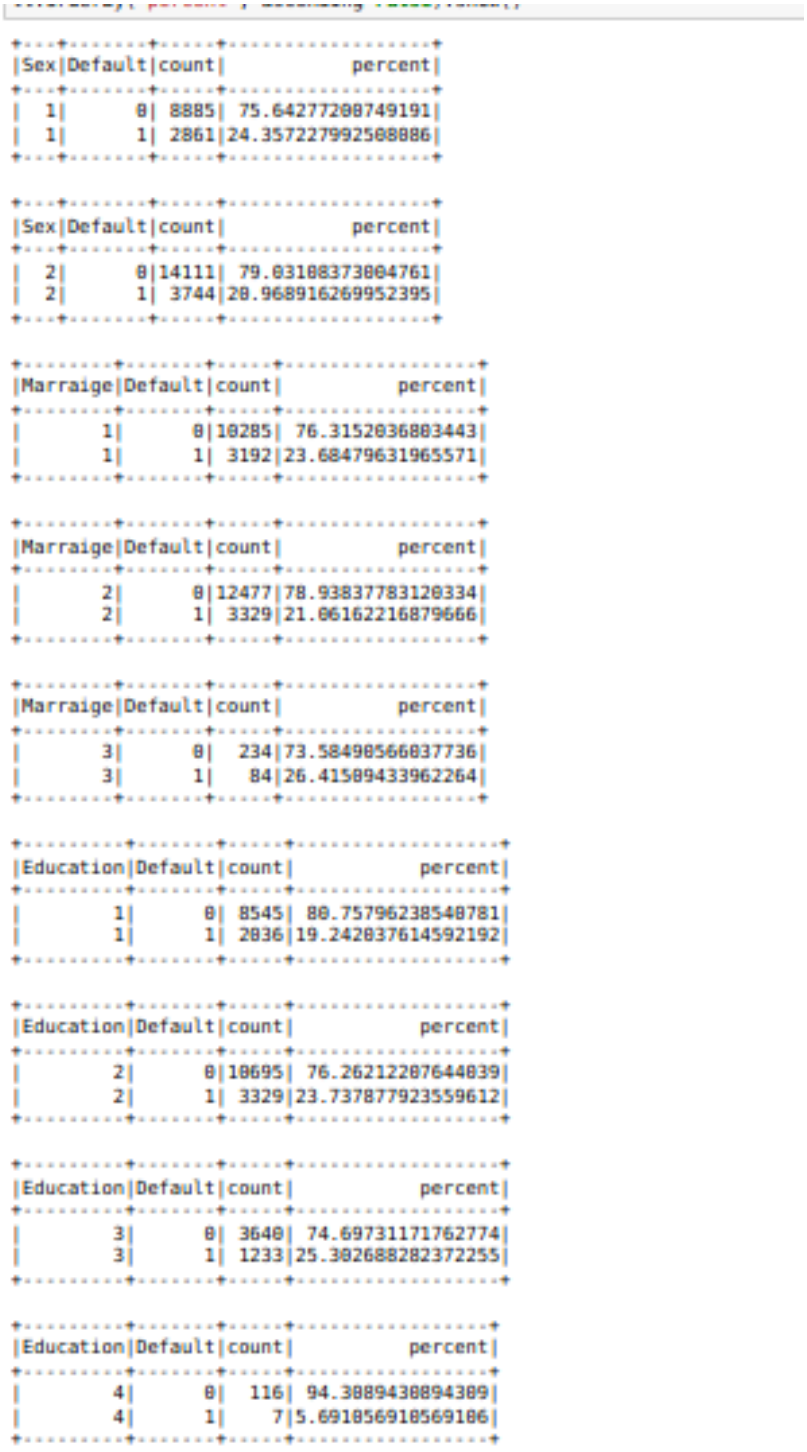
Multivariate analysis

The following code follows on from the code in appendix 3.

In [28]:
```
#grouped barplot of catagorical counts that did default and did not default

df_i = df.groupBy('Sex','Default').count().toPandas()

plt.figure()
df_i.pivot(index='Default', columns = 'Sex', values = 'count').plot(kind='bar')
plt.xticks([0,1],['No Default', 'Yes Default'])
plt.xlabel("Default")
plt.ylabel("Count")
plt.title("Male and Female Default count")
labels = ('Female', 'Male')
plt.legend(labels)
plt.show()


df_i = df.groupBy('Marraige','Default').count().toPandas()

plt.figure()
df_i.pivot(index='Default', columns = 'Marraige', values = 'count').plot(kind='bar')
plt.xticks([0,1],['No Default', 'Yes Default'])
plt.xlabel("Default")
plt.ylabel("Count")
plt.title("Marraige Default class count")
labels = ('Married', 'Single', 'Others')
plt.legend(labels)
plt.show()


df_i = df.groupBy('Education','Default').count().toPandas()

plt.figure()
df_i.pivot(index='Default', columns = 'Education', values = 'count').plot(kind='bar')
plt.xticks([0,1],['No Default', 'Yes Default'])
plt.xlabel("Default")
plt.ylabel("Count")
plt.title("Education Default class count")
labels = ('Graduate school', 'University', 'High school', 'Others')
plt.legend(labels)
plt.show()
```

In [42]:

```
import pyspark.sql.functions as f
from pyspark.sql.window import Window
#proportion calulations for Sex

pp = df.groupBy('Sex', 'Default').count()


df_2 = pp.filter(pp['Sex']==1)


ll = df_2.withColumn('percent', 100*f.col('count')/f.sum('count').over(Window.partitionBy()))
ll.orderBy('percent', ascending=False).show()


pp = df.groupBy('Sex', 'Default').count()


df_2 = pp.filter(pp['Sex']==2)


ll = df_2.withColumn('percent', 100*f.col('count')/f.sum('count').over(Window.partitionBy()))
ll.orderBy('percent', ascending=False).show()

#proportion calculations for marraige

pp = df.groupBy('Marraige', 'Default').count()


df_2 = pp.filter(pp['Marraige']==1)


ll = df_2.withColumn('percent', 100*f.col('count')/f.sum('count').over(Window.partitionBy()))
ll.orderBy('percent', ascending=False).show()







df_2 = pp.filter(pp['Marraige']==2)

ll = df_2.withColumn('percent', 100*f.col('count')/f.sum('count').over(Window.partitionBy()))
ll.orderBy('percent', ascending=False).show()



df_2 = pp.filter(pp['Marraige']==3)

ll = df_2.withColumn('percent', 100*f.col('count')/f.sum('count').over(Window.partitionBy()))
ll.orderBy('percent', ascending=False).show()


#proportion calculations from Education

pp = df.groupBy('Education', 'Default').count()


df_2 = pp.filter(pp['Education']==1)

ll = df_2.withColumn('percent', 100*f.col('count')/f.sum('count').over(Window.partitionBy()))
ll.orderBy('percent', ascending=False).show()







df_2 = pp.filter(pp['Education']==2)

ll = df_2.withColumn('percent', 100*f.col('count')/f.sum('count').over(Window.partitionBy()))
ll.orderBy('percent', ascending=False).show()







df_2 = pp.filter(pp['Education']==3)

ll = df_2.withColumn('percent', 100*f.col('count')/f.sum('count').over(Window.partitionBy()))
ll.orderBy('percent', ascending=False).show()




df_2 = pp.filter(pp['Education']==4)

ll = df_2.withColumn('percent', 100*f.col('count')/f.sum('count').over(Window.partitionBy()))
ll.orderBy('percent', ascending=False).show()
```



```
+---+-------+-----+------------------+
|Sex|Default|count|           percent|
+---+-------+-----+------------------+
|  1|      0| 8885| 75.64277208749191|
|  1|      1| 2861|24.357227992508886|
+---+-------+-----+------------------+

+---+-------+-----+------------------+
|Sex|Default|count|           percent|
+---+-------+-----+------------------+
|  2|      0|14111| 79.03108373804761|
|  2|      1| 3744|20.968916269952395|
+---+-------+-----+------------------+

+--------+-------+-----+------------------+
|Marraige|Default|count|           percent|
+--------+-------+-----+------------------+
|       1|      0|10285| 76.3152036803443|
|       1|      1| 3192|23.68479631965571|
+--------+-------+-----+------------------+

+--------+-------+-----+------------------+
|Marraige|Default|count|           percent|
+--------+-------+-----+------------------+
|       2|      0|12477|78.93837783120334|
|       2|      1| 3329|21.06162216879666|
+--------+-------+-----+------------------+

+--------+-------+-----+------------------+
|Marraige|Default|count|           percent|
+--------+-------+-----+------------------+
|       3|      0|  234|73.58490566037736|
|       3|      1|   84|26.41509433962264|
+--------+-------+-----+------------------+

+---------+-------+-----+------------------+
|Education|Default|count|           percent|
+---------+-------+-----+------------------+
|        1|      0| 8545| 80.75796238548781|
|        1|      1| 2036|19.242037614592192|
+---------+-------+-----+------------------+

+---------+-------+-----+------------------+
|Education|Default|count|           percent|
+---------+-------+-----+------------------+
|        2|      0|10695| 76.26212287644039|
|        2|      1| 3329|23.737877923559612|
+---------+-------+-----+------------------+

+---------+-------+-----+------------------+
|Education|Default|count|           percent|
+---------+-------+-----+------------------+
|        3|      0| 3640| 74.69731171762774|
|        3|      1| 1233|25.302688282372255|
+---------+-------+-----+------------------+

+---------+-------+-----+------------------+
|Education|Default|count|           percent|
+---------+-------+-----+------------------+
|        4|      0|  116| 94.3089430894309|
|        4|      1|    7|5.691056910569106|
+---------+-------+-----+------------------+
```

The percentage results were then inputted manually into the following tables to create the plots.

In [43]:

```
#Bar plot for proportion of Sex in each default class
df5 = spark.createDataFrame([
                             ('Male', 0, 75.64277200749191),
                             ('Male', 1, 24.357227992508086),
                             ('Female', 0, 79.03108373004761),
                             ('Female', 1, 20.968916269952395)],
                            ['Sex','Default','percentage'])




plt.figure()
df5=df5.toPandas()
df5.pivot(index='Default', columns = 'Sex', values = 'percentage').plot(kind='bar')
plt.xticks([0,1],['No Default', 'Yes Default'])
plt.xlabel("Default")
plt.ylabel("percentage")
plt.title("Male and Female Default percentage")
plt.show()


#Bar plot for proportion of marraige in each default class


df5 = spark.createDataFrame([
                             ('Single', 0, 78.93837783120334),
                             ('Single', 1, 21.06162216879666),
                             ('Married', 0, 76.3152036803443),
                             ('Married', 1, 23.68479631965571),
                             ('Other',   1, 26.41509433962264),
                             ('Other',    0,  73.58490566037736)],
                            ['Marriage','Default','percentage'])

plt.figure()
df5=df5.toPandas()
df5.pivot(index='Default', columns = 'Marriage', values = 'percentage').plot(kind='bar')
plt.xticks([0,1],['No Default', 'Yes Default'])
plt.xlabel("Default")
plt.ylabel("percentage")
plt.title("'The percentage of each Marriage catagory population who did default and did not default")
plt.show()

#Bar plot for proportion of education in each default class

df5 = spark.createDataFrame([
                             ('University', 0, 76.26212207644039),
                             ('University', 1, 23.737877923559612),
                             ('Graduate School', 0, 80.75796238540781),
                             ('Graduate School', 1, 19.242037614592192),
                             ('High School',   0, 74.69731171762774),
                             ('High School',    1,  25.302688282372255),
                             ('Other',   0, 94.3089430894309),
                             ('Other',    1,  5.691056910569106)],
                            ['Education','Default','percentage'])

plt.figure()
df5=df5.toPandas()
df5.pivot(index='Default', columns = 'Education', values = 'percentage').plot(kind='bar')
plt.xticks([0,1],['No Default', 'Yes Default'])
plt.xlabel("Default")
plt.ylabel("percentage")
plt.title("'The percentage of each Education catagory population who did default and did not default")
plt.show()
```

In [44]:
```
#mean age for default and no default class
cc = df.groupBy("Age","Default").count().show()

cc

uu = df.filter(df['Default']==1).groupBy("Age", 'Default').mean('Age').toPandas()

uu = spark.createDataFrame(uu)
uu.show()
ii = df.filter(df['Default']==0).groupBy("Age", 'Default').mean('Age').toPandas()


ii = spark.createDataFrame(ii)
ii.show()


ii.groupBy("Default").mean('Age').show()
uu.groupBy("Default").mean('Age').show()

ll = spark.createDataFrame([
                            ('Age', 1, 47.075471698113205),
                            ('Age', 0, 48.55357142857143)],
                           ['Age','Default','Mean'])

plt.figure()
ll = ll.toPandas()
ll.pivot(index='Default', columns = 'Age', values = 'Mean').plot(kind='bar')
plt.xticks([0,1],['No Default', 'Yes Default'])
plt.xlabel("Default")
plt.ylabel("Mean Age")
plt.title("Mean Age per Default Class")
plt.show()
```

In [48]:
```
df_HB = df.groupBy('Bill_stat_Sept', 'Bill_stat_Aug','Bill_stat_July',
        'Bill_stat_June',
        'Bill_stat_Apr', 'Bill_stat_may'
                        ,'Paid_AMT_Sept',
 'Paid_AMT_Aug',
 'Paid_AMT_July',
 'Paid_AMT_June',
 'Paid_AMT_may',
 'Paid_AMT_Apr','Limit_Balance','Default').count()

df_HB
def average(columns):
    for i in columns:
        df_HB.groupBy("Default").mean(i).show()

print(average(df_HB.columns[:13]))
```

```
+-------+-------------------+
|Default|avg(Bill_stat_Sept)|
+-------+-------------------+
|      1|  58765.21827734437|
|      0|  52803.34261974585|
+-------+-------------------+

+-------+------------------+
|Default|avg(Bill_stat_Aug)|
+-------+------------------+
|      1| 49540.24677599189|
|      0| 50538.47134897574|
+-------+------------------+

+-------+-------------------+
|Default|avg(Bill_stat_July)|
+-------+-------------------+
|      1| 47368.579366342936|
|      0|  48331.45778903404|
+-------+-------------------+

+-------+-------------------+
|Default|avg(Bill_stat_June)|
+-------+-------------------+
|      1|  44134.89921986945|
|      0|  44392.99844485915|
+-------+-------------------+

+-------+------------------+
|Default|avg(Bill_stat_Apr)|
+-------+------------------+
|      1|  48221.8662314918|
|      0|39877.026526259666|
+-------+------------------+

+-------+------------------+
|Default|avg(Bill_stat_may)|
+-------+------------------+
|      1|41550.823913389584|
|      0|  41317.0564738292|
+-------+------------------+

+-------+------------------+
|Default|avg(Paid_AMT_Sept)|
+-------+------------------+
|      1|3538.7648463620444|
|      0| 6442.083821203235|
+-------+------------------+

+-------+-----------------+
|Default|avg(Paid_AMT_Aug)|
+-------+-----------------+
|      1|3556.935679832001|
|      0|6759.728339189571|
+-------+-----------------+

+-------+------------------+
|Default|avg(Paid_AMT_July)|
+-------+------------------+
|      1|   3524.4588441331|
|      0|    5852.874477917|
+-------+------------------+

+-------+------------------+
|Default|avg(Paid_AMT_June)|
+-------+------------------+
|      1| 3319.886877885687|
|      0| 5423.739136230339|
+-------+------------------+

+-------+-----------------+
|Default|avg(Paid_AMT_may)|
+-------+-----------------+
|      1|3364.693679358422|
|      0|5367.004976458724|
+-------+-----------------+

+-------+-----------------+
|Default|avg(Paid_AMT_Apr)|
+-------+-----------------+
|      1|3588.167811622353|
|      0|5812.764373944726|
+-------+-----------------+

+-------+------------------+
|Default|avg(Limit_Balance)|
+-------+------------------+
|      1|126191.31985352651|
|      0|  177960.366124589|
+-------+------------------+

None
```

The mean values were taken from the tables above and manually put into a pyspark data frame.

In [50]:
```
df5 = spark.createDataFrame([
                              ('April', 0, 39877.826526259666),
                              ('April', 1, 40221.8662314918),
                              ('May', 0, 41317.0564738292),
                              ('May', 1, 41550.823913389584),
                              ('June', 0, 44392.99844485915),
                              ('June', 1, 44134.89921986945),
                              ('July', 0, 48331.45778983404),
                              ('July', 1, 44134.89921986945),
                              ('August', 0, 50538.47134897574),
                              ('August', 1, 49540.24677599189),
                              ('September', 0, 52883.34261974585),
                              ('September', 1, 50765.21827734437)],
                             ['Month','Default','mean'])
df5.show()


aa = df5.filter(df5['Default']==1).toPandas()
ab = df5.filter(df5['Default']==0).toPandas()


# multiple line plots
plt.plot( 'Month', 'mean', data=aa, marker='o', markerfacecolor='blue', markersize=12, color='skyblue', linewidth=4
plt.plot( 'Month', 'mean', data=ab, marker='o', markerfacecolor='Red', markersize=12, color='Green', linewidth=4)

plt.xlabel("Month")
plt.ylabel("Mean")
# show legend
plt.legend(['Yes Default', 'No Default'])

plt.title("Mean bill statment amount for each default class per month")
# show graph
plt.show()
```

In [51]:
```
df5 = spark.createDataFrame([
                              ('April', 0, 5812.764373944726),
                              ('April', 1, 3588.167011622353),
                              ('May', 0, 5367.004976458724),
                              ('May', 1, 3364.693679358422),
                              ('June', 0, 5423.739136230339),
                              ('June', 1, 3319.806877885687),
                              ('July', 0, 5852.874477917),
                              ('July', 1, 3524.4588441331),
                              ('August', 0, 6759.728339109571),
                              ('August', 1, 3556.935679032001),
                              ('September', 0, 6442.003821283235),
                              ('September', 1, 3538.7648463620444)],
                             ['Month','Default','mean'])

aa = df5.filter(df5['Default']==1).toPandas()
ab = df5.filter(df5['Default']==0).toPandas()

print(aa)
print(ab)

# multiple line plots
plt.plot( 'Month', 'mean', data=aa, marker='o', markerfacecolor='blue', markersize=12, color='skyblue', linewidth=4
plt.plot( 'Month', 'mean', data=ab, marker='o', markerfacecolor='Red', markersize=12, color='Green', linewidth=4)

plt.xlabel("Month")
plt.ylabel("Mean")
# show legend
plt.legend(['Yes Default', 'No Default'])

plt.title("Mean paid statment amount for each default class per month")
# show graph
plt.show()
```

In [52]:
```
#mean Limit balance comparisons
x = ['Deafult', 'No Default']
y = [126253.6137667304, 178002.31131656148]

plt.xlabel("Limit Balance amount")
plt.ylabel("Default classes")
plt.title("Horizontal bar plot to show the difference in mean Limit Balance for each class of default")
plt.barh(x,y)
plt.show()
```

Appendix 5:

Multicollinearity

The following code follows on from appendix 4.

In [37]:
```
from pyspark.mllib.stat import Statistics
import pandas as pd

#correlation matrix
corr_data = df.select(df.columns)

colnames = corr_data.columns
features = corr_data.rdd.map(lambda row: row[0:])
corrmatrix=Statistics.corr(features, method="pearson")
corrdf1 = pd.DataFrame(corrmatrix)
corrdf1.index, corrdf1.columns = colnames, colnames

print(corrdf1.to_string())

#easier to view
corrdf1
```

`

In [54]:
```
import seaborn as sns
#plot scatter matrix for Bill statements and Paid amounts
sns.pairplot(df.toPandas(), vars=df.columns[5:11], kind='scatter')
sns.pairplot(df.toPandas(), vars=df.columns[11:17])

```

In [70]:
```
#add mean bill
from pyspark.sql.functions import col, lit
df = df.withColumn("Mean_Bill",((col('Bill_stat_Sept') +
                col('Bill_stat_Aug')+
                col('Bill_stat_July')+ col('Bill_stat_June')+
                col('Bill_stat_may')+
                col('Bill_stat_Apr'))/lit(6)).alias("mean"))

#Drop bill statment columns
df  = df.drop('Bill_stat_Sept','Bill_stat_Aug','Bill_stat_July','Bill_stat_June',
              'Bill_stat_may', 'Bill_stat_Apr')
```

Appendix 6:

Class imbalance

The following code follows on from appendix 6.

In [56]:
```
#check for class imbalance
import pyspark.sql.functions as f
from pyspark.sql.window import Window

df.groupby('Default').count().sort("Default",ascending=True).show()

df_ci = spark.createDataFrame([(0,22996),
                               (1, 6605)],
                              ['Default','count'])

df_ci = df_ci.withColumn('percent', f.col('count')/f.sum('count').over(Window.partitionBy()))
df_ci.orderBy('percent', ascending=False).show()
```

Before one hot encoding can be done the repayment, columns needed all their categories to be positive or 0.

In [58]:
```
#make negative values positive
from pyspark.sql.functions import *



U = U.withColumn('Repayment_sept', regexp_replace('Repayment_sept', '-2', '11'))


U = U.withColumn('Repayment_Aug', regexp_replace('Repayment_Aug', '-2', '11'))
U = U.withColumn('Repayment_July', regexp_replace('Repayment_July', '-2', '11'))


U = U.withColumn('Repayment_June', regexp_replace('Repayment_June', '-2', '11'))


U = U.withColumn('Repayment_may', regexp_replace('Repayment_may', '-2', '11'))


U = U.withColumn('Repayment_Apr', regexp_replace('Repayment_Apr', '-2', '11'))


U = U.withColumn('Repayment_sept', regexp_replace('Repayment_sept', '-1', '10'))


U = U.withColumn('Repayment_Aug', regexp_replace('Repayment_Aug', '-1', '10'))


U = U.withColumn('Repayment_July', regexp_replace('Repayment_July', '-1', '10'))


U = U.withColumn('Repayment_June', regexp_replace('Repayment_June', '-1', '10'))


U = U.withColumn('Repayment_may', regexp_replace('Repayment_may', '-1', '10'))


U = U.withColumn('Repayment_Apr', regexp_replace('Repayment_Apr', '-1', '10'))
```

The variables needed to be changed back to integers and floats.

In [59]:
```
U = U.withColumn('Limit_Balance', U['Limit_Balance'].cast(FloatType()))\
    .withColumn('Sex', U['Sex'].cast(IntegerType()))\
    .withColumn('Education', U['Education'].cast(IntegerType()))\
    .withColumn('Age', U['Age'].cast(IntegerType()))\
    .withColumn('Marraige', U['Marraige'].cast(IntegerType()))\
    .withColumn('Repayment_sept', U['Repayment_sept'].cast(IntegerType()))\
    .withColumn('Repayment_Aug', U['Repayment_Aug'].cast(IntegerType()))\
    .withColumn('Repayment_July', U['Repayment_July'].cast(IntegerType()))\
    .withColumn('Repayment_June', U['Repayment_June'].cast(IntegerType()))\
    .withColumn('Repayment_may', U['Repayment_may'].cast(IntegerType()))\
    .withColumn('Repayment_Apr', U['Repayment_Apr'].cast(IntegerType()))\
    .withColumn('Paid_AMT_Sept', U['Paid_AMT_Sept'].cast(FloatType()))\
    .withColumn('Paid_AMT_Aug', U['Paid_AMT_Aug'].cast(FloatType()))\
    .withColumn('Paid_AMT_July', U['Paid_AMT_July'].cast(FloatType()))\
    .withColumn('Paid_AMT_June', U['Paid_AMT_June'].cast(FloatType()))\
    .withColumn('Paid_AMT_may', U['Paid_AMT_may'].cast(FloatType()))\
    .withColumn('Paid_AMT_Apr', U['Paid_AMT_Apr'].cast(FloatType()))\
    .withColumn('Default', U['Default'].cast(IntegerType()))\
    .withColumn('Mean_Bill', U['Mean_Bill'].cast(FloatType()))
```

In [60]:
```
from pyspark.ml.feature import OneHotEncoder
#One hot encoding

encoder = OneHotEncoder(inputCols=["Sex", "Education" ,"Marraige","Repayment_sept", "Repayment_Aug",
                                   "Repayment_July", "Repayment_June", "Repayment_may", "Repayment_Apr"],
                        outputCols=["Sex_V", "Edu_V", "Marraige_V", "RepaymentS_V", "RepaymentA_V",
                                    "RepaymentJULV", "RepaymentJUNV", "RepaymentmV", "Repayment_ApV"])
model = encoder.fit(U)
encoded = model.transform(U)
```

In [61]:
```
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler

#vector assembling

assembler = VectorAssembler( inputCols=['Limit_Balance',
                          'Age',
                          'Mean_Bill',
                          'Paid_AMT_Sept',
                          'Paid_AMT_Aug',
                          'Paid_AMT_July',
                          'Paid_AMT_June',
                          'Paid_AMT_may',
                          'Paid_AMT_Apr',
                          'RepaymentS_V',
                          'Marraige_V',
                          'RepaymentJULV',
                          'RepaymentmV',
                          'Edu_V',
                          'RepaymentA_V',
                          'RepaymentJUNV',
                          'Sex_V',
                          'Repayment_ApV'],
                         outputCol = "features")

output = assembler.transform(encoded)
output.select("features", "Default").show(truncate=False)
```

```
+-----------------------------------------------------------------------------------------------------------------------
```

In [62]:
```
Pysparkdf = output.select("features", "Default")
```

In [53]:
```
#Undersampling
Pysparkdf = output.select("features", "Default")

from pyspark.sql.functions import col, explode, array, lit
major_df = Pysparkdf.filter(col("Default") == 0)
minor_df = Pysparkdf.filter(col("Default") == 1)
ratio = int(major_df.count()/minor_df.count())
print("ratio: {}".format(ratio))


sampled_majority_df = major_df.sample(False, 1/ratio)
combined_df_2 = sampled_majority_df.unionAll(minor_df)
combined_df_2.show()
```

Appendix 7:

Logistic regression with under sampling

In [55]:
```
from pyspark.ml.classification import LogisticRegression
#Logistic regression model
train, test = combined_df_2.randomSplit([0.7, 0.3], seed = 2018)
lr = LogisticRegression(featuresCol = 'features', labelCol = 'Default', maxIter=10)
lrModel_3 = lr.fit(train)
predictions = lrModel_3.transform(test)

accuracy = predictions.filter(predictions.Default == predictions.prediction).count() / float(predictions.count())
print("Accuracy : ",accuracy)
```

Logistic regression without under sampling

In [56]:
```
from pyspark.ml.classification import LogisticRegression
#Logistic regression model
train, test = Pysparkdf.randomSplit([0.7, 0.3], seed = 2018)
lr = LogisticRegression(featuresCol = 'features', labelCol = 'Default', maxIter=10)
lrModel_3 = lr.fit(train)
predictions = lrModel_3.transform(test)

accuracy = predictions.filter(predictions.Default == predictions.prediction).count() / float(predictions.count())
print("Accuracy : ",accuracy)
```

Code adapted from Ogundur, (2020).