FINAL REPORT

Vehicle Loan Default Prediction

Abstract

Data preparation, Exploratory Data Analysis and selection of Machine Learning Classification models on a vehicle loan dataset to make accurate predictions regarding loan default

Rafael Silveira

rafaelmoreirasilveira@gmail.com

Table of Contents

Introduction

Background

The immediate consequences of not paying a loan are easy to understand: the borrower may incur in extra expenses, have their Credit Score dropped and not be able to get more loans in the future. It is common knowledge that it is a terrible idea not to fulfill the loan instalments.

However, a little less evident is the impact of a great number of loan defaults have on companies granting the loans. Typically, those companies have insurance contracts with other companies in case several clients default their loans at the same time, for example. It is for their best interest to have a low loan default rate, otherwise the insurance premium is just too high.

Simply limiting loans to a small group of clients that falls into a specific category is not good enough because these companies make profit by lending money. Less business means less profit, leading them to run out of business. In addition, "raising the bar" to grant loans affects potential clients that could benefit from taking loans to invest in their business, for example. Limiting loans affects the whole economy.

The ideal scenario would be to know beforehand who has a high probability of fulfilling the monthly payments, and that is where my project comes in handy.

Proposal

The goal of this project is to increase the predictability of a vehicle loan default by analysing historical loans data. I will first prepare the data for analysis, which consists of checking for missing data, outliers, mistyped information, wrong data type and other actions that may be needed along the process. Some relationship between columns will arise, as well as some plots will be shown, to help us understand the data better.

The dataset is considered "treated" when it is ready to be imputed in a Machine Learning Pipeline, which is the second part of this project. I will define and measure the accuracy of several ML Classification models and choose the best one for this kind of data.

Dataset

The dataset consists of borrower data (Demographic data like age, income, Identity proof etc.), loan data (Disbursal details, amount, EMI, loan to value ratio etc.) and Bureau data (Bureau score, number of active accounts, the status of other loans, credit history etc.). The .csv files may be found on the following URL:

https://www.kaggle.com/avikpaul4u/vehicle-loan-default-prediction

Ps: this link will not work on the pdf version of this document.

It consists of 233,154 rows and 40 columns.

The target column is called "LOAN_DEFAULT", where 1 is the label associated to default.

The data types of the columns consist of integers, floats, strings, dates formatted as strings and intervals formatted as strings. This will become clearer as we advance in the report.

It is also important to note that a description of the columns of the dataset is presented in the appendix.

Data Preparation

This section of the report is based on the Jupyter Notebook that I prepared called "Data Preparation.ipynb", located in the same folder where this report is.

The language used for this project is Python and Pandas is the main library I am using to manipulate the data.

I will walk though all the process of preparing the dataset for Machine Learning models application. As part of the project, I will generate two datasets, as follows:

- df --> Full DataFrame. No columns will be dropped, and a few columns will be added. For example, columns that are linear combinations of other columns, like the age of the borrower at the moment of the disbursement, will be kept.
- df_drop --> No columns will be added, correlated columns will be dropped and columns with too many categories will be dropped.

The goal is to try both DataFrames on the Machine Learning Classification Models that will be tested on the Final part of the project and check which one renders the most accurate model.

Missing Data

To find the number of NANs ("Not A Number", which is missing data) in each column, I used the following command:

```
In [4]:  # Count the missing data in each column
         df.isnull().sum()
```

The output shows that only the column "EMPLOYMENT_TYPE" presents NANs. There are 7,661 missing records, and, as this represents only 3% of the dataset, I will simply remove those rows:

```
# Drop N/A
df.dropna(inplace=True)
```

Data Type Conversions

Using the command "df.info()", we are able to inspect the datatype of all the columns. With a quick inspection, we note that the following conversions are needed:

1) Columns DATE_OF_BIRTH and DISBURSAL_DATE: convert from string to DateTime format

   This can be accomplished with the following command:

```
In [7]:  #1) Convert Columns DATE_OF_BIRTH and DISBURSAL_DATE from string to DateTime format
         df['DATE_OF_BIRTH'] = pd.to_datetime(df['DATE_OF_BIRTH'])
         df['DISBURSAL_DATE'] = pd.to_datetime(df['DISBURSAL_DATE'])
```

Please note that, for the analysis part, DateTime format will be better. However, for the ML Classification part, we will convert this datetime object into the number of years (float) since the earliest date (min(date)), for each column.

2) Columns AVERAGE_ACCT_AGE and CREDIT_HISTORY_LENGTH: should be numbers.

   First, let us check how the data looks like:

```
In [8]: df[['AVERAGE_ACCT_AGE', 'CREDIT_HISTORY_LENGTH']].head()
```

Out[8]:

UNIQUEID	AVERAGE_ACCT_AGE	CREDIT_HISTORY_LENGTH
420825	0yrs 0mon	0yrs 0mon
537409	1yrs 11mon	1yrs 11mon
417566	0yrs 0mon	0yrs 0mon
624493	0yrs 8mon	1yrs 3mon
539055	0yrs 0mon	0yrs 0mon

These columns need to have their values converted to a float type, and the units should be years. For such, the following code takes care of it:

```
In [9]:
# Import module for string manipulation
import re

# Define a function that strips the numbers from the string. The first number (token[0]) is the number of years,
# the second number (token[1]) is the number of months, which is converted to year and added to token[0]
def convertPeriod(str):
    tokens = re.findall("\d+", str)
    return int(tokens[0]) + round(int(tokens[1])/12,2)

# loop through the two columns and apply the conversion
for col in ['AVERAGE_ACCT_AGE', 'CREDIT_HISTORY_LENGTH']:
    df[col] = df[col].apply(lambda x:convertPeriod(x))

# Check to see if successful
df[['AVERAGE_ACCT_AGE', 'CREDIT_HISTORY_LENGTH']].head()
```

Out[9]:

UNIQUEID	AVERAGE_ACCT_AGE	CREDIT_HISTORY_LENGTH
420825	0.00	0.00
537409	1.92	1.92
417566	0.00	0.00
624493	0.67	1.25
539055	0.00	0.00

The conversion was successful, as we may see on the output above.

3) Every column ending in "ID" should be converted to categorical.

Later, when I apply the classification methods, I will use the pandas get_dummies functions to generate numerical columns out of these categories, which is needed for the Scikit learn library. This is how the get_dummies function works: if a column called “Groups” has 2 groups: “A” and “B”, calling the get_dummies function will generate two columns: “Groups_A” and “Groups_B”. Records that belongs to Group A will have a 1 on the first column and a 0 in the second one.

Based on the explanation above, it makes sense to check how many unique values each of the categorical columns have, so we can have an idea of how many columns will be generated.

First, let us convert all columns ending in ID to strings:

```
# Create a list with columns ending in ID
ID_cols = [col for col in df.columns if col[-2:] == "ID"]

# Convert to category
for col in ID_cols:
    df[col] = df[col].astype(str)
```

Mistyped Data

The next step is to identify categorical columns and check unique values. We may find inconsistencies or typos:

```
# Create a list of columns with data type object
cat_cols = [col for col in df.columns if df[col].dtypes == "O"]

# Print unique values from each column
def printUniqueValues(colsList):
    print("Columns name  -  Unique values")
    for col in colsList:
        print("-----------------------------------------------------------------------------------")
        print(col," - ",sorted(df[col].unique()))

printUniqueValues(cat_cols)
```

And here is part of the output:

```
Columns name  -  Unique values
-----------------------------------------------------
BRANCH_ID  -  ['1', '10', '100', '101', '103', '104'
'138', '14', '142', '146', '147', '15', '152', '153'
0', '202', '207', '217', '248', '249', '250', '251',
5', '36', '42', '43', '48', '5', '61', '62', '63', '
'77', '78', '79', '8', '82', '84', '85', '9', '97']
-----------------------------------------------------
SUPPLIER_ID  -  ['10524', '12311', '12312', '12374',
8', '13131', '13295', '13317', '13448', '13512', '1
```

First thing to note is that the columns "SUPPLIER_ID", "CURRENT_PINCODE_ID" and "EMPLOYEE_CODE_ID" have way too many unique values. I will use these columns for analysis, however, for the classification process, I'll generate one DataFrame called df_slim, where these columns will be removed, and later the accuracy of the models will be compared (df X df_slim).

```
# list of columns to drop
toDrop = ["SUPPLIER_ID", "CURRENT_PINCODE_ID", "EMPLOYEE_CODE_ID"]

# Drop columns from the list and store on a new DataFrame df_slim
df_slim = df.drop(labels=toDrop, axis=1)
```

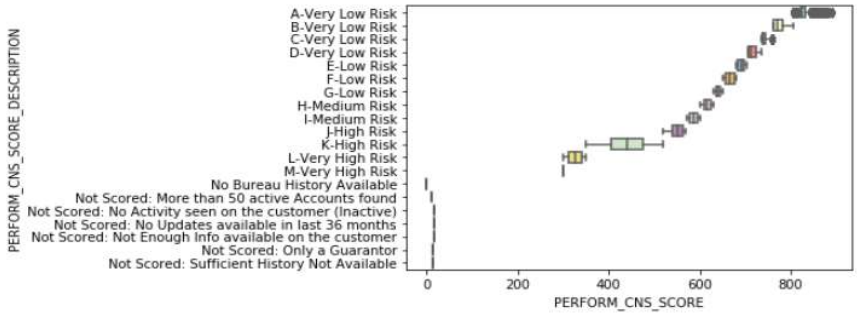
As for the other columns, no typos were found.

Inconsistencies Among Columns

I noticed that the columns PERFORM_CNS_SCORE_DESCRIPTION and PERFORM_CNS_SCORE are associated: each range of credit score from the second column corresponds to a “credit score grade” from the first column.

The goal here is check for inconsistencies between those 2 columns, and it can be easily spotted in graphical form. I will use a boxplot from the Seaborn library to check that there are no overlaps. This is needed because I do not know if the data on the description column was automatically filled when the score was entered, so I need to verify.

```
# Import Seaborn and matplotlib.pyplot Libraries
import seaborn as sns
import matplotlib.pyplot as plt

# Sort values by column description and store in a DataFrame object
df_sorted = df[["PERFORM_CNS_SCORE_DESCRIPTION", "PERFORM_CNS_SCORE"]].sort_values(by="PERFORM_CNS_SCORE_DESCRIPTION")

# Boxplot
ax = sns.boxplot(x="PERFORM_CNS_SCORE", y="PERFORM_CNS_SCORE_DESCRIPTION",palette="Set3", data=df_sorted)
plt.rcParams["figure.figsize"] = (1,2)
plt.show()
```



We can easily notice that there are no overlaps. Good data input methods were applied to generate this data.

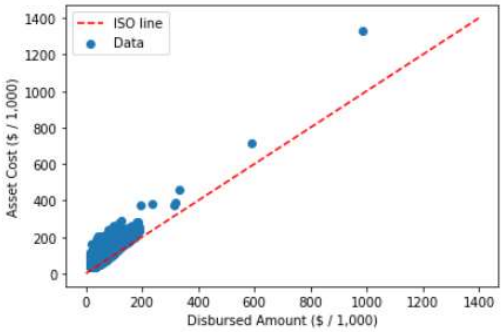
Those two columns are correlated, which means that the column PERFORM_CNS_SCORE will be dropped from the df_drop Dataframe. The bins convey more information because it sets categories to all data, as opposed to the actual scores that sets codes for 7 categories (the 7 last categories on the plot above). Whereas we can compare a credit score of 700 to a score of 800, we cannot say that 600 is greater 17 because 17 is a code for “Not Scored: Not Enough Info available on the customer”.

Outliers

Let us look for outliers in the numerical columns.

The two most important columns are "DISBURSED_AMOUNT" and "ASSET_COST". Even though the percentage of the Asset Cost may vary, we should see a positive correlation between these amounts. Let us do a scatter plot to verify that:

In [29]:
```
# Create scatter plot
plt.scatter(x=(df.DISBURSED_AMOUNT)/1000, y=(df.ASSET_COST)/1000, label='Data')
plt.plot([0,1400], [0,1400], '--',color='red', label='ISO line' )
plt.xlabel('Disbursed Amount ($ / 1,000)')
plt.ylabel('Asset Cost ($ / 1,000)')
plt.rcParams["figure.figsize"] = (6,4)
plt.legend()
plt.show()
```

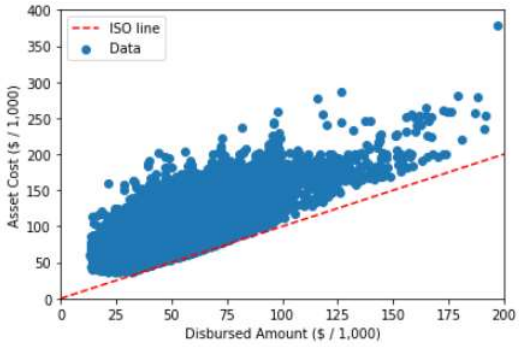


We notice that the data has outliers, however they do not alter the shape of a regression line (which is not plotted, by the way). They are "aligned" with the rest of the data, so they will be kept.

I included an ISO line (red doted line) to verify that every Asset Cost Value is greater than the Disbursed Amount. To double-check it, I will use two methods:

1) Zoom in on the plot for Disbursed Amounts between 0 and $200,000

```
In [34]:
# Zooming in on x=[0, 200000]
plt.scatter(x=(df.DISBURSED_AMOUNT)/1000, y=(df.ASSET_COST)/1000, label='Data')
plt.plot([0,1400], [0,1400], '--',color='red', label='ISO line' )
plt.xlabel('Disbursed Amount ($ / 1,000)')
plt.ylabel('Asset Cost ($ / 1,000)')
plt.rcParams["figure.figsize"] = (6,4)
plt.xlim(0,200)
plt.ylim(0, 400)
plt.legend()
plt.show()
```



Looks like all data is concentrated above the ISO line, meaning that the Asset cost is always greater than the Disbursed Amount. Let us double-check with another method, shown below.

2) Count the number of rows where the Disbursed Amount is greater than the Asset Cost.

```
In [35]:
print(df[df.DISBURSED_AMOUNT > df.ASSET_COST]["DISBURSED_AMOUNT"].count())
```

```
0
```

As expected, there are 0 rows that satisfy the condition Disbursed amount > Asset Cost.

Exploratory Data Analysis - EDA

For this section, let us analyse to the most important Column: our target column LOAN_DEFAULT. A few questions come to mind:

1) Is the Bureau Score reliable?
2) Does the default rate increase as the Disbursed Amount increase?
3) What is the average default rate by Employment type, State, Branch, Employee ID, ...
4) Is the age of the borrower relevant?

To answer these questions, I defined two functions to help me in this process: buildDict and plotBarPlot. You may find their definition in the Appendix, but here is what they do:

- buildDict – returns a dictionary where the keys = bin and values = Default Rate. There is a parameter to limit only bins with more than 10 data points in it. For example, consider that the bins are employees and one employee only granted 1 loan. This employee's Default Rate would be either 100% or 0%, which wouldn't be a valuable information.
- plotBarPlot - For any categorical column, this function plots, in descending order, up to the top 10 Default Rate for each bin. If there are more than 10 bins, just the top 10 will be plotted.

Is the Bureau Score Reliable?

We expect that low-risk clients should almost never default, and high-risk clients should have a higher default rate.

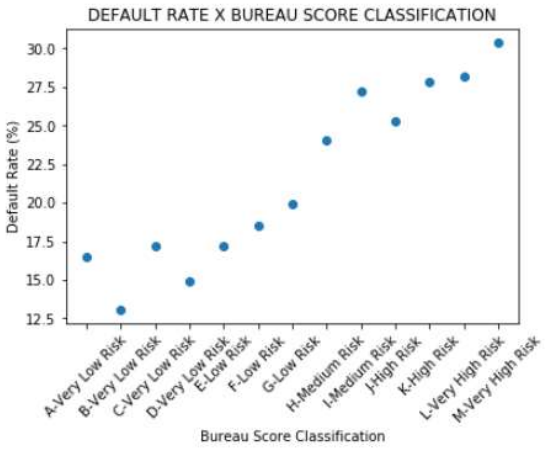
To answer these questions, I will plot the "Default Rate" X "BUREAU SCORE CLASSIFICATION". The column PERFORM_CNS_SCORE_DESCRIPTION is already separated into bins.

I will only consider the bins that starts with a letter, which corresponds to the Bureau's classification. The other bins, as stated earlier in this report, are merely codes, not bearing any sense when compared to actual credit scores. To make my job easier, I noticed that all bins that start with the letter "N" are used as codes, so I just need to list the bins where the description does not start with the letter "N", as seen in the first line of the code below.

The default rate for each bin is calculated using the buildDict function defined earlier (check the appendix), which is the second line of the code below.

```
# create a list of bins to be considered
score_bins = [bin for bin in sorted(df.PERFORM_CNS_SCORE_DESCRIPTION.unique()) if bin[0] != "N"]

# call dictionary building function
myDict = buildDict(df, score_bins, "PERFORM_CNS_SCORE_DESCRIPTION")

# plot decorations
plt.scatter(x=myDict.keys(), y=myDict.values())
plt.xticks(rotation=45)
plt.ylabel("Default Rate (%)")
plt.xlabel("Bureau Score Classification")
plt.title("DEFAULT RATE X BUREAU SCORE CLASSIFICATION")
plt.rcParams["figure.figsize"] = (6,4)
plt.show()
```

Which produces the following output:



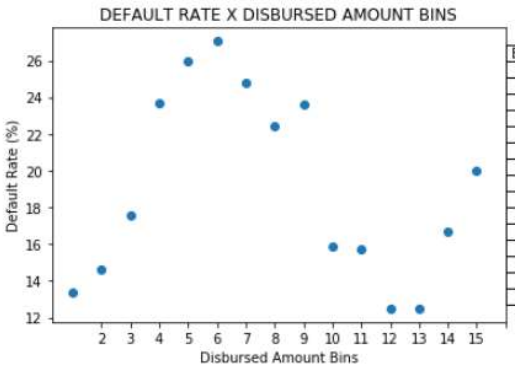
We can easily identify an upwards trend, as expected. It is interesting to notice that someone classified as Very High Risk is almost 20% more likely to default their loans, compared to someone classified as Very Low Risk.

Does the Default Rate Increase as the Disbursed Amount Increases?

To tackle this question, I will separate the Disbursed Amounts into bins. As there are outliers, I will disregard Disbursed Amounts greater than $200,000, otherwise several bins will have no data. Finally, I will follow the steps taken to answer the previous question.

```
# Import numpy
import numpy as np

# Create DataFrame with only Disbursed Amounts < $200,000
df_toPlot = df[df.DISBURSED_AMOUNT < 200000]

# Create the bins
df_toPlot.loc[:,'DisbBins'], lbBins = pd.cut(df_toPlot.loc[:, "DISBURSED_AMOUNT"], 15, labels=np.arange(1,16), retbins=True)

# Round lower Bound of the bins to have 2 decimal places
lbBins = ["{0:,.2f}".format(x) for x in lbBins]

# Call dictionary building function defined on previous cell
myDict = buildDict(df_toPlot, np.arange(1,16), "DisbBins")

# Prepare Scatter plot
plt.scatter(x=myDict.keys(), y=myDict.values())
plt.xticks(np.arange(1,16))
plt.ylabel("Default Rate (%)")
plt.xlabel("Disbursed Amount Bins")
plt.title("DEFAULT RATE X DISBURSED AMOUNT BINS")
plt.rcParams["figure.figsize"] = (6,4)

# Plot table with bins and the lower bounds for reference
the_table = plt.table(cellText=list(zip(myDict.keys(),lbBins)),
                      colLabels=["Bin", "Lower bound ($)"],
                      loc = 'right',
                      colWidths = [0.08,0.3])
the_table.auto_set_font_size(False)
the_table.set_fontsize(11)
plt.show()
```

Please note that I am also producing a table with the lower bounds of the bins for reference. Here is the output:



Bin	Lower bound ($)
1	13,136.32
2	25,565.20
3	37,810.40
4	50,055.60
5	62,300.80
6	74,546.00
7	86,791.20
8	99,036.40
9	111,281.60
10	123,526.80
11	135,772.00
12	148,017.20
13	160,262.40
14	172,507.60
15	184,752.80

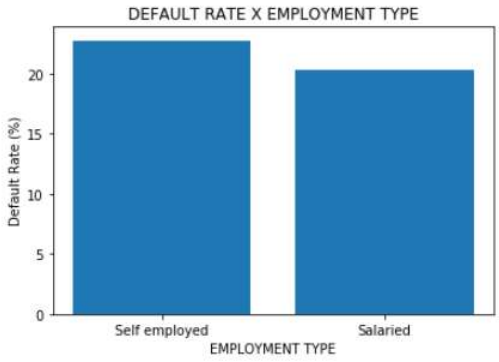
Contrary to my expectation, **the Default Rate does not have a linear relationship with the Disbursed Amount.**

What is the Average Default Rate by Employment Type, State and Employee ID?

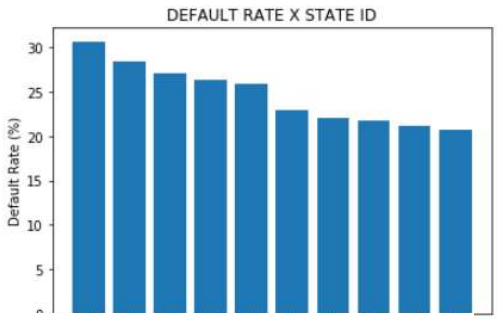
These questions can all be answered using the same logic and, for that reason, I defined the function “plotBarPlot” (check appendix). Once the function is defined, it suffices to call the function for the columns we want answers, as we can see in the code below:

```
In [53]:  # Call the function for 3 categorical columns
          for col in ["EMPLOYMENT_TYPE", "STATE_ID", "EMPLOYEE_CODE_ID"]:
              plotBarPlot(df, col)
              plt.rcParams["figure.figsize"] = (6,4)
```

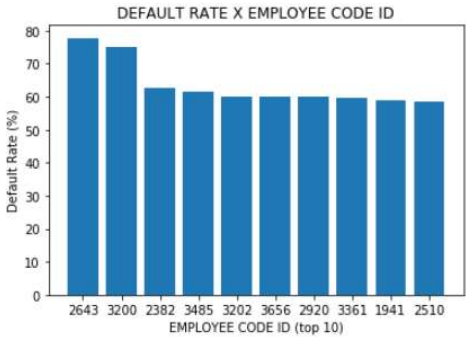
Which produces the following plots:



Contrary to my expectation, **there is not a significant difference on the Default Rate when we compare Salaried borrowers with Self-employed**. I imagined that irregular income from self-employed clients would cause them to default their loans more often then clients with regular income, however the data does not suggest this implication.



The analysis for State ID would be comparable to a Branch ID, for example. It can be very meaningful for the managers of a multi-branch financial institution to compare the performance of each branch or state where they operate. Maybe they should abide by different state rules and this graph might reflect it.



This is the most interesting plot in this exploratory section. it amazes me that there are **employees who have a Default Rate of almost 80%.** Remember that, for these three plots, each bin has at least 10 records, meaning that these are employees approved loans for at least 10 clients and their clients defaulted 80% of the time. Considering that the average Default Rate is around 20%, some measures should be taken regarding this issue: Training? Investigate the client and the employee? Improve data intake? Improve software?

Is the Age of the Client Relevant?

To answer this question, I will first create a column with the age of the borrower on the date of the disbursement. This column will be kept in the DataFrame df for later ML analysis, as opposed to df_slim, which is a slimmer version of the dataset.

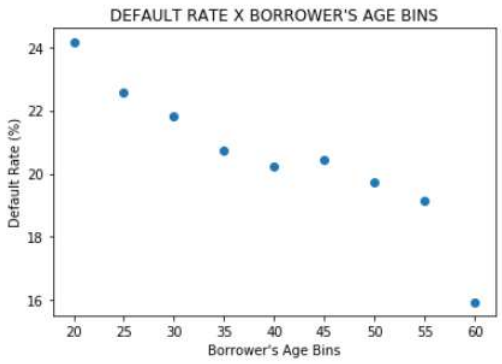
Here is the code to create the Age column:

```
df["AGE"] = (df["DISBURSAL_DATE"] - df["DATE_OF_BIRTH"])/np.timedelta64(1,'Y')
```

And here is the code to create the age bins and create a Scatter plot:

```
# Build Age Bins
df.loc[:,'ageBins'], lbBins = pd.cut(df.loc[:, "AGE"], bins = np.arange(20,70, 5), labels=np.arange(20,65, 5), retbins=True)

# Call dictionary building function
myDict = buildDict(df, np.arange(20,65, 5), "ageBins")

# Prepare Scatter plot
plt.scatter(x=myDict.keys(), y=myDict.values())
plt.ylabel("Default Rate (%)")
plt.xlabel("Borrower's Age Bins")
plt.title("DEFAULT RATE X BORROWER'S AGE BINS")
plt.rcParams["figure.figsize"] = (6,4)
plt.show()
```

Which outputs the following plot:



Even though the differences are not huge, we can see a downwards trend, meaning that **younger people are (a little) more likely to default their loans.**

It is also curious to see a **gap between the bins 55 and 60**. I checked the number of records that fit in the 60's bin and there are more than a thousand, which means that it's not only a few outliers that happened to fall in the same category: It means that **the whole bin is an outlier**!

How can we explain that? Maybe some states have more strict rules to lend money to people over 60? Or is it related to a fundamental psychological change that human beings go through at that age, fearing they would leave debt to their families on the event of their death? Although figuring that out would be fun, this in-depth analysis is out of the context of the project.

Last Datatype Conversion: From Dates to Floats

Now the last step is to convert the columns "DATE_OF_BIRTH" and "DISBURSAL_DATE" to an integer or float. There are several ways of doing it, but I will convert these dates to a float, representing the number of years passed since the earliest date (min(date)) of each column. A function will be handy.

```
# Define a function
def convertToYears(df, col):
    # For a DataFrame df, this function replaces a datetime column col with the number of years passed since
    # the min(date) of col, in float format
    minDate = min(df[col])
    df[col] = df[col].apply(lambda x: (x - minDate)/np.timedelta64(1,'Y'))

# Call the function for both columns from both DataFrames
for df_aux in [df, df_drop]:
    for col in ["DATE_OF_BIRTH" , "DISBURSAL_DATE"]:
        convertToYears(df_aux, col)
```

Data Preparation and Analysis Takeaways

This concludes the data preparation part of the project. We were able to clean the dataset, find some interesting facts and get two datasets ready to apply some Machine Learning Classification Methods. Let us proceed to the next section and find the best combination of ML Model X Dataset.

Applying Machine Learning Classification Models

The goal of this notebook is to apply some classification models to the two datasets previously prepared. As stated earlier, this report covers the content of two Jupyter Notebooks: “Data Preparation” and “ML Algorithms”. To transfer the data from one notebook to another I used the pandas “to_csv” and “read_csv” functions, meaning that I stored the two Dataframes in an external .csv file.

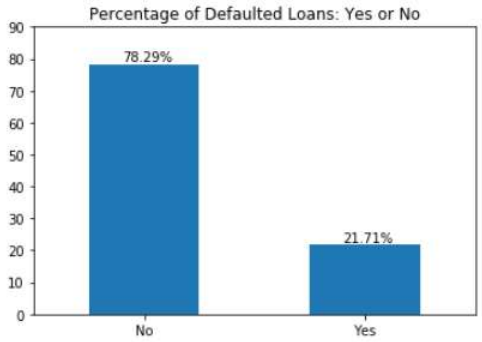
In this section I will load the datasets from the Data Preparation Notebook, generate dummy columns, split the datasets into training and testing, define a scoring benchmark, test several models, and elect a winner!

Imbalanced Data

First thing we need to do is to account for the fact that our Dataset is imbalanced, as far as the target column is concerned. We noticed that on the Data Analysis part of the previous Notebook, but we can do a quick bar chart to illustrate this fact:

```
# Bar plot with the number of occurencies of each label ("Yes" or "No") of the target column ("LOAN_DEFAULT")

# Import Pandas and matplotlib.pyplot
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Import raw dataset
df = pd.read_csv("train.csv")

# Plot
ax=plt.subplot(1, 1, 1)
(df.LOAN_DEFAULT.map({0: 'No', 1: 'Yes'}).value_counts()*100/len(df)).plot(kind="bar",
                                                                           title="Percentage of Defaulted Loans: Yes or No",
                                                                           rot = 0,
                                                                           ylim = [0,90])
for p in ax.patches:
    ax.annotate(str(round(p.get_height(), 2))+"%", ((p.get_x()+p.get_width()/2.-0.1), (p.get_height() + 1)))

plt.show()
```

Which yields the following bar chart:



As the data is imbalanced, I need to take that into account when splitting into train and test dataset. For that, I will use the module train_test_split from sklearn and, more speciffically, the stratify=y argument, so the proportions of labels are kept on both training and testing dataset.

Functions to Load and Split the Dataset

Below, I define a function that takes care of loading a dataset (specified by its name) and splits into train and test datasets

```python
# Import necessary Libraries
from sklearn.model_selection import train_test_split

# Define a function that loads the dataset, generates dummy columns and split into training and testing
def prepData(file_name, slim=False):

    # load datasets into pandas
    df = pd.read_csv(file_name)

    # Identify column "LOAN_DEFAULT" as target
    if("LOAN_DEFAULT" in df.columns):
        y = df.LOAN_DEFAULT
        df = df.drop(labels="LOAN_DEFAULT", axis=1)

    # Read the note on the cell below to understand why this part of the code is necessary
    # Remove columns ending in ID from the slim version (Branch ID, Manufacturer ID and State ID)
    if(slim==True):

        # Create a list with columns ending in ID
        ID_cols = [col for col in df.columns if col[-2:] == "ID"]

        #drop these columns
        df = df.drop(labels=ID_cols, axis=1)

    # Generate dummy columns on the categorical columns of df
    if(slim==True):
        df = pd.get_dummies(df, drop_first=True)
    else:
        df = pd.get_dummies(df)

    # Split into trainig and testing
    X_train, X_test, y_train, y_test = train_test_split(df, y, test_size=0.30, random_state=42, stratify=y, shuffle=True)

    return X_train, X_test, y_train, y_test
```

To better understand this function, we need to take the following into consideration:

After running this function for the first time, I realized that the data type of the columns ending in ID (employee_ID, branch_ID, etc.) were imported as integers. I used the following lines to convert to categorical datatype:

Create a list with columns ending in ID

ID_cols = [col for col in df.columns if col[-2:] == "ID"]

Convert to category

for col in ID_cols:

df[col] = df[col].astype('category')

Once I ran the pd.get_dummies() function, I received a memory error. I tried again with the argument "sparse=True" and the code ran for 20 minutes without resuming, forcing me to interrupt execution.

With that in mind, I did the following adaptations to the code:

- df_full --> Columns ending in ID will be kept as integers
- df_slim --> Columns ending in ID will be removed

However, as I carried on with testing a few models, I realized that I should try an even slimmer version of the dataset because df_slim was outperforming the full df in several models and some models were not converging when using the full dataset. For that reason, I defined the function prepDataSlimmer as follows:

```
# Function to remove more columns
def prepDataSlimmer(file_name, extreme=False, no_dummies=False):
    # extreme: When set to True, only the Bureau's Score description column is kept.
    # As a few models were not converging, I felt the need to leave the one column that I am sure that can be used to make
    # good predictions. This way I guarantee I have no noise in the data

    # no_dummies: Do not generate dummy columns, needed for comparing my model to the raw's file accuracy

    # load datasets into pandas
    df = pd.read_csv(file_name)

    # Identify column "LOAN_DEFAULT" as target
    if("LOAN_DEFAULT" in df.columns):
        y = df.LOAN_DEFAULT
        df = df.drop(labels="LOAN_DEFAULT", axis=1)

    # Remove columns ending in ID from the slim version (Branch ID, Manufacturer ID and State ID)
    # Create a list with columns ending in ID
    ID_cols = [col for col in df.columns if col[-2:] == "ID"]

    # Drop these columns
    df = df.drop(labels=ID_cols, axis=1)

    # List of columns to drop  -  for more noise reduction
    toDrop = ["LTV", "MOBILENO_AVL_FLAG", "AADHAR_FLAG", "PAN_FLAG", "VOTERID_FLAG", "DRIVING_FLAG", "PASSPORT_FLAG"]

    df = df.drop(labels=toDrop, axis=1)

    # Extreme - leave only the bureau's description, to test models' conversion
    if(extreme == True):
        df = df["PERFORM_CNS_SCORE_DESCRIPTION"]

    # Generate dummy columns on the categorical columns of df if no_dummies set to False
    if(no_dummies == False):
        df = pd.get_dummies(df, drop_first=True)

    # Split into trainig and testing
    X_train, X_test, y_train, y_test = train_test_split(df, y, test_size=0.30, random_state=0, stratify=y, shuffle=True)

    return X_train, X_test, y_train, y_test
```

Now I have 4 datasets to try: 2 generated by the function prepData using the full dataset and the slimmer version as arguments, and another 2 generated by prepDataSlimmer. Instead of creating 4 sets of X_train, X_test, y_train, y_test variables, one for each combination of “function X dataset”, I used the same variables and made the calls in separate cells, like shown below:

```
# Call prepData function for the slim dataset
X_train, X_test, y_train, y_test = prepData("Cleaned-slim.csv", True)
```

```
# Call prepData function for the full dataset
X_train, X_test, y_train, y_test = prepData("Cleaned-full.csv")
```

```
# call the function to split into train and test
# Load slim Dataset
X_train, X_test, y_train, y_test = prepDataSlimmer("Cleaned-slim.csv")
```

```
# call the function to split into train and test
# Load full Dataset
X_train, X_test, y_train, y_test = prepDataSlimmer("Cleaned-full.csv")
```

Then I just go back and forth: first I run one of the four cells to load the train and test variables, then I run the model. Then I repeat this process four times to each model.

The last thing we need to do before jumping in the models testing is a benchmark, which is the subject of the next section.

Benchmark

I want to beat two benchmarks: Sklearn's DummyClassifier and another one that I will define later. Accuracy will be measured by **F1 Score**, given that the classes are imbalanced. It accounts for both the precision and the recall. If we

measure accuracy with R squared, we could predict only 0 (no loan default) and the accuracy would be 78%, but my model would be useless as far as loan default detection is concerned. To help me with that, I created another function called printData, where the most relevant info about the model's performance is printed.

```
from sklearn.metrics import f1_score, confusion_matrix, accuracy_score

# Function that prints the relevant data about a model
def printData(clf, X_test, y_test):
    y_pred = clf.predict(X_test)
    print("total predicted Defaults: ",sum(y_pred))
    print("total Defaults on test: ", sum(y_test))
    print("R Squared: ", accuracy_score(y_test, y_pred))
    print("F1Score: ", f1_score(y_test, y_pred))
    cm = confusion_matrix(y_test, y_pred)
    print(cm)
    print("Recall: ", cm[1,1]/(cm[1,0]+cm[1,1]))
```

First, let us start with the dummy model. The goal is to beat the F1 score. Below, I build the benchmark model:

```
from sklearn.dummy import DummyClassifier

# Define and fit dummy classifier
dummy_clf = DummyClassifier(random_state=0).fit(X_train, y_train)

# Print relevant metrics
printData(dummy_clf, X_test, y_test)
```

Which yields the following output:

```
total predicted Defaults:  14622
total Defaults on test:  14690
R Squared:  0.6611281929990539
F1Score:  0.2179312227074236
[[41530 11428]
 [11496  3194]]
Recall:  0.21742682096664398
```

The score to beat is F1Score = 0.218. However, in addition to beating this metric, **I want to define a more challenging benchmark: the current model's performance!**

Yes, the current model! Allow me to elaborate: the raw data comes from actual loans, which means that these records are the current model's prediction = 0 (client will not default). We do not have the information about how many clients were rejected, so our **challenging benchmark will be defined by the raw data and compared to my model's predictions = 0.**

Models Exploration

I ran a few models and summarized the most relevant metrics on the table below. Please refer to the appendix for the full code used for each model.

ID	Model Family	Description (*1)	F1 Score	Recall (*2)
1	Logistic Regression	max_iter=300, solver='sag'	0 (no prediction =1)	0
2	kNeighbors	n_neighbors = 5	0.154	0.107
4	Perceptron	(tol=1e-3, random_state=42, class_weight={0:1,1:3}	0.333	0.629
5.1	Linear Discriminant Analysis (*3)	'priors': [0.4, 0.5] NOTE: true priors: [0.78, 0.22]	0.379	0.901
5.2	Linear Discriminat Analysis	No tweaking	0.007	0.004
6	Random Forest Classifier	n_estimators=100	0.061	0.033

7	Multi-layer Perceptron classifier	max_iter=300	0.000	0.000
8	Gradient Boosting Classifier	No tweaking	0.006	0.003
9.1	Support Vector Machine	gridCV (to define the gamma parameter) and StandardScaler()	Execution interrupted (> 20 minutes)	
9.2	Support Vector Machine	gridCV only	Execution interrupted (> 20 minutes)	
9.3	Support Vector Machine	StandardScaler() only	Execution interrupted (> 20 minutes)	
3	Stochastic Gradient Descent Classifier	class_weight='balanced',	0.377	0.657

(*1) – I used gridSearchCV to define the parameters. The scores presented uses the best configuration found for each model.
(*2) – Recall is the hit rate related to the class = 1 (loan default): out of all the customers who defaulted their loans, how many did my model identify?
(*3) – Artificial priors. Overfitting the testing data, cannot be used.
General note 1: once I defined the best dataset (slimmer version of df_slim), I used that dataset to generate the scores presented on this table. More on the "best dataset" later.
General note 2: models with F1 scores below the "Sklearn Dummy Model Benchmark" are marked in red.

By analysing this table, I noticed that:

- Considering the models' "out-of-the-box" versions, SGDClassifier turns out to have the best F1 score.
- **The dummy model's F1 score was already beaten by a lot** (0.37 > 0.21), even before dataset selection and parameter tweaking.

Now let us improve this model. I want the following answers:

- Which dataset is performing better? Full, slim, or slimmer (using function prepDataSlimmer)?
- What parameters can we tweak in the model to make better predictions?
- And finally: **Is this model performing better than the current model (challenging benchmark)**?

Dataset Selection

Now that I have chosen the best model, let us see how the 4 datasets perform. I prepared the table below with the results:

ID	Model Family	Dataset used	F1 Score	Recall (*2)
3.1	Stochastic Gradient Descent Classifier	- Full Dataset - function used: prepData	0.379	0.666
3.2	Stochastic Gradient Descent Classifier	- Full Dataset - function used: prepDataSlim	0.378	0.657
3.3	Stochastic Gradient Descent Classifier	- Slim Dataset - function used: prepData	0.388	0.677
3.4	Stochastic Gradient Descent Classifier	- Slim Dataset - function used: prepDataSlim	0.380	0.752

To choose between the last two models, I need to decide what I want from them. The third model has a better F1 score. However, the fourth model has a 75% Recall rate, which, for our purposes, is better. I want to have a decent accuracy with a strong ability to identify potential defaults, so it makes sense to sacrifice the precision to have a significant increase in the recall rate.

With that in mind, the **best result is achieved when I used the Slim dataset without the columns LTV and the ones marked as flags**. From now on, I am using this dataset.

Next, we need to see if we can improve this score by tweaking a few parameters.

Improving the Best Model: Parameters Tweaking

Penalty

First, let us inspect the penalty. I noticed that the models performed better when I used the slimmer version of the dataset. With that in mind, dimensionality reduction may help the model eliminate some features, so using an elasticnet will give the model the possibility of using L1 penalty and reduce a few dimensions.

Here follows the code:

```
# SGDClassifier Tweaking: Penalty
parameters = {'l1_ratio':np.arange(0.0, 1.0, 0.1)}
cl = SGDClassifier(class_weight='balanced', penalty='elasticnet', random_state=42)
cl_grid = GridSearchCV(cl, parameters, scoring='f1', cv=5)
clf = make_pipeline(StandardScaler(), cl_grid).fit(X_train, y_train)
print(cl_grid.best_params_)
printData(clf, X_test, y_test)
```

Which outputs:

```
{'l1_ratio': 0.7000000000000001}
total predicted Defaults:  37471
total Defaults on test:  14690
R Squared:  0.5205327578051088
F1Score:  0.3781752650447652
[[25350 27608]
 [ 4827  9863]]
Recall:  0.6714091218515997
```

For some reason, GridSearchCV did not choose the best model because using the default parameter L2 (meaning L1_ratio = 0) yields a better F1 score (and Recall rate). No success here, **I am keeping the default**.

Alpha

Now let us investigate the parameter alpha, which regulates the "regularization power".

```
# SGDClassifier Tweaking: Alpha
parameters = {'alpha':[0.001, 0.01, 0.1]}
cl = SGDClassifier(class_weight='balanced', random_state=42)
cl_grid = GridSearchCV(cl, parameters, scoring='f1', cv=3)
clf = make_pipeline(StandardScaler(), cl_grid).fit(X_train, y_train)
print(cl_grid.best_params_)
printData(clf, X_test, y_test)
```

Which outputs:

```
{'alpha': 0.01}
total predicted Defaults:  40360
total Defaults on test:  14690
R Squared:  0.49866958372753073
F1Score:  0.3839418710263397
[[23166 29792]
 [ 4122 10568]]
Recall:  0.7194009530292717
```

Even though there is a slight improvement on the F1 score, this model yields a worse recall rate. I rather stick with the default model (alpha = 0.0001) as the trade-off is not great. Note: I tested more values, the screenshot shows the last try.

As far as parameters tweaking is concerned, the only parameter that improved the F1 score, compared to the "out-of-the-box" version, is the class_weight="balanced".

Current Model X My Model – Challenging Benchmark

Overall Default Rate Prediction Accuracy

We defined the best dataset and the best parameters. Now we need to check if our predictions are better than the current model's predictions. Here is how I will calculate it:

Considering all the zeros in my model (loans predicted as non-default), my precision would be 3637/(3637+20511) ~ **15%** (numbers extracted from the confusion matrix), **significantly lower than the 21.7% default rate we observe in the raw data.** My model makes 50 % error when it comes to identifying non-default customers, but is 75% accurate to identify customers likely to default. We do not have these metrics regarding the model that originated the raw data, so let us be happy with what we got.

Default Rate Prediction Accuracy by Credit Score Bins

For the last part of this section, I want to see how the two models behave for each credit score bin. I want to see **how accurate my model is for each score bin when my prediction is 0** (non-default loan). The goal here is to have a plot similar to what I produced in the "Data Preparation" Notebook, but now with two curves: one for the raw data, another one for my predictions. And may the best one win!

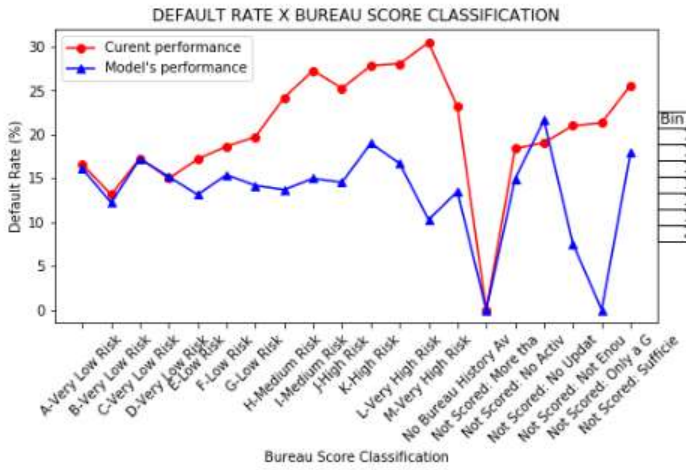
To help me conceive this plot, I will define the following function, which is a simpler version of buildDict defined in the appendix:

```
# First, let's define a function to build a dictionary where the keys = bin and values = Default Rate
def buildDict(df, bins):
# Arguments description:
#   - df - DataFrame that contains the data
#   - bins - how the data is grouped. Very important to note: only the first 20 characters of the score bin description
# is kept, otherwise the plot gets messy with too much text

    #create an empty dictionary
    aDict = {}

    #loop through bins
    for bin in bins:
        # For each bin, calculate the total defaults / total data points in that bin
        aDict[bin[:20]] = 100*(sum(df[df['PERFORM_CNS_SCORE_DESCRIPTION'] == bin]["LOAN_DEFAULT"])/
                               df[df['PERFORM_CNS_SCORE_DESCRIPTION'] == bin]["LOAN_DEFAULT"].count())

    return aDict
```

Now we can leverage this function to generate the plot with the two curves, as we can see on the screenshot below:

```
# Now let's build the plot with the two curves. The whole raw dataset is used to plot the ACTUAL default rate per
# bin, whereas the PREDICTION uses only the records where prediction = 0.

import matplotlib.pyplot as plt

# Load the raw data
df_actual = pd.read_csv("train.csv", header=0, index_col=0)

# Define the bins from PERFORM_CNS_SCORE_DESCRIPTION column
score_bins = [bin for bin in sorted(df_actual.PERFORM_CNS_SCORE_DESCRIPTION.unique())]

# Run the function to define X_test again, now without the pd.get_dummies (otherwise the column
# PERFORM_CNS_SCORE_DESCRIPTION won't exist, only the "dummy versions" of it)
X_train, X_test, y_train, y_test = prepDataSlimmer("Cleaned-slim.csv", extreme=False, no_dummies=True)

# Add columns "PREDICTION" and "LOAN_DEFAULT" (which is y_test) to X_test
X_test["PREDICTION"] = y_pred
X_test["LOAN_DEFAULT"] = y_test

# Keep the records where prediction is 0 (no loan default)
df_pred = X_test[X_test.PREDICTION == 0]

# Build dictionary for actual values and predictions values
actualDict = buildDict(df_actual, score_bins)
predDict = buildDict(df_pred, score_bins)

# Built plot
fig=plt.figure()
ax=fig.add_subplot(111)

# Curves
ax.plot(list(actualDict.keys()), list(actualDict.values()),c='r',ls='-', marker="o", label = 'Curent performance')
ax.plot(list(actualDict.keys()), list(predDict.values()),c='b',ls='-', marker="^", label = 'Model\'s performance')

# Decorations
plt.xticks(rotation=45)
plt.ylabel("Default Rate (%)")
plt.xlabel("Bureau Score Classification")
plt.title("DEFAULT RATE X BUREAU SCORE CLASSIFICATION")
plt.rcParams["figure.figsize"] = (8,4)
plt.legend(loc=2)

# Table with full bin description
# Create a list of bin description that got truncated
full_descr = [x for x in sorted(df_actual.PERFORM_CNS_SCORE_DESCRIPTION.unique()) if(x[0] == 'N')]

# Create table
the_table = plt.table(cellText=list(zip(np.arange(14,21), full_descr)),
                      colLabels=["Bin ID","Full bin description"],
                      loc = 'right',
                      colWidths = [0.08,0.8])
the_table.auto_set_font_size(False)
the_table.set_fontsize(11)
plt.show()
```

Which yields the following plot:



Bin ID	Full bin description
14	No Bureau History Available
15	Not Scored: More than 50 active Accounts found
16	Not Scored: No Activity seen on the customer (Inactive)
17	Not Scored: No Updates available in last 36 months
18	Not Scored: Not Enough Info available on the customer
19	Not Scored: Only a Guarantor
20	Not Scored: Sufficient History Not Available

As far as estimating the probability of a loan default is concerned, it is very easy to notice that **my model outperforms the current model in almost all ranges of credit score**. Even when the customer is not scored (please check the several reasons why a customer may not be scored on the table to the right of the plot), **following my predictions would result in a lower default rate**.

Models Exploration key Takeaways

From the technical perspective, it comes to me as a surprise that the slimmer version of the dataset, with just a few features, would yield the best prediction. As our personal computers have sufficient processing power nowadays, we tend to use all the features we have and, in this particular case, this proved to be a bad strategy.

As a final note, here is what I would recommend: If a customer asks for a loan, first run my model. If the prediction is zero, maybe a lower rate could be applied because the probability of a loan default is very low (15%). However, as noted when the confusion matrix was presented, this model makes a considerable amount of false negative predictions. Even though it detects 75% of true negatives, only one out of three default predictions are true negatives. That being said, further investigation should be done when the prediction is 1, in case the number of clients being rejected is considered "bad for the business". Maybe another model could be generated for these predictions, along with a "Loan Interest Rate X Default Risk" analysis. Needless to say, this is out of the scope of this project.

Conclusion

As promised, I went all the way from loading the raw dataset to proposing a good classification model to predict whether a customer is likely to default their loans, based on client's information, loan information and bureau's Credit Score information. All the data preparation process is clearly documented, as well as the exploration of different Machine Learning Classification models.

The model proposed is better than Scikit Learn's Dummy Model, but this was not enough to prove that this model is good. A more "challenging" benchmark was proposed, which consists of comparing the model to the existing model, the one that generated the raw file. Fortunately, the model proposed has a better predicting accuracy for every type of client, according to the bureau's classification: from the most "risky" (low score) to the less risky (high score).

As stated earlier, before granting a loan, the company should first run my model. In case the prediction is 0 (not likely to default), the company could even propose a better rate for the client, given that the probability of loan default is around 15%. On the flip side, if the prediction is 1, further analysis would be required, according to the company's risk appetite, as well as some other business metrics, which is not the scope of this project.

Appendix

Relevant codes

Please find some relevant codes that are not listed in the body of report.

Function “buildDict”

```
# First, let's define a function to build a dictionary where the keys = bin and values = Default Rate
def buildDict(df, bins, col_name, gt10=False):
# Arguments description:
#  - df - DataFrame that contains the data
#  - bins - how the data is grouped
#  - col_name - Column from where the bins will be extracted
#  - gt10 - Boolean. If True, only bins that have more than 10 data points will be taken into consideration

    #create an empty dictionary
    aDict = {}

    #loop through bins
    for bin in bins:
        if(gt10==False):
            # For each bin, forced to be a string (otherwise employee code is interpreted as number),
            # I calculate the total defaults / total data points in that bin
            aDict[str(bin)] = 100*(sum(df[df[col_name] == bin]["LOAN_DEFAULT"])/
                                   df[df[col_name] == bin]["LOAN_DEFAULT"].count())
        else:
            # Only bins with more than 10 data points will be considered
            if(df[df[col_name] == bin]["LOAN_DEFAULT"].count()>10):
                aDict[str(bin)] = 100*(sum(df[df[col_name] == bin]["LOAN_DEFAULT"])/
                                   df[df[col_name] == bin]["LOAN_DEFAULT"].count())

    return aDict
```

Function “plotBarPlot

```
def plotBarPlot(df, col):
# For any categorical column, this function plots, in descending order, the top 10 Default Rate for each bin

    # The bins are the unique values of the categorical column
    bins = sorted(df[col].unique())

    # Call the function defined earlier that builds the dictionary (key=bin, value=Default Rate)
    myDict = buildDict(df, bins, col, True)

    # Create a DataFrame out of the dictionary to make ordering simpler
    df_aux = pd.DataFrame(data=myDict.values(), index=myDict.keys(), columns=["Default_Rate"])

    # Boolean to verify if there are more than 10 categories
    mt10 = False
    # Slice the top 10
    if(len(df_aux)>10):
        df_aux = df_aux.sort_values(by=['Default_Rate'], ascending=False)[:10]
        mt10 = True
    else:
        df_aux = df_aux.sort_values(by=['Default_Rate'], ascending=False)

    # Prepare bar plot
    plt.bar(x=df_aux.index, height=df_aux.Default_Rate)
    plt.ylabel("Default Rate (%)")
    if(mt10==True):
        plt.xlabel(col.replace("_", " ") + " (top 10)")
    else:
        plt.xlabel(col.replace("_", " "))
    plt.title("DEFAULT RATE X " + col.replace("_", " "))
    plt.show()
```

Models tested

Note that these cells have some commented code left on purpose. Each of these cells were ran a few times, with different parameters. For the sake of simplicity, only the best versions of each try are shown below.

1-Logistic Regression

```
# Trial 1 - grid search CV with logistic regression - Poor performance or did not converge
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

#parameters = {'solver':('newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga')}
clf = LogisticRegression(random_state=42, max_iter=300, solver='sag').fit(X_train, y_train)
#clf = GridSearchCV(cl, parameters, cv=3).fit(X_train, y_train)

#y_pred

# y_pred = clf.predict(X_test)
# print(sum(y_pred))
# #result = 0 for both extreme = True or False

#y_prob
#y_prob = clf.predict_proba(X_test)
# print(np.average(y_prob[:,1])
#print(np.sort(y_prob[:,1])[-10:])
#print(np.count_nonzero(y_prob[y_prob[:,1] > 0.2]))

#print(clf.best_params_)
# print(sum(clf.predict(X_test)))
# print(clf.score(X_test, y_test))

printData(clf,X_test, y_test )
```

```
total predicted Defaults:  0
total Defaults on test:  14690
R Squared:  0.7828464995269631
F1Score:  0.0
[[52958     0]
 [14690     0]]
Recall:  0.0
```

2-KNeighbors

```
#trial 2 - K neighbors  -  very poor performance
from sklearn.neighbors import KNeighborsClassifier

#parameters = {'n_neighbors':np.arange(3, 7, 2)}
#, 'weights':('uniform', 'distance')
#cl = KNeighborsClassifier(n_neighbors=5)
clf = KNeighborsClassifier(n_neighbors=5).fit(X_train, y_train)
#clf = GridSearchCV(cl, parameters, cv=3).fit(X_train, y_train)

#print(clf.best_params_)
# y_pred = clf.predict(X_test)
# print(sum(y_pred))
# print(sum(y_test))
# print(clf.score(X_test, y_test))
# print(f1_score(y_test, y_pred))
printData(clf, X_test, y_test)
```

```
total predicted Defaults:  5801
total Defaults on test:  14690
R Squared:  0.7436583491012299
F1Score:  0.15372602606022157
[[48732  4226]
 [13115  1575]]
Recall:  0.10721579305650102
```

4-Perceptron

```
#trial 4 - Perceptron  -  Good performance
from sklearn.linear_model import Perceptron

clf = Perceptron(tol=1e-3, random_state=42, class_weight={0:1,1:3}).fit(X_train, y_train)
#penalty = 'elasticnet'
printData(clf, X_test, y_test)
```

```
total predicted Defaults:  40831
total Defaults on test:  14690
R Squared:  0.45235631504257334
F1Score:  0.3327389636353812
[[21364 31594]
 [ 5453  9237]]
Recall:  0.6287950987066031
```

5-Linear Discriminant Analysis

```
#trial 5.1 - Linear Discriminat Analysis - prior tweaking -  DID NOT RUN FOR FULL DATASET - "VARIABLES ARE COLLINEAR"
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
parameters = {'priors':[[x,1-x] for x in np.arange(0.2, 0.5, 0.1)]}

cl = LinearDiscriminantAnalysis()
clf = GridSearchCV(cl, parameters, scoring='f1', cv=3).fit(X_train, y_train)

print(clf.best_params_)
printData(clf, X_test, y_test)
```

```
{'priors': [0.4000000000000001, 0.5999999999999999]}
total predicted Defaults:  55061
total Defaults on test:  14690
R Squared:  0.3603210737937559
F1Score:  0.3796074608249344
[[11136 41822]
 [ 1451 13239]]
Recall:  0.9012253233492171
```

```
#trial 5.2 - Linear Discriminat Analysis  -  No priors tweaking
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

clf = LinearDiscriminantAnalysis().fit(X_train, y_train)
print(clf.priors_)

printData(clf, X_test, y_test)
```

```
[0.78284393 0.21715607]
total predicted Defaults:  135
total Defaults on test:  14690
R Squared:  0.7824473746452223
F1Score:  0.007284991568296796
[[52877    81]
 [14636    54]]
Recall:  0.0036759700476514637
```

6-Random Forest Classifier

```
# Trial 6 - Random Forest Classifier  -  Very poor performance, when ran with max_depth = 3
# I had no prediction for the positive class
from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier(random_state=0, n_estimators=100).fit(X_train, y_train)

printData(clf, X_test, y_test)
```

```
total predicted Defaults:  1274
total Defaults on test:  14690
R Squared:  0.7784117786187322
F1Score:  0.06101227762465548
[[52171   787]
 [14203   487]]
Recall:  0.03315180394826413
```

,

7-Multi Layer Perceptron Classifier

```
# Trial 7 -  Multi-Layer Perceptron classifier  -  Very poor performance
from sklearn.neural_network import MLPClassifier

clf = MLPClassifier(random_state=42, max_iter=300).fit(X_train, y_train)

printData(clf, X_test, y_test)
```

```
total predicted Defaults:  3
total Defaults on test:  14690
R Squared:  0.7828317171239356
F1Score:  0.00013611924045463825
[[52956     2]
 [14689     1]]
Recall:  6.807351940095303e-05
```

8-Gradient Boosting Classifier

```
# Trial 8 -  Gradient Boosting Classifier  -  Very poor performance
from sklearn.ensemble import GradientBoostingClassifier

clf = GradientBoostingClassifier(random_state=42).fit(X_train, y_train)

printData(clf, X_test, y_test)
```

```
total predicted Defaults:  80
total Defaults on test:  14690
R Squared:  0.7829056291390728
F1Score:  0.005687203791469195
[[52920    38]
 [14648    42]]
Recall:  0.0028590878148400272
```

9-Support Vector Machine

```
# Trial 9.1 - Support Vector Machine,gridCV - Took too long, interrupted.
from sklearn.svm import SVC

parameters = {'gamma':['scale', 'auto']}
cl = SVC()
clf = GridSearchCV(cl, parameters, scoring='f1', cv=3).fit(X_train, y_train)

print(clf.best_params_)
printData(clf, X_test, y_test)
```

```
# Trial 9.2 - Support Vector Machine,gridCV and Scaler - Took too long, interrupted.
from sklearn.svm import SVC

parameters = {'gamma':['scale', 'auto']}
cl = SVC(class_weight='balanced')  # or try different penalty parameters C
cl_grid = GridSearchCV(cl, parameters, scoring='f1', cv=3)
clf = make_pipeline(StandardScaler(), cl_grid).fit(X_train, y_train)

print(clf.best_params_)
printData(clf, X_test, y_test)
```

```
# Trial 9.3 - Support Vector Machine, Scaler - Took too long, interrupted.
from sklearn.svm import SVC

cl = SVC(class_weight='balanced')  # or try different penalty parameters C
clf = make_pipeline(StandardScaler(), cl).fit(X_train, y_train)

print(clf.best_params_)
printData(clf, X_test, y_test)
```

Columns description

Column Name	Description
UniqueID	Identifier for customers
loan_default	Payment default in the first EMI on due date
disbursed_amount	Amount of Loan disbursed
asset_cost	Cost of the Asset
ltv	Loan to Value of the asset
branch_id	Branch where the loan was disbursed
supplier_id	Vehicle Dealer where the loan was disbursed
manufacturer_id	Vehicle manufacturer(Hero, Honda, TVS etc.)
Current_pincode	Current pincode of the customer
Date.of.Birth	Date of birth of the customer
Employment.Type	Employment Type of the customer (Salaried/Self Employed)
DisbursalDate	Date of disbursement
State_ID	State of disbursement
Employee_code_ID	Employee of the organization who logged the disbursement
MobileNo_Avl_Flag	if Mobile no. was shared by the customer then flagged as 1
Aadhar_flag	if aadhar was shared by the customer then flagged as 1
PAN_flag	if pan was shared by the customer then flagged as 1
VoterID_flag	if voter was shared by the customer then flagged as 1

Driving_flag	if DL was shared by the customer then flagged as 1
Passport_flag	if passport was shared by the customer then flagged as 1
PERFORM_CNS.SCORE	Bureau Score
PERFORM_CNS.SCORE.DESCRIPTION	Bureau score description
PRI.NO.OF.ACCTS	count of total loans taken by the customer at the time of disbursement
PRI.ACTIVE.ACCTS	count of active loans taken by the customer at the time of disbursement
PRI.OVERDUE.ACCTS	count of default accounts at the time of disbursement
PRI.CURRENT.BALANCE	total Principal outstanding amount of the active loans at the time of disbursement
PRI.SANCTIONED.AMOUNT	total amount that was sanctioned for all the loans at the time of disbursement
PRI.DISBURSED.AMOUNT	total amount that was disbursed for all the loans at the time of disbursement
SEC.NO.OF.ACCTS	count of total loans taken by the customer at the time of disbursement
SEC.ACTIVE.ACCTS	count of active loans taken by the customer at the time of disbursement
SEC.OVERDUE.ACCTS	count of default accounts at the time of disbursement
SEC.CURRENT.BALANCE	total Principal outstanding amount of the active loans at the time of disbursement
SEC.SANCTIONED.AMOUNT	total amount that was sanctioned for all the loans at the time of disbursement
SEC.DISBURSED.AMOUNT	total amount that was disbursed for all the loans at the time of disbursement
PRIMARY.INSTAL.AMT	EMI Amount of the primary loan
SEC.INSTAL.AMT	EMI Amount of the secondary loan
NEW.ACCTS.IN.LAST.SIX.MONTHS	New loans taken by the customer in last 6 months before the disbursment
DELINQUENT.ACCTS.IN.LAST.SIX.MONTHS	Loans defaulted in the last 6 months
AVERAGE.ACCT.AGE	Average loan tenure
CREDIT.HISTORY.LENGTH	Time since first loan
NO.OF_INQUIRIES	Enquries done by the customer for loans