Project Report

Michael Impey - E10569412 - October 2021

Title: *"Repay or not Repay that is the question?"*

This is my submission to UCD, to demonstrate how I thought and put course concepts, course learning into practice. The course was titled "Specialist Certificate in Data Analytics Essentials".

The associated files are saved to GitHub on the following GitHub URL:

https://github.com/UCDPA-*E10569412*/Michael_2021_Specialist_Cert_Data.A

Table of Contents

Abstract

The project attempts to build a supervised machine learning classifier model that will predict if a loan is likely to be repaid or defaulted on. The data file was downloaded from Kaggle.com using the following link.

https://www.kaggle.com/ajay1735/hmeq-data

To fulfil the requirements of the course report, I have included examples and project code snippets that I used. Where the project did not specifically request something that was required for grading, I have submitted an example and code in appendix. These items include the API example and an example of using Regex.

Introduction

The Inspiration for this project is to predict clients who might default on their loans. The story was the consumer credit department of a bank wants to automate the decision-making process for approval of home equity lines of credit. A model was built to use a machine learning algorithm to complete this task. The model will be a supervised classifier model, as it is a binary target decision.

The decision is to determine if the loan is likely to be repaid or defaulted on. I have written the report to bring the reader along the various steps involved. I have created a flow chart to visualize this, chart 1. The project starts by reviewing the bank loan data-set, then reviews the project problem. Then the reader will be brought through the various steps of building the data-set and modelling. Finally, I will discuss some of the insight gained when reviewing the data and model.

Data-set

The data file is from https://www.kaggle.com/ajay1735/hmeq-data. The data-set (HMEQ) contains loan performance information for 5,960 recent home equity loans. The target is a binary variable indicating whether an applicant eventually repaid or defaulted on the loan. Loan default occurred in 1,189 cases (20%). For each applicant, 12 input feature variables were recorded.

I chose this data-set as it looked interesting to me as I am currently applying for a mortgage. I was initially interested in the Titanic data-set but this is heavily covered by other data analyst reports and I liked the thought of a similar challenge but in a different context. I was interested to see can a machine learning model actually be used to predict an outcome and could I configure it to do it successfully. I would like to follow up on this project to see can I use machine learning, to predict if a stock price is more probable to go up or down (Classification) as opposed to a target price (Linear Regression).

The legend for the original data-set is as follows:

- BAD: 1 = client defaulted on loan 0 = loan repaid
- LOAN = Amount of the loan request
- MORTDUE = Amount due on existing mortgage
- VALUE = Value of current property
- REASON = DebtCon = debt consolidation / HomeImp = home improvement
- JOB = Six occupational categories
- YOJ = Years at present job
- DEROG = Number of major derogatory reports
- DELINQ = Number of delinquent credit lines
- CLAGE = Age of oldest trade line in months
- NINQ = Number of recent credit lines
- CLNO = Number of credit lines
- DEBTINC = Debt-to-income ratio

I changed the legend of the original data to bring more meaning to the features headers and I added an “ID” column. I modified the original data-frame slightly adding in a “ID” column and splitting into two files so that I could demonstrate a merge function which is asked for in the project report. This detail and more will be covered below in the implementation process.

Implementation Process

Below is an overview of the project steps taken, table 1. I will include some detail on what the step entails. After this initial description, I will detail each step on its own. The outstanding elements required in report but not covered in the project are included in the appendices.

You will notice that I like to load the data-fame needed for the step at the start of a program and save the outputs at the end of the step. I did this as it was very cumbersome and time consuming to run the whole project as one whole program and it also allowed me to test the impact of changes on one step at a time before moving onto the next step. It also facilitated going back on earlier steps, isolating and modifying.

I used functions where possible and where they made sense. While this code was for this project I also wanted to make it re-usable and to be the foundation for other projects I intend to work on. For example step 3 could be used by any supervised classifier project where the user wants to run data through a number of classifier machine learning models.

I also created a pause() function, so that I could pause the program as it cycled through the various steps. This gave me time to review the program output, as sometimes it can loop through very quick and you end up missing the displayed output. It is implemented through a simple input command to proceed. Simple and effective.

```
def pause():
    '"used to pause program to view output"'
    input('===> Press Return to Continue Program ?')
```

The plan when writing this code was to build a program that builds on the previous step. I have attached a flow diagram in chart 1 below of how the steps interact in more detail. Submitted with this report is a more comprehensive detail of what each step contains in table 1. The project starts with perhaps the most important step. The problem statement, “What problem are we trying to solve?” then I was essentially building, cleaning data and testing models until I was at a stage were I was happy enough with the model. Then it was time to test on the validation data-set to evaluate the models predictive ability. I was constantly gaining insights as I went through the project and they are recorded in the report as they occurred.

Project “Steps”.

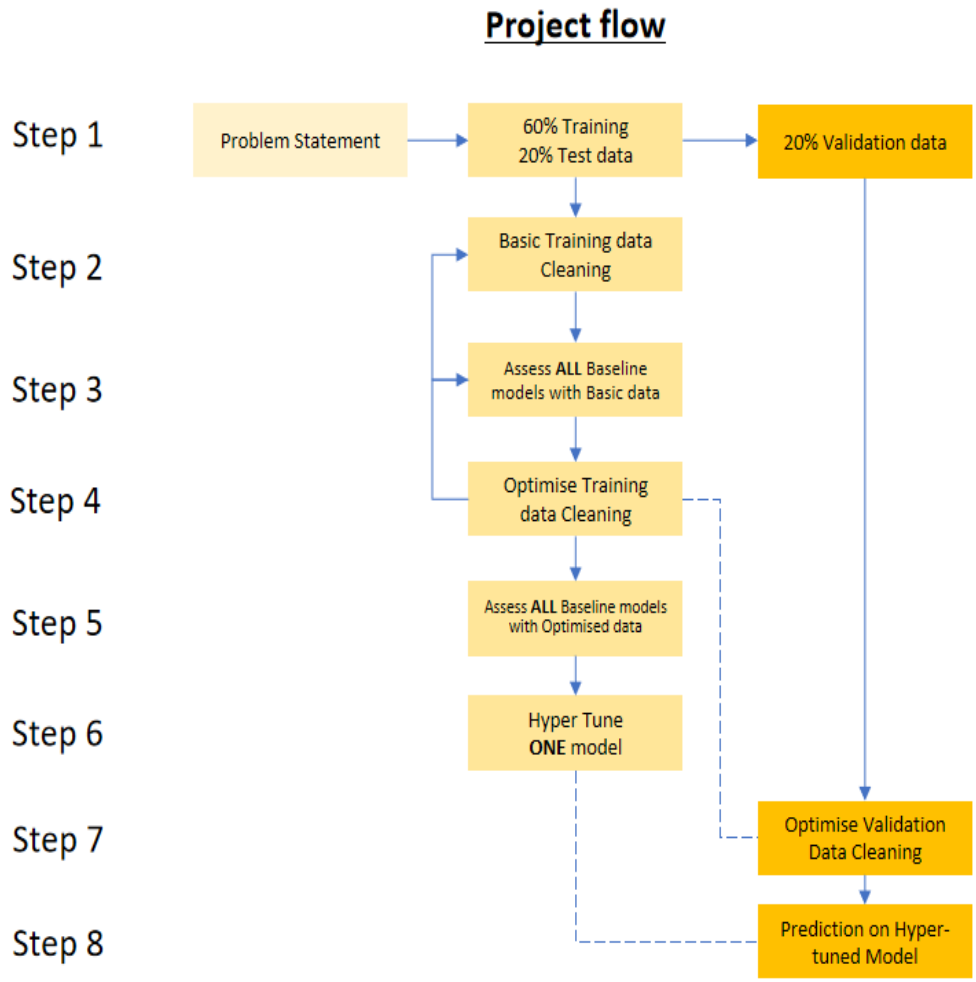
Table 1 - project steps

Step	*Title*	*Overview*
intro	*Project_Problem*	*Preliminary step. What is the problem the project is trying to answer and what data do we need answer it.*
1	*Data_Gathering*	*Import CSV data, Merge data-frames. I randomly split the data to create a Train, Test Data-set and Validation Data-set*
2	*Basic_Data_Cleaning*	*Perform EDA, basic data cleaning so that we can create basic data to run through several models to get initial model performance results on the Training data-set*
3	*Baseline_Model_Testing*	*Run a number of models on the basic cleaned data*
4	*Optimise_Data_Cleaning*	*Optimize data cleaning so that we can run through the model again to get final performance results*
5	*Tune_Model_and Select*	*Run a number of models on the Optimized cleaned data. From this trial I selected a model.*
6	*HyperTune_model*	*On the selected model, Hyper-tune model and assess performance.*
7	*Validation_Data_Cleaning*	*Perform basic and Optimized data cleaning on the Validation data set*
8	*Model_Predictions*	*Perform prediction using the Validation training set on the hyper-tuned model and asses against several classifier evaluation metrics*

Project flow chart:

This is an overview of the project flow. There is more detail in the file attached called **"Project Steps Template.xls"**

Chart 1 - Flow diagram of project flow



Project_Problem

This step was very important to the project's success. It involved understanding what problem the project was trying to answer. The project is trying to predict the clients that would default on their loans. To do this I had to hypothesize what information would be required and compare this to what was available. It was not possible to get information on all the data I needed as I was limited to the data-set from Kaggle.

If I was stating from my own data-set, some of the features that I would expect to need to include the following. Those highlighted in yellow below are the items provided in the Kaggle data-frame file. I added my hypothesis regrading each feature.

1. How much do you need? Possibly the higher the loan, the higher the risk of not being repaid. LOAN = Amount of the loan request

2. How long do you want the loan for? Possibly the longer the duration of the loan, the higher the risk of loan default through Job loss, death, etc.

3. Interest repayments? What is cost of loan? Higher the interest rate the higher overall cost of the loan, this may contribute to a loan default being more likely.

4. Do they work? Not working is a significant risk YOJ = Years at present job

5. Is employment permanent or temporary employment? In permanent employment you could assume the risk of default is lower.

6. How long in current employment? I would imagine the longer the applicant is in their current job, the more reliable the job is and means of paying loan. Possibly the more reliable the person in the role is. YOJ = Years at present job

7. Current job category? I would imagine certain jobs have higher established pays and security of paying - Temporary worker versus Professor. JOB = Six occupational categories

8. Existing mortgage debt? I would guess that the higher the current debt the applicant has, the more risk they expose the loan provider to. MORTDUE = Amount due on existing mortgage

9. Existing credit card age - line of credit? I would assume a person with an established line of credit for a lengthy period is more likely to pay back the loan.
   CLAGE = Age of oldest trade line in months

10. How many existing lines of credit? I would imagine having many lines of credit to be negative, regarding repaying a loan back. Not sure you can say this but I would imagine the higher number of lines of credit the higher the indebtedness
    NINQ = Number of recent credit lines & CLNO = Number of credit lines

11. How much do they earn? I would imagine this is very important as this would dictate the loan repaying power if indebtedness was known

12. Do they have a good credit score? This would be important in indicating are they a good customer or had credit issues in the past. DEROG = Number of major derogatory reports

13. Savings history? I would imagine if you have a track record for saving you will have a higher potential and discipline for paying back a loan

14. Existing assets? Important for securing loan as collateral VALUE = Value of current property

15. Have they dependents / re-occurring costs? Similar to current debts. Can the income exceed the existing costs to cover loan and interest repayments.

16. Reason for loan? Not sure the reason for the loan would impact repayment but perhaps if its for an asset that can be sold in the event of default, this may reduce loan default risk.

REASON = DebtCon = debt consolidation / HomeImp = home improvement

These are items I did not think of but where provided in the Kaggle data-set;

a) DELINQ = Number of delinquent credit lines - have they paid back loans in the past, I am not to 100% clear what DELINQ means.

b) CLAGE = Age of oldest trade line in months - are they struggling to pay and need credit or have an existing mortgage.

c) DEBTINC = Debt-to-income ratio - I would imagine this is important as it should gauge their ability to pay. Lower debt to income being preferred.

When looking at the data provided for the project, I notice two issues. 1) Some of the features I would have thought important are not available for modelling. For example, there is no income feature, and 2). The quality of the information available was not complete. For example the “”Debt_to_Income” is missing 20% of the instances and there is no feature relating to income that would allow me back-engineer for this missing data. Also, It was hard to get information from the internet on the exact meaning for some of the features.

For evaluation, there are two approaches I will take. Initially, the models are going to be evaluated using classification accuracy which is the ratio of number of correct predictions to the total number of input samples. There is a risk with this approach, if the data-sets have a high number of samples belonging to one class of prediction, the prediction may not be reliable.

The code from Scikit-learn is;

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = [0, 2, 1, 3]
>>> y_true = [0, 1, 2, 3]
>>> accuracy_score(y_true, y_pred)
Fractional accuracy = 0.5
>>> accuracy_score(y_true, y_pred, normalize=False)
Number of correct predictions = 2
```

The Confusion Matrix gives you a lot of information about how well your model does. When performing classification predictions, there's four types of outcomes.

```
print("True negatives - correctly classified as not Target: ", confusion_matrix_results[0][0])
print("False negatives - wrongly classified as not Target: ",confusion_matrix_results[1][0])
print("False positives - wrongly classified as Target: ", confusion_matrix_results[0][1])
print("True positives - correctly classified as Target: " ,confusion_matrix_results[1][1]
```

The code from Scikit-learn is;

```
>>> from sklearn.metrics import confusion_matrix
>>> y_true = [0, 1, 0, 1]
>>> y_pred = [0, 1, 0, 0]
>>> confusion_matrix(y_true, y_pred)
confusion_matrix_results = confusion_matrix(y_test, y_pred)
```

Step.1 - Data_Gathering

This step is where I import the original Kaggle data, reviewed it and prepared it for the following steps.

This step has the following actions.

1. Import file and create data-frame
2. Explore and tidy data-frame
3. Create Train and Test data-sets
4. Save to file

Initially, I saved the csv data file as **"loan_bad_Orignal.csv"** to my desktop from Kaggle. This was then modified to create two files, **'loan_bad_ID_Target.csv'** and **'loan_bad_ID_Features.csv'.** I then imported and merge both to make a project data-frame. They were merged on the 'ID' column. I created functions to complete the task, as I plan to use this program as a template for future projects. This will be a theme throughout the project.

I modified the original data-set so that I could demonstrate the MERGE function on two data-frames using the 'ID'. The following is the detail from both data-frames before the merge:

"loan_bad_ID_Target.csv" consisted of the following (5963 Rows, 2 Columns).

"loan_bad_ID_Features.csv" consisted of the following (5963 Rows, 13 Columns).

I used the following code to complete the MERGE. You can see I created functions so that the code is reusable. The filename is sent to the function. This is a little overkill, but it helped with repeatability and standardizing my code.

Code:

```
def import_file(filename):
    '''Import data - import and set up data frames'''
    file = pd.read_csv(filename)
    print("\n"+str(filename)+" in imported file:\n", file.info())
    return file

def create_project_file(A, B, Merge_on):
    '''create project file from imported files'''
    file = pd.merge(A, B, on=merge_on)
    print("\nMerged file info():\n", file.info())
    return file

#import file for project
filename1 = 'loan_bad_ID_Target.csv'
df_a = import_file(filename1)

#import file for project
filename2 = 'loan_bad_ID_Features.csv'
```

```
df_b = import_file(filename2)
#create project file and merge on 'ID'
merge_on = "ID"
df_merge = create_project_file(df_a, df_b, merge_on)

#make a copy in case we ruin orignal dataframe
data = df_merge.copy()
```

Merging both produced the following data-frame info: (5963 Rows, 14 Columns)

I then modified the legend so that the column titles were clear to me. I did not find all the original column titles that descriptive.

Code:

```
#rename columns to more understandable titles

data.rename(columns={'BAD': 'BAD_LOAN',
                     'LOAN': 'AMOUNT_REQUESTED',
                     'MORTDUE':'EXIST_MORTG_DEBT',
                     'VALUE': 'EXIST_PROPERTY_VALUE',
                     'REASON':'LOAN_REASON',
                     'YOJ': 'EMPLOYED_YEARS',
                     'DEROG': 'DEROG_REPORTS',
                     'DELINQ': 'DELINQ_CR_LINES',
                     'CLAGE': 'CR_LINES_AGE(MTS)',
                     'NINQ': 'NO_OF_RECENT_CR_LINES',
                     'CLNO': 'NO_OF_CR_LINES',
                     'DEBTINC': 'DEBT_TO_INCOME'}, inplace=True)
```

I then split this data-frame into (80%) Train, Test and (20%) Validation data-sets. Row selection was random using this code *(msk = np.random.rand(len(data)) < 0.8)*. The 20% Validation data-set will be used later in the project to make the predictions on a clean validation data-frame to avoid over-fitting.

The most important thing you can do, to properly evaluate your model, is not train the model on the entire data set. The train/test split I used was 70% for training and 30% for testing. I will be using K-Fold Cross validation where appropriate. I usually used K-Fold CV when I was trying to determine the model performance metrics. I evaluated my model as I was building it so I could find that best parameters. I did not use the validation data-set for evaluation as it might end up selecting parameters that perform best on the test data, but these may not be the parameters that generalize best. From the internet a typical train/test/validation split would be 60% of the data for training, 20% of the data for testing, and 20% of the data for validation. https://www.jeremyjordan.me/evaluating-a-machine-learning-model/

The following code shows how that was achieved and saved. I had an issue saving and reloading the files. The data-frame was loading with a new index and putting my column number out. To resolved this I use ‘index=False‘ when saving;

Code:

```
#create a test and traing dataframe that has not been cleaned
#use the random function to select random rows and assign to a mask
msk = np.random.rand(len(data)) < 0.8

#save the test dataframe - not in mask
test = data[~msk]
filename1 = 'S1_test_Loan_Basic_Data_Cleaning.csv'
test.to_csv(filename1, index=False)
print("\n>>Saved test data.shape: ", test.shape);print(test.info())

#save the train dataframe in mask
train = data[msk]
filename2 = 'S1_train_Loan_Basic_Data_Cleaning.csv'
train.to_csv(filename2, index=False)
print("\n>>Saved train data.shape: ", train.shape);print(train.info())

#Load df from file - used to see how the saved file loads back as I had an issue with index column
df = pd.read_csv(filename2)
print("\n<<Loaded dataframe.shape: ", df.shape);print(df.info())#(4826, 14)
```

"S1_Train_Test_Loan_Basic_Data_Cleaning.csv" contains 14 columns – 4779 rows. Unfortunately, due to the restriction I have had to drop a lot of screen shots to make the file size ok to upload. The screen shot would show the column data types and missing ness.
"S1_Validation_Loan_Basic_Data_Cleaning.csv" contains 14 columns – 1190 rows.

Step.2 - Basic_Data_Cleaning

This step used the data merged from the previous step. The overall aim of this step is to perform basic Exploratory Data Analysis (EDA). I performed basic data cleaning with minimal cleaning so that I can use this data quickly in the next step and get performance results for several classifier Machine Learning models.

This step has the following actions.

1. Import data and perform basic Exploratory Data Analysis (EDA)
2. Perform basic data cleaning
3. Save a data-frame to use in step 4 optimise data clean (without imputations)
4. Impute data for baseline models and save the data-frame for the next step

Action 1 - Perform basic Exploratory Data Analysis

In this action I conducted these three checks.

1. Column d.type check: What were the data types for the individual columns? I need to know are they correct for the information contained as this would affect the machine learning models. For example, some models cannot handle categorical data. I also printed out the numeric and non-numeric column titles.
2. EDA Descriptive: This action consisted of displaying the shape of the data-frame and info on the data-frame. I completed a for-loop where I iterate through the data-frame columns and printed out a description of the column content. I used describe(include=all) so I could get details on counts and max min etc. The (include=all) setting is very useful as it gives a fuller description of the column contents.
3. EDA Visual: This function displayed a chart of the missing data and a chart of any possible correlations.

The missingness chart is an excellent way to see how much missing data we have and where very quickly. The heat map is an excellent first step in examining any possible correlations in the data.

Missingness chart:

The missingness chart is excellent for quickly assessing which columns are really affected by missing data. It appears there are 11 columns missing data. We can see that “Debt_to_Income” appears to be the worst affected column. Column “Derog_Reports” appearing to be the next worst affected. If you look carefully it appears some rows are missing several columns of data.

Heat map:

The heat map is an excellent way of visually inspecting correlations or relationships between the features. Taking the extremes, it showed that (excluding “ID”) “Amount_requested” to “Existing_Mortg_Debt” and “Existing_Mortg_Debt” to “Existing_property_value” would appear to have some correlation. Perhaps the existing debt is for an existing loan on that existing property. In general, I only work off the extreme correlations as they are more reliable.

Action 2 - Perform Basic data cleaning

In this action I conducted the following checks.

1. Descriptive review of the dataframe missing data. I ran a function to table all the columns and their content that was missing. The detail was combined and recorded in my notes for correction.

2. Describe the unique content in the columns. I used a function to iterate through the data-frame columns looking at the unique data in each column. This was especially important as I can see very quickly any possible text errors for example miss-spelt “DebtCon” or “debtCon” and missing values “nan”. Also, duplicated rows for example, 4 counts of ‘ID’ = 26, 27 and 2 counts of ‘ID’ = 6 were I expected a count of 1 is an issue.

3. Using the information gathered so far, I was able to drop rows and standardize the content of several columns very quickly. The purpose was to do the most obvious basic data clean to allow me run and score the models on the basically cleaned data-frame. The following code was used to complete this.

```
##is there missing data?
print(draw_missing_data_table(df));pause()

##What is the unique data?
dataframe_unique_check(df)

##Drop all duplicate rows based
print("\nNumber of rows before drop_duplicated: ",len(df))
df.drop_duplicates(subset=None, keep='first', inplace=True, ignore_index=False)
print("\nNumber of rows After drop_duplicated: ", len(df));pause()

##Lets re-examine the offending unique data of the dataframe after the duplicate row drops and the drop ID column?
check_columns = df[['ID', 'BAD_LOAN','LOAN_REASON','JOB']]
dataframe_unique_check(check_columns)
```

```
# BAD_LOAN: Expected TWO unique values but got FIVE. I will change to column data to Repaid, Defaulted.
# Then I will change to 1:Default and 0:Repaid as its affecting categorising
df['BAD_LOAN'].replace(['paid','Repaid'],0,inplace=True)
df['BAD_LOAN'].replace([ 'default', 'Dfault','Default'],1,inplace=True)

# LOAN_REASON: Expected TWO unique values but got SIX. will change to DebtCon, HomeImp
df['LOAN_REASON'].replace(['homeImp','Homeimp'],'HomeImp',inplace=True)
df['LOAN_REASON'].replace([ 'debtCon' , 'debtcon'],'DebtCon',inplace=True)
#Going to use this oppourtunity to impute a value other than leave empty
df['LOAN_REASON'].fillna('Other', inplace = True)

# Expected SIX unique values but got SEVEN. Will investigate empty features and call Other
# replacing na values in 'JOB' with 'Other'
df['JOB'].fillna('Other', inplace = True)

##Lets rexamine after replace function has been used
check_columns = df[['ID', 'BAD_LOAN','LOAN_REASON','JOB']]
dataframe_unique_check(check_columns)
```

Action 3 - Save data-frame to use when Optimize Data Clean

Now I saved the data-frame that would later be used when I wanted to optimize the data in step 4. I saved this data-frame before imputing for missing data. This was going to be my import data-frame into the optimize data clean step and I did not want the imputed data completed next, to affect this. In the optimize data step 4, I intend to take a closer look at the following imputations and optimize them. To state again, my aim here is to create a clean data-frame so I can assess the baseline models performances. These scores are then compared and becomes my baseline score. This data-frame was saved as **'S2_Loan_Basic_Data_Cleaning.csv.**

Action 4 - Impute data for baseline models and Save data-frame for next step

For this action I created a simple function that would cycle through the data-frame columns and where there was a missing value, insert a multiple of 10 by the max value of the column. I chose 10 times the max value of the column, as I wanted any impute to negatively affect the model's performance or make no difference. In the optimize data step 4 I would correct this. The columns were now all numeric, so I did not need to worry about non-numeric errors from the compiler. The loop was a simple for-loop that iterated through the data-frame columns. I checked for missing values before and after the change was completed.

The output of the imputation can be seen below:

```
column is BAD_LOAN
Filled column[BAD_LOAN] with 10

column is AMOUNT_REQUESTED
Filled column[AMOUNT_REQUESTED] with 892000

column is EXIST_MORTG_DEBT
Filled column[EXIST_MORTG_DEBT] with 3995500.0

column is EXIST_PROPERTY_VALUE
Filled column[EXIST_PROPERTY_VALUE] with 8559090.0
```

```
column is EMPLOYED_YEARS
Filled column[EMPLOYED_YEARS] with 99990.0

column is DEROG_REPORTS
Filled column[DEROG_REPORTS] with 100.0

column is DELINQ_CR_LINES
Filled column[DELINQ_CR_LINES] with 150.0

column is CR_LINES_AGE(MTS)
Filled column[CR_LINES_AGE(MTS)] with 11682.33561

column is NO_OF_RECENT_CR_LINES
Filled column[NO_OF_RECENT_CR_LINES] with 170.0

column is NO_OF_CR_LINES
Filled column[NO_OF_CR_LINES] with 710.0

column is DEBT_TO_INCOME
Filled column[DEBT_TO_INCOME] with 2033.121487
```

The code to complete this is:

```
##Impute columns missing data - lets use a basic impute of the max value (by 10) in the column
for col in df.columns:
    print("column is "+str(col))
    n = df[col].max()
    n = n * 10
    df[col].fillna(n, inplace=True)
    print("Filled column["+str(col)+"] with "+str(n))

##Correct the missing data - review changes
print(draw_missing_data_table(df));pause()
```

The file was saved and called **'S2_Loan_Basic_Data_for_Baseline_Models.csv'.** This data-frame which is now 'Basically Cleaned' and was to be use in the next step where I would apply this data to the models and evaluate their performance.

Step.3 - Baseline Model Testing

This step used the data-frame created in the last step 2. The purpose of this step is to run the data-frame through several Supervised Classifier Machine Learning algorithm models and assess their performance on basic cleaned data.

This step has the following actions.

1. Import data and transform the categorical variables.
2. Create an array of features values 'X' and target array of values called 'y'.
3. Normalize the data-frame?

4. Perform K-Fold cross validation on several models

Action 1 - Transform the categorical variables

After importing, I transformed the categorical features to numeric variables. This is the first pre-processing activity. To do this I passed the data-frame to a function I created. This returned a new data-frame of numeric values. One concern I have with this activity is will it affect feature importance. This is because when I look at feature importance chart, you see that the categorized "JOB" features are lower, as their total has been split across several columns. My fear is that the impact of the "JOB" feature will have less importance on the model's performance. Qualitatively this was not the case as the underlying data has not been changed.

The categorical data-frame column for "JOB" was:

JOB
Mgr
Other
Office
Self
ProfExe

I use the following code to achieve numerical variables. I passed it the data-frame I wanted to transform:

```
def  transform_categorical_variables(dataframe):
    ''' Transform categorical variables into dummy variables - - known as one-hot encoding of the data.
        This process takes categorical variables, such as days of the week
          and converts it to a numerical representation without an arbitrary ordering.'''
    dataframe = pd.get_dummies(dataframe, drop_first=True)  # To avoid dummy trap
    return dataframe
```

Side note : *Dummy Variable Trap:* *When the number of dummy variables created is equal to the number of values the categorical value can take on. This leads to multi-collinearity, which causes incorrect calculations of regression coefficients and p-values.* https://www.statology.org/dummy-variable-trap/

Job column was transformed and is now are like this:

JOB_Office	JOB_Other	JOB_ProfExe	JOB_Sales	JOB_Self
0	1	0	0	0
0	0	0	0	0
0	0	0	0	0
1	0	0	0	0
0	0	0	0	1
0	0	1	0	0
0	1	0	0	0

Action 2 - Create Target and Feature variables

In this part I created the Target and Feature arrays. There are many different options for completing this. The most effective way I found allows me to name a Target column and send it as an argument to the function. I like this approach in my code as it allows me easily to re-use my code but also it does not matter where the

Target column is the data-frame columns. It then returns the X and y arrays. This is important as the column headers need to be removed as the machine learning model cannot accept the string title, they need numerical values.

```
def create_X_y_datasets(df,target_column_name):
    '''create features and target datasets'''
    X = df[df.loc[:, df.columns != target_column_name].columns]
    y = df[target_column_name]
    return X, y

# Create datasets for model
target_column_name = 'BAD_LOAN'
X, y = create_X_y_datasets(df, target_column_name)
```

Action 3 - Normalize the data-frame

In this part I would normalize the data. This consisted of setting the max and min of the column to 0 and 1 respectively and scaling the data appropriately. I ran the data with and without Normalizing as I read some models do not need the data normalized and I was curious to see the effect.

Code:

```
def scale_data_normalisation(X):
    '''pre-processing - Normalisation'''
    scaler = MinMaxScaler()
    scaled = scaler.fit_transform(X)
return scaled

#rescale X between 0 - 1
X = scale_data_normalisation(X)
```

Action 4 - Perform K-Fold Cross Validation on a number of models

Here I built on an idea I found on towardsdatascience.com. The idea was to send the X and y data arrays and perform K-Fold cross validation against several classifier models. The function tests all the models for a specific K-Fold value, score them using cross_val_score() and display their accuracy.
https://towardsdatascience.com/cross-validation-and-hyperparameter-tuning-how-to-optimise-your-machine-learning-model-13f005af9d7d

This was then appended to a data-frame I created and saved. The idea for this step was to get baseline performance metrics for the various models using their standard model parameters. In Step 5 this was repeated on optimised data and I needed baseline metrics for comparison.
Step 3 saved the baseline performance metrics to **"ML3_Loans_Models_Results_on_Basic_Data.csv"**.

Code:

```
def Classifier_models_test(df_model_values, a, b):
    '''Test data on a number of different classifier algorithims, using KFold CV and save performance data'''
    # get the list of models to consider
    models = get_models()
```

```
            # define test conditions
            Kfold_number = range(a,b,1)

            for CV_val in Kfold_number:
                #https://www.askpython.com/python/examples/k-fold-cross-validation
                kf = KFold(n_splits=CV_val, shuffle=True, random_state=42)
                # evaluate each model
                for model in models:
                    print("\nKfold_number = ", CV_val)

        #Implementing cross validation and get y_p
        #https://www.bitdegree.org/learn/train-test-split
        import sklearn.model_selection as model_selection
        X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, train_size=0.7, random_state=CV_val)
        #Fit model and predict on training data
        model.fit(X_train,y_train)
        y_pred = model.predict(X_test)

        ##Test 1 - Accruacy_score
        #Implement accuracy_score() function
        model_accuracy_score= accuracy_score(y_test, y_pred)
        print("\nAccuracy_score() is: ", round(model_accuracy_score, 3))

        ##Test 2 - Confusion matrix
        confusion_matrix_results = confusion_matrix(y_test, y_pred)
        print("True negatives - correctly classified as not Target: ", confusion_matrix_results[0][0])
        print("False negatives - wrongly classified as not Target: ",confusion_matrix_results[0][1])
        print("False positives - wrongly classified as Target: ", confusion_matrix_results[1][0])
        print("True positives - correctly classified as Target: " ,confusion_matrix_results[1][1])
        confusion_matric_accuracy = (confusion_matrix_results[0][0]+confusion_matrix_results[1][1])/len(y_pred)

 #just want to make sure program stops if these couts are dirrenent as it mean my accuracy will not be correct
        assert
        len(y_pred)==(confusion_matrix_results[0][0]+confusion_matrix_results[0][1]+confusion_matrix_results[1][0]+confusion_matrix
_results[1][1])
        print("Confusion Matric - Accuracy: " ,confusion_matric_accuracy)

        ##Test 3 - Coss_Val_Score
        # evaluate model using each test condition on cross_val_score()
        #https://scikit-learn.org/stable/modules/cross_validation.html
        scores = cross_val_score(model,X,y,scoring='accuracy', cv=kf, n_jobs=None)
        cv_mean = mean(scores)
        # check for invalid results
        if isnan(cv_mean):
            continue
        # Model performances
        model_name = type(model).__name__
        print(str(model_name)+' Cross Val Score - Accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))

        #Append data to dataframe to record results
        df_model_values = df_model_values.append({'CV':CV_val,'Model':str(model_name),
                                    'CVS_Accuracy':round((np.mean(scores)),3),
                                    'CVS_STD':round((np.std(scores)),3),
                                    'Accuracy_Score':round(model_accuracy_score,3),
```

```
                    'C_M_Accuracy':round(confusion_matric_accuracy,2),
                    'True_Neg':confusion_matrix_results[0][0],
                    'False_Neg':confusion_matrix_results[1][0],
                    'False_Pos':confusion_matrix_results[0][1],
                    'True_Pos':confusion_matrix_results[1][1]},ignore_index = True)
    #Sort the values
    df_model_values.sort_values(by=['CVS_Accuracy'], axis=0, ascending=False,inplace=True,
kind='quicksort',na_position='last',ignore_index=False, key=None)
    #save the dafatframe to file
    df_model_values.to_csv("ML3_Loans_Models_Results_on_Basic_Data.csv")#use this to see what the data looks like after
lateststep
     return df_model_values
```

The models I had to choose from are as follows. However, I had an issue with my PC running Linear SVC model.

```
def get_models():
    models = list()
    models.append(LogisticRegression())
    models.append(RidgeClassifier())
    models.append(SGDClassifier())#
    models.append(PassiveAggressiveClassifier())
    models.append(KNeighborsClassifier())
    models.append(DecisionTreeClassifier())
    models.append(ExtraTreeClassifier())
    ##models.append(LinearSVC())# gives low reading but gives fault
    models.append(SVC())
    models.append(GaussianNB())
    models.append(AdaBoostClassifier())
    models.append(BaggingClassifier())
    models.append(RandomForestClassifier())
    models.append(ExtraTreesClassifier())
    models.append(GaussianProcessClassifier())
    models.append(GradientBoostingClassifier())
    models.append(LinearDiscriminantAnalysis())
    models.append(QuadraticDiscriminantAnalysis())
     return models
```

The performance of the classifiers was saved to file **“ML3_Loans_Models_Results_on_Basic_Data.csv”.** Random Classifier performed the best at this point. It did this with or without Normalization. This is interesting as it confirms what I read, that this model does not require scaling.
https://datascience.stackexchange.com/questions/62031/normalize-standardize-in-a-random-forest

Step.4 - Optimise_Data_Cleaning

This step used the data from step 2, before the imputed values was completed. The purpose of this step was to try optimizing the data before trying all the classifiers models again. I expected to have an improvement in model performances.

This step has the following actions.

1. Import data and perform basic Exploratory Data Analysis (EDA)

2. Perform basic data cleaning
3. Perform data-frame tidying
4. Optimize data cleaning (including Box plot and Histogram)
5. Scatter plot of single Feature to Target
6. Feature engineering
7. Checked for multi-collinearity in features
8. Identify most important features

Action 1 - Action 2

These action where covered in step 2 - Basic_Data_Cleaning and data was imported in with basic cleaning. Duplicates etc were removed but no imputing for missing values.

Action 3 - Perform data-frame tidying

In this action I dropped the 'ID' column and removed rows with many empty cells. I am aware it is nearly a "cardinal sin" to delete data, but I found that a row with up to 8 of 9 columns missing would not affect model performance. I ran the model with and without the drop in rows. Random Forest model did not show a change in accuracy.

I used this code to complete this. Basically, this deletes row up to a number 'n' which I set before running the program.

```
n = 8#1#8 #we are allowing rows with up to 7 empty cells
df = df[df.isnull().sum(axis=1) < n]
```

Action 4 - Optimize data cleaning (including Box plot and Histogram)

In this action I reviewed charts for the features. I measured what % of the data was missing. I will discuss some of these below. Due to the restriction on the file size I was not able to show all graphs.

"DEBT_TO_INCOME" Total Percent missing 21%

In this chart we can see some outliers but without an income I was not able to correct or access if it needed to be corrected. I used the KNN_Impute to resolve the missing data.

"DEROG_REPORTS" Total Percent missing 12%

I could not say with any certainty what this should be. The figure must be taken as correct and I did not want to impute an artificially high or low value as this may affect the model's performance. I used the mean to resolve the missing data.

"DELINQ_CR_LINES" Total Percent missing 10%

I was not sure what this meant. I could not find this on the internet. The closest identifier I could find was to do with credit lines. If this is what this is, it would seem a very high number for a single person. But I was not able to challenge these figures, so I used the mean to resolve the missing data.

"EXIST_MORTG_DEBT" Total Percent missing 10%

This did not seem unreasonable and to be honest, I would need to know the standard house price to have an opinion. For example in Ireland this mean would seem quite low. And the high mortgage debt identified in the box plot may be for a person with multiple mortgages I.e. 2nd or 3rd home? I used the mean to resolve the missing data.

"NO_OF_RECENT_CR_LINES" Total Percent missing 8%

We did have one extreme outlier but perhaps this is correct, and I did not make any correction. I used the mean to resolve the missing data.

"EMPLOYED_YEARS" Total Percent missing 8%

For Column 'EMPLOYED_YEARS' The box plot showed an obvious outlier in the data, where the employed years was 9999. In theory the max working age would be retirement age, less starting age. Let say 50 years max employment. However, the box plot shows clearly this to be 10000 years. I used the mean to resolve the missing data and change the outlier. It's extremely interesting the impact that outlier had on the histogram.

"CR_LINES_AGE(MTS)" Total Percent missing 5%

There appears to be something unusual regarding this outlier. The problem I have is if I change it, what do I change it to? I was not sure what this feature meant. I imputed the mean of the column for the missing values.

"NO_OF_CR_LINES" Total Percent missing 4%

I used the mean to resolve the missing data. This was because I am not sure what the original column description meant, so I was not able to challenge these figures. The missing data was relatively small.

"EXIST_PROPERTY_VALUE" Total Percent missing 2%

I used the mean to resolve the missing data. This was because I was not able to challenge these figures. The missing data was relatively small.

Action 5 - Categorical Scatter plot of single Feature to Target "BAD_LOAN"

In this section these are some of the insights I gained by examining the relationship of the individual features to "BAD_LOAN" feature. This was completed after the missing cells were imputed with the mean and "DEBT_TO_INCOME" feature was imputed with KNN-Impute.

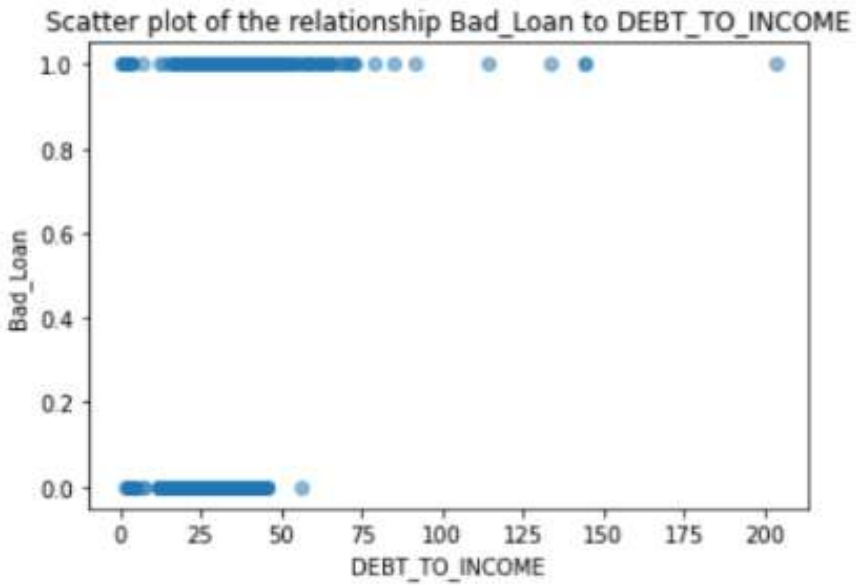
https://www.westga.edu/academics/research/vrc/assets/docs/scatterplots_and_correlation_notes.pdf

BAD_LOAN: 1 = client defaulted on loan 0 = loan repaid

DEBT_TO_INCOME:

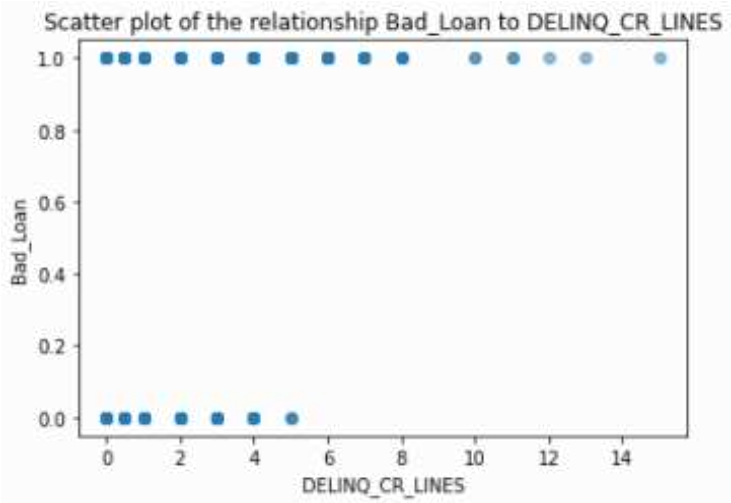
In this scatter plot you could suggest that client defaults increase with "DEBT_TO_INCOME". Which makes sense. The more indebted you are, it could be argued the greater the risk that you will be unable to pay off or struggle to pay your loan.

DEBT_TO_INCOME *after* KNN-Impute



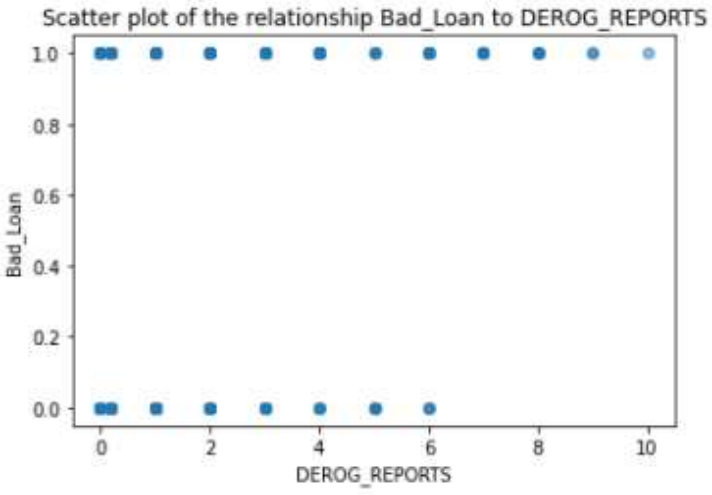
DELINQ_CR_LINES:

Possibly the more “Delinquent_CR_lines” a person has the more likely to default.



DEROG_REPORTS:

It would appear, the more “Derogatory reports” a person has the more likely to default.



Action 6 - Feature engineering

In this section I used the KNN-Imputer to impute missing values for column “DEBT_TO_INCOME” which had a total Percent missing of 21%. What made this a difficult column to impute missing values, was that I could not find any column to indicate an income or any means to back-engineer the “Debt-to-Income” ratio. Scikit learn describe the process of KNN-Imputation as completing missing values using the k-Nearest Neighbours and can be found at:
https://scikit-learn.org/stable/modules/generated/sklearn.impute.KNNImputer.html

I created optimized data for KNN-Impute parameter n= 2, 4, 6, 8, 10 using the code below. Then I applied each to the Random Forest Classifier algorithm to compute a model performance.

This is the KNN-Impute code to impute the missing values.

```
from sklearn.impute import KNNImputer
df = transform_categorical_variables(df)

# from sklearn.preprocessing import MinMaxScaler
# scaler = MinMaxScaler()
# df = pd.DataFrame(scaler.fit_transform(df), columns = df.columns)

imputer = KNNImputer(n_neighbors=n)
df = pd.DataFrame(imputer.fit_transform(df),columns = df.columns)
```

This is the model performance for each value of n, using a K-Fold CV value of 5. The Random Forest Classifier model had the following results. N=4 was the best in this very limited trial.

N=2

Accuracy_score() is: 0.913
True negatives - correctly classified as not Target: 1132
False negatives - wrongly classified as not Target: 9
False positives - wrongly classified as Target: 115
True positives - correctly classified as Target: 171
Confusion Matric - Accuracy: 0.913104414856342
RandomForestClassifier Cross Val Score - Accuracy: 0.924 +/- 0.005

N=4

Accuracy_score() is: 0.919
True negatives - correctly classified as not Target: 1132
False negatives - wrongly classified as not Target: 9
False positives - wrongly classified as Target: 107
True positives - correctly classified as Target: 179
Confusion Matric - Accuracy: 0.9187105816398038
RandomForestClassifier Cross Val Score - Accuracy: 0.928 +/- 0.006

N=6

Accuracy_score() is: 0.914
True negatives - correctly classified as not Target: 1136

False negatives - wrongly classified as not Target: 5
False positives - wrongly classified as Target: 118
True positives - correctly classified as Target: 168
Confusion Matric - Accuracy: 0.9138051857042747
RandomForestClassifier Cross Val Score - Accuracy: 0.922 +/- 0.008

N=8

Accuracy_score() is: 0.916
True negatives - correctly classified as not Target: 1137
False negatives - wrongly classified as not Target: 4
False positives - wrongly classified as Target: 116
True positives - correctly classified as Target: 170
Confusion Matric - Accuracy: 0.9159074982480728
RandomForestClassifier Cross Val Score - Accuracy: 0.926 +/- 0.008

N=10

Accuracy_score() is: 0.917
True negatives - correctly classified as not Target: 1135
False negatives - wrongly classified as not Target: 6
False positives - wrongly classified as Target: 113
True positives - correctly classified as Target: 173
Confusion Matric - Accuracy: 0.9166082690960056
RandomForestClassifier Cross Val Score - Accuracy: 0.923 +/- 0.007

Action 7 - Check for multi-collinearity in features

I want to access the impact of similar features, so I completed a multi-collinearity test. I got information regarding this from:
https://towardsdatascience.com/everything-you-need-to-know-about-multicollinearity-2f21f082d6dc
And,
https://www.geeksforgeeks.org/detecting-multicollinearity-with-vif-python/

Although used for linear regression, it is nice to see if the VIF will indicate if there is high collinearity between these features. This is a simple method to detect multi-collinearity in a model. It uses Variance Inflation Factor or the VIF for each predicting feature. The VIF measures the ratio between the variance for a given regression coefficient with only that variable in the model versus the variance for a given regression coefficient with all variables in the model. A VIF of 1 (the minimum possible VIF) means the tested predictor is not correlated with the other predictors.

The code was as follows:

```
# Create datasets for model
target_column_name = 'BAD_LOAN'
X, y = create_X_y_datasets(df_cat, target_column_name)

# VIF dataframe
vif_data = pd.DataFrame()
vif_data["feature"] = X.columns
```

```
# calculating VIF for each feature
vif_data["VIF"] = [variance_inflation_factor(X.values, i)for i in range(len(X.columns))]

# vif_data.drop(vif_data[0], axis=1, inplace = True)
vif_data.set_index('feature', inplace = True)
vif_data.sort_values(by=['VIF'], inplace=True, ascending=False)
print(vif_data)


print(vif_data.info())
vif_data.plot.bar()
plt.show()
```

The results would suggest that there is multi-collinearity with some of the features. Above 10 was stated as less than desirable. “Feature_property_Value”, “Debt_to_Income” and “Existing_Mortg_Debt” appear to be possibly correlated. Perhaps having a high mortgage debt is reflected in a high existing property value. Which likely to lead to a higher debt?

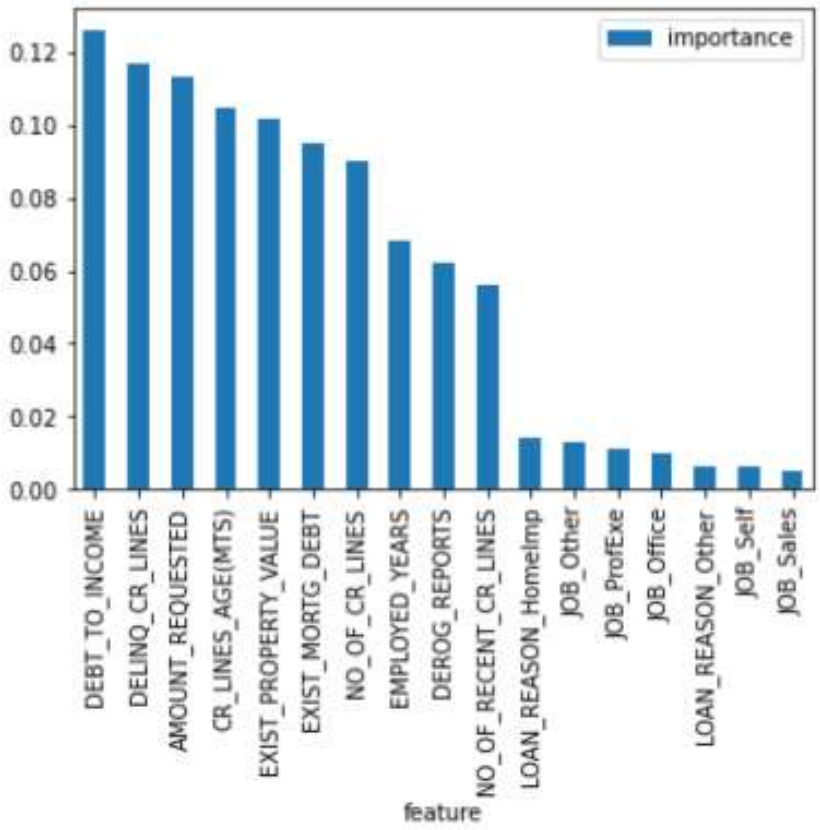
Action 8 - Identify most important features

In this section I wanted to see what features were having the most significant impact on the output. With this information I removed the lesser important features and assessed model performance. I used the Random Forest Classifier and tested with one less feature each time. In chart 6 we can see feature importance.

This is the code I used:

```
def most_important_features(X,y):
    #Feature Importance - https://www.kaggle.com/niklasdonges/end-to-end-project-with-python
    random_forest = RandomForestClassifier(n_estimators=100)
    random_forest.fit(X, y)
  importances = pd.DataFrame({'feature':X.columns,'importance':np.round(random_forest.feature_importances_,3)})
    importances = importances.sort_values('importance',ascending=False).set_index('feature')
    print('\nimportances.head(15):\n',importances.head(15))
    #show bar plot on impotant features
    importances.plot.bar()
    plt.show()

#=============================================================
#Identiy important features
#=============================================================

#call function to plot and describe important features
most_important_features(X,y)
```

Chart 6 - Feature importance



This is the model performance result for Random Forest classifier with KNN impute N=4, and CV = 5 which I will be using to compare against as I remove features.

Accuracy_score() is: 0.919

True negatives - correctly classified as not Target: 1132

False negatives - wrongly classified as not Target: 9

False positives - wrongly classified as Target: 107

True positives - correctly classified as Target: 179

Confusion Matric - Accuracy: 0.9187105816398038

RandomForestClassifier Cross Val Score - Accuracy: 0.928 +/- 0.006

This is what I found when I dropped the least important feature “JOB”(s) .

No significant real change.

Accuracy_score() is: 0.919

True negatives - correctly classified as not Target: 1135

False negatives - wrongly classified as not Target: 6

False positives - wrongly classified as Target: 109

True positives - correctly classified as Target: 177

Confusion Matric - Accuracy: 0.9194113524877365

RandomForestClassifier Cross Val Score - Accuracy: 0.923 +/- 0.007

This is what I found for when I dropped the next least important feature “Loan_Reason”(s) . Model Accuracy decreased but not significantly.

Accuracy_score() is: 0.908

True negatives - correctly classified as not Target: 1132

False negatives - wrongly classified as not Target: 9

False positives - wrongly classified as Target: 122

True positives - correctly classified as Target: 164
Confusion Matric - Accuracy: 0.9081990189208129
RandomForestClassifier Cross Val Score - Accuracy: 0.919 +/- 0.008

This is what I found for when I dropped the next least important feature “NO_OF_RECENT_CR_LINES” .
Model Accuracy decreased the most but not significantly.

Accuracy_score() is: 0.901
True negatives - correctly classified as not Target: 1130
False negatives - wrongly classified as not Target: 11
False positives - wrongly classified as Target: 130
True positives - correctly classified as Target: 156
Confusion Matric - Accuracy: 0.9011913104414856
RandomForestClassifier Cross Val Score - Accuracy: 0.912 +/- 0.008

Conclusion: The removal of features of less importance in this case did not improve the model predictive performance. I decided not to remove any features from the data-frame.

I checked out medium web site for feature importance and I found this article. The article reviewed feature importance. The post covered four classes of feature importance:

- ensemble tree specific feature importance (local model-specific),
- permuted feature importance (global model-agnostic),
- LIME (local model-agnostic), and
- Shapley values (local model-agnostic).

https://colab.research.google.com/drive/1as0n3ozs4ut7-KbQX-d1NVeP37-E6kkC#scrollTo=mEp0fCnZPoMr
https://medium.com/bigdatarepublic/feature-importance-whats-in-a-name-79532e59eea3
https://shap-lrjball.readthedocs.io/en/latest/generated/shap.force_plot.html
https://towardsdatascience.com/decrypting-your-machine-learning-model-using-lime-5adc035109b5
https://towardsdatascience.com/shap-explain-any-machine-learning-model-in-python-24207127cad7
https://www.analyticsvidhya.com/blog/2019/11/shapley-value-machine-learning-interpretability-game-theory/

Step.5 - Tune_Model_and Select

This step is like step 3. In step 3, I ran the models on basically cleaned data but in this step, I ran the model on optimized cleaned data as described in step 4. The purpose of this step is to run the optimized data through several supervised classifier machine learning models and evaluate their performance. Then I selected one to hyper-tune.

This step has the following actions.

1. Import data and transform the categorical variables.
2. Create an array of Features values ‘X’ and Target array of values called ‘y’.
3. Normalize the data-frame
4. Perform K-Fold cross validation on several models

The performance of the top classifiers is Extra trees classifier. The Random Classifier performs excellently but not the best at this point. The Random Classifier shows an increase in performance of 1% with the optimised cleaned data.

Conclusion: Out of this step I selected the Random Forrest Classifier. The Random Forest Classifier had performed best on the basic cleaned data and second best on the optimised cleaned data. I am aware that the Extra Trees Classifier has a better performance over the Random Forrest Classifier at this stage, I am familiar with the setting up of the Random Forest Classifier and have decided to use it. After this project is completed, I would like to compare the performance of both. Below is a link regarding a comparison of both and a link to setting the Grid Search CV parameters for the Extra Trees.
https://quantdare.com/what-is-the-difference-between-extra-trees-and-random-forest/
https://www.kaggle.com/eikedehling/extra-trees-tuning

Step.6 - HyperTune_model

This step is like step 3 and step 5 in that I am trying to run a model and assess performance. I have chosen to use the Random Forest Classifier as it performed excellent on the data-set to date. In this step, I am going to Hyper-tune its parameters. To complete this I am going to use Grid Search Cross Validation. Grid Search CV is a method for estimating the best parameters for a model. It provides an exhaustive search over specified parameter values for a model.
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

This step has the following actions.

1. Pre-processing data
2. Hyper-parameter Tuning and Saving best parameters to file
3. Classifier performance after hyper tuning
4. Evaluating final model and saving model

Action 1 - Pre-processing data

This has been described in step 3 and step 5 and involves setting up the data so that it can run effectively in the model. This consists of transforming the data, identifying the features, target variables, Scaling (not in this case) the variables and splitting the data into train (70%) and test (30%) sets.

Action 2 - Hyper parameter Tuning

In this action I created a parameter grid which was loaded into the Grid Search CV function. You can see the dictionary below called “param_grid”. This has the values I will be applying to the model. I created a second test “param_grid” which is hashed out. This was to allow a more rapid testing as Hyper tuning can take up to 40 minutes on this PC. [Parallel(n_jobs=-1)]: Done 2400 out of 2400 | elapsed: 47.2min finished

```
print("\nHyperparameter Tuning:")
param_grid = { "criterion" : ["gini", "entropy"],
        "min_samples_leaf" : [1, 5, 10, 25, 50, 70],
        "min_samples_split" : [2, 4, 10, 12, 16, 18, 25, 35],
        "n_estimators": [100, 400, 700, 1000, 1500]}
```

I used the following code to identify the best parameters from the model. These are then displayed.

```
print("\nBest parameters found as per parama grid")
print("\nclf.best_params_", clf.best_params_)
```

Using Pickle, I saved the best model parameter to a file in the project folder. This was so I had a copy of the parameters for Step 8. Making Predictions.

```
# Import pickle Package
import pickle

# Save the Modle to file in the current working directory
model_filename = "6_Best_Model_Params.pkl"

with open(model_filename, 'wb') as file:
    pickle.dump(clf.best_params_, file)
```

Action 3 - Classifier performance after hyper tuning

I used the following code to test the model's performance.

```
from sklearn.metrics import classification_report
print("\nDetailed classification report for HyperTuned model:")

print("Train scores:")
y_pred = clf.predict(X_train)
print(classification_report(y_train, y_pred))

print("Test scores:")
y_pred = clf.predict(X_test)
print(classification_report(y_test, y_pred));pause()
```

Detailed classification report for Hyper-tuned model:

	Precision	Recall	F1-score	Support
0.0	0.91	0.99	0.95	1158
1.0	0.95	0.59	0.73	273
			0.93	1431
#macro avg	0.93	0.79	0.84	1431
#weighted avg	0.92	0.92	0.91	1431

Action 4 - Evaluating final model and save

I then re-loaded the Hyper-tuned parameters into the Random Forest Classifier model. I did this to practice setting up the model.

```
#Test new best paramters: Random Forest with TEST data
model = RandomForestClassifier(criterion = "gini",
                               min_samples_leaf = 1,
                               min_samples_split = 2,
                               n_estimators=700,
                               max_features='auto',
                               oob_score=True,
                               random_state=1,
                               n_jobs=-1)

##clf.best_params_ {'criterion': 'gini', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 700}
#Check model parameters loaded
print('\nConfirm parameters currently in use:\n')
pprint(model.get_params())

#get oob score
model.fit(X_train, y_train)
print("\nHypertuned - oob score:", round(model.oob_score_, 2)*100, "%")

#Implement accuracy_score() function on model
y_pred = model.predict(X_test)
accuracy_test = accuracy_score(y_test, y_pred)
print("Accuracy_score() is: ", round(accuracy_test, 3));pause()

#Confusion matrix
confusion_matrix_results = confusion_matrix(y_test, y_pred)
print("True negatives - correctly classified as not Target: ", confusion_matrix_results[0][0])
print("False negatives - wrongly classified as not Target: ",confusion_matrix_results[1][0])
print("False positives - wrongly classified as Target: ", confusion_matrix_results[0][1])
print("True positives - correctly classified as Target: " ,confusion_matrix_results[1][1])

confusion_matric_accuracy = (confusion_matrix_results[0][0]+confusion_matrix_results[1][1])/len(y_pred)

#just want to make sure program stops if these counts are not correct
assert len(y_pred)==(confusion_matrix_results[0][0]+confusion_matrix_results[0][1]+
        confusion_matrix_results[1][0]+confusion_matrix_results[1][1])

print("Confusion Matric - Accuracy: " ,confusion_matric_accuracy)
```

I got the following results for the Random Forest Classifier after Hyper-tuning. Table 3 below are the model performance figures for the test data. The test data has a different number of rows (30% of the rows). You can see similar predictive performance on the Test data.

Table 3. - Random Forest Classifier after Hyper-tuning - performance figures for the test data

Accuracy_Score		True_Neg	False_Neg	False_Pos	True_Pos
0.917		1152	113	6	160

I then saved the model to the project file. The reason I did this is because I was testing each component of the project individually and wanted to be able to re-load the working model when required.

https://www.kaggle.com/prmohanty/python-how-to-save-and-load-ml-models

This was saved as follows:

```
# Import pickle Package
import pickle

# Save the Modle to file in the current working directory
model_filename = "6_Loan_UCD_ML_Model.pkl"

with open(model_filename, 'wb') as file:
    pickle.dump(model, file)
```

Additional item: After I wrote this, I did give Extra Trees a try and got this error on max_features, I set this to 'auto' to get around the issue but that restricts my grid search. https://stackoverflow.com/questions/42072721/valueerror-max-features-must-be-in-0-n-features-in-scikit-when-using-rand

Then I got this error: *ValueError: Classification metrics can't handle a mix of binary and continuous targets.* So I used the following code to create a list of predictions: *y_pred = [round(x[0]) for x in y_pred]* but it did not work. Then I had the realization that while GridSearch CV was identifying the optimal parameters from the parameter list, I then had to set up the model with these parameters. These are the optimal parameters identified:

```
Best: 0.926260 using {'criterion': 'gini', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 75}
```

My final code was:

```
#========================================================================
#Extra TreeClassifier
#========================================================================
print("\nHyperparameter Tuning Extra Trees:")
# to identify the optimal parameters from this dictionary
# param_grid_ET={'criterion': ["gini", "entropy"],
#     'n_estimators': range(50,126,25),
#         'min_samples_leaf': range(1,30,2),
#         'min_samples_split': range(2,50,2)}
#for testing
param_grid_ET={'n_estimators': [50],
       'min_samples_leaf': [20],
       'min_samples_split': [15] }

model1 = ExtraTreesClassifier(random_state=1)
gsc = GridSearchCV(estimator=model1, param_grid=param_grid_ET, n_jobs=-1, verbose=3)

Extra_trees_model = gsc.fit(X_train, y_train)
print("Best: %f using %s" % (Extra_trees_model.best_score_, Extra_trees_model.best_params_))

# #================================================
# # Extra Trees Classification Report after Hypertuning
# #================================================

from sklearn.metrics import classification_report
print("\nDetailed confusion matrix for Extra trees HyperTuned model:")
```

```
#Test new best paramters: Random Forest with TEST data
model2 = ExtraTreesClassifier(criterion='gini', min_samples_leaf=1, min_samples_split=2, n_estimators=75)

#Check model parameters loaded
print('\nConfirm parameters currently in use:\n')
pprint(model2.get_params())

#get oob score
model2.fit(X_train, y_train)

#Implement accuracy_score() function on model
y_pred2 = model2.predict(X_test)

accuracy_test = accuracy_score(y_test, y_pred2)
print("Accuracy_score() is: ", round(accuracy_test, 2))

print("Confusion Matric - Accuracy: ")
print(confusion_matrix(y_test, y_pred2))
```

Extra Tree performance metrics:

Accuracy_score()	0.92
Confusion Matric	
1152	6
104	169

Conclusion: the performance between the Random Forest Classifier and Extra Tress classifier was similar.

Step.7 - Optimise_Validation_Data_Cleaning

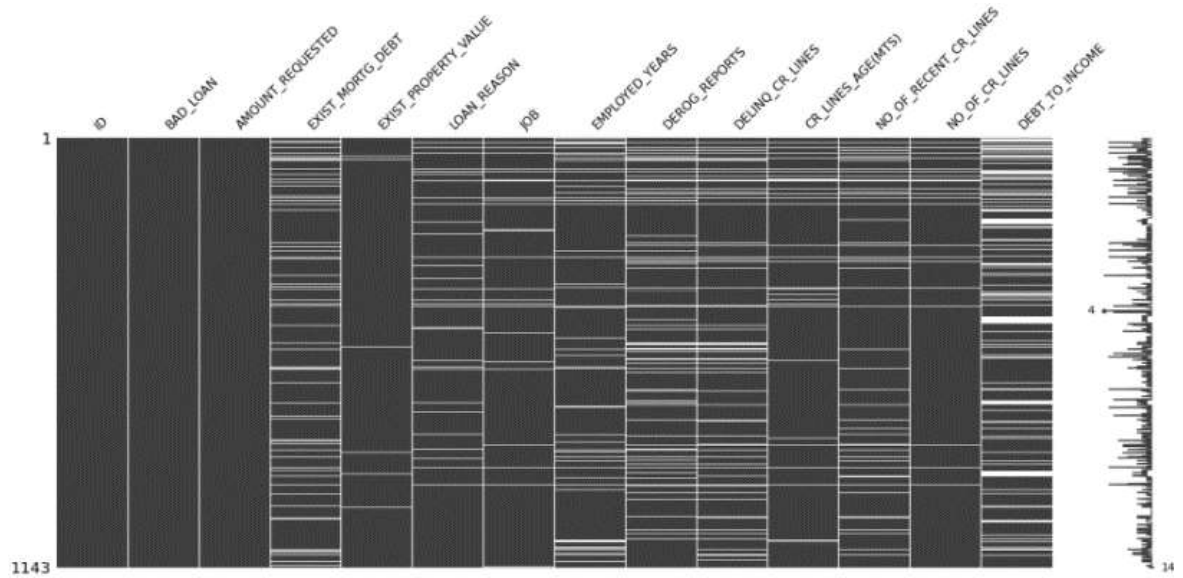
In this section, I perform the optimized data cleaning techniques in one program on the validation data-set. The purpose of the validation data-frame was to test the model on unseen data. This is so we can get a true indication of how well the model performs. I created this validation data-frame at the start of the project. This data-set will then be used to make the final predictions and check the performance of the model.

This is an important step as there may have been anomalies that were not in the Training / Testing data-set. For example, there may have been different text issues, miss spellings, outliers.

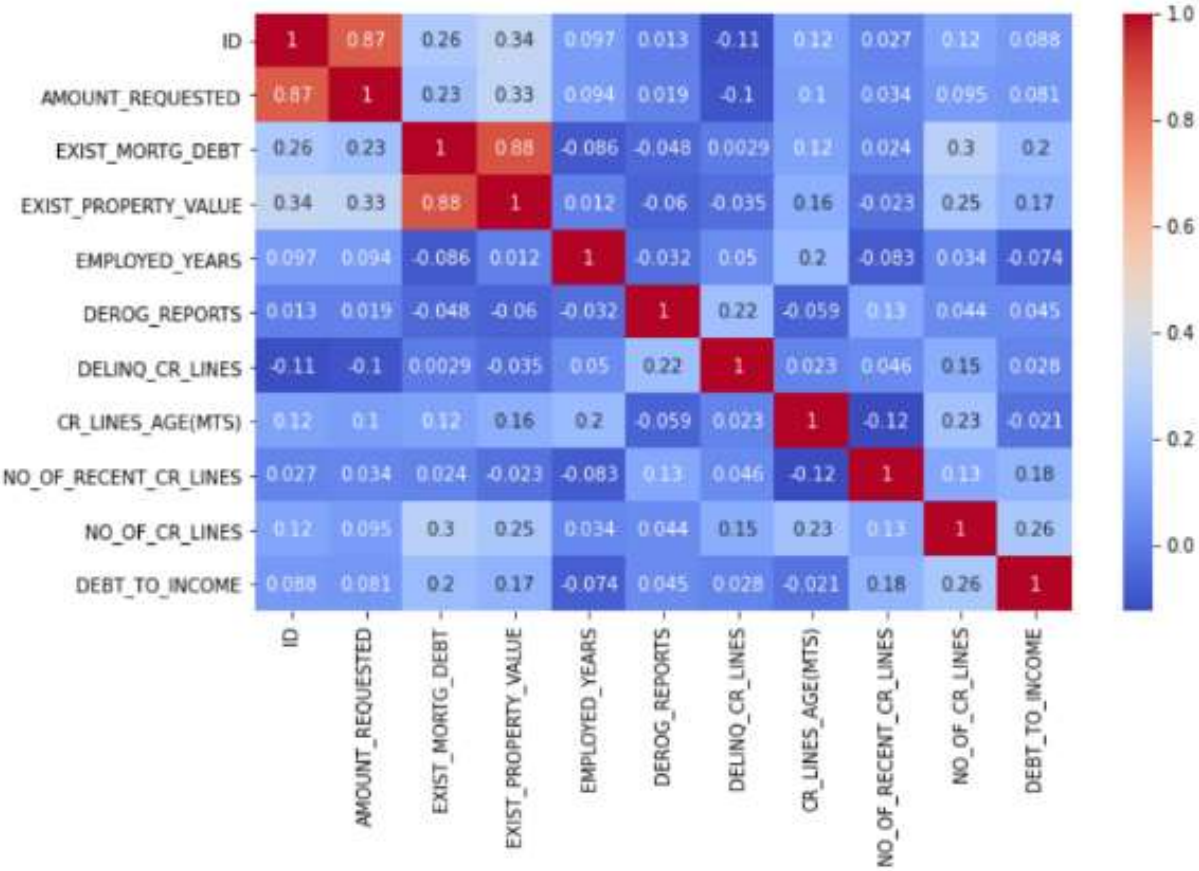
This step consisted of some of the following, but no additional anomalies were found that had not been seen already.

- Data checking the columns data types
- EDA Descriptive: shape was (1143, 14)
- EDA Exploratory

Missingness



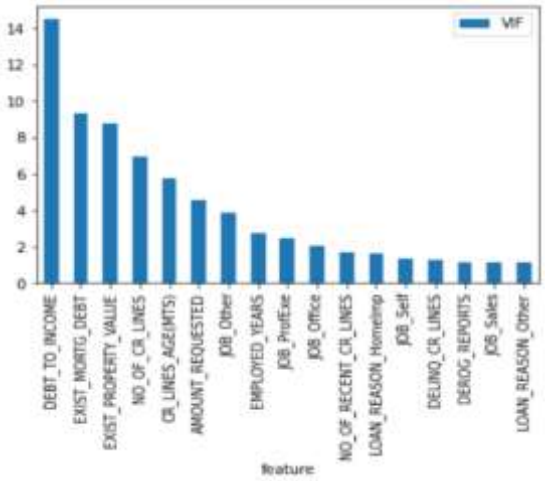
Heat Map



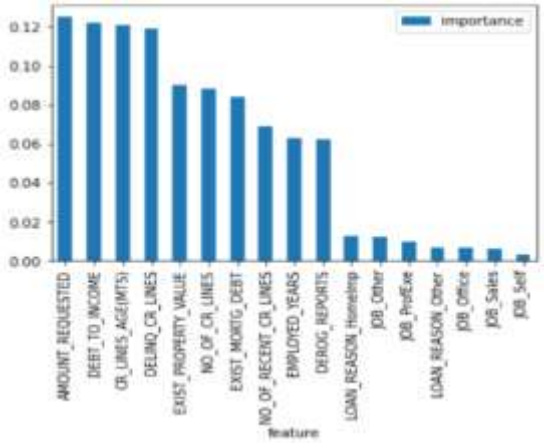
- Dropping duplicate rows

- Standardized categorial column contents: “BAD_LOAN” and “JOB” columns
- Check scatter plots - these will be reviewed in the insight section
- Imputed missing values
- imputed values for “DEBT_TO_INCOME” using KNN-Imputer
- Descriptive missing data check.

- Then I checked the data-frame for multi-collinearity



Then I checked the data-frame for feature importance



Conclusion: This data-frame was saved to **"S7_Loan_Optimised_Data_Cleaning.csv"**. After screening the scatter plots and graphs above is the changes were similar to those performed earilier. I did not have to perform any additional changes. But this should not be taken for granted. When doing a project again, I would probably do this activity for the entire data-set together, at the start before splitting in to Train-Test and Validation data-sets.

Step.8 - Model_Prediction

This step uses the optimized data-set from "Step 7 - Optimise_Data_Cleaning". The purpose of creating this new cleaned data-set was to see how accurately the model predicts on clean data. In this step, we will also review several different ways to evaluate the model. The three main metrics used to evaluate a classification model are accuracy, precision, and recall.

This step has the following actions.

1. Import Hyper-tuned model and get parameters
2. Import the optimized data and pre-process for model
3. Import Hyper-tuned model parameters and examine

4. Evaluate model predictions

Action 1 - Import Hyper-tuned Model and get parameters

In this action I imported the model code using pickle. I used the following guide to complete this activity. https://www.kaggle.com/prmohanty/python-how-to-save-and-load-ml-models

I imported the model and checked the best parameters. I was checking to see were they the same from "Step 6 - Hyper tuning". I was interested to see if any information was lost or modified in the save & load process.

Action 2. Import the Optimized data and pre-process for model

I will not spend a lot of time describing this action as it is mostly a repeat of pre-processing in Step 6. In this action I imported the optimised data, checked its shape to make sure the columns matched what the model was expecting.

df shape was : (1246, 18)
X shape is : (1246, 17)
y shape is: (1246,)

Action 3 - Import Hyper tuned model parameters and examine

For my own interest I imported the hyper tuned best parameters which I had saved and created a Random Forest Classifier model with these parameters. The purpose was just to compare how would the model saved, that was reloaded, perform compared to a model that was created by defining it parameters. The answer was no different.

Action 4 - Evaluate model predictions

Below is the evaluation metric I used to evaluate the model. While I did not need all of these, I thought it would be interesting to see how they evaluated the model's performance.

Evaluation methods used

- Method 1 - Model accuracy
- Method 2 - oob score
- Method 3 - Confusion Matrix
- Method 4 - Classification_report

Method 1 - Models accuracy

This checks the model's prediction accuracy. Classification accuracy is what we mean when we say accuracy. It is the ratio of number of correct predictions to the total number of input samples. You should have equal number of samples belonging to each class. A class being the potential prediction (Loan Repaid or Loan Defaulted). If one class is very high, say 98%. Then our model can easily get 98% training accuracy by simply predicting every training sample belonging to that class. This is a very important consideration.
https://towardsdatascience.com/metrics-to-evaluate-your-machine-learning-algorithm-f10ba6e38234
https://scikit-

learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html#sklearn.metrics.accuracy_score

Code:

```
from sklearn.metrics import accuracy_score
y_pred = model.predict(X_test)

prediction_accuracy = accuracy_score(y_test, y_pred) # fraction of correctly classified samples
print("prediction_accuracy:", round(prediction_accuracy,2))
```

Output Accuracy_score for this model: 0.876

Method 2 - oob score

The Random Forest Classifier is trained using bootstrap aggregation. The internet states that the out-of-bag (OOB) error is the average error for each calculated using predictions from the trees that do not contain in their respective bootstrap sample. This allows the Random Forest Classifier to be fit and validated whilst being trained. The result allows a practitioner to approximate a suitable value of n_estimators at which the error stabilizes. From; https://scikit-learn.org/stable/auto_examples/ensemble/plot_ensemble_oob.html

Code:

```
#get oob score
model.fit(X_train, y_train)
print("\nHypertuned - oob score:", round(model.oob_score_, 2)*100, "%")  ;pause()
```

Output oob for this model: 88%

Method 3 - Confusion Matrix

A confusion matrix gives you a lot of information about how well your model does. When performing classification predictions, there's four types of outcomes that could occur.

- *True positives: are when you predict an observation belongs to a class and it actually does belong to that class.*
- *True negatives: are when you predict an observation does not belong to a class and it actually does not belong to that class.*
- *False positives: occur when you predict an observation belongs to a class when in reality it does not.*
- *False negatives: occur when you predict an observation does not belong to a class when in fact it does.*

These four outcomes are plotted on a confusion matrix for binary classification.
https://www.jeremyjordan.me/evaluating-a-machine-learning-model/

Code:

```
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import confusion_matrix

predictions = cross_val_predict(model, X_train, y_train, cv=5)
confusion_matrix_results = confusion_matrix(y_train, predictions)
print("\nConfusion Matrix: \n",confusion_matrix_results)
print("\nConfusion Matrix: \nThe first row is about the not-target-predictions:")
print("True negatives - correctly classified as not Target: ", confusion_matrix_results[0][0])
```

```
print("False negatives - wrongly classified as not Target: ",confusion_matrix_results[0][1])
print("\nThe second row is about the Target-predictions:")
print("False positives - wrongly classified as Target: ", confusion_matrix_results[1][0])
print("True positives - correctly classified as Target: " ,confusion_matrix_results[1][1]);pause()
```

Confusion Matrix for this model: I am note really impressed with the high number of false negatives, this suggests that the model predicted a lot of actual positive targets incorrectly (bottom row of confusion matrix). The model performed a lot better of the negative targets (top row of confusion matrix).

TN:932	FP:3
FP:103	TP134

Accuracy for the confusion matrix can be calculated by taking average of the values lying across the "main diagonal"

Accuracy = (TP + TN) / Total sample

Confusion Matrix forms the basis for the other types of metrics. I got the following accuracy score using this formula: 0.909

Method 4 - Classification_report

The Classification report shows the main classification metrics. It computes the Precision, Recall, F-measure and support for each class.

The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The recall is intuitively the ability of the classifier to find all the positive samples.

The F score can be interpreted as a weighted harmonic mean of the precision and recall.

The support is the number of occurrences of each class in y_true.

```
from sklearn.metrics import classification_report
target_names = ['class 0', 'class 1']
print(classification_report(y_true, y_pred, target_names=target_names))
```

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_fscore_support.html#sklearn.metrics.precision_recall_fscore_support

Code:

```
#Classification report
from sklearn.metrics import classification_report

print("Test scores:")
y_pred = model.predict(X_test)
print(classification_report(y_test, y_pred));pause()
```

```
Validation scores, Classification Report:
              precision    recall  f1-score   support

         0.0       0.90      1.00      0.95       935
         1.0       0.98      0.57      0.72       237

    accuracy                           0.91      1172
   macro avg       0.94      0.78      0.83      1172
weighted avg       0.92      0.91      0.90      1172
```

This is an excellent report as it gives all the metrics so far in one report. The only issue I have with it is that it can be hard to pull the individual results from it. For example, there is a function to pull recall from the report, but it only gives it for the positive result. In this report, at recall of 0.57 I would suggest that the model is struggling to predict the positive class of outcome - defaulted loans which could be dangerous for a bank of financial institution if they were to rely on this model.

Deep Learning

This section briefly examines how a deep learning model would perform on this data-set. To complete this I built a program to find the optimal parameters and then used these to create a model. I used the following article to help me.
https://towardsdatascience.com/designing-your-neural-networks-a5e4617027ed
https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/

Here I was interested to compare the predictive performance with machine learning. I used the clean data-set to Train and Test for the model parameters. This data-set was developed in Step 4:

'S4_Loan_Optimised_Data_Cleaning.csv'

and I used the validation data from Step 7, to test the model on clean data:

'S7_Loan_Validation_Optimised_Data_Cleaning.csv'

Parameter's investigation: I created a for-loop to iterate through list of parameters. This was very interesting. I was amazed at how many parameters there are to consider. I wanted to iterate through the parameters and plot the different results. Some parameter consideration included:

- What type of optimizer was I to going to use? I trailed the Adam optimization algorithm (ADAM) and Stochastic gradient descent (SGD).
- For those optimisers, what parameters was I going to set - learning rate, momentum? After trial and error

and enough time I decided to use the standard parameters for the Optimiser - for example 0.01 learning rate for SGD.

- How many epochs or complete training passes was I going to make? I set this to 1000 as I was using an early stop.
- Was used the Early-Stop to monitor a break-out after a training pass if there was no change in loss or Accuracy? I initially tried 'accuracy' but I could not get the model to make reliably consistent prediction. So I used the 'loss' monitor to check for change. Set patients to 5.
- How many layers was I going to use? I read on the internet that 1 layer was not enough and eventually decided to use 3 layers.

https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw

https://www.heatonresearch.com/2017/06/01/hidden-layers.html

From the internet it suggests there are many rule-of-thumbs for determining the correct number of neurons to use in the hidden layers, these include the following:

1. The number of hidden neurons should be between the size of the input layer and the size of the output layer.
2. The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
3. The number of hidden neurons should be less than twice the size of the input layer.

These three rules provide a starting point for me to consider. however, the selection of an architecture for my neural network came down to trial and error.

- What activation would I use for the hidden layers? I only tried ''relu'.
- What would I set the output layer activation to? As it was binary output classifier, I set it to activation='sigmoid'.
- How many nodes per layer? To discover this I tried various combinations, from 128/64/32/16 to 3/3/3. eventually I created a program to loop through:

```
hidden_layers  = range(10, 30, 2)
hidden_layers1 = range(5, 30, 2)
hidden_layers2 = range(3, 30, 2)
```

- What would I set the batch to, how many samples to process before the batch update? I initially tried: *range(8, 12, 1)* but settled using *batch = 10*

- How would I evaluate model? I will discuss this shortly but first to note that you can try any combination of parameter if you have the time. Time becomes key as it takes so long when you try different combinations. Then unfortunately there are the errors which can spoil a couple of days work. One such error was my miss-interpretation of the confusion matrix. The error was that I confused the True Negative and the True Positive. When you search the internet, you must be careful which version of the confusion matrix you use. For example, this chart shows TP as Top left:

However, when examining the output from the confusion matrix, I should have been examining it like this:

TN = 5 [0][0]	FP = 2 [0][1]
FN = 0 [1][0]	TP = 3 [1][1]

This led me to this article on the problem:
https://towardsdatascience.com/the-two-variations-of-confusion-matrix-get-confused-never-again-8d4fb00df308

I investigated further. I ran the following code to help me understand how I should be interpreting the confusion matrix properly. Confusion matrix is a great name after all!!
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.plot_confusion_matrix.html

```
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.metrics import plot_confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

X, y = make_classification(n_samples=20,n_features=4,random_state=0);print(X);print(y)
X_train, X_test, y_train, y_test = train_test_split(X, y,train_size=0.5, random_state=0)

clf = SVC(random_state=0)
clf.fit(X_train, y_train)

plot_confusion_matrix(clf, X_test, y_test)
plt.show()

y_pred = clf.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print("y_test: ",y_test)
print("y_pred: ",y_pred)
```

Output:

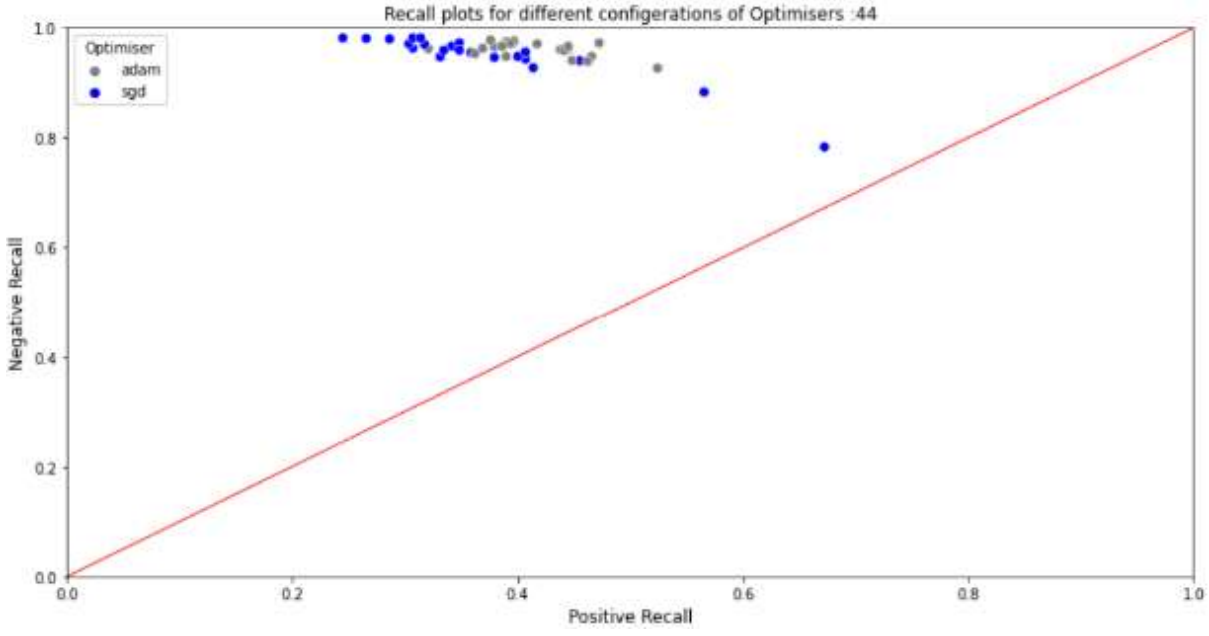
```
y_test:  [0 1 1 0   1 0 1 0 0   1]
y_pred:  [0 1 1 1fp 1 0 1 0 1 fp1]
```

[[3 2]
[0 5]]

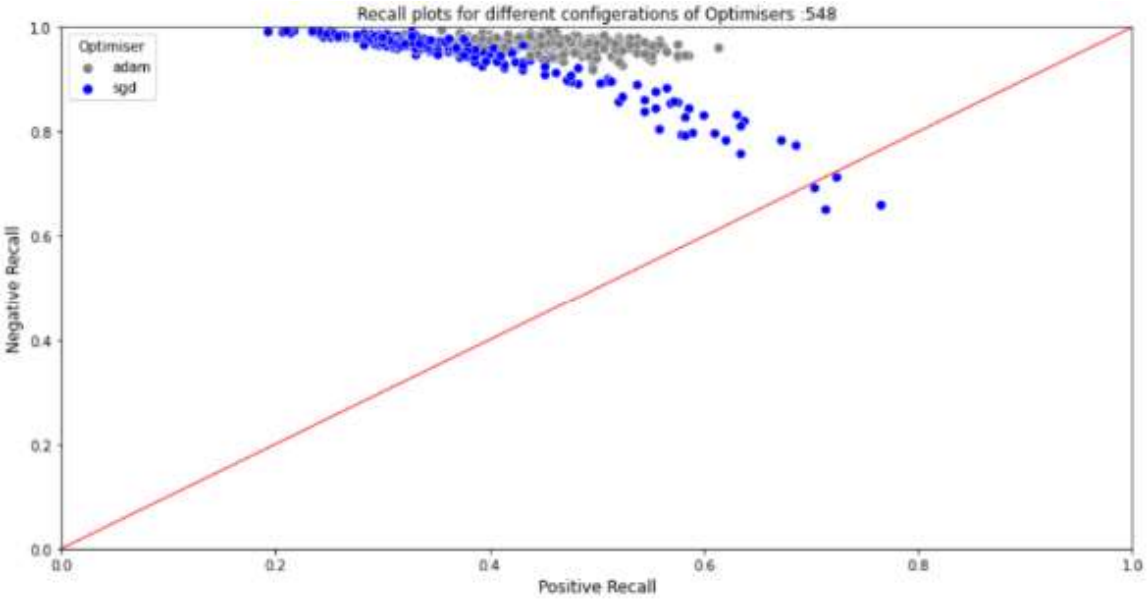
FP = confusion_matrix_results[0][1] = 2
TN = confusion_matrix_results[0][0] = 3
FN = confusion_matrix_results[1][0] = 0
TP = confusion_matrix_results[1][1] = 5

I like the recall metric. I think it is a clean result and better than an accuracy scores alone. This is because it can be used to measure the total amount of correct positive prediction with the total amount of negative predictions to evaluate the model.

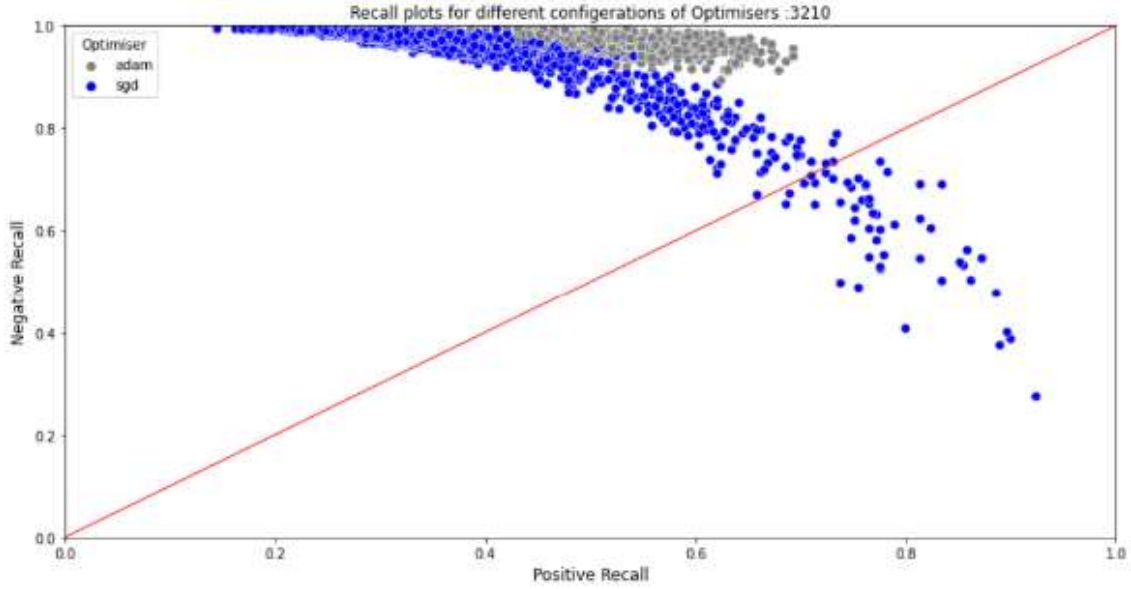
Below you can see were the scatter-plot of the positive recall and negative recall changes as the program iterates through the various parameters. This is soon after I have started to iterate through the loops, this is the 44th loop. Notice the SGD optimiser is starting to perform best as I iterate through the loops. Please note this is using the test data from step 4 and is a subset of the original data-set. Nodes per layers are approximately: 10/7/5



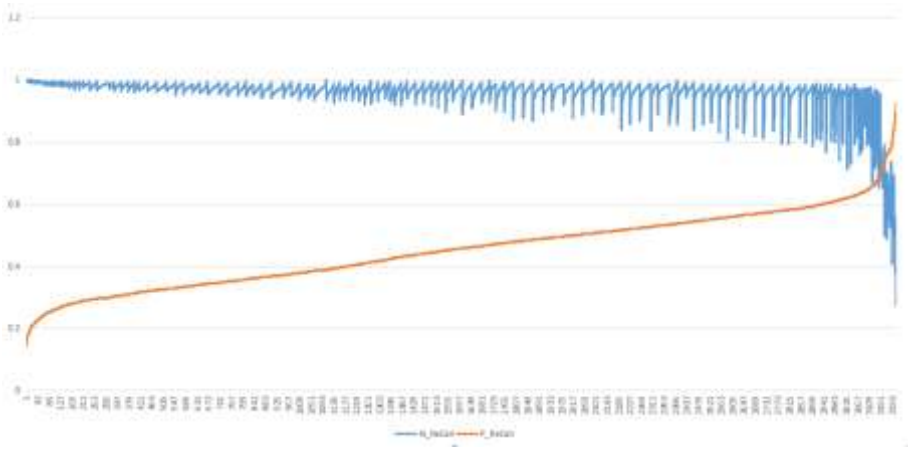
As the program iterated on through the different nodes per layer you can see below that the test set are converging from left to right toward the centre, but ADAM optimiser is lagging SGD. Nodes per layers are approximately now: 12/17/19. This is the Test data-set.



This chart is after 3210 iterations. We are now at approximately 26 nodes on the first layer. Below you can see the test data-set which the SGD optimiser appears to be the best performing.



I charted the Positive and Negative Recall to gauge the best parameters. It's interesting the fluctuation in the negative class recall result.



My aim was to located parameters that would give the best positive recall and negative recall. And looking at the scatter plot you can see this is where the SGD optimiser performed best for this data-set. I used the file to search positive and negative recall at 0.6.

One strange issue I had was that the behaviour of the model on the test data-set was very different to the behaviour of the model on the validation data-set. I rechecked my code several times as I thought I had mixed up a variable or made negative a setting but was unable to find such an error.

So, I ended up using a parameter that gave the best recall figures on both test and validation data-sets. This meant that I ended up using the ADAM optimiser. I am not able to spend any more time on this topic at this time, but it is something I would like to investigate further.

The final settings of my model are as follows.

```
model = Sequential()
model.add(Dense(26, input_dim=n_cols, activation='relu'))#input_dim=8
model.add(Dense(21, activation='relu'))
model.add(Dense(19, activation='relu'))

model.add(Dense(1, activation='sigmoid'))#model.add(Dense(2, activation='softmax'))
model.compile(loss='binary_crossentropy', optimizer=my_otimizer, metrics=['accuracy'])#sgd#adam

early_stopping_monitor = EarlyStopping(monitor='loss', patience=5)
model.fit(X, y, epochs=1000, batch_size=10, callbacks=[early_stopping_monitor])
```

Validation Data-set Results

Below are the predicted results using the Random Forest Classifier Hyper-tuned model in result 1 and the Keras deep learning model in results 2. Provided the precision result is above 85% I only focus on the recall number for each class of prediction. For me this is how I gauge how well the model performed at each class of prediction. The Train / Test predictions are not being replicated in the validation data? The support is an interesting number as it provides a way to assess was the data-set biased, ideally this should be a 50:50 split.

Result 1 - Random Forest Classifier. These results are not great, to me I have a problem as it feels like the

model is biased to predict a negative outcome:

```
Validation scores, Classification Report:
              precision    recall  f1-score   support

         0.0       0.90      1.00      0.95       935
         1.0       0.98      0.57      0.72       237

    accuracy                           0.91      1172
   macro avg       0.94      0.78      0.83      1172
weighted avg       0.92      0.91      0.90      1172
```

Result 2 - Keras model below, deep learning 26/21/19 - activation='sigmoid'. These results are also not great, to me I have a problem as it appears the model is failing to accurately predict the negative class:

```
Validation scores, Classification Report:
              precision    recall  f1-score   support

         0.0       0.94      0.20      0.33       935
         1.0       0.23      0.95      0.37       237

    accuracy                           0.35      1172
   macro avg       0.59      0.58      0.35      1172
weighted avg       0.80      0.35      0.34      1172
```

Insights

Many insights were omitted due to file size constraints, some further examples are below.

- Project plan

  The importance of a plan was critical for me in completing this project in a systematic way. Before I constructed the plan, I was more randomly completing actions and not really achieving anything. The problem statement formed the foundation of the project and really helped guide my analysis.

- Data-set

  When I started to do this project, my focus was on running & optimizing the model. In retrospect while this is very important my focus may have been miss guided. Looking back, I now think that examining the data visually, descriptively, imputing and feature engineering are perhaps more important. I am not saying that tuning - hyper tuning the model is not important. This is very important and needs to be correct for the data-set. What I am suggesting, is that this can be relatively automated, if you have the time and computational power, you can check any parameter combinations available. But this is relatively easy and available to every data analyst. It is optimizing your data, feature engineering that appear to be key to your model gaining an edge.

- Quality of data-set

  As noted above this is very important. Getting the right data-set is key. The considerations include,

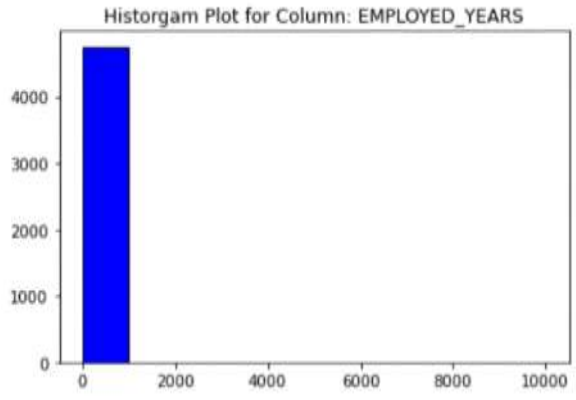
  1) Does this satisfy your problem statement?

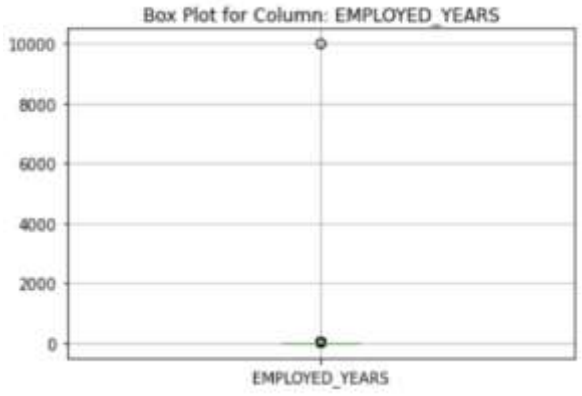
2) Does it have the appropriate number of rows?
3) Does it have an adequate number of columns?
4) How much missing data is there and where?
5) Can you get details on the feature to aid your understanding?
6) Is it possible to feature engineer?

If I was running this project again and wanted to do a loan prediction project, I am not sure that this data-set is complete enough to do a project like this. I found it hard to get detail from the internet that would allow me to understand the features adequately enough. This made it difficult to make meaningful imputations and feature engineering.
For example, there was no income feature to back-engineer the “Debt-to-Income” feature for missing data. I could not get any information on some of the features, so I could not say with any certainty that some of box-plot outliers were outliers.

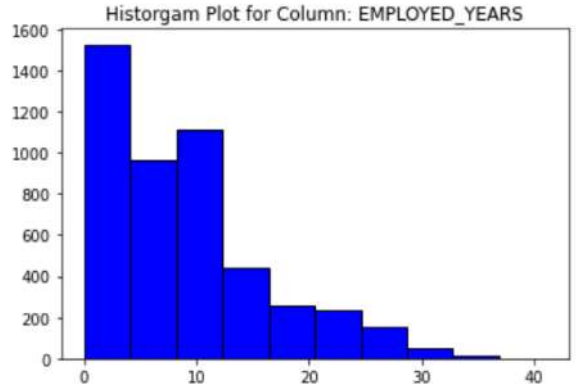
- Interpreting Features

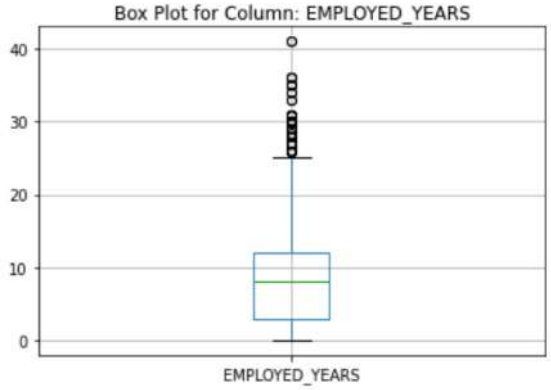
Understanding the importance of a feature to a project is very important and I used several approaches. I used scatter plots, box plots, histograms but it was with experience I realized how misleading these plots can be. For example, when I looked at the employed years charts below, I initially missed how the single outlier was affecting the chart. It was only with some experience that I realized this must be an outlier and when I changed it for the mean, you can see how chart changed to display more valuable information.





#replacing outlier with mean of column
EMPLOYED_YEARS = round((df['EMPLOYED_YEARS'].mean()),1)
df['EMPLOYED_YEARS'].replace(9999,EMPLOYED_YEARS,inplace=True)





Although with a bit more experience I now find that these charts are also a bit limited. I now use a combination of Sea-born scatter plots, violin-plot, and strip-plots. With these I used hue="BAD_LOAN". This give a better

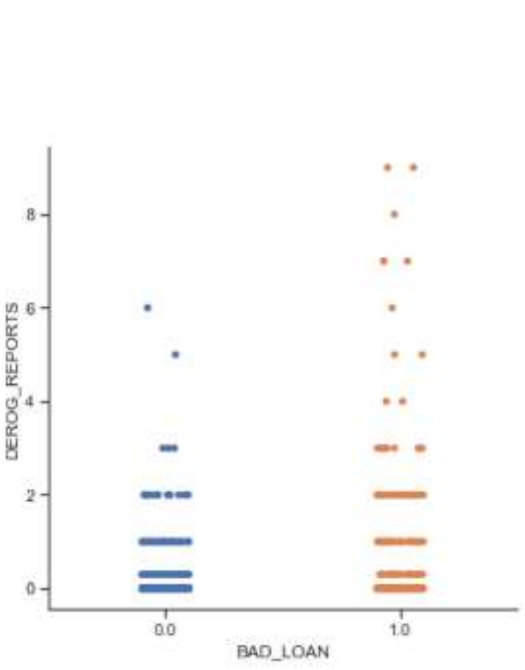
clue to what is happening in the feature or relationship. Using the code below, I will briefly describe any insights or observations in relationships using a strip-plot for Bad Loans against the other features. I will comment on the significant charts.

Code:

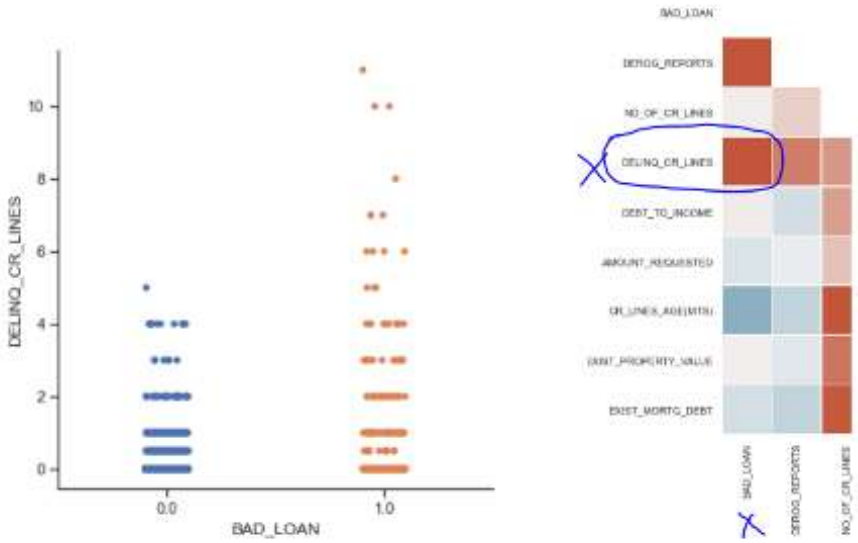
```
#list of features o be examined
Examine = ['BAD_LOAN']
#list of all fetaure i desired to do relationship against
Examine2 =['BAD_LOAN', 'AMOUNT_REQUESTED', 'EXIST_MORTG_DEBT', 'EXIST_PROPERTY_VALUE',
'EMPLOYED_YEARS',  'DEROG_REPORTS', 'DELINQ_CR_LINES', 'CR_LINES_AGE(MTS)',
'NO_OF_RECENT_CR_LINES', 'NO_OF_CR_LINES',   'DEBT_TO_INCOME', 'LOAN_REASON_HomeImp',
'LOAN_REASON_Other', 'JOB_Office', 'JOB_Other', 'JOB_ProfExe',  'JOB_Sales', 'JOB_Self']

# loop1 to go through list of features
for col in df[Examine]:
    #loop2 to go through list of features
   for col2 in df[Examine2]:
       #create a relationship plot
       sns.catplot(x=col, y=col2, hue="BAD_LOAN", data=df)
       #show plot
       plt.show()
```

Bad loans - “1” is the loan was defaulted on. “0” is the loan was repaid.

Derogatory Reports: the internet states derogatory information is any reported negative credit information which can be used to deny an individual a loan. They may be for late payments etc and it’s probably not surprising to see higher loan defaults the more derogatory reports that exists. This was also reflected in the heat map below.

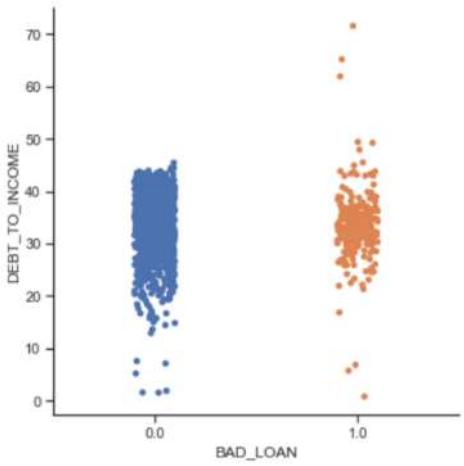


Delinquent credit lines: The internet states that a line of credit becomes delinquent when a borrower does not make the minimum required payments 30 to 60 days past the day on which the payments were due. At this point, the lender often stops the line of credit until payments have caught up with the payments that are due. It's probably not surprising to see higher loan defaults with increased Delinquent credit lines.



Credit line age (months): one internet page suggests this is to do with the applicant age and another suggest credit line age is to do with how long the loan is in place. I initially thought this was to do with a person's age in months, but its lower limit is too low. An 18-year-old would be 216 months which is not the case below. Regarding credit time in place, the internet suggests a long track record without any major slip-ups suggests that your credit behaviour will be similar in the future — and lenders and credit card issuers like that. This appears correct below but what is odd is the outliers. The 360 months would suggest a 30-year mortgage but without being able to interrogate the source of the data I was not confident in correcting the outliers (600 months) to perhaps the mean of the set or 360-month standard mortgage period?

Debt to Income: The internet suggests your “debt-to-income” ratio is all your monthly debt payments divided by your gross monthly income. This number is one-way lenders measure your ability to manage the monthly payments to repay the money you plan to borrow. 43% is seen as a cut off in lending and this is apparent in the chart below. https://www.consumerfinance.gov/ask-cfpb/what-is-a-debt-to-income-ratio-why-is-the-43-debt-to-income-ratio-important-en-1791/ I would have expected the impact on whether a loan was to be repaid or not, to be more pronounced in this chart. With higher “debt-to-income” figures suggesting a stronger link to loan default.

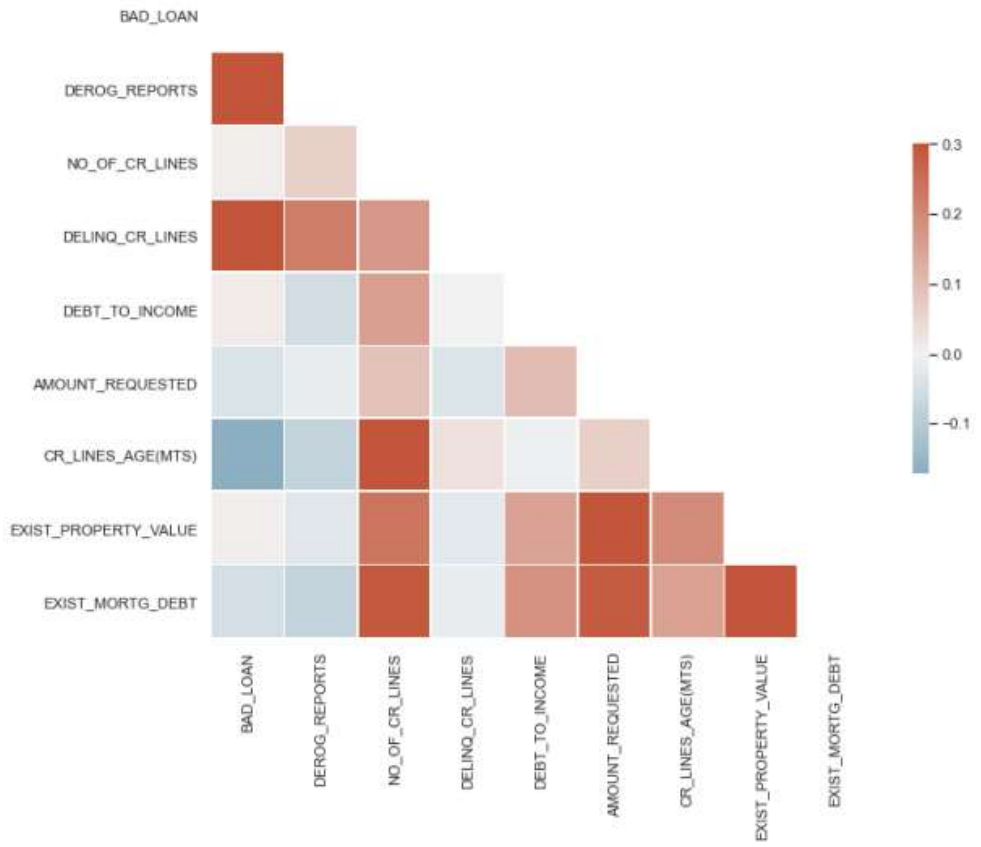


For the next part of the discussion, I created nested for-loops, where I looped through the columns to examine relationships. To focus the discussion. I will use the following pieces of information: Feature VIF, Importance and Correlation plot. I will take the top 5 items from each of the lists below and compare in a heat map below.

feature	VIF
EXIST_PROPERTY_VALUE	12.524296
DEBT_TO_INCOME	11.577433
EXIST_MORTG_DEBT	11.128488
NO_OF_CR_LINES	6.783816
CR_LINES_AGE(MTS)	5.961612
AMOUNT_REQUESTED	4.376698
JOB_Other	3.634433
EMPLOYED_YEARS	2.729321
JOB_ProfExe	2.383282
JOB_Office	1.950662
NO_OF_RECENT_CR_LINES	1.630586
LOAN_REASON_HomeImp	1.549707
JOB_Self	1.296268
DELINQ_CR_LINES	1.252867
DEROG_REPORTS	1.182894
JOB_Sales	1.129283
LOAN_REASON_Other	1.089437

importances.head(15):

feature	importance
DEBT_TO_INCOME	0.129
DELINQ_CR_LINES	0.121
AMOUNT_REQUESTED	0.112
CR_LINES_AGE(MTS)	0.104
EXIST_PROPERTY_VALUE	0.100
EXIST_MORTG_DEBT	0.096
NO_OF_CR_LINES	0.090
EMPLOYED_YEARS	0.068
DEROG_REPORTS	0.059
NO_OF_RECENT_CR_LINES	0.055
LOAN_REASON_HomeImp	0.014
JOB_Other	0.014
JOB_Office	0.011
JOB_ProfExe	0.010
LOAN_REASON_Other	0.006

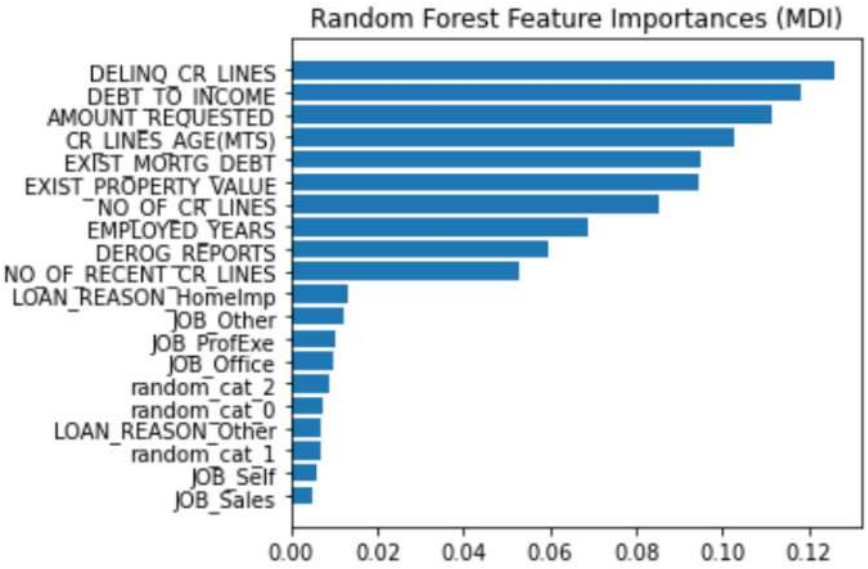


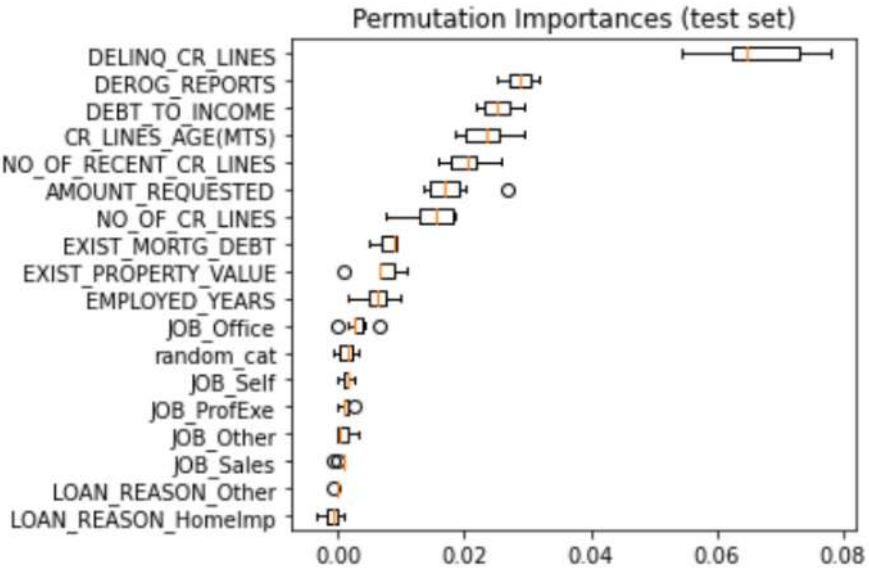
From the VIF table and importance table the following features look like their impact is significant on the model. These are; *['NO_OF_CR_LINES', 'DELINQ_CR_LINES', 'DEBT_TO_INCOME', 'AMOUNT_REQUESTED', 'CR_LINES_AGE(MTS)', 'EXIST_PROPERTY_VALUE']*

However, examining the correlation heat map, we also expected to see *['DEROG_REPORTS']* but this did not appear highly in the feature importance table. It's interesting from the heat map that there does exist correlation between Amount Requested and Existing Mortgage Debt, Amount Requested and Existing Property Value. I tend to only take the correlation to mean something significant at extremes. The correlation heat map is a very interesting way to examine the data. However, it should not be used in isolation.

I have since used code I got from sci-kit-learn to examine feature importance. This example from them compares the impurity-based feature importance of Random Forest Classifier with the permutation importance on the bad loan data-set using permutation_importance. The example shows that the impurity-based feature importance can inflate the importance of numerical features. It also suggests the impurity-based feature importance of random forests suffers from being computed on statistics derived from the training data-set. The importance can be high even for features that are not predictive of the target variable, as long as the model has the capacity to use them to over fit. This is an area of focus I will have to study some more to fully understand.

This example shows how to use Permutation Importance as an alternative that can mitigate those limitations. https://scikit-learn.org/stable/auto_examples/inspection/plot_permutation_importance.html#sphx-glr-auto-examples-inspection-plot-permutation-importance-py





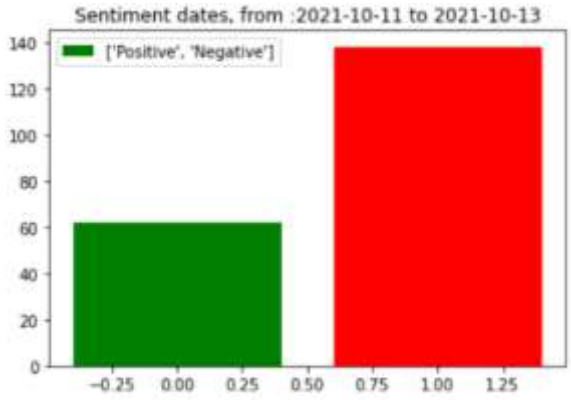
- Model evaluation

This took me awhile to get my head around. My instinct was to interpret a good accuracy score as sufficient in reviewing a model's performance. I was seriously mistaken. When I examined a classification report properly you start to understand the importance of having a balanced target to predict on. Similar number of Target classes. It's important we use precision, recall, and have a balance target group. I really like how the confusion matrix describes the data. My aim is predicting as few False Positives and False Negatives.

Appendix 1 - API - Twitter

Below is code that I have used in other projects. I am collecting the sentiment from twitter which can be positive or Negative against a key word. In this case I put in **"AI"** and the search dates where start = '2021-10-11' to end = '2021-10-13'. Twitter only allows you go back 7 days for a data search and collect a max of 3500 calls in that period. I am currently using this to look at Bitcoin sentiment and used it in another project I did with UCD introduction to data Analytic. I am collecting sentiment data but did not have enough to use in this machine

learning project. The program produces a plot of Positive versus Negative sentiment. The API code is quite simple, and it is amazing how much you can get done in python with two lines of code. Below is the output for the search on “AI” and comments are in the code of what it is doing.



This is where I got the code for this Sentiment analysis.

https://www.youtube.com/watch?v=dSOUd9Sm1gI

```
import tweepy
import textblob
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import re
```

I got keys by logging onto Twitter and creating a developers account.

```
# keys and tokens from the Twitter Dev Console
consumer_key = ''
api_key = consumer_key
consumer_secret = 'b'
api_key_secret = consumer_secret
access_token = '9'
access_token_secret = 'tI'
```

I used this to hide my keys which I put in a file in the same location as I opened the python file from

```
# all_keys = open('twitterKeys','r').read().splitlines()

# api_key = all_keys[0]
# api_key_secret = all_keys[1]
# access_token = all_keys[2]
# access_token_secret = all_keys[3]
```

This handle opening the API link and handshakes

```
#Connecting to API
authenticator = tweepy.OAuthHandler(api_key,api_key_secret)
authenticator.set_access_token(access_token, access_token_secret)
```

This opens the communication channel

```
#Create the API
TwitterAPI = tweepy.API(authenticator,wait_on_rate_limit=(True))
```

I set the Topic to search for here and the look back period

```
#Topic to check for sentiment
topic = 'AI'
search = f'#{topic} - filter:retweets'

#search peroid - look back max 7 days
start   = '2021-10-11'
end     = '2021-10-13'
```

We make the search here and the number of tweets to search - you have a max search so you have to spread of the look back period

```
# createing cursor
tweet_cursor = tweepy.Cursor(TwitterAPI.search, q=search, lange='en', until=end, since=start,
tweet_mode='extended').items(200)#since goes back 7 days
```

This collects the Tweets

```
#getting tweets
tweets = [tweet.full_text for tweet in tweet_cursor]
```

Now we are putting them in to a data-frame and using REGEX to sub out characters (@, #) we do not want for the Tweets data-frame.

```
#turn in to data frame and get polarity
tweets_df = pd.DataFrame(tweets, columns=['Tweets'])
```

Using a for-loop we iterate through the rows

```
for _,row in tweets_df.iterrows():
    row['tweets'] = re.sub('http\s+',   '', row['Tweets'])#substitute out
    row['tweets'] = re.sub('#\s+',      '', row['Tweets'])#substitute out
    row['tweets'] = re.sub('@\s+',      '', row['Tweets'])#substitute out
    row['tweets'] = re.sub('\\n',       '', row['Tweets'])#substitute out
```

Using textblob we identify the sentiment of the tweets

```
#identift the number of positive and negative tweets with detail above substituted out
tweets_df['Polarity'] = tweets_df['Tweets'].map(lambda tweet:textblob.TextBlob(tweet).sentiment.polarity)
tweets_df['Result'] = tweets_df['Polarity'].map(lambda pol: '+' if pol > 0 else'-')
```

I am creating the plot of the sentiment below, first get the counts and second plot it

```
#Count number of negative and positive tweets
positive = tweets_df[tweets_df.Result == '+'].count()['Tweets']
negative = tweets_df[tweets_df.Result == '-'].count()['Tweets']

#Create plot
plt.bar([0,1], [positive, negative], label=['Positive', 'Negative'], color=['green', 'red']) # colort positive = green and neg = red
plt.legend()
plt.title (f'Sentiment dates, from :{start} to {end}')
plt.show()
```

Appendix 2 - Regex

The files associated with this are.

- 'S1_Data_Gathering_Titanic.csv' is input file
- Regex test.py is the project file
- 'S1_Data_Gathering_Regex_output.csv

Below is an example of code I created to use REGEX to get the prefix of the passengers for the Titanic data. This is so that I so that I could assign an age base on the prefix for missing rows. REGEX was not required in this project, so I am using this as an example I created for my Titanic submission.

1st I imported the file and set the strings to upper case to remove any errors with capitalization.
I used REGEX to get the Prefix. First, I substituted the string with “” up to the comma, then substituted the string with “” after the full stop. This was then saved to a list called prefix. I then use the set function to identify the unique contents of the prefix list.

2nd I created a new column for each of the prefix’s. I used the contains function to put a true where the prefix was contained in the “Name” column.

3rd I created a for loop to iterate through the list of prefixes and if the prefix was the same as a pre-set prefix like “Mr” or “Mrs” etc I loaded the data in the “Age” column at that row to the prefix column.
I used https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.iloc.html to help with this. I created different if-conditional statements for each prefix. This would be better achieved with a function as the code is repeated had I more time.

4th I then got the mean age for each of the prefixes. Basically, the mean of the column. This was my output

```
# Mean age on Titanic:  29.7
# imputed_age_Mr:  32.4
# imputed_age_Mrs:  35.9
# imputed_age_Miss:  21.8
# imputed_age_Don:  40.0
# imputed_age_Master:  4.8
```

5th In this step was basically looping through the rows of “Age” and where they were empty inserting the mean for that prefix. If the prefix was not available, I used the mean of the column. I had to reset the True and False of the prefix column as this was used in the conditional check. I set the column location by their column name location.

```
col_no_ID = df.columns.get_loc('PassengerId')
```

As the columns could shift if I created a new column in the project and this would put the program in error, so I made them dynamic. There is no doubt in my mind that there is an easier way to achieve what I have achieve in much less code and on reflection I could have used function to reduce the number of rows required.

Output of changes for missing ages, you can see that it reads in the name, location and empty age cell “nan” and repeats with the mean age for that prefix in place of the “nan”.

```
1111111:  693 LAM, MR. ALI nan
Changed to:  693 LAM, MR. ALI 32.4

3333333:  698 MULLENS, MISS. KATHERINE "KATIE" nan
change to:  698 MULLENS, MISS. KATHERINE "KATIE" 21.8

5555555:  710 MOUBAREK, MASTER. HALIM GONIOS ("WILLIAM GEORGE") nan
change to:  710 MOUBAREK, MASTER. HALIM GONIOS ("WILLIAM GEORGE") 4.8
```

Code:

```
import pandas as pd
import numpy as np
import re

#import files for project - Load df from file
filename = 'S1_Data_Gathering_Titanic.csv'
df = pd.read_csv(filename)
print("\nLoaded df.shape: ", df.shape);print("\ndf.info(): ",df.info())

#to convert all strings in "Name"to upper case just case there is a mix in the data
df['Name'] = df['Name'].str.upper()

#==============================================================================
# 1st we make a list fo prefixs
#==============================================================================

# fill in missing ages using MR, Master, Mrs, Ms, Don, other. This is the use of REGEX for the Titanic data set to
# get a mena age appropiate to the prefix.

#find prefixs in Name column
prefix = [] # list to hold prefixes
#iterate through rows of dataframe column
for row in df['Name']:
    # strip after comma - start of string
    row = re.sub(r'^.*,\s','',row)
    # strip after full stop - end of string
```

```
    row = re.sub(r'(?<=\.)[^.]*$','',row)
    #append prefix to last prefix in list - prefix
    prefix.append(row)

#used for testing
# mylist = ['nowplaying', 'PBS', 'PBS', 'nowplaying', 'job', 'debate', 'thenandnow', 'impet', 'impet','impet']
# myset = set(mylist)
# print(myset)

#set() -> new empty set object set(iterable) -> new set object
#Build an unordered collection of unique elements.
unique_prefix = set(prefix)
print(unique_prefix)

#output was:

#{'JONKHEER.', 'MAJOR.', 'THE COUNTESS.', 'MR.', 'MISS.', 'MRS. MARTIN (ELIZABETH L.',
#'COL.', 'MME.', 'DR.', 'MS.', 'SIR.', 'LADY.', 'MASTER.', 'MRS.', 'CAPT.', 'REV.', 'DON.', 'MLLE.'}

# I am only selecting the following from the list for convienance - issue with would need to be resolved
list_of_prefixs = ('Mr','Mrs','Miss','Don','Master')

#==============================================================================
# 2nd  we have to find the title and make a colum where its True or false for each prefix
#==============================================================================

#create a new column in dataframe with prefixs - find rows in `df` which contain r'\,\sTEXT.\s'
# df['Mr'] = (df[df['Name'].str.contains(r'\w,\sMR.\s\w')])# if upper() set
df['Mr'] = df['Name'].str.contains(r'\,\sMR.\s')
df['Mrs'] = df['Name'].str.contains(r'\,\sMRS')
df['Miss'] = df['Name'].str.contains(r'\,\sMISS.\s')
df['Don'] = df['Name'].str.contains(r'\,\sDON.\s')
df['Master'] = df['Name'].str.contains(r'\,\sMASTER\.\s')


#==============================================================================
#3rd interate through the list of prefixs and piut age in to the associated column prefix
#==============================================================================

for prefix in list_of_prefixs:
    # if iterate is the same as prefix do code
    if prefix == 'Mr':
        # find the column index no for prefix
        col_no = df.columns.get_loc(prefix)
        #f find the column number for Age
        col_no_age = df.columns.get_loc('Age')
        #using key and value take number for column with this prefix
        #interate through the prefix column rows and is the value is true do condition
        for index, value in df[prefix].items():
            if value == True:
                #https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.iloc.html
                #assign the value in the age column at that location to the prefix column
                df.iloc[index, col_no] = df.iloc[index, col_no_age]

    if prefix == 'Mrs':
```

```
        col_no = df.columns.get_loc(prefix)
        col_no_age = df.columns.get_loc('Age')
        for index, value in df[prefix].items():
            if value == True:
                df.iloc[index, col_no] = df.iloc[index, col_no_age]

    if prefix == 'Miss':
        col_no = df.columns.get_loc(prefix)
        col_no_age = df.columns.get_loc('Age')
        for index, value in df[prefix].items():
            if value == True:
                df.iloc[index, col_no] = df.iloc[index, col_no_age]

    If prefix == 'Don':
        col_no = df.columns.get_loc(prefix)
        col_no_age = df.columns.get_loc('Age')
        for index, value in df[prefix].items():
            if value == True:
                df.iloc[index, col_no] = df.iloc[index, col_no_age]

    if prefix == 'Master':
        col_no = df.columns.get_loc(prefix)
        col_no_age = df.columns.get_loc('Age')
        for index, value in df[prefix].items():
            if value == True:
                df.iloc[index, col_no] = df.iloc[index, col_no_age]

#==============================================================================
# 4th Get mean of Ages in prefix column
#==============================================================================
age_mean = round((df["Age"].mean()), 1)
print("\nMean age on Titanic: ",age_mean)

#problem with nan skewing results - Get ages for prefixs
df['Mr_Age_Numeric'] = pd.to_numeric(df['Mr'], errors='coerce')
df['Mr_Age_Numeric'] = df['Mr_Age_Numeric'].replace(0, np.NaN)
imputed_age_Mr      = round((df['Mr_Age_Numeric'].mean()), 1)
print("imputed_age_Mr: ", (imputed_age_Mr))

df['Mrs_Age_Numeric'] = pd.to_numeric(df['Mrs'], errors='coerce')
df['Mrs_Age_Numeric'] = df['Mrs_Age_Numeric'].replace(0, np.NaN)
imputed_age_Mrs      = round((df['Mrs_Age_Numeric'].mean()), 1)
print("imputed_age_Mrs: ", (imputed_age_Mrs))

df['Miss_Age_Numeric'] = pd.to_numeric(df['Miss'], errors='coerce')
df['Miss_Age_Numeric'] = df['Miss'].replace(0, np.NaN)
imputed_age_Miss      = round((df['Miss_Age_Numeric'].mean()), 1)
print("imputed_age_Miss: ", (imputed_age_Miss))

df['Don_Age_Numeric'] = pd.to_numeric(df['Don'], errors='coerce')
df['Don_Age_Numeric'] = df['Don'].replace(0, np.NaN)
imputed_age_Don      = round((df['Don_Age_Numeric'].mean()),1)
print("imputed_age_Don: ", (imputed_age_Don))

df['Master_Age_Numeric'] = pd.to_numeric(df['Master'], errors='coerce')
```

```
df['Master_Age_Numeric'] = df['Master'].replace(0, np.NaN)
imputed_age_Master      = round((df['Master_Age_Numeric'].mean()), 1)
print("imputed_age_Master: ", (imputed_age_Master))

#output:
# Mean age on Titanic:  29.7
# imputed_age_Mr:  32.4
# imputed_age_Mrs:  35.9
# imputed_age_Miss:  21.8
# imputed_age_Don:  40.0
# imputed_age_Master:  4.8
#=============================================================================
# 5th Loop through rows, if df['Age'] is empty insert age depending on prefix
#=============================================================================
#get column locations for
col_no_age = df.columns.get_loc('Age')
col_no_Name = df.columns.get_loc('Name')
col_no_ID = df.columns.get_loc('PassengerId')

#iterate through the dataframe, row by row
for i in range(len(df)):
    #if the age in the row is empty
    if str(df.iloc[i, col_no_age]) == str(np.nan):
        #assign True to each prefix and assin to variable
        df['Mr'] = df['Name'].str.contains(r'\,\sMR.\s')
        #assign prefix column location to variable
        col_no_mr = df.columns.get_loc('Mr')
        df['Mrs'] = df['Name'].str.contains(r'\,\sMRS.\s')
        col_no_mrs = df.columns.get_loc('Mrs')
        df['Miss'] = df['Name'].str.contains(r'\,\sMISS.\s')
        col_no_miss = df.columns.get_loc('Miss')
        df['Don'] = df['Name'].str.contains(r'\,\sDON.\s')
        col_no_don = df.columns.get_loc('Don')
        df['Master'] = df['Name'].str.contains(r'\,\sMASTER\.\s')
        col_no_master = df.columns.get_loc('Master')
        #if the prefix colum is True and the age was empty
        if df.iloc[i, col_no_mr] == True:
            print("1111111: ",  df.iloc[i, col_no_ID], df.iloc[i, col_no_Name], df.iloc[i, col_no_age])
            df.iloc[i, col_no_age] = imputed_age_Mr
            print("Changed to: ", df.iloc[i, col_no_ID], df.iloc[i, col_no_Name], df.iloc[i, col_no_age])
        elif df.iloc[i, col_no_mrs] == True:
            print("2222222: ",  df.iloc[i, col_no_ID], df.iloc[i, col_no_Name], df.iloc[i, col_no_age])
            df.iloc[i, col_no_age] = imputed_age_Mrs
            print("change to: ", df.iloc[i, col_no_ID], df.iloc[i, col_no_Name], df.iloc[i, col_no_age])
        elif df.iloc[i, col_no_miss] == True:
            print("3333333: ",  df.iloc[i, col_no_ID], df.iloc[i, col_no_Name], df.iloc[i, col_no_age])
            df.iloc[i, col_no_age] = imputed_age_Miss
            print("change to: ", df.iloc[i, col_no_ID], df.iloc[i, col_no_Name], df.iloc[i, col_no_age])
        elif df.iloc[i, col_no_don] == True:
            print("4444444: ",  df.iloc[i, col_no_ID], df.iloc[i, col_no_Name], df.iloc[i, col_no_age])
            df.iloc[i, col_no_age] = imputed_age_Don
            print("change to: ", df.iloc[i, col_no_ID], df.iloc[i, col_no_Name], df.iloc[i, col_no_age])
        elif df.iloc[i, col_no_master] == True:
            print("5555555: ",  df.iloc[i, col_no_ID], df.iloc[i, col_no_Name], df.iloc[i, col_no_age])
            df.iloc[i, col_no_age] = imputed_age_Master
```

```
            print("change to: ", df.iloc[i, col_no_ID], df.iloc[i, col_no_Name], df.iloc[i, col_no_age])
        else:
            #Catch any prefixs that are not above
            print("Other: ",  df.iloc[i, col_no_ID], df.iloc[i, col_no_Name], df.iloc[i, col_no_age])
            df.iloc[i, col_no_age] = age_mean
            print("change to: ", df.iloc[i, col_no_ID], df.iloc[i, col_no_Name], df.iloc[i, col_no_age])
        print()

#save to file so that we can see the changes
filename1 = 'S1_Data_Gathering_Regex_output.csv'
df.to_csv(filename1)
```