Supervised Machine Learning Classification: Home Loan Default Data

By: Denis O'Byrne

Date: 4/18/2022

IBM Supervised Machine Learning: Classification Final Project

Purpose and Associated Jupyter Notebook

This report is part of my completion of the IBM Machine Learning Professional Certificate. All work related to this Certificate Can be found on my GitHub Linked here:

< https://github.com/DenisOByrne/IBM-Machine-Learning-Professional-Certificate >

This project is specifically for the IBM Supervised Machine Learning Classification Course. The folder containing the dataset, this report and the Jupyter notebook used to complete this assignment can be found at the link below:

< https://github.com/DenisOByrne/IBM-Machine-Learning-Professional-Certificate/tree/main/Supervised-Machine-Learning-Classification >

Introduction

This project is a continuation of my project for the course Exploratory Data Analysis for Machine Learning. Continuing from my previous analysis I will be building a classification model to predict whether a customer for a bank will default on their home loan. For the purposes of this assignment, we will be attempting to build a classifier model with a strong specificity to identify the defaulters so that we minimize the risk of lending out money and losing it all. To keep the business profitable, we still need to be able to identify customers who will pay back their loans, so we cannot just choose a model that predicts everyone will default for 100% specificity and 0% sensitivity, so we will be looking to find a model with a strong enough specificity and sensitivity to remain profitable.

This report will assume that the reader has not read my previous assignment, so the data analysis prior to modeling will be exactly the same as the previous report. If you have already read my previous report on this dataset you may skip to the section on Revisions to Data Analysis starting on page 20, where I discuss a discovery regarding the dataset which led me to drop certain variables which were left null for the majority of the customers who paid their loans. Now we begin.

Banks earn a majority of their revenue from lending loans. However, this is often associated with risk. The borrowers may default on the loan. To mitigate this issue, the banks have decided to use Machine Learning to overcome this issue. They have collected past data on the loan borrowers & would like to develop a strong ML Model to classify if any new borrower is likely to default or not.

Data Set Description

For this assignment I will be analyzing the Loan Default Dataset from Kaggle seen here. The data contains 148670 observations with 33 columns of features per observation along with a labeled categorical status (1 = paid, 0 = defaulted). The data specifically seems to cover home loans as many of the features included in the data are only relevant to home loans, so I will be analyzing the data under the assumption that these are home loans although there is no information provided to definitively confirm this.

Note that the dataset does not provide descriptions for the variables so the understanding of the variables and factors is not clear in some cases, but I will still provide a detailed description of the variables based on my understanding of the dataset.

Target:

Status (Categorical)- 1 / 0 - Each datapoint is labeled either 1 for paid or 0 for defaulted

Features:

1. ID (Integer) - Row labels for the data from the source. 24890 — 173559 (this column is not important and will be removed)
2. Year (Integer) - Year the Loan payment was due. All loans in this dataset are from 2019 so this column should be removed as it provides no information
3. Loan_Limit (Categorical) - cf /ncf - meaning conforming or nonconforming. A conforming loan is a loan that cannot exceed a specified amount. Therefore, a loan limit exists for a cf loan and there is no limit on a ncf loan. We do not know what the limit is, just that it exists.
4. Gender (Categorical) - Male/Female/Joint/Sex Not Available - A Joint loan is a loan with multiple cosigners not necessarily of mixed gender.
5. Approv_In_Adv (Categorical) - pre/nopre - A preapproved loan is a loan that was given by a bank prior to making a purchase to let the buyer know how much they can afford. A non-preapproved loan is a loan is for a buyer who made a purchase or found a price for an item prior to speaking with a bank or lender.
6. Loan_Type (Categorical) - type1/type2/type3 - no information on what these types mean is provided
7. Loan_Purpose (Categorical) - p1/p2/p3/p4 - no information on what these purposes are
8. Credit_Worthiness (Categorical) – L1/L2 - no information on what this means. I think it refers to habitual defaulters.
9. Open_Credit (Categorical) - nopc/opc - Do they have an open monthly personal credit (opc) card/line with the bank or not?
10. Business_Or_Comercial (Categorical) – nob/c or b/c - Is this a business or personal expense
11. Loan_Amount (Continuous) - Dollar amount of the original loan. Range (16,500 – 32,576,500)

12. Rate_Of_Interest (Continuous) - Percentage of interest accrued annually. Range (0.00 - 8.00)
13. Interest_Rate_Spread (Continuous) - The net interest rate spread is the difference between the interest rate a bank pays to depositors and the interest rate it receives from loans to consumers. Range (-3.638 - +3.357). A negative spread implies the loan holder could deposit the loan in the bank and earn money on the interest faster than the loan grows with interest.
14. Upfront_Charges (Continuous) - Down payments and fees associated with the loan. Range (0.00 - 60,000.00)
15. Term (Integer)- Days Until the Loan is to be paid off in full. Range (96 - 365). Note more than 75% of the loans in this dataset have a loan term of 365. It Is more helpful to consider this as a categorical variable as there are only a few possible values for the loan terms.
16. Neg-Amortization (Categorical) - not_neg / neg_amm - An amortized loan is paid so that you pay more than the accrued interest at the end of each term. Amortized payments are higher but eventually the loan is paid off in full. Negative-amortized loans pay less than the accrued interest on the loan so that the amount owed on the loan is more at the end of each term. A loan will never be paid off if it remains in negative amortized status.
17. Interest_only (Categorical) - not_int / int_only - An interest-only loan is a loan in which the borrower pays only the interest for some or all of the term, with the principal balance unchanged during the interest-only period
18. Lump_Sum_Payment (Categorical) - not_lpsm / lpsm - A lump-sum payment is an often-large sum that is paid in one single payment instead of broken up into installments.
19. Property_Value (Continuous)- Value of the building being purchased without other fees. Range (8,000 - 16,508,000)
20. Construction_Type (Categorical) - sb / mh - I don't know what these mean
21. Occupancy_Type (Categorical) - pr / sr/ ir - I don't know what these mean
22. Secured_By (Categorical) - home / land
23. Total_Units (Categorical) - 1U / 2U / 3U / 4U – number of units refers to how many separate families can live in the home
24. Income (Continuous) - How much does the property earn in the term of the loan. If this is a personal home then the value should be 0. If the property is being rented then it should be nonzero. Range (0 – 578,580)
25. Credit_Type (Categorical) - EXP / EQUI / CRIF / CIB – Credit rating type
26. Credit_Score (Integer) - Credit Score of Applicant at the time of the loan. Range (500-900)
27. Co-Applicant-Credit-Type (Categorical) - CIB / EXP
28. Age (Categorical) - < 25 / 25-34 / 35-44 / 45-54 / 55-64 / 65-74 / > 74 – Age range of loan applicant
29. Submission_Of_Application (Categorical) - to_inst / not_inst – to inst means they submitted
30. LTV (Continuous) - The loan-to-value ratio is a financial term used by lenders to express the ratio of a loan to the value of an asset purchased. In Real estate, the term is commonly

used by banks and building societies to represent the ratio of the first mortgage line as a percentage of the total appraised value of real property. A higher LTV is better for the buyer since they are getting a better deal. Range (0.967478 - 7831.25)

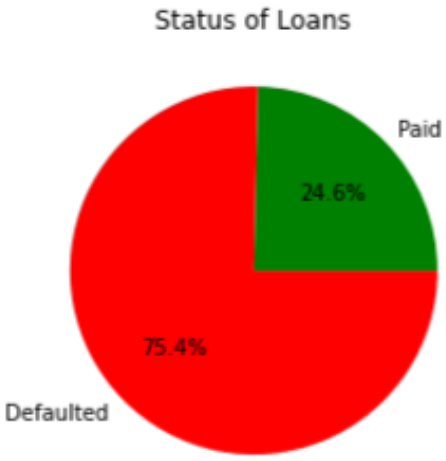
31. Region (Categorical) - South / North / Central / North-East – Region of the US where the Applicant lives
32. Security_Type (Categorical) - direct / indirect – not sure what this means
33. Dtir1 (Continuous) - Debt to income ratio percentage. Your debt-to-income ratio is all your monthly debt payments divided by your gross monthly income. Range (5.00 - 61.00)

Exploratory Data Analysis and Data Cleaning

For a better understanding of the data set we will begin looking at the data visually. To start we can see the summary statistics for the numeric features omitting the year and ID columns

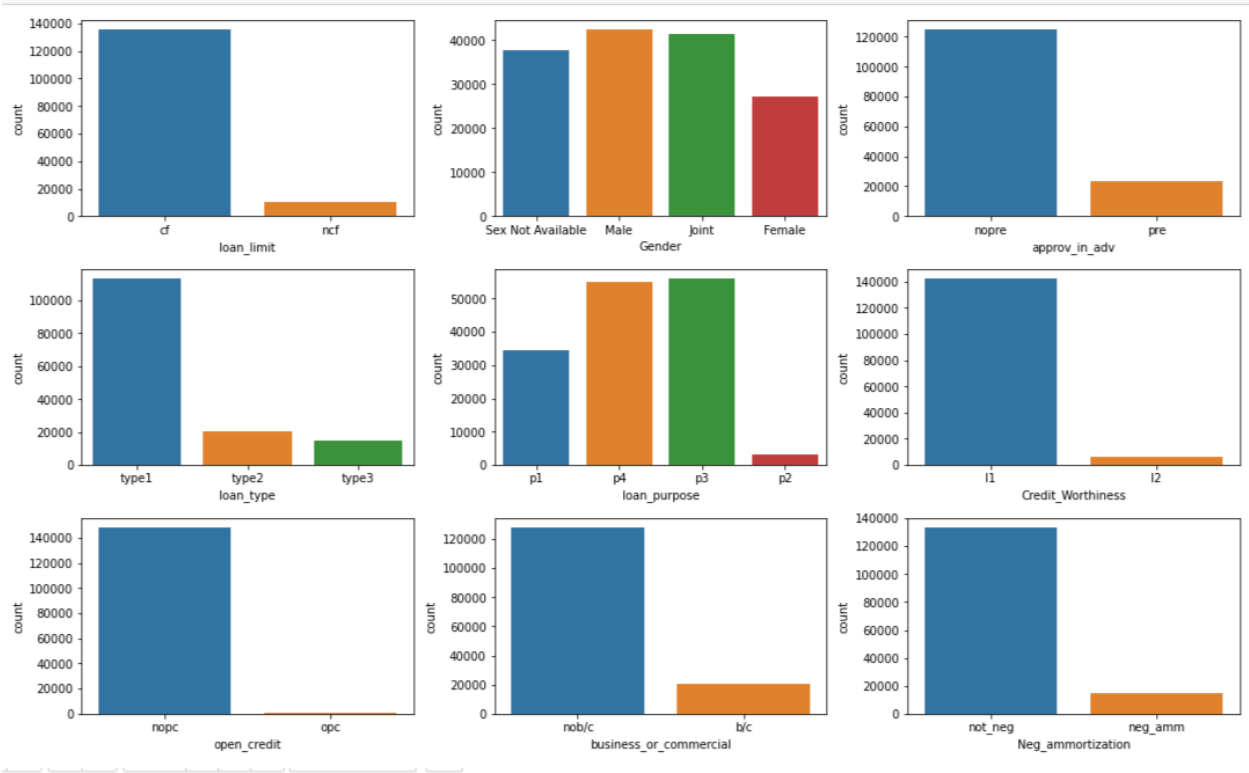
	loan_amount	rate_of_interest	Interest_rate_spread	Upfront_charges	term	property_value	income	Credit_Score	LTV	Status	dtir1
count	1.486700e+05	112231.000000	112031.000000	109028.000000	148629.000000	1.335720e+05	139520.000000	148670.000000	133572.000000	148670.000000	124549.000000
mean	3.311177e+05	4.045476	0.441656	3224.996127	335.136582	4.978935e+05	6957.338876	699.789103	72.746457	0.246445	37.732932
std	1.839093e+05	0.561391	0.513043	3251.121510	58.409084	3.599353e+05	6496.586382	115.875857	39.967603	0.430942	10.545435
min	1.650000e+04	0.000000	-3.638000	0.000000	96.000000	8.000000e+03	0.000000	500.000000	0.967478	0.000000	5.000000
25%	1.965000e+05	3.625000	0.076000	581.490000	360.000000	2.680000e+05	3720.000000	599.000000	60.474860	0.000000	31.000000
50%	2.965000e+05	3.990000	0.390400	2596.450000	360.000000	4.180000e+05	5760.000000	699.000000	75.135870	0.000000	39.000000
75%	4.365000e+05	4.375000	0.775400	4812.500000	360.000000	6.280000e+05	8520.000000	800.000000	86.184211	0.000000	45.000000
max	3.576500e+06	8.000000	3.357000	60000.000000	360.000000	1.650800e+07	578580.000000	900.000000	7831.250000	1.000000	61.000000

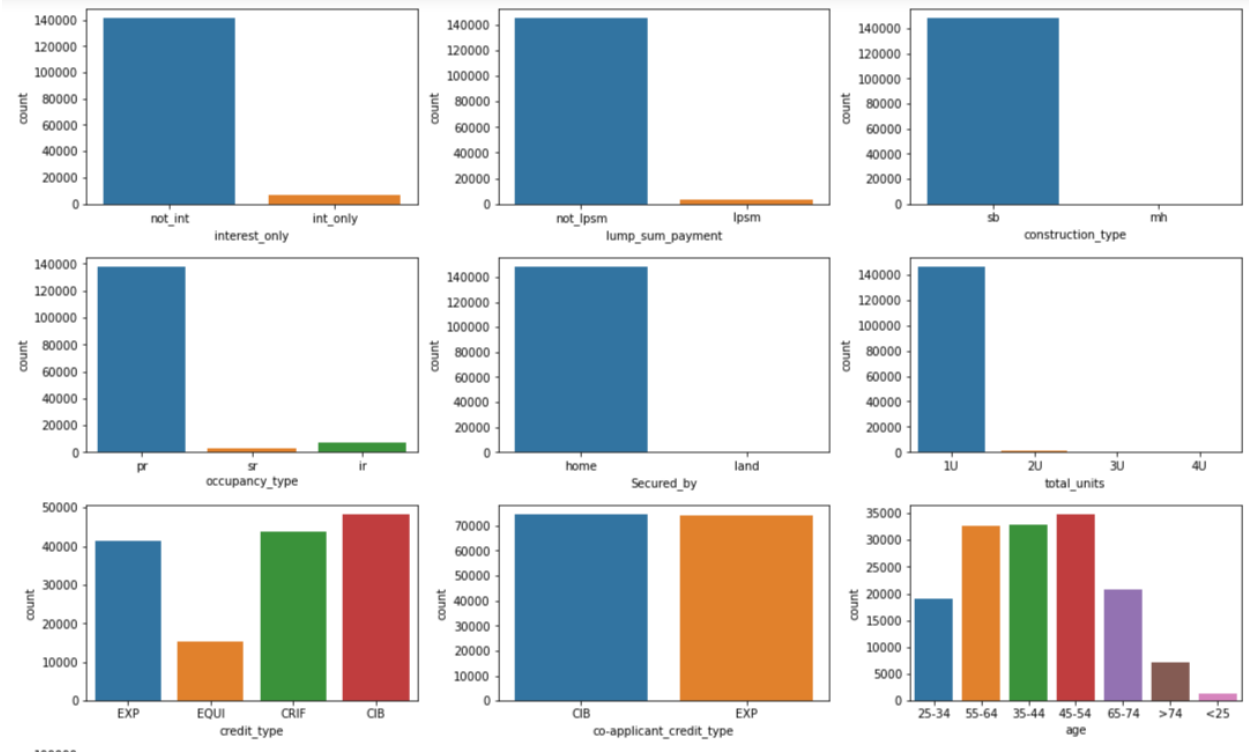
Most of the information here is no more informative than what I had provided in the original data description for the ranges however we can see that the status label for the majority of loans in this data set is defaulted, with only 24.6% of loans being paid. We can see this better in a pie chart.

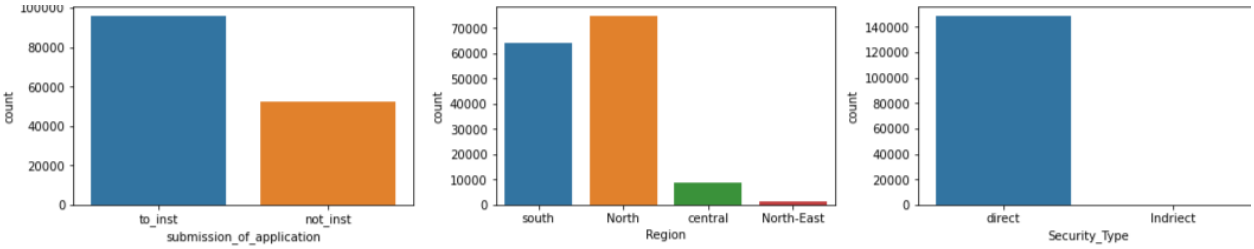


Although in most cases this imbalance of classes would seem bad for building a predictive model, we need to recognize that in the case of loan defaults we would like to only lend out to customers who we know will pay back the loan, so by oversampling the default class, we can help any model we build in the future spot more indicators of a defaulter. Any model built on this data should be optimized to improve sensitivity to default instead of specificity of paid loans or total accuracy, so that the bank does not make risky loans.

We can view bar charts to see the distributions of the other categorical variables in the data set as well.


count
loan_limit
cf
ncf
Gender
Sex Not Available
Male
Joint
Female
approv_in_adv
nopre
pre
loan_type
type1
type2
type3
loan_purpose
p1
p4
p3
p2
Credit_Worthiness
l1
l2
open_credit
nopc
opc
business_or_commercial
nob/c
b/c
Neg_ammortization
not_neg
neg_amm


count
interest_only
not_int
int_only
lump_sum_payment
not_lpsm
lpsm
construction_type
sb
mh
occupancy_type
pr
sr
ir
Secured_by
home
land
total_units
1U
2U
3U
4U
credit_type
EXP
EQUI
CRIF
CIB
co-applicant_credit_type
CIB
EXP
age
25-34
55-64
35-44
45-54
65-74
>74
<25



Looking at the splits in the data here we can see that we have extreme class imbalances in many of the features meaning they will almost surely be unhelpful in developing a predictive model as the information gain from using these features such as Security_Type, Construction_Type, or Secured_By to develop a model will be minute and any information gleaned from these features will not extrapolate to new data as our sample size is too small. For now, I will leave them in, but we can get even more information on these 3 features specifically by looking at the exact counts in the groups for these data types.

```
data['construction_type'].value_counts()
```

```
sb    148637
mh        33
Name: construction_type, dtype: int64
```

```
33/(148637+33)
```

```
0.00022196811730678686
```

```
data['Secured_by'].value_counts()
```

```
home    148637
land        33
Name: Secured_by, dtype: int64
```

```
data['Security_Type'].value_counts()
```

```
direct      148637
Indriect        33
Name: Security_Type, dtype: int64
```

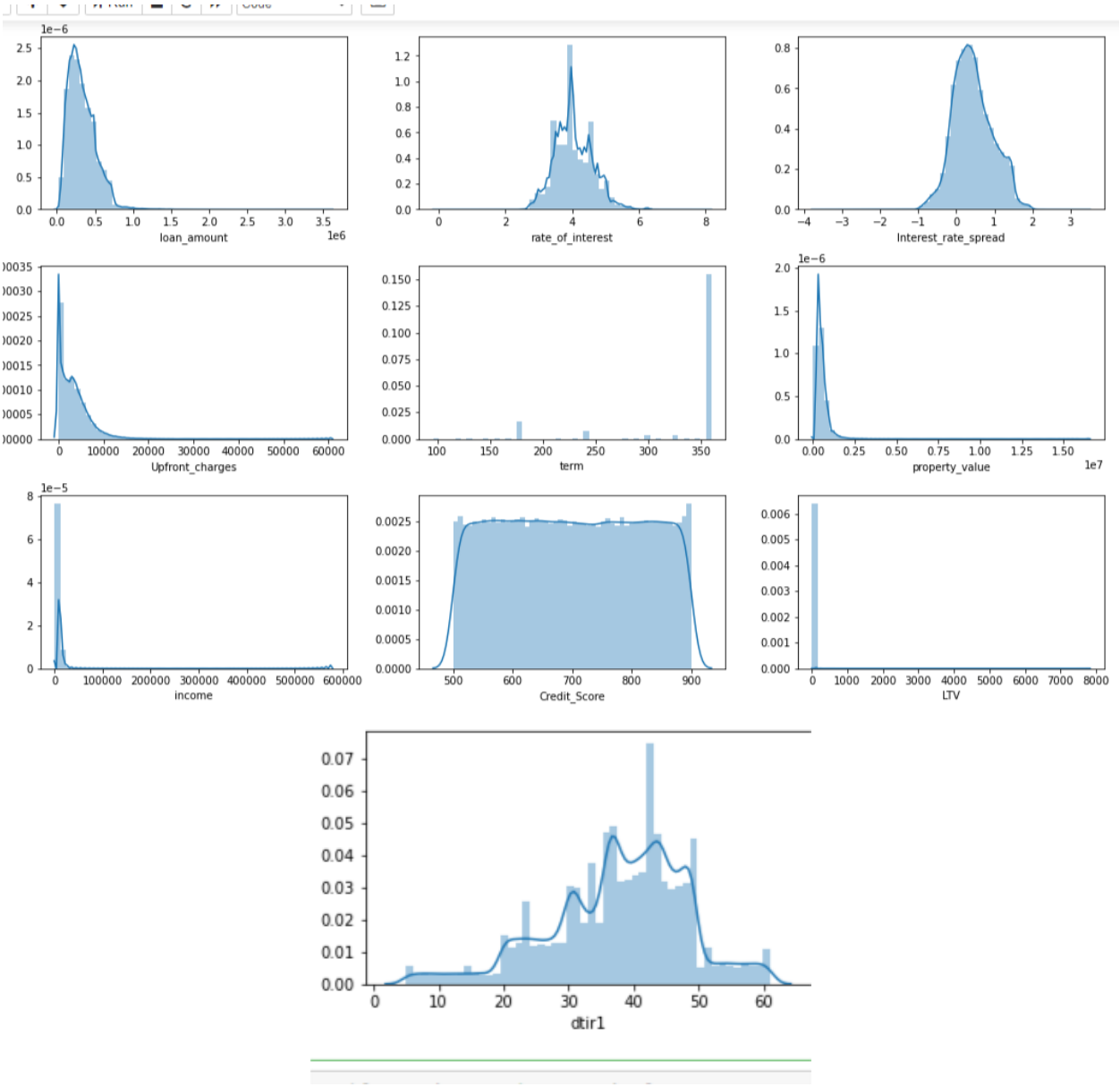
As we can see above, the three variables seem to be indicators of the same observations in the data. In other words, if a loan is construction type mh, then it will be secured by land and security type indirect and if it is construction type sb, then it will be secured by home and security type will be direct.

```
tmp = data[['Security_Type','Secured_by', 'construction_type']]
tmp[tmp['construction_type'].isin(['mh'])]
```

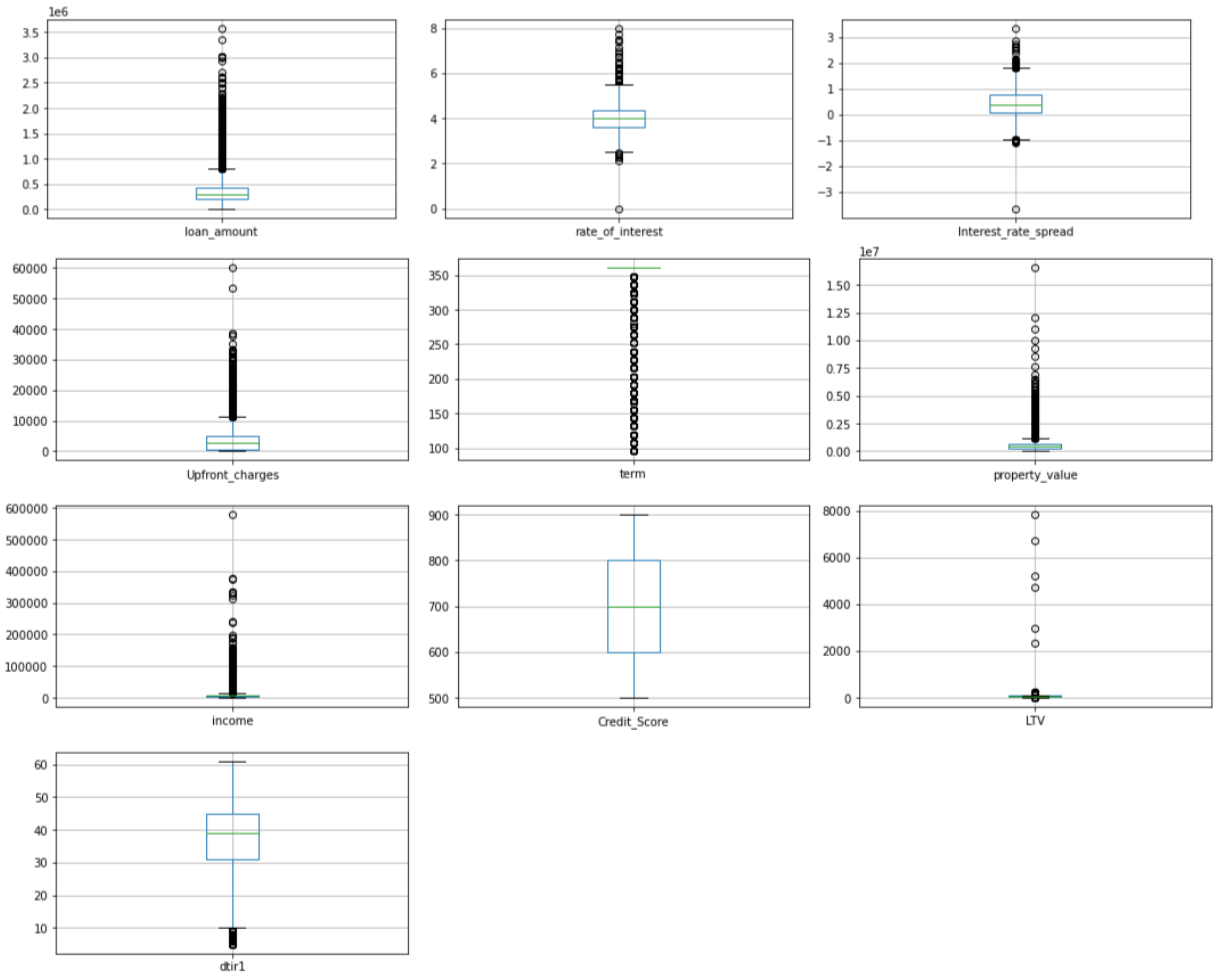
	Security_Type	Secured_by	construction_type
7973	Indriect	land	mh
32312	Indriect	land	mh
34412	Indriect	land	mh
35368	Indriect	land	mh
36155	Indriect	land	mh
41151	Indriect	land	mh
44592	Indriect	land	mh
46022	Indriect	land	mh
47828	Indriect	land	mh
56153	Indriect	land	mh
59732	Indriect	land	mh
60122	Indriect	land	mh
62581	Indriect	land	mh
68927	Indriect	land	mh
82520	Indriect	land	mh
85185	Indriect	land	mh
90473	Indriect	land	mh
91255	Indriect	land	mh
104761	Indriect	land	mh
105639	Indriect	land	mh
106363	Indriect	land	mh
109160	Indriect	land	mh
109448	Indriect	land	mh

Checking this we see this assumption is true and so we can at least drop two of these columns. I will leave the secured_by column as it is the easiest column to understand as a variable.

Lastly, we can investigate the numeric features using plots of the distribution functions of the data



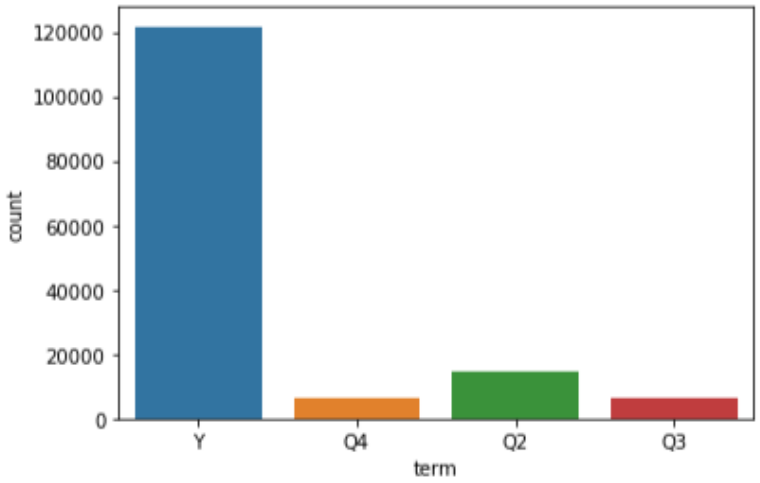
Here we can see the sample probability mass functions with the overlaid maximum likelihood density of each of the numeric variables. Here we can see that none of these variables follow a normal distribution except possibly the rate of interest and the interest rate spread. Credit score appears to follow a uniform distribution between 500 and 900. Dtir1 appears to follow a right skewed heavy tailed distribution. The Term looks to be discrete however inspecting further there do appear to be continuous possible values, just a majority fall on certain dates. The rest of the variables follow some form of exponential distributions. We can observe box plots to see if any outliers exist.



From these plots we can see that each variable besides Credit score is plagued with outliers and it is more obvious now that most of these variables are skewed to the right or left meaning they would be well suited for a log transform.

Data Cleaning

Also, I would like to restructure the term variable into a categorical variable, splitting the data into 5 groups, separating the year into quarters such that terms between 0-90 days are in group Q1, 91-180 days are in group Q2, 181-240 days are in group Q3, 241-359 days are in group Q4, and 360 days exclusively are in group Y for Year. Below is the histogram of the term data after this transformation.



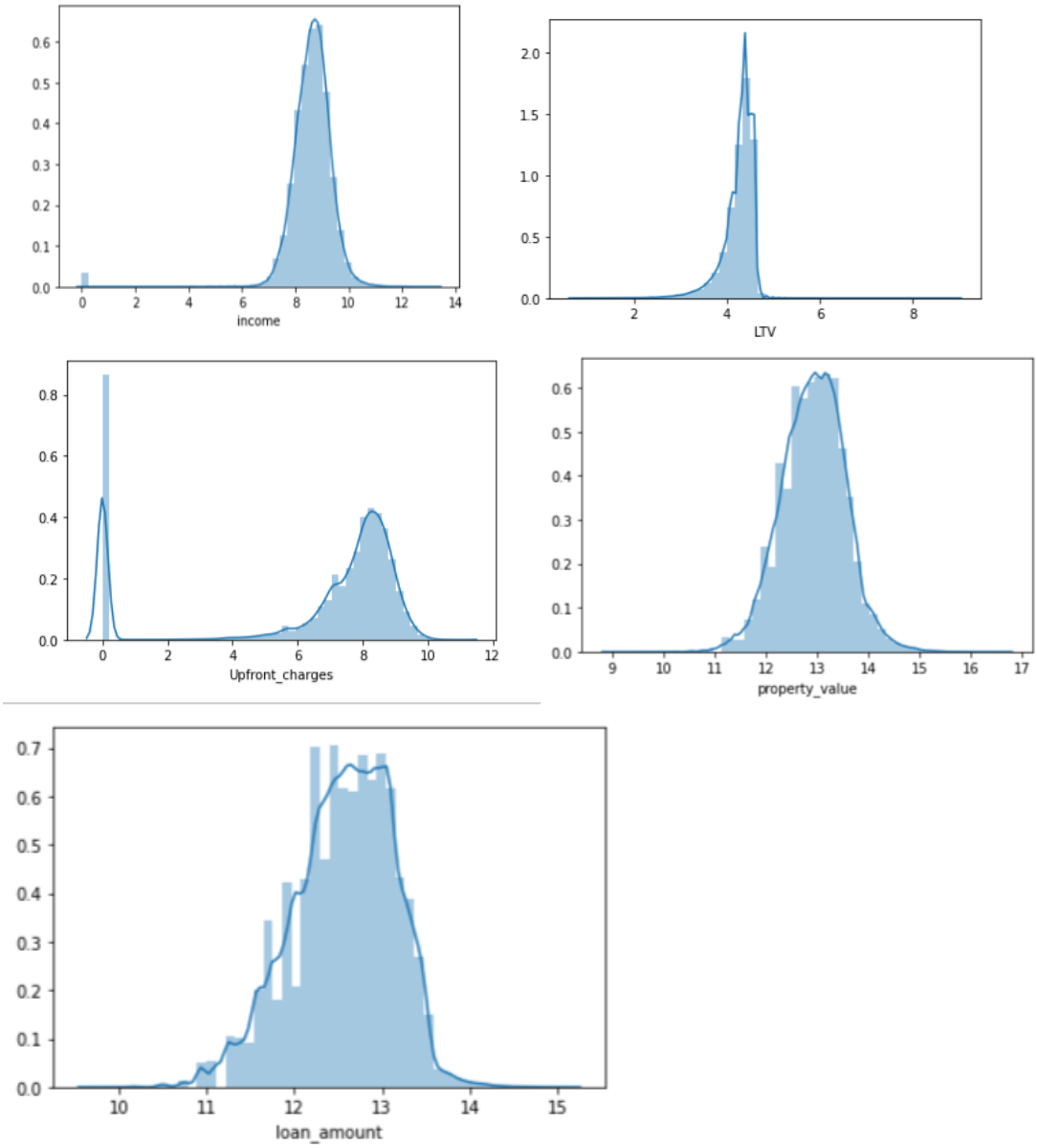
To see which variables need the log transform, we can check the skewness of the numeric variables to find the following skewness values:

```
: print(skew_vals)

loan_amount              1.666998
rate_of_interest         0.388406
Interest_rate_spread     0.280762
Upfront_charges          1.754076
property_value           4.586276
income                  17.307695
LTV                    120.615337
dtir1                   -0.551465
Credit_Score             0.004767
dtype: float64
```

Following the recommendations of this course I only applied the log transformation to the features with a skewness above 0.75, which applies to loan_amount, Upfront_charges, property_value, income, and LTV.

Following the transformation, we find that the data looks much more normally distributed with the possible exception of zero inflation.



Imputing Missing Values

Now that the data is properly transformed, we can impute missing values. First, we can look at the number of missing values for each variable presented below:

```
data.isnull().sum()
```

```
loan_limit                3344
Gender                       0
approv_in_adv              908
loan_type                    0
loan_purpose               134
Credit_Worthiness            0
open_credit                  0
business_or_commercial       0
loan_amount                  0
rate_of_interest         36439
Interest_rate_spread     36639
Upfront_charges          39642
term                        41
Neg_ammortization          121
interest_only                0
lump_sum_payment             0
property_value           15098
occupancy_type               0
Secured_by                   0
total_units                  0
income                    9150
credit_type                  0
Credit_Score                 0
co-applicant_credit_type     0
age                        200
submission_of_application  200
LTV                      15098
Region                       0
Status                       0
dtir1                    24121
dtype: int64
```

We see that for the categorical variables we have at most 3344 missing variables which accounts for 2% of total observations but most are much less than this, so for convenience I will impute the most frequent value. We also see that the target value, Status has no missing values. Lastly, we see that the numeric variables have at minimum 9150 missing values or 6.15% of the observations for the income feature all the way up to 39642 missing values or 26.67% of the observations for the Upfront_charges feature. For these numeric features we should be more cautious of what value we impute.

For the variables income and Upfront_charges, I will impute the value 0 for missing data, as it is the most frequent value for the Upfront_charges and I am making the assumption that anyone who did not fill out the income for a property left it blank for the purpose that the property will not be used for income. Meanwhile for the rate_of_interest, Interest_rate_spread, LTV, dtir1, and property_value features I will impute the mean of the data as we have transformed the data to be somewhat normalized for these variables, making the mean an acceptable average observation for the sample date on these features.

We can achieve these imputation specifications using the following code:

```
: from sklearn.impute import SimpleImputer

nums = ['rate_of_interest', 'Interest_rate_spread', 'LTV', 'dtir1','property_value']
data['income'] = data['income'].fillna(0)
data['Upfront_charges'] = data['Upfront_charges'].fillna(0)

for col in nums:
    SI = SimpleImputer(strategy='mean')
    data[col] = SI.fit_transform(data[[col]])

mask = data.dtypes == np.object
obj_cols = data.columns[mask]

for col in obj_cols:
    SI = SimpleImputer(strategy='most_frequent')
    data[col] = SI.fit_transform(data[[col]])
```

Standard Scaling and One Hot Encoding:

For one hot encoding we convert all factors for each feature to their own dummy variable with values of 1 or 0 to indicate if the observation belongs to the group. We drop 1 factor from each feature as it will be redundant since we can equally say an observation is not in all other groups for that feature. We do this with the following lines of code:

```
for i in obj_cols:
    #print(i)
    if data[i].nunique()==2:
        data[i]=pd.get_dummies(data[i], drop_first=True, prefix=str(i))
    if data[i].nunique()>2:
        data = pd.concat([data.drop([i], axis=1), pd.DataFrame(pd.get_dummies(data[i], drop_first=True, prefix=str(i)))],axis=1)
```

Lastly for feature engineering we apply the Standard Scalar to the data to standardize the scales on the data for future modeling. We do this only to features which are not binary. This can be accomplished with the following code:

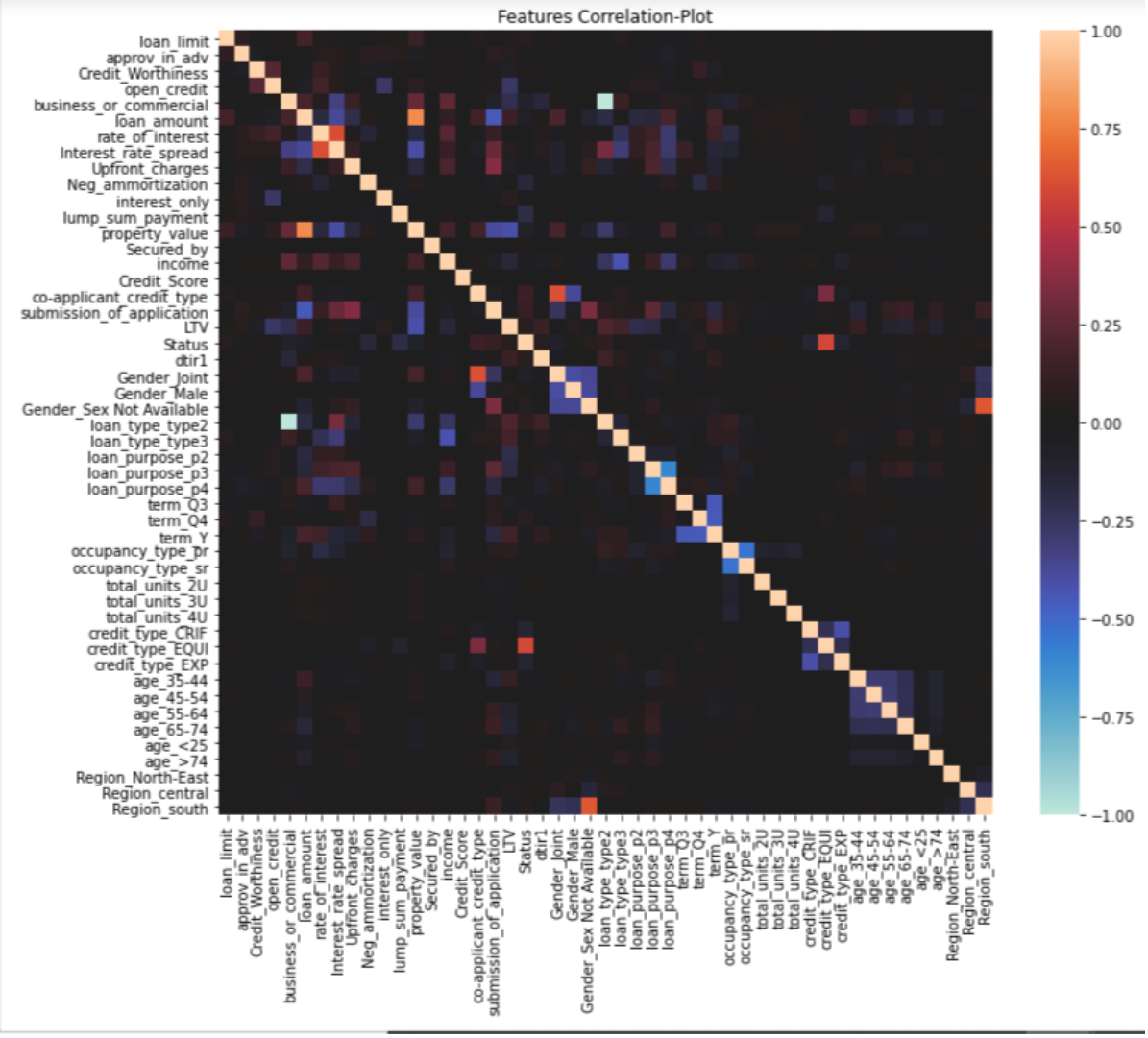
```
#Feature Scaling (Standardization)
nums = []
for col in data.columns:
    if data[col].nunique()>2:
        nums.append(col)

std = StandardScaler()

X_std = std.fit_transform(data[nums])
X_std = pd.DataFrame(X_std, columns=nums)

for col in nums:
    data[col] = X_std[col]
```

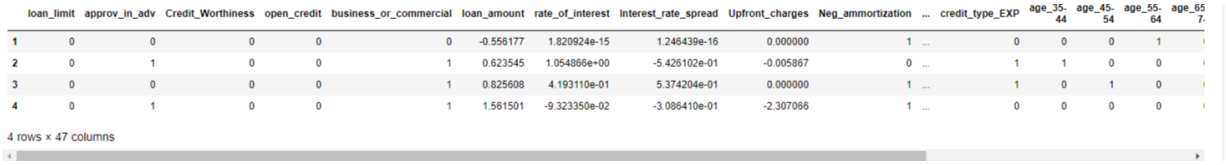
We can now look at the correlation heatmap of the data below.



Here we can see that there is some multi-correlation in our features and we also find that the feature buisness_or_commercial is the exact opposite indicator as a loan_type of type 2 meaning we can drop one of these two variables to avoid redundancy. We also see that rate_of_interest and interest_rate_spread are highly corelated with an exact correlation of 0.6143209863866121. This makes sense that they would be correlated as the bank is more likely to have a strong or weak spread if they give out a strong or weak interest rate, but it is possible these two variables can have different effects on loan repayment so we should keep them in. Lastly the loan amount and the property value are highly correlated as the loan is meant to pay for the property, with an exact correlation of 0.7919049032238942. With this level of correlation, it is probably best practice to drop one of the variables. I will drop the property value for this as

we are trying to detect what makes people not pay back their loans after all, so the loan amount is more important for understanding the data in this case.

At this point the data cleaning is complete, so let's take a look at the head of the dataset



	loan_limit	approv_in_adv	Credit_Worthiness	open_credit	business_or_commercial	loan_amount	rate_of_interest	Interest_rate_spread	Upfront_charges	Neg_ammortization	...	credit_type_EXP	age_35-44	age_45-54	age_55-64	age_65-7
1	0	0	0	0	0	-0.556177	1.820924e-15	1.246439e-16	0.000000	1	...	0	0	0	1	
2	0	1	0	0	1	0.623545	1.054866e+00	-5.426102e-01	-0.005867	0	...	1	1	0	0	
3	0	0	0	0	1	0.825608	4.193110e-01	5.374204e-01	0.000000	1	...	1	0	1	0	
4	0	1	0	0	1	1.561501	-9.323350e-02	-3.086410e-01	-2.307066	1	...	0	0	0	0	

4 rows × 47 columns

We can see that we have a total of 46 features left and we never removed any observations so we still have 148670 rows.

Testing Hypotheses About the Data

Hypothesis #1

We have been making some assumptions on the distributions of the features in the data set, so we should test these assumptions to ensure that the transformations made earlier actually normalized the data.

H_o: The loan_amount variable follows a Standard normal distribution after standard scaling and log transforming the data

H_a: The loan_amount variable does not follow a Standard normal distribution after standard scaling and log transforming the data

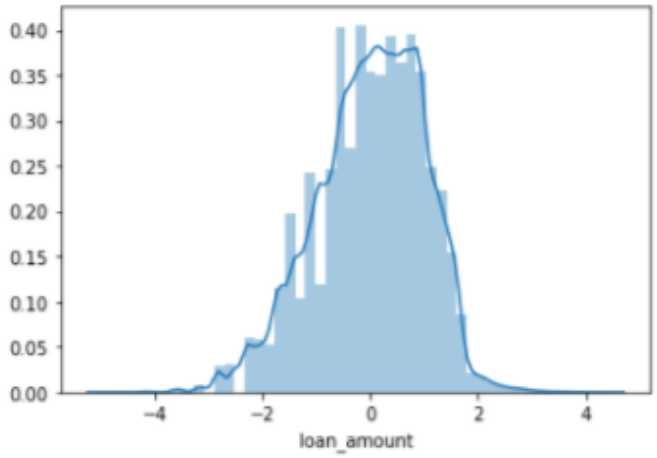
We can test this assumption using a Shapiro-Wilks test for normality or the D'Agostino's K^2 test. We will use an alpha = 0.05 level of significance for both tests.

```
from scipy.stats import shapiro
# normality test
stat, p = shapiro(data['loan_amount'])
print('Statistics=%.3f, p=%.3f' % (stat, p))
# interpret
alpha = 0.05
if p > alpha:
    print('Sample looks Gaussian (fail to reject H0)')
else:
    print('Sample does not look Gaussian (reject H0)')
```

```
Statistics=0.987, p=0.000
Sample does not look Gaussian (reject H0)
```

```
sns.distplot(data['loan_amount'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a427a2e670>
```




```
from scipy.stats import normaltest
stat, p = normaltest(data['loan_amount'])
print('Statistics=%.3f, p=%.3f' % (stat, p))
# interpret
alpha = 0.05
if p > alpha:
    print('Sample looks Gaussian (fail to reject H0)')
else:
    print('Sample does not look Gaussian (reject H0)')
```

```
Statistics=3629.487, p=0.000
Sample does not look Gaussian (reject H0)
```

We can see that the test statistic is too large in both tests leading to a p-value below 0.001. Since our p-value is below 0.05, we reject that the data for loan amounts is normally distributed at the alpha = 0.05 level of significance.

Hypothesis #2

We can test with the data if the gender of an applicant has an effect on the rate of interest for the loan. We may want to know if men or women are better at bargaining for terms in their loan agreements, or is it possible that couples/joint buyers receive better rates. For the specific test we can use a one-way Anova F test to see if the mean interest rate is equal for all groups. We can structure the test as follows:

H_o:

$\mu_{Male}=\mu_{Female}=\mu_{joint}=\mu_{Unspecified}$

H_a: At least one of the mean values is different from the others.

We again will run the test at the alpha = 0.05 level of significance

```
: import scipy.stats as stats

  stats.f_oneway(data['rate_of_interest'][data['Gender_Male'] == 1],
                 data['rate_of_interest'][data['Gender_Joint'] == 1],
                 data['rate_of_interest'][data['Gender_Sex Not Available'] == 1],
                data['rate_of_interest'][(data['Gender_Male'] == 0) & (data['Gender_Joint'] == 0) & (data['Gender_Sex Not Available'] == 0)])

: F_onewayResult(statistic=173.5264048391031, pvalue=2.6024115620818563e-112)
```

Here we see that that gender has a significant effect on the mean rate of interest for home loans with a p-value of $2*10^{-112}$ thus we reject the null hypothesis at any rational alpha level including our choice of 0.05

I decided to run the same test for just males and females as well below:

```
1]: stats.f_oneway(data['rate_of_interest'][data['Gender_Male'] == 1],
                   data['rate_of_interest'][(data['Gender_Male'] == 0) & (data['Gender_Joint'] == 0) & (data['Gender_Sex Not Available'] == 0)])

1]: F_onewayResult(statistic=224.67150557693387, pvalue=1.039243440002079e-50)
```

We see again that we reject the hypothesis that men and women receive equal loan interest rates with a p value of $1* 10^{-50}$ thus again we reject the null hypothesis at any rational alpha level including our choice of 0.05

We can see the sample mean and standard deviation for the standardized rate of interest on loans for males and females below. Recall that the data has been standardized prior to these calculations so the data is measuring the average Z score, not the average value.

Loan Interest Rate Z Score Statistics by Gender			
Gender	Sample Size	Sample Mean Z score	Sample Z score Standard Deviation
Male	42346	-0.010968703013856218	0.9917286229656858
Female	27266	0.1026491404400715	0.9516309894572063

We see that men receive loan interest rates relatively close to the total sample average while women usually get interest rates about 0.1 standard deviation higher than the average loan.

If we run the same test on the original data we get the same p-value as the standard scaling has a 1 to 1 correspondence as shown below, but we can look at the exact loan rates.

```
SI = SimpleImputer(strategy='mean')
df['rate_of_interest'] = SI.fit_transform(df[['rate_of_interest']])
```

```
df['rate_of_interest'][df['Gender'] == "Male"]
```

```
1         4.045476
2         4.560000
3         4.250000
10        4.045476
15        4.045476
            ...
148631    4.875000
148652    4.045476
148663    4.045476
148666    5.190000
148667    3.125000
Name: rate_of_interest, Length: 42346, dtype: float64
```

```
stats.f_oneway(df['rate_of_interest'][df['Gender'] == "Male"],
               df['rate_of_interest'][df['Gender'] == "Female"])
```

```
F_onewayResult(statistic=224.67150557693404, pvalue=1.039243440002079e-50)
```

We find the table for loan rates becomes as follows:

Loan Interest Rate Z Score Statistics by Gender			
Gender	Sample Size	Sample Mean Interest Rate	Sample Standard Deviation
Male	42346	4.040125682383571	0.48372803062761205
Female	27266	4.095544205542036	0.464169908737542

We see that the difference in interest rates between the sexes is very miniscule but the evidence from the One-way Anova test is overwhelmingly significant due to the sample sizes that there is a difference between the interest rates for men and women with women receiving slightly higher interest rates.

Hypothesis #3

Lastly, we know that a high credit rating indicates a person who pays their debts and bills on time regularly, so the credit score should be correlated with the status of loan so we should check using a Pearson correlation test we test this under the original assumption that there is no correlation.

H_o: The $r^2 = 0$ between Credit Score and Status

H_a: r^2 != 0 and there is a correlation between Credit Score and Status

We test this hypothesis with a Pearson Correlation Test at alpha = 0.05 level significance

```
from scipy import stats
stats.pearsonr(data['Credit_Score'],data['Status'])
```

```
(0.0040036935955881 73, 0.1226544025327888)
```

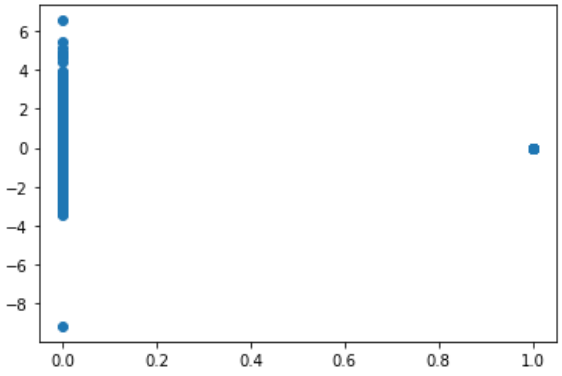
We see from the above test that there is a very miniscule correlation in this sample of 0.004 between the Credit Score and the default Status of the loan. From our intuition this seems nonsensical as the whole point of a credit score is to show if a person will pay their bills or loans back so that the banks know who they should lend to, but we need to remember that the purpose of this data set is to find reasons why people who passed loan acceptance decided not to pay. Therefore, our data includes tons of people with great credit scores who still defaulted which the bank did not expect and the bank wants to know what caused this. Thus, credit scores alone in this dataset should not be a good indicator of default since anyone we loaned money to is someone who we thought would pay their bills to begin with. That being said, we see that the p-value for this test is 0.1227 which is greater than 0.05 and thus we fail to reject the null hypothesis and conclude that there is no statistically significant correlation between Credit Score and Default Status.

Revisions to Data Analysis:

Before we begin to fit the any model to the data, I discovered an error in my analysis of the data. When examining the missing data, I had not considered if the status of a loan had an effect on the probability that certain feature variable would be left missing. When I began to fit a Random Forest model to the data, I found that my model had achieved perfect accuracy on the testing and training data which seemed entirely improbable, so I decided to look at the most important features in the model, and found that the feature Interest_Rate_Spread had an importance 0.24 which seemed rather excessive. Upon plotting the Loan status vs the Interest_Rate_Spread I was presented with the following plot:

```
plt.scatter(data['Status'], data['Interest_rate_spread'])
```

```
<matplotlib.collections.PathCollection at 0x1c09126dc40>
```



We find in the plot that every paid loan (Status = 1) is reported to have an interest_rate_spread of 0, the value imputed for missing data in this column. I decided at this point to check the count of missing data in the dataset with respect to the Status of the loans.

Recall we had found the following counts of missing data for each variable:

```
data.isnull().sum()
```

```
loan_limit                3344
Gender                       0
approv_in_adv              908
loan_type                    0
loan_purpose               134
Credit_Worthiness            0
open_credit                  0
business_or_commercial       0
loan_amount                  0
rate_of_interest         36439
Interest_rate_spread     36639
Upfront_charges          39642
term                        41
Neg_ammortization          121
interest_only                0
lump_sum_payment             0
property_value           15098
occupancy_type               0
Secured_by                   0
total_units                  0
income                    9150
credit_type                  0
Credit_Score                 0
co-applicant_credit_type     0
age                        200
submission_of_application  200
LTV                      15098
Region                       0
Status                       0
dtir1                    24121
dtype: int64
```

Note that we have 36639 observations where the Loan is Paid and we have 36639 observations where the Interest_Rate_Spread is missing. This is not coincidental.

Now we can view the count of missing data only for observations where the Status = 1 (paid)

```
: data_v2[data_v2['Status']==1].isnull().sum()

: ID                          0
  year                        0
  loan_limit                881
  Gender                      0
  approv_in_adv             241
  loan_type                   0
  loan_purpose               35
  Credit_Worthiness           0
  open_credit                 0
  business_or_commercial      0
  loan_amount                 0
  rate_of_interest        36439
  Interest_rate_spread    36639
  Upfront_charges         36486
  term                       15
  Neg_ammortization          32
  interest_only               0
  lump_sum_payment            0
  property_value          15096
  construction_type           0
  occupancy_type              0
  Secured_by                  0
  total_units                 0
  income                   1239
  credit_type                 0
  Credit_Score                0
  co-applicant_credit_type    0
  age                       200
  submission_of_application 200
  LTV                     15096
  Region                      0
  Security_Type               0
  Status                      0
  dtir1                   16310
  dtype: int64
```

We find that the majority of columns where missing data is present are only missing for observations where the loan Status is Paid.

Noting that there are only 36639 total observations in the dataset with status= 1, we should remove the following features which have missing data for more than half of these observations:

rate_of_interest, Interest_rate_spread, Upfront_charges, property_value, LTV, and dtir1

As these variables are missing for more than 50% of all paid loans and paid loans account for over 90% of all missing values in these columns, thus these variables present systematic bias if used in any classification model.

Note that I already removed property_value earlier in the report since it is highly correlated with the loan_amount which had no missing data.

With this out of the way we can now begin using the data to build a classifier model.

(Note that if you are following along with my Jupyter Notebook for this project I had not noticed this issue until around line 192. I had fit a Logistic Regression Model and Random Forest model to the data at this point so these models were refit after removing the problematic columns from the Testing and Training sets)

Splitting the Data into Training and Testing Sets

To make sure that any model we build will retain its strength for prediction on new data, I chose to validate my models with a test train split, using 70% of the sample for training and 30% of the sample for testing with Sklearn's train_test_split function as seen below:

```
y = data['Status']
x_cols = []
for i in data.columns:
    if i != 'Status':
        x_cols.append(i)
X = data[x_cols]
```

```
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=80)
```

```
X_train.shape
```

```
(104069, 46)
```

```
X_test.shape
```

```
(44601, 46)
```

In the end we have 104069 training observations and 44601 testing observations. The number of features in this image is prior to removing the faulty variables, we actually have a total of 41 feature variables, not 46.

Predictive Modeling

To make the best predictions we will be comparing multiple classifiers using cross validation methods to find the best hyperparameters for each model. In the end, the criterion for the best model will come down to a few factors:

1. We want to find the model with the highest specificity for identifying Defaults (Status = 0) so that we minimize the risk in loans that we approve.
2. We want to find a high enough sensitivity to still allow the bank to make a profit by loaning to good customers
3. We want to minimize computation time

For the most part we care about model strength over explain-ability of the model as there is very high risk involved if customers default on their home loans, however we would like to know the top 20 or so factors which may cause a person to default on their loans, so that we can advise our employees of which customers they should be skeptical of.

Random Forest Model

A random forest model is an ensemble model which uses multiple decision tree models to create a consensus prediction on the data. Each individual decision tree in the forest is trained on bootstrap aggregated (bagged) samples with replacement of the original dataset with each tree using a random sample of the total m features, usually of size $\sqrt{m}$, to split the data at each node, of which the model picks the feature with the highest Gini impurity at that node. With a large number of trees Random Forest models usually perform well on new data since each tree in the forest is trained on a random subset of the data with replacement. Adding trees to the model usually has no value for the total accuracy after the forest reaches a certain size at which point the model is minimally efficient.

Decision trees on their own are very prone to overfitting to the data they are trained on, so it is important to limit the tree depth so that the model is not overfit to the training data. It should also be considered that with a large dataset, increasing the number of trees in the model can drastically increase computation time and memory requirements. Increasing tree depth however is much more likely to increase the computation time for fitting and predicting as each tree potentially doubles in size as the depth increases by 1, which potentially doubles the number of splitting nodes in the entire forest, while adding 1 more tree to a forest does not double the size of the forest (unless there is only 1 other tree). In the case of our dataset, Random Forest actually runs somewhat efficiently compared to other models and we will find that we achieve the best results with a relatively small forest. For this dataset, Random Forest actually runs quite efficiently compared to other models, so computation time is not a major concern in this case but it is important to consider these facts if we were working with a larger dataset.

To find the best forest size and tree depth for the model, I used Grid Search Cross Validation to compare models with different choices for the number of trees and maximum tree depths using K = 3 k-fold cross validation. The models were compared based on their cross validated accuracy scores, and the model with the highest out of sample accuracy on the training data was selected. At the end the model was refit to the entire set of training data using the best parameters. For the Max Depth, I let the model select from the choices [5, 10, 15, 20, 25], while for the number of trees in the forest I let the model select from the choices [25, 50, 100]. In the end, the

GridSearchCV found that the best model would be obtained using Max Depth = 20 and Number of Trees = 100 as seen below.

```
kf_rf = KFold(shuffle=True, random_state=3, n_splits=3)
estimator_rf = Pipeline([ ("RF", RandomForestClassifier(random_state = 18))])
params_rf = {
    'RF__max_depth': [5, 10, 15, 20, 25],
    'RF__n_estimators': [25, 50, 100]
}

grid_rf = GridSearchCV(estimator_rf, params_rf, cv=kf_rf)
```

```
grid_rf.fit(X_train, y_train)

grid_rf.best_score_
grid_rf.best_params_
```

```
{'RF__max_depth': 20, 'RF__n_estimators': 100}
```

We can view the actual predictive power on the training data below:

```
RF_cv_best = grid_rf.best_estimator_['RF']
```

```
rf_cv_train_pred = RF_cv_best.predict(X_train)
```

```
rf_cv_train_acc = metrics.accuracy_score(y_train, rf_cv_train_pred)
rf_cv_train_sen = metrics.recall_score(y_train, rf_cv_train_pred, pos_label=1)
rf_cv_train_spec = metrics.recall_score(y_train, rf_cv_train_pred, pos_label=0)
print("Trainging Accuracy  = " + str(rf_cv_train_acc) +"\n"
      "Training Sensitivity = " + str(rf_cv_train_sen) + "\n"
      "Training Specificity = " + str(rf_cv_train_spec))
```

```
Trainging Accuracy  = 0.9119718648204557
Training Sensitivity = 0.6416585175044005
Training Specificity = 1.0
```

```
confusion_matrix(y_train, rf_cv_train_pred)
```

```
array([[78504,     0],
       [ 9161, 16404]], dtype=int64)
```

We find that the model is able to properly identify all 78,504 defaulters in the training data, however we only properly predict that 16,404 will pay us back (64.17% of the total 25,565 paying customers). This model achieves a total accuracy of 91.12% with a resounding 100% specificity to identify defaulters, meaning that this model is 100% risk free on the training data.

Moving over to the out of sample testing data, we find that the Specificity is still quite strong, but the sensitivity drops considerably. We see the exact results below:

```
rf_cv_test_pred = RF_cv_best.predict(X_test)
```

```
rf_cv_test_acc = metrics.accuracy_score(y_test, rf_cv_test_pred)
rf_cv_test_sen = metrics.recall_score(y_test, rf_cv_test_pred, pos_label=1)
rf_cv_test_spec = metrics.recall_score(y_test, rf_cv_test_pred, pos_label=0)
print("Trainging Accuracy  = " + str(rf_cv_test_acc) +"\n"
      "Training Sensitivity = " + str(rf_cv_test_sen) + "\n"
      "Training Specificity = " + str(rf_cv_test_spec))
```

```
Trainging Accuracy  = 0.8680298648012377
Training Sensitivity = 0.49611703088314973
Training Specificity = 0.9908730277090106
```

```
confusion_matrix(y_test, rf_cv_test_pred)
```

```
array([[33221,   306],
       [ 5580,  5494]], dtype=int64)
```

We find that on the testing data we have a 99% specificity, properly labeling 33,221 of the total 33,527 defaulters. This is near perfect, which means that if used on new data we can be assured that the model will minimize risk to predict if new customers will default. Looking at the sensitivity however, we find that the model is only able to detect that 5,494 of the total 11,074 paying customers will actually pay. This means that our profits are limited by our model since we are too selective in who we approve as a customer.

Random Forest also allows us to do some feature selection, as it can inform us which variables helped reduce the Gini entropy the most. In Sklearn we can find this information stored as the feature importance, which gives us the average decrease in Gini entropy across all trees as a proportion of the total decrease when using that feature to split the data. A higher score indicates the feature is more important to the model. Presented below are the feature importance of each variable sorted in ascending order.

```
for i in np.argsort(RF_cv_best.feature_importances_):
    print(X_train.columns[i], RF_cv_best.feature_importances_[i])
```

```
open_credit 0.0005229532361141318
Secured_by 0.0005234875764654587
total_units_4U 0.0007550259655018447
total_units_3U 0.00113663960774069
age_<25 0.001574406488789218
Region_North-East 0.0016689058627255917
occupancy_type_sr 0.002291779651766697
total_units_2U 0.0028560964286164893
term_Q3 0.002883791014551416
loan_purpose_p2 0.0033969734603913728
age_>74 0.004360441693209426
Region_central 0.005554105478851039
interest_only 0.005670001555695373
occupancy_type_pr 0.005816136479412175
term_Q4 0.006415271860530305
loan_limit 0.006463517425791018
Gender_Sex Not Available 0.00651106621459675
loan_type_type3 0.006575412065338226
age_65-74 0.006730751362389395
age_35-44 0.007231678879027033
term_Y 0.0076010091300197135
age_55-64 0.007879428476424056
age_45-54 0.008023512562727275
Region_south 0.009119219371658473
loan_purpose_p3 0.009292776493516253
approv_in_adv 0.01036618314616569
loan_purpose_p4 0.010617397205480675
Gender_Male 0.012102874077467612
business_or_commercial 0.01232734200989304
Credit_Worthiness 0.012939055051759107
Gender_Joint 0.01464781978557182
submission_of_application 0.014953631529980096
credit_type_CRIF 0.02107743682635064
credit_type_EXP 0.024412407757585602
co-applicant_credit_type 0.025648534776467176
Neg_ammortization 0.027723414789205134
lump_sum_payment 0.038105062977970026
Credit_Score 0.06852965629351837
loan_amount 0.07122426553311503
income 0.08370894634199542
credit_type_EQUI 0.4307615835556251
```

We find that the top 10 features for determining the default status of our customers are ['credit_type_EQUI', 'income', 'loan_amount', 'Credit_Score', 'lump_sum_payment', 'Neg_ammortization', 'co-applicant_credit_type', 'credit_type_EXP', 'credit_type_CRIF', 'submission_of_application'].

The exact feature importances of a Random Forest model are subject to change slightly depending on the random state used to initialize the trees, but since we have cross validated and are using a large number n=100 trees we can be confident that these are the top features for predicting the default status of a customer. (Prior to using cross validation, I made a preliminary model on the data to get an idea of the time required for running grid search and that model produced very similar results and used these same 10 features as the most important features in

the model although the exact order among these 10 were slightly different. This model is included in the Jupyter Notebook).

I decided to fit a model using just the top 10 features for comparison. We will use the same cross validation and grid search settings as before and see how the model performs with just the most important features. This model fits much faster using less than ¼ of the original features, and still retains very strong specificity and accuracy. For this model the best cross validated performance was found using n=100 trees and max_depth = 15, so our forest is still just as large but not as deep, meaning overall, computation times for predictions will be smaller. We can see the exact scores below.

First, we look at the training data.

```
rf_imp_cv_train_pred = RF_imp_cv_best.predict(X_train[imp_feats])
```

```
rf_imp_cv_train_acc = metrics.accuracy_score(y_train, rf_imp_cv_train_pred)
rf_imp_cv_train_sen = metrics.recall_score(y_train, rf_imp_cv_train_pred, pos_label=1)
rf_imp_cv_train_spec = metrics.recall_score(y_train, rf_imp_cv_train_pred, pos_label=0)
print("Trainging Accuracy  = " + str(rf_imp_cv_train_acc) +"\n"
      "Training Sensitivity = " + str(rf_imp_cv_train_sen) + "\n"
      "Training Specificity = " + str(rf_imp_cv_train_spec))
```

```
Trainging Accuracy  = 0.8895636548828181
Training Sensitivity = 0.5515353021709368
Training Specificity = 0.9996433302761643
```

```
confusion_matrix(y_train, rf_imp_cv_train_pred)
```

```
array([[78476,    28],
       [11465, 14100]], dtype=int64)
```

We find that on the training data using only the 10 most important features our model achieves 88.96% accuracy with 99.96% specificity (only failing to catch 28 defaulters) and 55.15% Sensitivity (finding 14,100 out of our 25,565 paying customers). This model is weaker for finding paying customers than the full model but still performs nearly perfect for finding defaulters.

On the testing data we find the following.

```
: rf_imp_cv_test_pred = RF_imp_cv_best.predict(X_test[imp_feats])
```

```
: rf_imp_cv_test_acc = metrics.accuracy_score(y_test, rf_imp_cv_test_pred)
  rf_imp_cv_test_sen = metrics.recall_score(y_test, rf_imp_cv_test_pred, pos_label=1)
  rf_imp_cv_test_spec = metrics.recall_score(y_test, rf_imp_cv_test_pred, pos_label=0)
  print("Trainging Accuracy  = " + str(rf_imp_cv_test_acc) +"\n"
        "Training Sensitivity = " + str(rf_imp_cv_test_sen) + "\n"
        "Training Specificity = " + str(rf_imp_cv_test_spec))

  Trainging Accuracy  = 0.8625591354453936
  Training Sensitivity = 0.48591294925049666
  Training Specificity = 0.9869657291138485
```

```
: confusion_matrix(y_test, rf_imp_cv_test_pred)

: array([[33090,   437],
         [ 5693,  5381]], dtype=int64)
```

The model performs slightly worse on new data but nearly as well as the full model. The sensitivity for paying customers on the new model is closer to that seen on the training data compared to the full model, indicating that this model is less overfit to the training data than the full model in regards to finding paying customers. The sensitivity on this model is nearly equal to the full model indicating that the extra 31 variables provide little information to distinguish a paying customer. Looking at the specificity however we find that the model fails to predict 437 customers who will default. This is 131 more than the full model which could cost the bank over $13,000 if each customer is defaulting on at least a $100 loan payment, which is rather low for a home mortgage. According to the U.S. Census Bureau's American Housing Survey, as of 2019 the average monthly mortgage in America is $1,487 while the median is $1,200. Thus, improperly predicting these 131 defaulters is much more likely to cost the bank upwards of $120,000 per month. Considering the cost of using less variables it makes sense to use the full 41 features to ensure our default predictions are as good as can be. That being said it is helpful to note these 10 features seem to account for the majority variability in whether a person defaults, so it would be helpful to inform our employees at the bank to consider these variables first when speaking to a new potential client to do an initial screening before continuing on in the loan application process.

We find in the end that the Random Forest model performs exceptionally well for finding customers who will default, however it struggles to predict customers who will actually pay back their loans, and tends to overfit to the data presented. This may be due to the fact that our dataset is oversaturated with defaulting customers. It would be helpful now that we have identified important features for classifying defaulters, if the bank would provide more information on their paying customers so that our model can find a better way to distinguish them. That being said, we will now move on to the next Machine Learning Model in our toolset.

Logistic Regression Model

For a logistic regression model, we run a regression to fit the data to the binary response variable Status = y with y = 1 = Paid and y = 0 = Default. The model outputs the predicted likelihood that the observation belongs to the y=1 class. The model is fit using the following formula:

$$y = p(X) = \left(1 + e^{-\alpha-\beta \cdot X}\right)^{-1} \quad \text{(Eq 1)}$$

Here X represents the columns of the feature variables and β represents the vector of coefficients for the feature variables, while α is the intercept. We find the coefficients α and β using a linear regression on the formula below:

$$Logit\big(p(x)\big) = \ln\left(\frac{p(x)}{1-p(x)}\right) = \alpha + \beta \cdot X \quad \text{(Eq 2)}$$

In Sklearn, this regression is actually performed using an L2 Regularized (Ridge) Regression by default, however since I am using 41 feature variables, many of which we have already seen to be unnecessary, I opted to instead use LASSO (L1 Regularized) Regression as it is known to remove more excess features than Ridge. In practice however, the model did not remove any features under these conditions as I did not set the regularization coefficient high enough to remove any features. However, considering the explanation given in the Random Forest section, we would like to maximize total specificity, so adding features to the model is worth the cost of computation time.

Reusing the formula from my Regression project we will be computing each coefficient by minimizing the function below:

(Eq 3)

$$L_{lasso}(\hat{\beta}) = \sum_{i=1}^{n}(y_i - x_i^T\hat{\beta})^2 + \lambda\sum_{j=1}^{m}|\hat{\beta}_j|$$

In this formulation X contains a column of all 1s to represent the intercept and the vector β contains the coefficient $\beta_0 = \alpha$. Y in this case is the Logit(p(X)). Here λ is the regularization coefficient which is a hyper parameter set by the user to determine the strength of regularization. In Sklearn's LogisticRegression(), the parameter C=1/λ so that small values of C represent strong regularization, and by default C=1. In my model I did not change this as fitting the model took a very long time on my laptop, so the model did not remove any features.

After determining the appropriate values for the coefficients, the model then uses (Eq 1) to produce a predicted probability for the Status of the observation. In Sklearn the model can output the predicted probabilities, or it will make predictions on the classes, whereby the model predicts the observation belongs to class 1 = Paid if the probability is above or equal to 0.5 and the class 0 = Defaulted if the probability is below 0.5. This threshold is arbitrary however, and it is important to find a proper prediction probability threshold to meet the needs of your model. In our case we want to improve the model Specificity, so in most cases we will want to increase the prediction threshold, so that we catch more defaulting customers.

Since we are using a Regularized Regression, it is important that the scale of our variables is similar, so that the size of the regression coefficient is not dependent on the feature. If this were the case then features with small scales would require large coefficients to have the same effect on the predicted probability meaning that features with smaller scales would be subject to harsher penalties. To fix this I applied Standard scaling in the pipeline for my model to all my features which calculates the Z score of each observation for its respective column and replaces the value with the column Z score. I am aware that I have actually already performed this in the data cleaning section, however I did the Data Analysis portion of this project for my previous project over a month ago so I forgot that I already had done this. This doesn't really affect anything however since the Standard scaling transformation is consistent meaning applying the transformation a second time produces the same result. This can be considered a waste of computation time in my model, but it is actually helpful in that if we are provided new data the model will standardize this data prior to prediction.

For this model I used Grid Search Cross Validation, however I only used one set of hyper parameters as the fit time for the model was rather long. For Cross Validation, I used k=3 fold cross validation using the same random state as for the Random Forest model so that the results are comparable.

```
kf_log = KFold(shuffle=True, random_state=3, n_splits=3)
estimator_log = Pipeline([("scaler", StandardScaler()),
                          ("polynomial_features", PolynomialFeatures()),
                          ("LogReg", LogisticRegression())])
params_log = {
    'polynomial_features__degree': [1],
    'LogReg__penalty': ['l1'],
    'LogReg__solver': ['liblinear']
}

grid_log = GridSearchCV(estimator_log, params_log, cv=kf_log)
```

```
grid_log.fit(X_train, y_train)

grid_log.best_score_
```

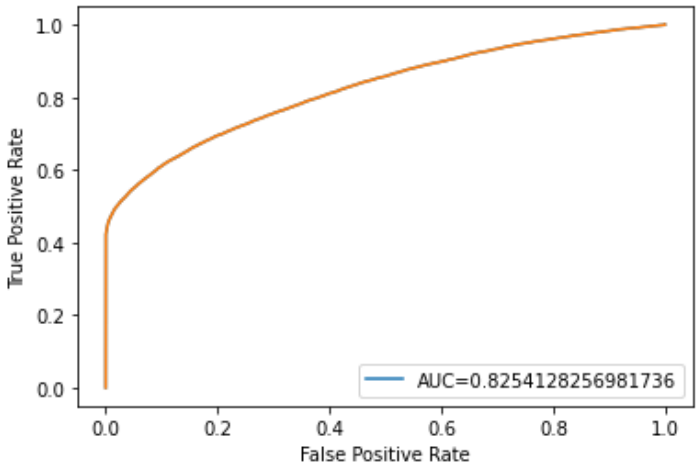
```
0.863369483764095
```

With the Lasso Regularized Logistic Regression model we were able to achieve an average cross validated accuracy of 0.86 using a prediction threshold of 0.5 on the training data.

We can investigate the predictions on the training data further by looking at the ROC curve, which plots the False Positive Rate vs the True Positive Rate of the model using different cutoff thresholds. The Area under the curve for the ROC curve measures the strength of the model, where a higher AUC indicates a stronger model. The maximum possible AUC is 1 for a perfect classifier model. The ROC curve for this Logistic Regression model on the training data is plotted below with the AUC included:

```
y_pred_proba = log_mod.predict_proba(X_train)[::,1]
fpr, tpr, thresh_log = metrics.roc_curve(y_train,  y_pred_proba)
auc = metrics.roc_auc_score(y_train, y_pred_proba)


#create ROC curve
plt.plot(fpr,tpr,label="AUC="+str(auc))
plt.plot(fpr,tpr)
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend(loc=4)
plt.show()
```



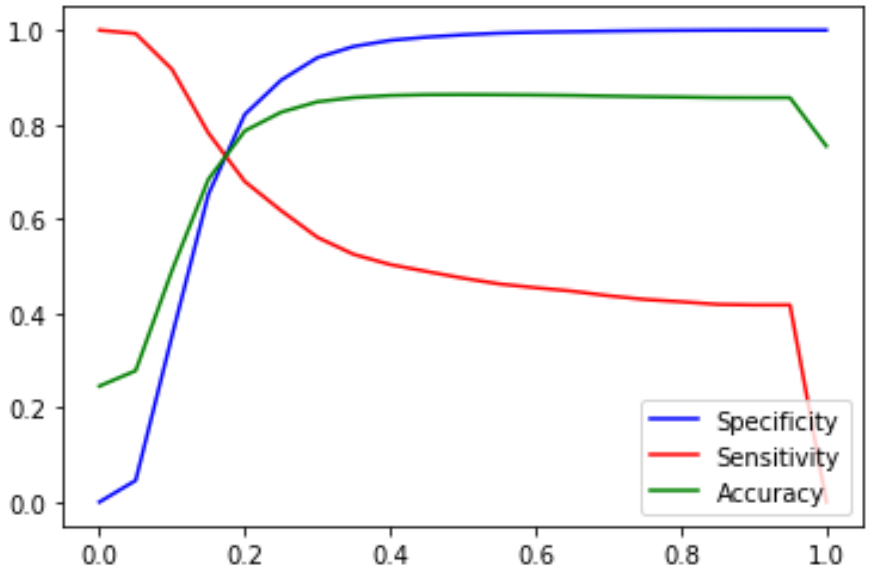
This model attains an AUC of 0.8254 which indicates we have a moderately strong classifier.

In many cases, the ROC plot is also used to find the appropriate probability prediction threshold, as the threshold which produces the point on the curve with the minimum distance from the point (0,1) has the highest overall model accuracy, however since we are not looking to maximize accuracy, we do not care about this property of the ROC curve for our model.

A better alternative for our case is to plot the Accuracy, Sensitivity, and Specificity as a function of the prediction threshold, which I have provided below. To produce the plot, I calculated these values for every 0.05 interval threshold from 0.0 up to 1.0

```
plt.plot(thresh,spec, c = 'Blue', label = 'Specificity')
plt.plot(thresh,sen, c = 'Red', label = 'Sensitivity')
plt.plot(thresh,acc, c = 'Green', label = 'Accuracy')
plt.legend(loc=4)
```

```
<matplotlib.legend.Legend at 0x1c08fd6fd30>
```



After looking at the exact values, for this model the specificity is strong (above 97%) for prediction probability thresholds above 0.4 and near perfect (above 99% specificity) for thresholds above 0.5, while the model sensitivity is between 50.3% and 41.8% for thresholds between 0.4 and 0.95. Since our model specificity is nearly stagnant for all thresholds above 0.5 while the sensitivity drops significantly for thresholds above 0.5 it is reasonable to set the model threshold at the default level of 0.5 however if we are more risk averse it may be reasonable to increase the threshold to 0.8, while a riskier client could justify a minimum threshold of 0.4 In the end, I decided to go with the default prediction threshold of 0.5.

Lastly, I decided to check the model coefficients to see if using a LASSO regression had helped the model to remove any features, however we find in the output below that all features including the intercept have nonzero coefficients.

```
log_mod['LogReg'].coef_
```

```
array([[ 0.16534186,  0.16765964, -0.08199953,  0.08230446, -0.07880211,
        -0.24421249, -0.02024511, -0.32410267, -0.04804077, -0.38254124,
         0.11342435, -0.01162487,  0.01256167, -0.15806016,  0.32682979,
         0.00823192,  0.05849972,  0.01190574, -0.00772284,  0.06098593,
         0.04523952,  0.05091021, -0.03045303,  0.07048312,  0.06280229,
        -0.13131301, -0.02804148,  0.06494437,  0.03487927,  0.01040688,
         0.00969897,  3.66681998, -0.02124635, -0.01030353,  0.03420034,
         0.0603466 ,  0.05304263,  0.03661127,  0.05706404,  0.02530557,
         0.06877489,  0.08868606]])
```

We can now view the performance of the model on the training data using the prediction threshold 0.5.

```
y_train_pred_proba = log_mod.predict_proba(X_train)[::,1]
train_pred = []
for j in y_train_pred_proba:
    if j < 0.5:
        train_pred.append(0)
    else:
        train_pred.append(1)
confusion_matrix(y_train, train_pred)
```

```
array([[77733,   771],
       [13434, 12131]], dtype=int64)
```

```
Using a prediction probability threshold of 0.5, the logistic regression model has the following scores on the training set
Trainging Accuracy  = 0.8635040213704369
Training Sensitivity = 0.4745159397613925
Training Specificity = 0.9901788443900947
```

We find that this model is significantly weaker on the training data than the Random Forest model, however the advantage to this model is that it gives us a predicted probability which we can use to advise our investors on which customers we should be wearier of even if we do approve them for a loan, allowing us to adjust their loan terms appropriately, or send possibly troublesome customers reminders to let them know their mortgage will be due in hopes it will help them stay on track of their finances.

We can now view the performance on new testing data.

Performance on Test Data

```
y_test_pred_proba = log_mod.predict_proba(X_test)[::,1]
test_pred = []
for j in y_test_pred_proba:
    if j < 0.5:
        test_pred.append(0)
    else:
        test_pred.append(1)
test_acc = metrics.accuracy_score(y_test, test_pred)
test_sen = metrics.recall_score(y_test, test_pred, pos_label=1)
test_spec = metrics.recall_score(y_test, test_pred, pos_label=0)
```

```
print("Logistic Regression test set accuracy = " + str(test_acc) + "\n"
      "test set sensitivity = " + str(test_sen) + "\n"
      "test set specificity = " + str(test_spec))
```

```
Logistic Regression test set accuracy = 0.8635232393892514
test set sensitivity = 0.4789597254831136
test set specificity = 0.9905449339338444
```

The model retains its strength on the testing c

```
confusion_matrix(y_test, test_pred)
```

```
array([[33210,   317],
       [ 5770,  5304]], dtype=int64)
```

Now we find that on the new testing data the Logistic Regression model is much closer to its reported sensitivity and specificity on the training data than the Random Forest model. In fact, this model performs only slightly worse than the Random Forest model on the new data. This may indicate that the Random Forest model is a poor choice over the Logistic Regression model as the Random Forest model is more likely to vary in strength on new data while the Logistic Regression model is proven to be consistent and has very similar strength to the Random Forest model on new data, however if we consider just the total specificity on the entire dataset, the Random Forest model is definitively better model.

Support Vector Machine Model

Support Vector Machines classify data by finding a hyperplane to separate the data into the two distinct groups. If our data has m feature variables, then an SVM model finds the (m-1) dimensional hyperplane that minimizes the classification error (maximizes the accuracy) such that observations where the response variable = 1 are “above” the hyperplane and observations where the response variable = 0 are “below” the plane. This is referred to as a linear classifier. In most cases there are infinitely many choices for the hyperplane that satisfy this definition, so the model tries to find the hyperplane that maximizes its distance from the points in each group that are closest to the hyperplane. If such a hyperplane exists, it is known as the maximum-margin hyperplane and the linear classifier it defines is known as a maximum-margin classifier.

This definition above assumes that the data are linearly separable, which is almost never the case, as we know a true classifier will always make classification errors. Under our original definition we assume there exists a “hard margin” between the two groups while a model that allows for misclassifications is said to have a “soft margin”. A secondary issue with this model is that the data may not have a linear margin between the data, and may instead have some nonlinear curved surface between the data. To work with nonlinear curved separations, we apply what is known as the kernel trick, to transform the data into a higher – possibly infinite – dimension where the data is linearly separable. For a much more in-depth analysis of the kernel trick I recommend my paper on Support Vector Regression which uses the same methods for transforming the data. That paper can be found on my GitHub linked below:

< https://github.com/DenisOByrne/Support-Vector-Regression-Report >

I will discuss briefly here the formulation of the hard margin, soft margin for the Linear SVM then discuss the kernel trick.

Linear SVM and a Hard Margin

Suppose we have a set of data $(x_1, y_1), \ldots, (x_n, y_n)$ where y_k indicates the class, -1 or 1, to which the observation belongs and each x_k is an m dimensional vector of feature observations. Then we want to find the maximum-margin m-1 dimensional hyperplane that separates all points where y = -1 from all points where y = 1 and the distance between the hyperplane and the closest point to the hyperplane in each group is maximized. The vectors normal to our hyperplane to the closest points are our support vectors, of which we wish to maximize the magnitude of each.

Any hyperplane can be written as the set of points x satisfying:

$$w^T x - b = 0 \qquad \text{(Eq. 4)}$$

where w is the vector normal to the hyperplane.

If the data is linearly separable, there exist two parallel hyperplanes that separate the two classes of data, so that the distance between them is as large as possible. The region bounded by these two hyperplanes is called the "margin", and the maximum-margin hyperplane is the hyperplane that lies halfway between them. With a normalized or standardized dataset, these hyperplanes can be described by equations 5 and 6:

$$w^T x - b = 1 \qquad \text{(Eq. 5)}$$

Any point on or above this boundary is in the positive y=1 class

$$w^T x - b = -1 \qquad \text{(Eq. 6)}$$

Any point on or above this boundary is in the positive y=1 class

Geometrically, the distance between these two hyperplanes is $\frac{2}{||w||}$ so, to maximize the distance between the planes we want to minimize $||w||$. The distance is computed using the distance from a point to a

plane equation. We also have to prevent data points from falling into the margin, we add the following constraint: for each k either:

(Eq. 7) $w^T x_k - b \geq 1$ if $y_i = 1$

or

(Eq. 8) $w^T x_k - b \leq -1$ if $y_i = -1$

These constraints state that each data point must lie on the correct side of the margin. This can be rewritten as:

(Eq. 9) $y_i(w^T x_k - b) \geq 1$ for all $i \leq 1 \leq n$

We now have the optimization problem to minimize $||w||$ subject to Eq. 9 for all i. The w and b that solve this problem determine our classifier $x \rightarrow \text{sgn}(w^T x - b)$ where sgn is the sign function. The max-margin hyperplane is completely determined by those x_i that are closest to it. These x_i are the support vectors.

Soft Margin

If the data are not linearly separable, we instead consider the hinge loss function:

(Eq. 10) $\max\left(0,\ 1 - y_i(w^T x_i - b)\right)$

For points which lie outside the margin and are properly classified, the hinge loss is 0. For data on the wrong side of the margin, the function's value is proportional to the distance from the margin.

We then wish to minimize the following:

(Eq. 11) $\lambda||w||^2 + \left[\frac{1}{n}\sum_{i=1}^{n}\left(\max\left(0,\ 1 - y_i(w^T x_i - b)\right)\right)\right]$

where $\lambda > 0$ determines the tradeoff between increasing the margin size and ensuring that the x_i lie on the correct side of the margin. For linearly separable data, if λ is small the model will be the same as a hard margin however the margin may not be maximized, while if the data is not linearly separable the model can still find an appropriate hyperplane to maximize the model accuracy.

Non-Linear SVM

The original maximum-margin hyperplane algorithm is only able to find a linear classifier, however if we replace the dot product $w^T x_i$ with a nonlinear kernel function, we are able to transform the feature space and fit the model in a transformed higher, possibly infinite, dimensional space where there does exist a linear hyperplane between the two classes. Although the classifier is a hyperplane in the transformed feature space, it may be nonlinear in the original input space.

For the purposes of this assignment, I will be comparing the results using the linear dot product, the Gaussian Radial Basis Kernel (RBF). I originally intended to check a polynomial kernel with degree d = 3 however the computation time took over 20 minutes for the linear kernel and 30 minutes for the RBF kernel. The polynomial kernel would take longer since the transformation is longer to compute, so I did not feel this was worth the effort.

Gaussian RBF Kernel Transformation:

$$k(x_i, x_j) = \exp\left[-\frac{1}{2\sigma^2}\left||x_i - x_j|\right|^2\right]$$

The formulation for this kernel is similar to the Gaussian Normal Distribution from which it derives its name, however it is used to separate circularly separable classes or curved separation hyperplanes. As we know from calculus, the exponential function is defined by a sum $\exp[x] = \sum_{k=0}^{\infty} \frac{x^k}{k!}$ which is where this kernel is actually derived. The RBF kernel is actually a transformation into an infinite dimensional space.

Before we fit an SVM model we should consider that Support Vector machines require using the training data to make predictions and all points which fall on or within their respective margins are needed to build the model. This means that with a large dataset with classes that are hard to separate, the SVM model can be extremely computationally intensive, not just for fitting, but also for predicting on new data as these training samples must all be stored in memory to rebuild the hyperplane used to classify the data. If we use a Kernel trick the support vectors are then converted to higher dimensions using more memory storage to get to the proper dimension where the hyperplane is linear. As such, SVM is usually only reserved for small datasets. If you wish to reproduce my results from my Jupyter notebook, just note that these models may take upwards of 30 minutes to fit and another 30 minutes to predict on the training data. As such, I was unable to properly optimize the choice of λ using cross validation for the tradeoff between classification error and margin size. That being said we now look at the SVM model on the Loan Default dataset.

Linear SVM can be fit in Python using sklearn's SVC model which I have shown below.

```
SV_lin = SVC(kernel = 'linear')
SV_lin.fit(X_train, y_train)
```

```
SVC(kernel='linear')
```

```
SV_lin_preds= SV_lin.predict(X_train)
```

```
SV_lin_train_acc = metrics.accuracy_score(y_train, SV_lin_preds)
SV_lin_train_sen = metrics.recall_score(y_train, SV_lin_preds, pos_label=1)
SV_lin_train_spec = metrics.recall_score(y_train, SV_lin_preds, pos_label=0)
print("Trainging Accuracy  = " + str(SV_lin_train_acc) +"\n"
      "Training Sensitivity = " + str(SV_lin_train_sen) + "\n"
      "Training Specificity = " + str(SV_lin_train_spec))
```

```
Trainging Accuracy  = 0.8615630014701784
Training Sensitivity = 0.4576178368863681
Training Specificity = 0.9931086314073169
```

```
confusion_matrix(y_train, SV_lin_preds)
```

```
array([[77963,   541],
       [13866, 11699]], dtype=int64)
```

```
SV_lin.n_support_
```

```
array([17026, 14526])
```

We find that on the training data a Linear Support Vector Machine model performs worse than the Random Forest model but better than the Logistic Regression model for identifying defaults. The model fails to identify 541 of the defaulters, with a specificity of 99.31%. However, this model performs worse than both the Random Forest and Logistic Regression models for detecting 11,699 paying customers, for a 45.76% sensitivity on the training data. Lastly, we find that the model requires using 17,026 observed defaults and 14,526 observed paying customers as Support Vectors to make predictions, which is extremely memory intensive.

Moving on to the testing data we find the following results.

```
SV_lin_test_preds= SV_lin.predict(X_test)
```

```
SV_lin_test_acc = metrics.accuracy_score(y_test, SV_lin_test_preds)
SV_lin_test_sen = metrics.recall_score(y_test, SV_lin_test_preds, pos_label=1)
SV_lin_test_spec = metrics.recall_score(y_test, SV_lin_test_preds, pos_label=0)
print("Trainging Accuracy  = " + str(SV_lin_test_acc) +"\n"
      "Training Sensitivity = " + str(SV_lin_test_sen) + "\n"
      "Training Specificity = " + str(SV_lin_test_spec))
```

```
Trainging Accuracy  = 0.860989663908881
Training Sensitivity = 0.4596351815062308
Training Specificity = 0.9935574313240075
```

```
confusion_matrix(y_test, SV_lin_test_preds)
```

```
array([[33311,   216],
       [ 5984,  5090]], dtype=int64)
```

This model has the best specificity to detect defaulting customers on new data failing to detect 216 defaulters for a 99.36% specificity, however it has the lowest sensitivity on the new data of 45.96% finding only 5090 paying customers. Although the sensitivity is rather poor, the model is extremely effective and consistent for finding defaulting customers, however this comes at an extreme computational overhead compared to the Random Forest model which performed much better on the training data and only slightly worse on the out of sample testing data.

Next up we consider the RBF kernel SVM model.

```
SV_rbf = SVC(kernel = 'rbf')
SV_rbf.fit(X_train, y_train)
```

```
SVC()
```

```
SV_rbf_train_preds = SV_rbf.predict(X_train)
```

```
SV_rbf_train_acc = metrics.accuracy_score(y_train, SV_rbf_train_preds)
SV_rbf_train_sen = metrics.recall_score(y_train, SV_rbf_train_preds, pos_label=1)
SV_rbf_train_spec = metrics.recall_score(y_train, SV_rbf_train_preds, pos_label=0)
print("Trainging Accuracy  = " + str(SV_rbf_train_acc) +"\n"
      "Training Sensitivity = " + str(SV_rbf_train_sen) + "\n"
      "Training Specificity = " + str(SV_rbf_train_spec))
```

```
Trainging Accuracy  = 0.8676647224437632
Training Sensitivity = 0.4837081947975748
Training Specificity = 0.9927010088657903
```

```
confusion_matrix(y_train, SV_rbf_train_preds)
```

```
array([[77931,   573],
       [13199, 12366]], dtype=int64)
```

```
SV_rbf.n_support_
```

```
array([18714, 14339])
```

We find that on the training data an RBF Support Vector Machine model performs slightly worse than the Linear SVM model for detecting defaults but better for identifying paying customers. The model fails to identify 573 of the defaulters, with a specificity of 99.27%. On the other hand, this model performs detects 12,366 paying customers, for a 48.37% sensitivity on the training data. Lastly, we find that the model requires using 18,714 observed defaults and 14,339 observed paying customers as Support Vectors to make predictions, which is about 1,500 more than the Linear model which added about 7 additional minutes to the computation time for predictions.

Predicting on the testing data we find the following results.

```
: SV_rbf_test_preds = SV_rbf.predict(X_test)
```

```
: SV_rbf_test_acc = metrics.accuracy_score(y_test, SV_rbf_test_preds)
  SV_rbf_test_sen = metrics.recall_score(y_test, SV_rbf_test_preds, pos_label=1)
  SV_rbf_test_spec = metrics.recall_score(y_test, SV_rbf_test_preds, pos_label=0)
  print("Trainging Accuracy  = " + str(SV_rbf_test_acc) +"\n"
        "Training Sensitivity = " + str(SV_rbf_test_sen) + "\n"
        "Training Specificity = " + str(SV_rbf_test_spec))
```

```
Trainging Accuracy  = 0.8662361830452232
Training Sensitivity = 0.48275239299259526
Training Specificity = 0.992901243773675
```

```
: confusion_matrix(y_test, SV_rbf_test_preds)
```

```
: array([[33289,   238],
         [ 5728,  5346]], dtype=int64)
```

The model retains a similar predictive power on new data. The model has a Specificity of 99.29% which is slightly lower than the Linear model but better than all other models failing to identify 238 defaulters. The RBF kernel model has a sensitivity of 48.28% finding 5346 paying customers, which is higher than the Linear SVM model and close to that of the Random Forest and Logistic Regression models. Again, the computation speed is really limiting the usefulness of this model, and I am sure that these SVM models could be refined further however the time required to tune these models makes such a process seem inefficient.

For model selection, as I am sure this bank has much more training data of paying customers to tune the model with, I would not recommend using a Support vector machine classifier as adding in more data of paying customers would only increase the computation time for prediction for such a model, making it useless as customers may take their business elsewhere if they are not approved for a loan in a timely manner. Seeing that the model is able to outperform Random Forest to detect defaulting customers on new data, it may still be of use to the bank if they wish to avoid the most risk, in which case I would recommend using the Linear SVM model.

K-Nearest Neighbors Classifier Model

K-Nearest Neighbors is a model which finds the k nearest points based on some distance metric in the training data to an observation and predicts the class of the new point as the majority class out of the neighboring points. If k is even, there can be a tie in which case the class with the lower average distance among the k points is selected as the predicted class. Since K Nearest neighbors uses the entire training dataset to make a prediction on new data, this model is computationally intensive and is not recommended for datasets as large as the one used here, so be warned that this model takes a while to run, however surprisingly, fitting and predicting with a KNN model was not as long as SVM, taking about 2 minutes to fit and another 15 minutes for predictions.

For KNN to work properly, the data must have consistent scales so that the distances along different variables are meaningful in reference to one another. If the scale on any one variable is too large, the distance between observations on that feature will be too large for points to be considered neighbors if they if they are close on all other variables besides the large scaled feature. To make sure all features were scaled appropriately, I used Standard Scaling to convert each observation to its associated Z-score for that feature variable as described earlier in this paper.

There are many different possible choices for distance metrics to use, however to be simple I chose to use the Euclidean distance which is calculated between two observations x_1 and x_2 with n features as:

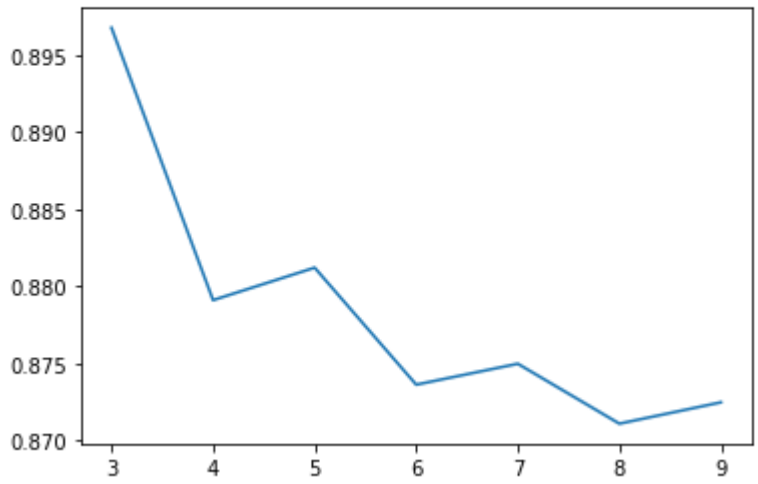
(Eq. 12) $$\sqrt{\sum_{i=1}^{n}\left(x_{1_i} - x_{2_i}\right)^2}$$

For K nearest neighbors, the choice of the number of neighbors to use for classification is a hyper parameter which should be determined by testing multiple choices of k. If we pick a large value of k, we risk making the radius of our neighborhood too large, and thus we use points that are not very similar to classify new data. Meanwhile if the choice of k is too small, we run the risk of overfitting since the consensus is made for a prediction using very few observations. In our dataset there is an additional problem with choosing k since our classes are very unbalanced, with 75% of observations belonging to the default class. If we make k too large, then it is inevitable that every neighborhood will contain more of the majority class, making our model always predict the majority class of defaulters. Thus, for unbalanced data it is important to choose a small value for k to avoid this problem. Lastly, increasing the value of k increases computation time as the model must find the k points with the smallest distance to the point being predicted, so as k increases, so does computation time. Taking all of this into consideration, I chose to test all integer values for k between 3 and 9 inclusive. To compare the models I plotted the Accuracy, Sensitivity, and Specificity of the KNN model as a function of k seen below.

KNN Accuracy vs Choice of K on training data

```
plt.plot(ks, knn_acc)
```

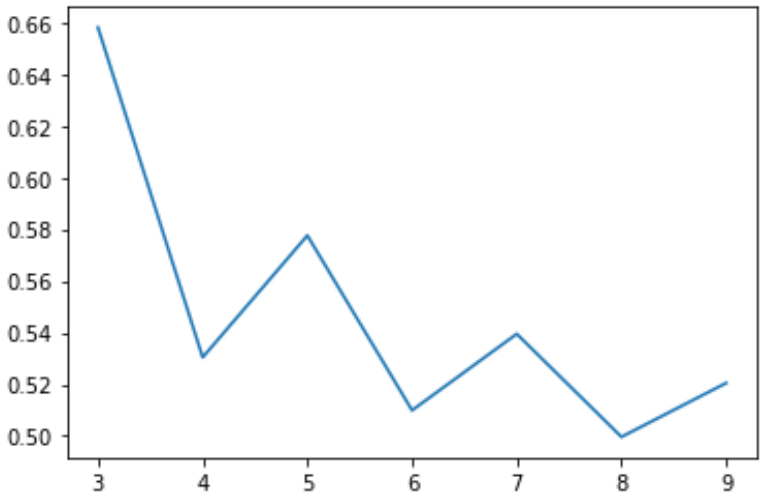
```
[<matplotlib.lines.Line2D at 0x1c08fe14e20>]
```



KNN Sensitivity vs Choice of K on training data

```
: plt.plot(ks, knn_sen)
```

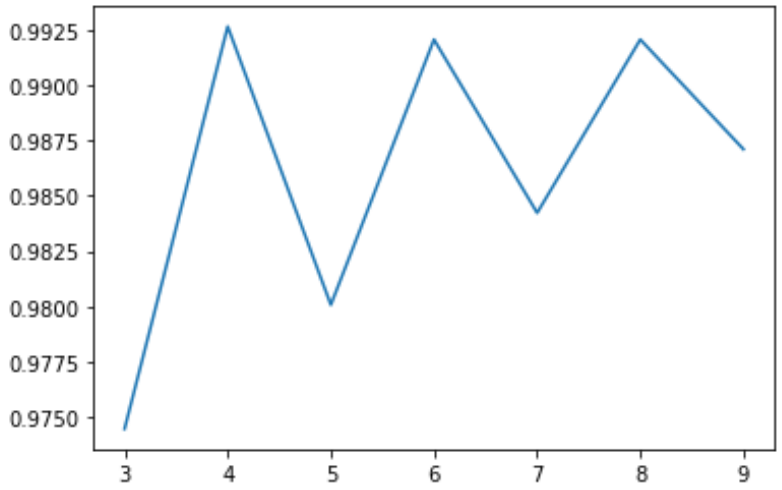
```
: [<matplotlib.lines.Line2D at 0x1c08fe847c0>]
```



KNN Specificity vs Choice of K on training data

```
plt.plot(ks, knn_spec)
```

```
[<matplotlib.lines.Line2D at 0x1c093177490>]
```



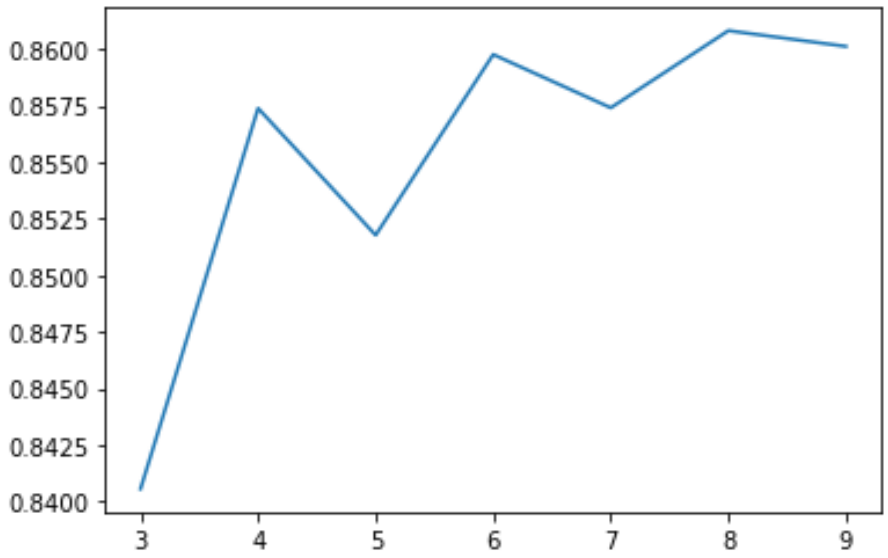
We find that as K increases the sensitivity of the model decreases significantly between 3 and 4 then continues to drop much slower for larger values of k. This indicates the model was probably overfit using k = 3. We also note that for even values of k the sensitivity is lower than for odd values above and below. This is probably due to the fact that the dataset is more densely filled with default data, so when we pick an even number for k and the model has to use the average distance of each class in the neighborhood, there are bound to be more defaulters closer to the point being predicted on. Meanwhile, the specificity jumps around randomly between 99.25% and 97.5%, which is generally stable but we find that as k increases the variance in the specificity drops and the score trends upwards. We note that for even values of k the specificity is strongest, for the same reason that it was lower on sensitivity. The best model for total accuracy seems to occur at k=3 however as mentioned choosing a small value for k has a tendency to perform worse on new data from overfitting. Since we are interested in maximizing specificity, it makes sense to choose a model with k as an even value, but we should check how the models perform on the testing data to see which models are overfit.

Below I have plotted the same plots on the testing data.

KNN Accuracy vs Choice of K on Testing data

```
plt.plot(ks, knn_testing_acc)
```

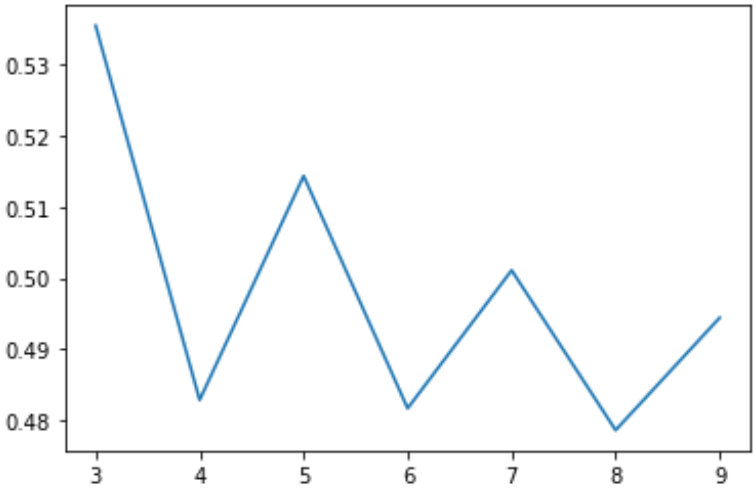
```
[<matplotlib.lines.Line2D at 0x1c0931ccc70>]
```



KNN Sensitivity vs Choice of K on Testing data

```
plt.plot(ks, knn_testing_sen)
```

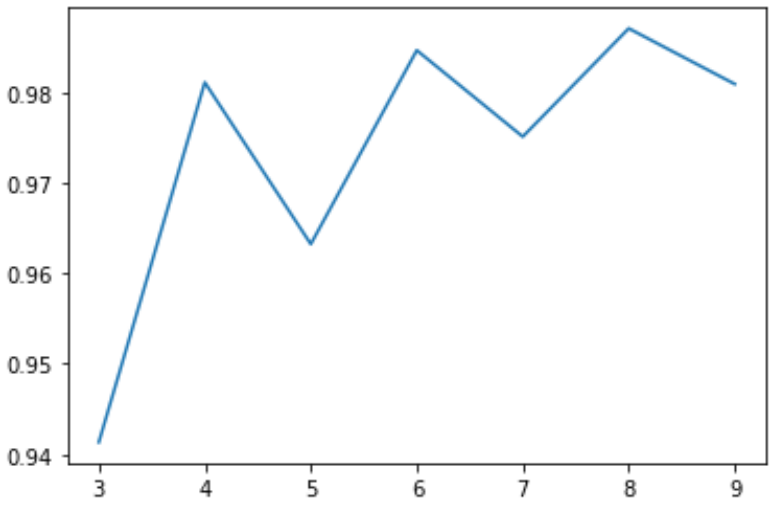
```
[<matplotlib.lines.Line2D at 0x1c09341aa00>]
```



KNN Specificity vs Choice of K on Testing data

```
plt.plot(ks, knn_testing_spec)
```

```
[<matplotlib.lines.Line2D at 0x1c093468f70>]
```



As suspected, the k=3 model was overfit to the training data and the performance is quite bad on new data. We find that in general the specificity tends to increase as k increases however as k increases the improvements begin to diminish. We still find that the model specificity is better for even values of k and improves across even values as k increases however we find surprisingly that if k is even the sensitivity is nearly constant at around 48%, while for odd values of k the sensitivity decreases as k increases.

Based on how the models perform on new data and considering that our goal is to maximize specificity, it seems reasonable to choose the model with k=8 as the best model for predicting whether a customer will default. For all choices of k used here the model fit times were rather quick however the prediction times were rather lengthy. Considering that the bank probably intends to improve the model with new data in the future by adding in data of paying customers, I would not recommend using a KNN model for this dataset.

Below are the exact performance metrics for the KNN model using k=8 neighbors for prediction

Training data Scores

```
Knn8_train_acc = metrics.accuracy_score(y_train, Knn8_train_preds)
Knn8_train_sen = metrics.recall_score(y_train, Knn8_train_preds, pos_label=1)
Knn8_train_spec = metrics.recall_score(y_train, Knn8_train_preds, pos_label=0)
print("Training Accuracy  = " + str(Knn8_train_acc) +"\n"
      "Training Sensitivity = " + str(Knn8_train_sen) + "\n"
      "Training Specificity = " + str(Knn8_train_spec))
```

```
Training Accuracy  = 0.8710951388021408
Training Sensitivity = 0.49962839820066496
Training Specificity = 0.9920640986446551
```

```
confusion_matrix(y_train, Knn8_train_preds)
```

```
array([[77881,   623],
       [12792, 12773]], dtype=int64)
```

On the training data this model, the specificity is weaker than the Random Forest models and both the Linear and RBF SVM models but stronger than the Logistic Regression model. The sensitivity of this model is the best of all models tested except the Random Forest model using all features.

The model fails to detect 623 defaulters and finds 12773 paying customers.

Testing data Scores

```
print("Testing Accuracy  = " + str(Knn8_test_acc) +"\n"
      "Testing Sensitivity = " + str(Knn8_test_sen) + "\n"
      "Testing Specificity = " + str(Knn8_test_spec))
```

```
Testing Accuracy  = 0.8608102957332795
Testing Sensitivity = 0.47859851905363915
Testing Specificity = 0.9870552092343484
```

```
confusion_matrix(y_test, Knn8_test_preds)
```

```
array([[33093,   434],
       [ 5774,  5300]], dtype=int64)
```

On the testing data the model retains its predictive power quite well. The model fails to detect 434 defaulters and finds 5300 paying customers.

On the new data this model is worse in both sensitivity and specificity to the RBF SVM model, the Logistic Regression model, and the Random Forest model using all features.

The Random Forest model using only 10 features has nearly equivalent specificity and much better sensitivity on new data.

Considering that this model is out performed by and slower than the Random Forest models on both testing and training, this model should not be used.

Gradient Boosted Tree Classifier Model

Gradient bosting is a method for developing a machine learning model from an ensemble of weak models, usually a collection of decision trees. Unlike Random Forest where each tree is fit independently however, gradient boosted trees are fit successively to correct for the errors of earlier trees in the model and each tree is fit to the entire training data instead of resampled bagged data as in Random Forest. Each decision tree in the model is trying to minimize the total error of the model, however future models add in a correction weighting to make sure that the newest model tries to fix the errors made by the previous models. Gradient Boosted trees are intended to be very weak learners, meaning that the max tree depth is very small so that future trees can find many corrections producing more complex decision boundaries in the end. In many texts GB Trees are often referred to as stumps as they are cut short. GB trees can still be fit with large trees; however, the model is usually strongest when initial trees are weak so that as the Forest grows more complex decision boundaries can be discovered whereas large trees may inhibit the learning of future trees as the algorithm is too greedy in the initial branches to make more complex decisions at the next tree's leaves. GB forests must be limited in the number of estimators however, as each tree is successively correcting for previous errors, meaning that increasing the size of the forest by adding more trees will eventually lead to overfitting, which is contrary to Random Forest where adding more trees can only improve or stagnate the model's performance on new data.

There are three main parameters which must be adjusted to tune a Gradient Boosted Tree classifier. So far, we have discussed the importance of the tree depth, and the number of trees, and we now introduce the learning rate. In GB Tree models, the learning rate λ is the weighting that the model will place on newer trees for their overall influence on the final consensus prediction. For forests with a large number of trees, a small learning rate can prevent overfitting as any single model will not have significant predictive power but only when enough trees find a similar decision boundary to overcome the small learning rate will the model adjust its consensus decision boundary. For forests with a high learning rate, we must use fewer trees otherwise the model will quickly become overfit. The converse is also true, if we have a forest with only a few trees we require a higher learning rate to improve the model otherwise the model will be severely underfit as each model is intended to be a weak learner. There is a tradeoff between the model learning rate and number of trees and the appropriate values should be found using Grid Search and Cross Validation.

Lastly, Gradient Boosted Trees work by minimizing a Loss function just as any other model, however when we build new trees our model must decide how to weight misclassifications from the previous trees in the loss function on new trees. To start we can imagine adding 1 to the loss function if the model misclassifies a point and 0 if it gets that point correct. This does not tell the whole story however, as we have an exact decision boundary, so we can know exactly how far in the wrong direction each point is from the decision boundary similar to the hinge loss of Support Vector Machines. For gradient boosting however, the model is actually learning by finding the direction to move the boundary that minimizes the loss function, which it can do by finding the direction vector where the slope (or gradient) of the Loss function is steepest (highest magnitude)

and moving the boundary in the negative direction to minimize the Loss along that vector. For this reason, Gradient Boosted models require a differentiable Loss Function but the hinge loss (including the 0 or 1 variant) is not differentiable. Instead, we must choose a loss function which penalizes correctly classified points with a smooth transition into incorrectly classified points. In Python's Gradient Boosted Trees Classifier we have a choice between two appropriate Loss Functions, the exponential AdaBoost, or the deviance calculated as the Log Likelihood Loss. The formulas for these two loss functions are presented below:

Given: $(x_1, y_1), \ldots, (x_m, y_m)$ where $x_i \in \mathcal{X}$, $y_i \in \{-1, +1\}$.
Initialize: $D_1(i) = 1/m$ for $i = 1, \ldots, m$.
For $t = 1, \ldots, T$:

- Train weak learner using distribution D_t.
- Get weak hypothesis $h_t : \mathcal{X} \to \{-1, +1\}$.
- Aim: select h_t with low weighted error:

$$\varepsilon_t = \Pr_{i \sim D_t}[h_t(x_i) \neq y_i].$$

- Choose $\alpha_t = \frac{1}{2} \ln\left(\frac{1 - \varepsilon_t}{\varepsilon_t}\right)$.
- Update, for $i = 1, \ldots, m$:

$$D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}$$

where Z_t is a normalization factor (chosen so that D_{t+1} will be a distribution).

Output the final hypothesis:

$$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right).$$

Fig. 1 The boosting algorithm AdaBoost.

Source: https://towardsdatascience.com/adaboost-for-dummies-breaking-down-the-math-and-its-equations-into-simple-terms-87f439757dcf

For simplicity we will reduce the formula to Eq 13 below:

(Eq. 13) $$A \cdot \exp[-\alpha \cdot y_i \cdot h(x_i)]$$

From the AdaBoost formula we then develop the Deviance below:

(Eq. 14) $$A \cdot Log(1 + \exp[-2 \cdot \alpha \cdot y_i \cdot h(x_i)])$$

The Log Likelihood "Deviance" loss formula is based on the AdaBoost formula. In practice the Deviance is more robust to outliers as the growth of the loss for points far away from the decision boundary is much less than that of the AdaBoost formula so AdaBoost tends to overfit more than the Deviance Loss. We can compare the results of using both loss functions to see the results on our dataset.

Now that we understand our model, we can train a Gradient Boosted Tree Classifier on our dataset. In the Jupyter notebook for this assignment, I ran the model using GridSearchCV using k=3 fold cross validation. I ran the Grid Search separately for set values for the number of estimators and a set max depth using the default Deviance Loss function, and allowed the grid search model to find the appropriate learning rate. I then compared the results for different choices for the number of estimators and tree depths. I then repeated all of this for the AdaBoost Exponential Loss function. The main reason I did not run all models in a single grid search is that doing so was much slower and I don't know why, but a secondary reason that is more useful is that we know the learning rate and the number of estimators should be inversely related, so we can skip checking sets of parameters with large numbers of trees and large learning rates or small numbers of trees and small learning rates as these models will never outperform models with inversely related learning rates and numbers of estimators.

Below I present a table of the results from different settings for the model ordered by max depth

Max Depth	Number of Trees	Best Cross Validated Learning Rate	Loss	Training Accuracy	Training Sensitivity	Training Specificity
2	100	0.75	Exponential	0.8717	0.5154	0.9878
2	100	0.75	Deviance	0.8732	0.5189	0.9886
2	500	0.5	Exponential	0.8751	0.5279	0.9882
2	500	0.25	Deviance	0.8742	0.5214	0.9891
3	100	0.5	Exponential	0.8750	0.5255	0.9889
3	100	0.5	Deviance	0.8765	0.5338	0.9882
3	200	0.25	Exponential	0.8747	0.5225	0.9894
3	200	0.25	Deviance	0.8762	0.5280	0.9896
5	100	0.25	Exponential	0.8789	0.5345	0.9910
5	100	0.25	Deviance	0.8819	0.5465	0.9911
5	200	0.25	Exponential	0.8837	0.5536	0.9913
5	200	0.1	Deviance	0.8795	0.5355	0.9915
10	100	0.05	Exponential	0.8859	0.5467	0.9964
10	100	0.05	Deviance	0.8901	0.5624	0.9969
10	200	0.05	Exponential	0.8981	0.5916	0.9979
10	200	0.05	Deviance	0.9036	0.6133	0.9981
15	100	0.05	Exponential	0.9240	0.6907	0.99997
15	100	0.025	Deviance	0.9136	0.6486	0.99989

The best model on the training data seems to be the model with max depth = 5, 200 trees, Loss function set to Exponential, and a learning rate of 0.25. Although the models with more depth have higher scoring metrics, the fit time is too slow to make them viable if new data with non-defaulting customers is added to the training set, also the cross-validation scores from grid search cv indicate that these models are overfit anyway. The models with depth = 5, 200 trees, learning rate of 0.25, and deviance loss function is the second best by a very slight margin.

I now present the same table to compare the models' performance on the test data

Max Depth	Number of Trees	Best Cross Validated Learning Rate	Loss	Testing Accuracy	Testing Sensitivity	Training Specificity
2	100	0.75	Exponential	0.8702	0.5144	0.9877
2	100	0.75	Deviance	0.8707	0.5166	0.9877
2	500	0.5	Exponential	0.8712	0.5215	0.9867
2	500	0.25	Deviance	0.8707	0.5150	0.9882
3	100	0.5	Exponential	0.8723	0.5213	0.9883
3	100	0.5	Deviance	0.8707	0.5212	0.9862
3	200	0.25	Exponential	0.8719	0.5177	0.9889
3	200	0.25	Deviance	0.8721	0.5204	0.9882
5	100	0.25	Exponential	0.8723	0.5229	0.9877
5	100	0.25	Deviance	0.8719	0.5250	0.9865
5	200	0.25	Exponential	0.8728	0.5307	0.9859
5	200	0.1	Deviance	0.8720	0.5194	0.9885
10	100	0.05	Exponential	0.8721	0.5150	0.9900
10	100	0.05	Deviance	0.8725	0.5223	0.9882
10	200	0.05	Exponential	0.8723	0.5246	0.9871
10	200	0.05	Deviance	0.8717	0.5311	0.9842
15	100	0.05	Exponential	0.8701	0.5219	0.9852
15	100	0.025	Deviance	0.8680	0.5208	0.9827

We find that most models with larger depths perform worse on out of sample data than they did on the training data. The best models on the new data are the model with a max depth of 5, 200 trees, deviance loss, and 0.1 learning rate, along with the model with a depth of 10, 100 trees, learning rate of 0.05 and exponential loss. We find that in most cases the Deviance loss function performs better on new data than the Exponential loss but only barely.

Since the difference in scores for the top two models is less than 0.01% in total accuracy, I would suggest choosing the model with reduced depth so that training on new data containing more paying customers in the future will not take as long for the model to fit.

Thus, we find that the best setting of hyper parameters for the Gradient Boosted Trees Classifier model is a max depth of 5, 200 trees, deviance loss, and 0.1 learning rate. With this model we achieve the following confusion matrix on the training data.

```
GBC_best = GradientBoostingClassifier(max_depth = 5, n_estimators = 200, learning_rate = 0.1, loss = 'deviance')
```

```
GBC_best.fit(X_train, y_train)
```

```
GradientBoostingClassifier(max_depth=5, n_estimators=200)
```

```
gbc_best_train_preds = GBC_best.predict(X_train)
```

```
gbc_best_train_acc = metrics.accuracy_score(y_train, gbc_best_train_preds)
gbc_best_train_sen = metrics.recall_score(y_train, gbc_best_train_preds, pos_label=1)
gbc_best_train_spec = metrics.recall_score(y_train, gbc_best_train_preds, pos_label=0)
print("Training Accuracy  = " + str(gbc_best_train_acc) +"\n"
      "Training Sensitivity = " + str(gbc_best_train_sen) + "\n"
      "Training Specificity = " + str(gbc_best_train_spec))
```

```
Training Accuracy  = 0.8795030220334585
Training Sensitivity = 0.5355368668100919
Training Specificity = 0.9915163558544787
```

```
confusion_matrix(y_train, gbc_best_train_preds)
```

```
array([[77838,   666],
       [11874, 13691]], dtype=int64)
```

This model fails to detect 666 defaulters in the training data (maybe we shouldn't have chosen this model for superstitious reasons ha-ha.) Meanwhile we are able to find 13691 paying customers.

Onto the testing data

```
gbc_best_test_preds = GBC_best.predict(X_test)
```

```
gbc_best_test_acc = metrics.accuracy_score(y_test, gbc_best_test_preds)
gbc_best_test_sen = metrics.recall_score(y_test, gbc_best_test_preds, pos_label=1)
gbc_best_test_spec = metrics.recall_score(y_test, gbc_best_test_preds, pos_label=0)
print("Testing Accuracy  = " + str(gbc_best_test_acc) +"\n"
      "Testing Sensitivity = " + str(gbc_best_test_sen) + "\n"
      "Testing Specificity = " + str(gbc_best_test_spec))
```

```
Testing Accuracy  = 0.87204322773032
Testing Sensitivity = 0.51950514719162
Testing Specificity = 0.9884868911623468
```

```
confusion_matrix(y_test, gbc_best_test_preds)
```

```
array([[33141,   386],
       [ 5321,  5753]], dtype=int64)
```

This model fails to detect 386 defaulters, and finds 5753 paying customers on the out of sample data.

Selecting the Best Overall Model

Reproduced below are the training and testing accuracy, sensitivity and specificity for the best settings on each type of model

Model	Training Accuracy	Training Sensitivity	Training Specificity	Testing Accuracy	Testing Sensitivity	Testing Specificity
Random Forest	0.91197186482045 57	0.6416585 175044005	1.0	0.8680298 64801237	0.4961170 30883149	0.9908730 27709010
Logistic Regression	0.863504 02137043 69	0.4745159 397613925	0.9901788 443900947	0.8635232 393892514	0.4789597 254831136	0.9905449 339338444
SVM Linear	0.861563 00147017 84	0.4576178 368863681	0.9931086 314073169	0.8609896 63908881	0.4596351 815062308	0.9935574 313240075
SVM RBF	0.867664 72244376 32	0.4837081 947975748	0.9927010 088657903	0.8662361 830452232	0.4827523 929925952 6	0.9929012 43773675
K Nearest Neighbors	0.871095 13880214 08	0.4996283 982006649 6	0.9920640 986446551	0.8608102 957332795	0.4785985 190536391 5	0.9870552 092343484
GB Trees	0.879503 02203345 85	0.5355368 668100919	0.9915163 558544787	0.8720432 2773032	0.5195051 4719162	0.9884868 911623468

The best model for detecting defaults would have to be the Linear kernel SVM model, as it has the highest accuracy on the out of sample data followed very closely behind by the RBF kernel SVM. We must remember that the memory requirement to run these models gets worse as we add new data to the training set however and the bank is keeping the data of paying customers secret to retrain the models in the future making these models a poor choice overall. The model with the best overall accuracy on out of sample data would be the GB Trees model. The GB Trees model would be a good choice if accuracy was our main concern, but we know that the most important classification is specifying defaulters, which the GB trees model is the worst at on the training data and the second worst at on the testing data.

The best overall model for the purposes of the bank however is the Random Forest model, which detects all defaulting customers in the training data and just over 99% of all defaulters on the testing data and still is able to find 49.6% of paying customers in the out of sample data on top of finding the highest proportion of 64.17% of paying customers in the training data.

On top of all this the Random Forest model prediction and training speed is the fastest of all models tested taking just under 12 seconds to fit and predict on the training data. Coupled with the easy to understand feature importances output, this makes for a great model to help advise our bank managers on which customers they should keep watch on more closely to make sure they pay their loans on time.

Recall that the best settings for the Random Forest model were n_estimators= 100 and max_depth = 20

Next Steps

To improve the model, as mentioned many times throughout this paper, we will need to include new data of customers who do pay their loans back so that the model is able to improve the sensitivity while retaining a strong specificity of defaulters. We know that the bank already has data on such customers, as if they did not, this bank would clearly be out of business by now since the dataset we have from them is 75% customers who didn't pay their loans. On top of adding in new data, I showed in the Random Forest model that the specificity for identifying defaulters only really depended on the 10 most important features, so it would be interesting to see if such performance is still just as strong when new data of paying customers is added. Lastly, the bank should try to do a better job of collecting data on the following features that we had to remove- rate_of_interest, Interest_rate_spread, Upfront_charges, property_value, LTV, and dtir1 - from their paying customers, as these may be valuable features, especially the property value since we are looking at home loans. Recall that most of these features were missing only for paying customers which made using them in the model lead to unintentional bias. As all of these are paying customers, the bank should have no problem attaining the information, as they have not tried to avoid payment so they would appear to be on good terms with the bank. I would recommend adding an incentive on their monthly bill to get them to call since this information could be valuable to the bank and most paying customers may ignore a simple request for the information otherwise.