**Problem Statement**

Understanding the problem statement is the first and foremost step. This would help you give an intuition of what you will face ahead of time. Let us see the problem statement -

***Dream Housing Finance company deals in all home loans. They have presence across all urban, semi urban and rural areas. Customer first apply for home loan after that company validates the customer eligibility for loan. Company wants to automate the loan eligibility process (real time) based on customer detail provided while filling online application form. These details are Gender, Marital Status, Education, Number of Dependents, Income, Loan Amount, Credit History and others. To automate this process, they have given a problem to identify the customers segments, those are eligible for loan amount so that they can specifically target these customers.***

It is a classification problem where we have to predict whether a loan would be approved or not. In a classification problem, we have to predict discrete values based on a given set of independent variable(s). Classification can be of two types:

* **Binary Classification** : In this classification we have to predict either of the two given classes. For example: classifying the gender as male or female, predicting the result as win or loss, etc.
* **Multiclass Classification** : Here we have to classify the data into three or more classes. For example: classifying a movie's genre as comedy, action or romantic, classify fruits as oranges, apples, or pears, etc.

Loan prediction is a very common real-life problem that each retail bank faces atleast once in its lifetime. If done correctly, it can save a lot of man hours at the end of a retail bank.

Although this case-study is specifically built to give you a walkthrough of Loan Prediction problem, you can always refer the content to get a comprehensive overview to solve a classification problem.

## Hypothesis Generation

After looking at the problem statement, we will now move into hypothesis generation. It is the process of listing out all the possible factors that can affect the outcome.

### **What is hypothesis generation?**

This is a very important stage in any data science/machine learning pipeline. It involves understanding the problem in detail by brainstorming as many factors as possible which can impact the outcome. It is done by understanding the problem statement thoroughly and before looking at the data.

Below are some of the factors which I think can affect the Loan Approval (dependent variable for this loan prediction problem):

* **Salary:** Applicants with high income should have more chances of loan approval.
* **Previous history:** Applicants who have re-paid their previous debts should have higher chances of loan approval.
* **Loan amount:** Loan approval should also depend on the loan amount. If the loan amount is less, chances of loan approval should be high.
* **Loan term:** Loan for less time period and less amount should have higher chances of approval.
* **EMI:** Lesser the amount to be paid monthly to repay the loan, higher the chances of loan approval.

These are some of the factors which I think can affect the target variable; you can come up with many more factors.

## Getting the system ready and loading the data

We will be using Python for this course along with the below listed libraries. The version of these libraries is mentioned below:

### **Specifications**

* Python = 3.7
* pandas = 0.20.3
* seaborn = 1.0.0
* sklearn = 0.19.1

To install the latest release of seaborn visit <https://seaborn.pydata.org/installing.html>

### **Loading Packages**

import pandas as pd

importnumpy as np # For mathematical calculations

importseaborn as sns # For data visualization

importmatplotlib.pyplot as plt # For plotting graphs

%matplotlib inline

import warnings # To ignore any warnings

warnings.filterwarnings("ignore")

### **Data**

For this practice problem, two CSV files are given: train and test.

* Train file will be used for training the model, i.e. our model will learn from this file. It contains all the independent variables and the target variable.
* Test file contains all the independent variables, but not the target variable. We will apply the model to predict the target variable for the test data.

### **Reading data**

train=pd.read\_csv("train.csv")

test=pd.read\_csv("test.csv")

Let’s make a copy of train and test data so that even if we have to make any changes in these datasets we would not lose the original datasets.

train\_original=train.copy()

test\_original=test.copy()

**Understanding the Data**

In this section, we will look at the structure of the train and test datasets. Firstly, we will check the features present in our data and then we will look at their data types.

train.columns

Index(['Loan\_ID', 'Gender', 'Married', 'Dependents', 'Education',

'Self\_Employed', 'ApplicantIncome', 'CoapplicantIncome', 'LoanAmount',

'Loan\_Amount\_Term', 'Credit\_History', 'Property\_Area', 'Loan\_Status'],

dtype='object')

We have 12 independent variables and 1 target variable, i.e. Loan\_Status in the train dataset. Let’s also have a look at the columns of test dataset.

test.columns

Index(['Loan\_ID', 'Gender', 'Married', 'Dependents', 'Education',

'Self\_Employed', 'ApplicantIncome', 'CoapplicantIncome', 'LoanAmount',

'Loan\_Amount\_Term', 'Credit\_History', 'Property\_Area'],

dtype='object')

We have similar features in the test dataset as the train dataset except the Loan\_Status. We will predict the Loan\_Status using the model built using the train data.

Given below is the description for each variable.

| **Variable** | **Description** |
| --- | --- |
| Loan\_ID | Unique Loan ID |
| Gender | Male/ Female |
| Married | Applicant married (Y/N) |
| Dependents | Number of dependents |
| Education | Applicant Education (Graduate/Under Graduate) |
| Self\_Employed | Self employed (Y/N) |
| ApplicantIncome | Applicant income |
| CoapplicantIncome | Coapplicant income |
| LoanAmount | Loan amount in thousands |
| Loan\_Amount\_Term | Term of loan in months |
| Credit\_History | Credit history meets guidelines |
| Property\_Area | Urban/ Semi Urban/ Rural |
| Loan\_Status | Loan approved (Y/N) |

# Print data types for each variable

train.dtypes

Loan\_ID object

Gender object

Married object

Dependents object

Education object

Self\_Employed object

ApplicantIncome int64

CoapplicantIncome float64

LoanAmount float64

Loan\_Amount\_Term float64

Credit\_History float64

Property\_Area object

Loan\_Status object

dtype: object

We can see there are three format of data types:

* object: Object format means variables are categorical. Categorical variables in our dataset are: Loan\_ID, Gender, Married, Dependents, Education, Self\_Employed, Property\_Area, Loan\_Status
* int64: It represents the integer variables. ApplicantIncome is of this format.
* float64: It represents the variable which have some decimal values involved. They are also numerical variables. Numerical variables in our dataset are: CoapplicantIncome, LoanAmount, Loan\_Amount\_Term, and Credit\_History

Let’s look at the shape of the dataset.

train.shape, test.shape

**Univariate Analysis**

In this section, we will do univariate analysis. It is the simplest form of analyzing data where we examine each variable individually. For categorical features we can use frequency table or bar plots which will calculate the number of each category in a particular variable. For numerical features, probability density plots can be used to look at the distribution of the variable.

**Target Variable**

We will first look at the target variable, i.e., Loan\_Status. As it is a categorical variable, let us look at its frequency table, percentage distribution and bar plot.

Frequency table of a variable will give us the count of each category in that variable.

train['Loan\_Status'].value\_counts()

# Normalize can be set to True to print proportions instead of number

train['Loan\_Status'].value\_counts(normalize=True)

train['Loan\_Status'].value\_counts().plot.bar()

Now lets visualize each variable separately. Different types of variables are Categorical, ordinal and numerical.

* Categorical features: These features have categories (Gender, Married, Self\_Employed, Credit\_History, Loan\_Status)
* Ordinal features: Variables in categorical features having some order involved (Dependents, Education, Property\_Area)
* Numerical features: These features have numerical values (ApplicantIncome, CoapplicantIncome, LoanAmount, Loan\_Amount\_Term)

Let’s visualize the categorical and ordinal features first.

**Independent Variable (Categorical)**

plt.figure(1)

plt.subplot(221)

train['Gender'].value\_counts(normalize=True).plot.bar(figsize=(20,10), title= 'Gender')

plt.subplot(222)

train['Married'].value\_counts(normalize=True).plot.bar(title= 'Married')

plt.subplot(223)

train['Self\_Employed'].value\_counts(normalize=True).plot.bar(title= 'Self\_Employed')

plt.subplot(224)

train['Credit\_History'].value\_counts(normalize=True).plot.bar(title= 'Credit\_History')

plt.show()

It can be inferred from the above bar plots that:

* 80% applicants in the dataset are male.
* Around 65% of the applicants in the dataset are married.
* Around 15% applicants in the dataset are self employed.
* Around 85% applicants have repaid their debts.

Now let’s visualize the ordinal variables.

**Independent Variable (Ordinal)**

plt.figure(1)

plt.subplot(131)

train['Dependents'].value\_counts(normalize=True).plot.bar(figsize=(24,6), title= 'Dependents')

plt.subplot(132)

train['Education'].value\_counts(normalize=True).plot.bar(title= 'Education')

plt.subplot(133)

train['Property\_Area'].value\_counts(normalize=True).plot.bar(title= 'Property\_Area')

plt.show()

Following inferences can be made from the above bar plots:

* Most of the applicants don’t have any dependents.
* Around 80% of the applicants are Graduate.
* Most of the applicants are from Semiurban area.

**Independent Variable (Numerical)**

Till now we have seen the categorical and ordinal variables and now lets visualize the numerical variables. Let’s look at the distribution of Applicant income first.

plt.figure(1)

plt.subplot(121)

sns.distplot(train['ApplicantIncome']);

plt.subplot(122)

train['ApplicantIncome'].plot.box(figsize=(16,5))

plt.show()

It can be inferred that most of the data in the distribution of applicant income is towards left which means it is not normally distributed. We will try to make it normal in later sections as algorithms works better if the data is normally distributed.

The boxplot confirms the presence of a lot of outliers/extreme values. This can be attributed to the income disparity in the society. Part of this can be driven by the fact that we are looking at people with different education levels. Let us segregate them by Education:

train.boxplot(column='ApplicantIncome', by = 'Education')

plt.suptitle("")

Text(0.5,0.98,'')

We can see that there are a higher number of graduates with very high incomes, which are appearing to be the outliers.

Let’s look at the Coapplicant income distribution.

plt.figure(1)

plt.subplot(121)

sns.distplot(train['CoapplicantIncome']);

plt.subplot(122)

train['CoapplicantIncome'].plot.box(figsize=(16,5))

plt.show()

We see a similar distribution as that of the applicant income. Majority of coapplicant’s income ranges from 0 to 5000. We also see a lot of outliers in the coapplicant income and it is not normally distributed.

Let’s look at the distribution of LoanAmount variable.

plt.figure(1)

plt.subplot(121)

df=train.dropna()

sns.distplot(df['LoanAmount']);

plt.subplot(122)

train['LoanAmount'].plot.box(figsize=(16,5))

plt.show()

We see a lot of outliers in this variable and the distribution is fairly normal. We will treat the outliers in later sections.

Now we would like to know how well each feature correlate with Loan Status. So, in the next section we will look at bivariate analysis.

**Bivariate Analysis**

Let’s recall some of the hypotheses that we generated earlier:

* Applicants with high income should have more chances of loan approval.
* Applicants who have repaid their previous debts should have higher chances of loan approval.
* Loan approval should also depend on the loan amount. If the loan amount is less, chances of loan approval should be high.
* Lesser the amount to be paid monthly to repay the loan, higher the chances of loan approval.

Let’s try to test the above mentioned hypotheses using bivariate analysis

After looking at every variable individually in univariate analysis, we will now explore them again with respect to the target variable.

**Categorical Independent Variable vs Target Variable**

First of all we will find the relation between target variable and categorical independent variables. Let us look at the stacked bar plot now which will give us the proportion of approved and unapproved loans.

Gender=pd.crosstab(train['Gender'],train['Loan\_Status'])

Gender.div(Gender.sum(1).astype(float), axis=0).plot(kind="bar", stacked=True, figsize=(4,4))

It can be inferred that the proportion of male and female applicants is more or less same for both approved and unapproved loans.

Now let us visualize the remaining categorical variables vs target variable.

Married=pd.crosstab(train['Married'],train['Loan\_Status'])

Dependents=pd.crosstab(train['Dependents'],train['Loan\_Status'])

Education=pd.crosstab(train['Education'],train['Loan\_Status'])

Self\_Employed=pd.crosstab(train['Self\_Employed'],train['Loan\_Status'])

Married.div(Married.sum(1).astype(float), axis=0).plot(kind="bar", stacked=True, figsize=(4,4))

plt.show()

Dependents.div(Dependents.sum(1).astype(float), axis=0).plot(kind="bar", stacked=True)

plt.show()

Education.div(Education.sum(1).astype(float), axis=0).plot(kind="bar", stacked=True, figsize=(4,4))

plt.show()

Self\_Employed.div(Self\_Employed.sum(1).astype(float), axis=0).plot(kind="bar", stacked=True, figsize=(4,4))

plt.show()

* Proportion of married applicants is higher for the approved loans.
* Distribution of applicants with 1 or 3+ dependents is similar across both the categories of Loan\_Status.
* There is nothing significant we can infer from Self\_Employed vs Loan\_Status plot.

Now we will look at the relationship between remaining categorical independent variables and Loan\_Status.

Credit\_History=pd.crosstab(train['Credit\_History'],train['Loan\_Status'])

Property\_Area=pd.crosstab(train['Property\_Area'],train['Loan\_Status'])

Credit\_History.div(Credit\_History.sum(1).astype(float), axis=0).plot(kind="bar", stacked=True, figsize=(4,4))

plt.show()

Property\_Area.div(Property\_Area.sum(1).astype(float), axis=0).plot(kind="bar", stacked=True)

plt.show()

* It seems people with credit history as 1 are more likely to get their loans approved.
* Proportion of loans getting approved in semiurban area is higher as compared to that in rural or urban areas.

Now let’s visualize numerical independent variables with respect to target variable.

**Numerical Independent Variable vs Target Variable**

We will try to find the mean income of people for which the loan has been approved vs the mean income of people for which the loan has not been approved.

train.groupby('Loan\_Status')['ApplicantIncome'].mean().plot.bar()

Here the y-axis represents the mean applicant income. We don’t see any change in the mean income. So, let’s make bins for the applicant income variable based on the values in it and analyze the corresponding loan status for each bin.

bins=[0,2500,4000,6000,81000]

group=['Low','Average','High', 'Very high']

train['Income\_bin']=pd.cut(df['ApplicantIncome'],bins,labels=group)

Income\_bin=pd.crosstab(train['Income\_bin'],train['Loan\_Status'])

Income\_bin.div(Income\_bin.sum(1).astype(float), axis=0).plot(kind="bar", stacked=True)

plt.xlabel('ApplicantIncome')

P = plt.ylabel('Percentage')

It can be inferred that Applicant income does not affect the chances of loan approval which contradicts our hypothesis in which we assumed that if the applicant income is high the chances of loan approval will also be high.

We will analyze the coapplicant income and loan amount variable in similar manner.

bins=[0,1000,3000,42000]

group=['Low','Average','High']

train['Coapplicant\_Income\_bin']=pd.cut(df['CoapplicantIncome'],bins,labels=group)

Coapplicant\_Income\_bin=pd.crosstab(train['Coapplicant\_Income\_bin'],train['Loan\_Status'])

Coapplicant\_Income\_bin.div(Coapplicant\_Income\_bin.sum(1).astype(float), axis=0).plot(kind="bar", stacked=True)

plt.xlabel('CoapplicantIncome')

P = plt.ylabel('Percentage')

It shows that if coapplicant’s income is less the chances of loan approval are high. But this does not look right. The possible reason behind this may be that most of the applicants don’t have any coapplicant so the coapplicant income for such applicants is 0 and hence the loan approval is not dependent on it. So we can make a new variable in which we will combine the applicant’s and coapplicant’s income to visualize the combined effect of income on loan approval.

Let us combine the Applicant Income and Coapplicant Income and see the combined effect of Total Income on the Loan\_Status.

train['Total\_Income']=train['ApplicantIncome']+train['CoapplicantIncome']

bins=[0,2500,4000,6000,81000]

group=['Low','Average','High', 'Very high']

train['Total\_Income\_bin']=pd.cut(train['Total\_Income'],bins,labels=group)

Total\_Income\_bin=pd.crosstab(train['Total\_Income\_bin'],train['Loan\_Status'])

Total\_Income\_bin.div(Total\_Income\_bin.sum(1).astype(float), axis=0).plot(kind="bar", stacked=True)

plt.xlabel('Total\_Income')

plt.ylabel('Percentage')

We can see that Proportion of loans getting approved for applicants having low Total\_Income is very less as compared to that of applicants with Average, High and Very High Income.

Let’s visualize the Loan amount variable.

bins=[0,100,200,700]

group=['Low','Average','High']

train['LoanAmount\_bin']=pd.cut(df['LoanAmount'],bins,labels=group)

LoanAmount\_bin=pd.crosstab(train['LoanAmount\_bin'],train['Loan\_Status'])

LoanAmount\_bin.div(LoanAmount\_bin.sum(1).astype(float), axis=0).plot(kind="bar", stacked=True)

plt.xlabel('LoanAmount')

plt.ylabel('Percentage')

It can be seen that the proportion of approved loans is higher for Low and Average Loan Amount as compared to that of High Loan Amount which supports our hypothesis in which we considered that the chances of loan approval will be high when the loan amount is less.

Let’s drop the bins which we created for the exploration part. We will change the 3+ in dependents variable to 3 to make it a numerical variable.We will also convert the target variable’s categories into 0 and 1 so that we can find its correlation with numerical variables. One more reason to do so is few models like logistic regression takes only numeric values as input. We will replace N with 0 and Y with 1.

train=train.drop(['Income\_bin', 'Coapplicant\_Income\_bin', 'LoanAmount\_bin', 'Total\_Income\_bin', 'Total\_Income'], axis=1)

train['Dependents'].replace('3+', 3,inplace=True)

test['Dependents'].replace('3+', 3,inplace=True)

train['Loan\_Status'].replace('N', 0,inplace=True)

train['Loan\_Status'].replace('Y', 1,inplace=True)

**Missing Value and Outlier Treatment**

After exploring all the variables in our data, we can now impute the missing values and treat the outliers because missing data and outliers can have adverse effect on the model performance.

**Missing value imputation**

Let’s list out feature-wise count of missing values.

train.isnull().sum()

Loan\_ID 0

Gender 13

Married 3

Dependents 15

Education 0

Self\_Employed 32

ApplicantIncome 0

CoapplicantIncome 0

LoanAmount 0

Loan\_Amount\_Term 14

Credit\_History 50

Property\_Area 0

Loan\_Status 0

dtype: int64

There are missing values in Gender, Married, Dependents, Self\_Employed, LoanAmount, Loan\_Amount\_Term and Credit\_History features.

We will treat the missing values in all the features one by one.

We can consider these methods to fill the missing values:

* **For numerical variables:** imputation using mean or median
* **For categorical variables:** imputation using mode

There are very less missing values in Gender, Married, Dependents, Credit\_History and Self\_Employed features so we can fill them using the mode of the features.

train['Gender'].fillna(train['Gender'].mode()[0], inplace=True)

train['Married'].fillna(train['Married'].mode()[0], inplace=True)

train['Dependents'].fillna(train['Dependents'].mode()[0], inplace=True)

train['Self\_Employed'].fillna(train['Self\_Employed'].mode()[0], inplace=True)

train['Credit\_History'].fillna(train['Credit\_History'].mode()[0], inplace=True)

Now let’s try to find a way to fill the missing values in Loan\_Amount\_Term. We will look at the value count of the Loan amount term variable.

train['Loan\_Amount\_Term'].value\_counts()

360.0 512

180.0 44

480.0 15

300.0 13

84.0 4

240.0 4

120.0 3

36.0 2

60.0 2

12.0 1

Name: Loan\_Amount\_Term, dtype: int64

It can be seen that in loan amount term variable, the value of 360 is repeating the most. So we will replace the missing values in this variable using the mode of this variable.

train['Loan\_Amount\_Term'].fillna(train['Loan\_Amount\_Term'].mode()[0], inplace=True)

Now we will see the LoanAmount variable. As it is a numerical variable, we can use mean or median to impute the missing values. We will use median to fill the null values as earlier we saw that loan amount have outliers so the mean will not be the proper approach as it is highly affected by the presence of outliers.

train['LoanAmount'].fillna(train['LoanAmount'].median(), inplace=True)

Now lets check whether all the missing values are filled in the dataset.

train.isnull().sum()

Loan\_ID 0

Gender 0

Married 0

Dependents 0

Education 0

Self\_Employed 0

ApplicantIncome 0

CoapplicantIncome 0

LoanAmount 0

Loan\_Amount\_Term 0

Credit\_History 0

Property\_Area 0

Loan\_Status 0

dtype: int64

As we can see that all the missing values have been filled in the test dataset. Let’s fill all the missing values in the test dataset too with the same approach.

test['Gender'].fillna(train['Gender'].mode()[0], inplace=True)

test['Dependents'].fillna(train['Dependents'].mode()[0], inplace=True)

test['Self\_Employed'].fillna(train['Self\_Employed'].mode()[0], inplace=True)

test['Credit\_History'].fillna(train['Credit\_History'].mode()[0], inplace=True)

test['Loan\_Amount\_Term'].fillna(train['Loan\_Amount\_Term'].mode()[0], inplace=True)

test['LoanAmount'].fillna(train['LoanAmount'].median(), inplace=True)

**Outlier Treatment**

As we saw earlier in univariate analysis, LoanAmount contains outliers so we have to treat them as the presence of outliers affects the distribution of the data. Let's examine what can happen to a data set with outliers. For the sample data set:

1, 1, 2, 2, 2, 2, 3, 3, 3, 4, 4

We find the following: mean, median, mode, and standard deviation

Mean = 2.58

Median = 2.5

Mode = 2

Standard Deviation = 1.08

If we add an outlier to the data set:

1, 1, 2, 2, 2, 2, 3, 3, 3, 4, 4, 400

The new values of our statistics are:

Mean = 35.38

Median = 2.5

Mode = 2

Standard Deviation = 114.74

It can be seen that having outliers often has a significant effect on the mean and standard deviation and hence affecting the distribution. We must take steps to remove outliers from our data sets.

Due to these outliers bulk of the data in the loan amount is at the left and the right tail is longer. This is called right skewness. One way to remove the skewness is by doing the log transformation. As we take the log transformation, it does not affect the smaller values much, but reduces the larger values. So, we get a distribution similar to normal distribution.

Let’s visualize the effect of log transformation. We will do the similar changes to the test file simultaneously.

train['LoanAmount\_log'] = np.log(train['LoanAmount'])

train['LoanAmount\_log'].hist(bins=20)

test['LoanAmount\_log'] = np.log(test['LoanAmount'])

Now the distribution looks much closer to normal and effect of extreme values has been significantly subsided. Let’s build a logistic regression model and make predictions for the test dataset.

**Evaluation Metrics for Classification Problems**

The process of model building is not complete without evaluation of model’s performance. Suppose we have the predictions from the model, how can we decide whether the predictions are accurate? We can plot the results and compare them with the actual values, i.e. calculate the distance between the predictions and actual values. Lesser this distance more accurate will be the predictions. Since this is a classification problem, we can evaluate our models using any one of the following evaluation metrics:

**Accuracy**: Let us understand it using the confusion matrix which is a tabular representation of Actual vs Predicted values. This is how a confusion matrix looks like:

* True Positive - Targets which are actually true(Y) and we have predicted them true(Y)
* True Negative - Targets which are actually false(N) and we have predicted them false(N)
* False Positive - Targets which are actually false(N) but we have predicted them true(T)
* False Negative - Targets which are actually true(T) but we have predicted them false(N)

Using these values, we can calculate the accuracy of the model. The accuracy is given by:

**Precision**: It is a measure of correctness achieved in true prediction i.e. of observations labeled as true, how many are actually labeled true.

Precision = TP / (TP + FP)

**Recall(Sensitivity)** - It is a measure of actual observations which are predicted correctly i.e. how many observations of true class are labeled correctly. It is also known as ‘Sensitivity’.

Recall = TP / (TP + FN)

**Specificity** - It is a measure of how many observations of false class are labeled correctly.

Specificity = TN / (TN + FP)

Specificity and Sensitivity plays a crucial role in deriving ROC curve.

**ROC curve**

* Receiver Operating Characteristic(ROC) summarizes the model’s performance by evaluating the trade offs between true positive rate (sensitivity) and false positive rate(1- specificity).
* The area under curve (AUC), referred to as index of accuracy(A) or concordance index, is a perfect performance metric for ROC curve. Higher the area under curve, better the prediction power of the model.

**Model Building : Part I**

Let us make our first model to predict the target variable. We will start with Logistic Regression which is used for predicting binary outcome.

* Logistic Regression is a classification algorithm. It is used to predict a binary outcome (1 / 0, Yes / No, True / False) given a set of independent variables.
* Logistic regression is an estimation of Logit function. Logit function is simply a log of odds in favor of the event.
* This function creates a s-shaped curve with the probability estimate, which is very similar to the required step wise function

Lets drop the Loan\_ID variable as it do not have any effect on the loan status. We will do the same changes to the test dataset which we did for the training dataset.

train=train.drop('Loan\_ID',axis=1)

test=test.drop('Loan\_ID',axis=1)

We will use scikit-learn (sklearn) for making different models which is an open source library for Python. It is one of the most efficient tool which contains many inbuilt functions that can be used for modeling in Python.

To learn further about sklearn, refer here: <http://scikit-learn.org/stable/tutorial/index.html>

Sklearn requires the target variable in a separate dataset. So, we will drop our target variable from the train dataset and save it in another dataset.

X = train.drop('Loan\_Status',1)

y = train.Loan\_Status

Now we will make dummy variables for the categorical variables. Dummy variable turns categorical variables into a series of 0 and 1, making them lot easier to quantify and compare. Let us understand the process of dummies first:

* Consider the “Gender” variable. It has two classes, Male and Female.
* As logistic regression takes only the numerical values as input, we have to change male and female into numerical value.
* Once we apply dummies to this variable, it will convert the “Gender” variable into two variables(Gender\_Male and Gender\_Female), one for each class, i.e. Male and Female.
* Gender\_Male will have a value of 0 if the gender is Female and a value of 1 if the gender is Male.

X=pd.get\_dummies(X)

train=pd.get\_dummies(train)

test=pd.get\_dummies(test)

Now we will train the model on training dataset and make predictions for the test dataset. But can we validate these predictions? One way of doing this is we can divide our train dataset into two parts: train and validation. We can train the model on this train part and using that make predictions for the validation part. In this way we can validate our predictions as we have the true predictions for the validation part (which we do not have for the test dataset).

We will use the train\_test\_split function from sklearn to divide our train dataset. So, first let us import train\_test\_split.

fromsklearn.model\_selection import train\_test\_split

x\_train, x\_cv, y\_train, y\_cv = train\_test\_split(X,y, test\_size =0.3)

The dataset has been divided into training and validation part. Let us import LogisticRegression and accuracy\_score from sklearn and fit the logistic regression model.

fromsklearn.linear\_model import LogisticRegression

fromsklearn.metrics import accuracy\_score

model = LogisticRegression()

model.fit(x\_train, y\_train)

LogisticRegression(C=1.0, class\_weight=None, dual=False, fit\_intercept=True,

intercept\_scaling=1, max\_iter=100, multi\_class='ovr', n\_jobs=1,

penalty='l2', random\_state=1, solver='liblinear', tol=0.0001,

verbose=0, warm\_start=False)

Here the C parameter represents inverse of regularization strength. Regularization is applying a penalty to increasing the magnitude of parameter values in order to reduce overfitting. Smaller values of C specify stronger regularization. To learn about other parameters, refer here: <http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html>

Let’s predict the Loan\_Status for validation set and calculate its accuracy.

pred\_cv = model.predict(x\_cv)

Let us calculate how accurate our predictions are by calculating the accuracy.

accuracy\_score(y\_cv,pred\_cv)

0.7945945945945946

So our predictions are almost 80% accurate, i.e. we have identified 80% of the loan status correctly.

Let’s make predictions for the test dataset.

pred\_test = model.predict(test)

Remember we need predictions in Y and N. So let’s convert 1 and 0 to Y and N.

**Logistic Regression using stratified k-folds cross validation**

To check how robust our model is to unseen data, we can use Validation. It is a technique which involves reserving a particular sample of a dataset on which you do not train the model. Later, you test your model on this sample before finalizing it. Some of the common methods for validation are listed below:

* The validation set approach
* k-fold cross validation
* Leave one out cross validation (LOOCV)
* Stratified k-fold cross validation

In this section we will learn about **stratified k-fold cross validation**. Let us understand how it works:

* Stratification is the process of rearranging the data so as to ensure that each fold is a good representative of the whole.
* For example, in a binary classification problem where each class comprises of 50% of the data, it is best to arrange the data such that in every fold, each class comprises of about half the instances.
* It is generally a better approach when dealing with both bias and variance.
* A randomly selected fold might not adequately represent the minor class, particularly in cases where there is a huge class imbalance.

Let’s import StratifiedKFold from sklearn and fit the model.

fromsklearn.model\_selection import StratifiedKFold

Now let’s make a cross validation logistic model with stratified 5 folds and make predictions for test dataset.

i=1

kf = StratifiedKFold(n\_splits=5,random\_state=1,shuffle=True)

fortrain\_index,test\_index in kf.split(X,y):

print('\n{} of kfold {}'.format(i,kf.n\_splits))

xtr,xvl = X.loc[train\_index],X.loc[test\_index]

ytr,yvl = y[train\_index],y[test\_index]

model = LogisticRegression(random\_state=1)

model.fit(xtr, ytr)

pred\_test = model.predict(xvl)

score = accuracy\_score(yvl,pred\_test)

print('accuracy\_score',score)

i+=1

pred\_test = model.predict(test)

pred=model.predict\_proba(xvl)[:,1]

1 of kfold 5

accuracy\_score 0.7983870967741935

2 of kfold 5

accuracy\_score 0.8306451612903226

3 of kfold 5

accuracy\_score 0.8114754098360656

4 of kfold 5

accuracy\_score 0.7950819672131147

5 of kfold 5

accuracy\_score 0.8278688524590164

The mean validation accuracy for this model turns out to be 0.81. Let us visualize the roc curve.

fromsklearn import metrics

fpr, tpr, \_ = metrics.roc\_curve(yvl, pred)

auc = metrics.roc\_auc\_score(yvl, pred)

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

plt.plot(fpr,tpr,label="validation, auc="+str(auc))

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.legend(loc=4)

plt.show()

We got an auc value of 0.77.

**Feature Engineering**

Based on the domain knowledge, we can come up with new features that might affect the target variable. We will create the following three new features:

* **Total Income** - As discussed during bivariate analysis we will combine the Applicant Income and Coapplicant Income. If the total income is high, chances of loan approval might also be high.
* **EMI** - EMI is the monthly amount to be paid by the applicant to repay the loan. Idea behind making this variable is that people who have high EMI’s might find it difficult to pay back the loan. We can calculate the EMI by taking the ratio of loan amount with respect to loan amount term.
* **Balance Income** - This is the income left after the EMI has been paid. Idea behind creating this variable is that if this value is high, the chances are high that a person will repay the loan and hence increasing the chances of loan approval.

train['Total\_Income']=train['ApplicantIncome']+train['CoapplicantIncome']

test['Total\_Income']=test['ApplicantIncome']+test['CoapplicantIncome']

Let’s check the distribution of Total Income.

sns.distplot(train['Total\_Income']);

We can see it is shifted towards left, i.e., the distribution is right skewed. So, let’s take the log transformation to make the distribution normal.

train['Total\_Income\_log'] = np.log(train['Total\_Income'])

sns.distplot(train['Total\_Income\_log']);

test['Total\_Income\_log'] = np.log(test['Total\_Income'])

Now the distribution looks much closer to normal and effect of extreme values has been significantly subsided. Let’s create the EMI feature now.

train['EMI']=train['LoanAmount']/train['Loan\_Amount\_Term']

test['EMI']=test['LoanAmount']/test['Loan\_Amount\_Term']

Let’s check the distribution of EMI variable.

sns.distplot(train['EMI']);

Let us create Balance Income feature now and check its distribution.

train['Balance Income']=train['Total\_Income']-(train['EMI']\*1000) # Multiply with 1000 to make the units equal

test['Balance Income']=test['Total\_Income']-(test['EMI']\*1000)

sns.distplot(train['Balance Income']);

Let us now drop the variables which we used to create these new features. Reason for doing this is, the correlation between those old features and these new features will be very high and logistic regression assumes that the variables are not highly correlated. We also wants to remove the noise from the dataset, so removing correlated features will help in reducing the noise too.

train=train.drop(['ApplicantIncome', 'CoapplicantIncome', 'LoanAmount', 'Loan\_Amount\_Term'], axis=1)

test=test.drop(['ApplicantIncome', 'CoapplicantIncome', 'LoanAmount', 'Loan\_Amount\_Term'], axis=1)

**Model Building : Part II**

After creating new features, we can continue the model building process. So we will start with logistic regression model and then move over to more complex models like RandomForest and XGBoost.

We will build the following models in this section.

* Logistic Regression
* Decision Tree
* Random Forest
* XGBoost

Let’s prepare the data for feeding into the models.

X = train.drop('Loan\_Status',1)

y = train.Loan\_Status # Save target variable in separate dataset

**Logistic Regression**

i=1

kf = StratifiedKFold(n\_splits=5,random\_state=1,shuffle=True)

fortrain\_index,test\_index in kf.split(X,y):

print('\n{} of kfold {}'.format(i,kf.n\_splits))

xtr,xvl = X.loc[train\_index],X.loc[test\_index]

ytr,yvl = y[train\_index],y[test\_index]

model = LogisticRegression(random\_state=1)

model.fit(xtr, ytr)

pred\_test = model.predict(xvl)

score = accuracy\_score(yvl,pred\_test)

print('accuracy\_score',score)

i+=1

pred\_test = model.predict(test)

pred=model.predict\_proba(xvl)[:,1]

1 of kfold 5

accuracy\_score 0.7983870967741935

2 of kfold 5

accuracy\_score 0.8306451612903226

3 of kfold 5

accuracy\_score 0.8032786885245902

4 of kfold 5

accuracy\_score 0.7540983606557377

5 of kfold 5

accuracy\_score 0.8278688524590164

The mean validation accuracy for this model is 0.812

So we can infer feature engineering has not improved the model. Let us look at some other algorithms.

**Decision Tree**

Decision tree is a type of supervised learning algorithm(having a pre-defined target variable) that is mostly used in classification problems. In this technique, we split the population or sample into two or more homogeneous sets(or sub-populations) based on most significant splitter / differentiator in input variables.

Decision trees use multiple algorithms to decide to split a node in two or more sub-nodes. The creation of sub-nodes increases the homogeneity of resultant sub-nodes. In other words, we can say that purity of the node increases with respect to the target variable.

For detailed explanation visit <https://www.analyticsvidhya.com/blog/2016/04/complete-tutorial-tree-based-modeling-scratch-in-python/#six>

fromsklearn import tree

Let’s fit the decision tree model with 5 folds of cross validation.

i=1

kf = StratifiedKFold(n\_splits=5,random\_state=1,shuffle=True)

fortrain\_index,test\_index in kf.split(X,y):

print('\n{} of kfold {}'.format(i,kf.n\_splits))

xtr,xvl = X.loc[train\_index],X.loc[test\_index]

ytr,yvl = y[train\_index],y[test\_index]

model = tree.DecisionTreeClassifier(random\_state=1)

model.fit(xtr, ytr)

pred\_test = model.predict(xvl)

score = accuracy\_score(yvl,pred\_test)

print('accuracy\_score',score)

i+=1

pred\_test = model.predict(test)

1 of kfold 5

accuracy\_score 0.6693548387096774

2 of kfold 5

accuracy\_score 0.7661290322580645

3 of kfold 5

accuracy\_score 0.7213114754098361

4 of kfold 5

accuracy\_score 0.6475409836065574

5 of kfold 5

accuracy\_score 0.7131147540983607

We got an accuracy of 0.63 which is much lesser than the accuracy from logistic regression model. So let’s build another model, i.e. Random Forest, a tree based ensemble algorithm and try to improve our model by improving the accuracy.

**Random Forest**

* RandomForest is a tree based bootstrapping algorithm wherein a certain no. of weak learners (decision trees) are combined to make a powerful prediction model.
* For every individual learner, a random sample of rows and a few randomly chosen variables are used to build a decision tree model.
* Final prediction can be a function of all the predictions made by the individual learners.
* In case of regression problem, the final prediction can be mean of all the predictions.

fromsklearn.ensemble import RandomForestClassifier

i=1

kf = StratifiedKFold(n\_splits=5,random\_state=1,shuffle=True)

fortrain\_index,test\_index in kf.split(X,y):

print('\n{} of kfold {}'.format(i,kf.n\_splits))

xtr,xvl = X.loc[train\_index],X.loc[test\_index]

ytr,yvl = y[train\_index],y[test\_index]

model = RandomForestClassifier(random\_state=1, max\_depth=10)

model.fit(xtr, ytr)

pred\_test = model.predict(xvl)

score = accuracy\_score(yvl,pred\_test)

print('accuracy\_score',score)

i+=1

pred\_test = model.predict(test)

1 of kfold 5

accuracy\_score 0.8145161290322581

2 of kfold 5

accuracy\_score 0.8064516129032258

3 of kfold 5

accuracy\_score 0.7540983606557377

4 of kfold 5

accuracy\_score 0.7213114754098361

5 of kfold 5

accuracy\_score 0.819672131147541

The mean validation accuracy for this model is 0.766

We will try to improve the accuracy by tuning the hyperparameters for this model. We will use grid search to get the optimized values of hyper parameters. Grid-search is a way to select the best of a family of hyper parameters, parametrized by a grid of parameters.

We will tune the max\_depth and n\_estimators parameters. max\_depth decides the maximum depth of the tree and n\_estimators decides the number of trees that will be used in random forest model.

fromsklearn.model\_selection import GridSearchCV

# Provide range for max\_depth from 1 to 20 with an interval of 2 and from 1 to 200 with an interval of 20 for n\_estimators

paramgrid = {'max\_depth': list(range(1, 20, 2)), 'n\_estimators': list(range(1, 200, 20))}

grid\_search=GridSearchCV(RandomForestClassifier(random\_state=1),paramgrid)

fromsklearn.model\_selection import train\_test\_split

x\_train, x\_cv, y\_train, y\_cv = train\_test\_split(X,y, test\_size =0.3, random\_state=1)

# Fit the grid search model

grid\_search.fit(x\_train,y\_train)

*GridSearchCV(cv=None, error\_score='raise',*

*estimator=RandomForestClassifier(bootstrap=True, class\_weight=None, criterion='gini',*

*max\_depth=None, max\_features='auto', max\_leaf\_nodes=None,*

*min\_impurity\_decrease=0.0, min\_impurity\_split=None,*

*min\_samples\_leaf=1, min\_samples\_split=2,*

*min\_weight\_fraction\_leaf=0.0, n\_estimators=10, n\_jobs=1,*

*oob\_score=False, random\_state=1, verbose=0, warm\_start=False),*

*fit\_params=None, iid=True, n\_jobs=1,*

*param\_grid={'max\_depth': [1, 3, 5, 7, 9, 11, 13, 15, 17, 19], 'n\_estimators': [1, 21, 41, 61, 81, 101, 121, 141, 161, 181]},*

*pre\_dispatch='2\*n\_jobs', refit=True, return\_train\_score='warn',*

*scoring=None, verbose=0)*

# Estimating the optimized value

grid\_search.best\_estimator\_

*RandomForestClassifier(bootstrap=True, class\_weight=None, criterion='gini',*

*max\_depth=3, max\_features='auto', max\_leaf\_nodes=None,*

*min\_impurity\_decrease=0.0, min\_impurity\_split=None,*

*min\_samples\_leaf=1, min\_samples\_split=2,*

*min\_weight\_fraction\_leaf=0.0, n\_estimators=41, n\_jobs=1,*

*oob\_score=False, random\_state=1, verbose=0, warm\_start=False)*

So, the optimized value for the max\_depth variable is 3 and for n\_estimator is 41. Now let’s build the model using these optimized values.

i=1

kf = StratifiedKFold(n\_splits=5,random\_state=1,shuffle=True)

fortrain\_index,test\_index in kf.split(X,y):

print('\n{} of kfold {}'.format(i,kf.n\_splits))

xtr,xvl = X.loc[train\_index],X.loc[test\_index]

ytr,yvl = y[train\_index],y[test\_index]

model = RandomForestClassifier(random\_state=1, max\_depth=3, n\_estimators=41)

model.fit(xtr, ytr)

pred\_test = model.predict(xvl)

score = accuracy\_score(yvl,pred\_test)

print('accuracy\_score',score)

i+=1

pred\_test = model.predict(test)

pred2=model.predict\_proba(test)[:,1]

1 of kfold 5

accuracy\_score 0.7983870967741935

2 of kfold 5

accuracy\_score 0.8225806451612904

3 of kfold 5

accuracy\_score 0.8032786885245902

4 of kfold 5

accuracy\_score 0.7622950819672131

5 of kfold 5

accuracy\_score 0.819672131147541

We got an accuracy of 0.7638 from the random forest model on leaderboard.

Let us find the feature importance now, i.e. which features are most important for this problem. We will use feature\_importances\_ attribute of sklearn to do so.

importances=pd.Series(model.feature\_importances\_, index=X.columns)

importances.plot(kind='barh', figsize=(12,8))

We can see that Credit\_History is the most important feature followed by Balance Income, Total Income, EMI. So, feature engineering helped us in predicting our target variable.

**XGBoost**

XGBoost is a fast and efficient algorithm and has been used to by the winners of many data science competitions.

XGBoost works only with numeric variables and we have already replaced the categorical variables with numeric variables. Let’s have a look at the parameters that we are going to use in our model.

* n\_estimator: This specifies the number of trees for the model.
* max\_depth: We can specify maximum depth of a tree using this parameter.

fromxgboost import XGBClassifier

i=1

kf = StratifiedKFold(n\_splits=5,random\_state=1,shuffle=True)

fortrain\_index,test\_index in kf.split(X,y):

print('\n{} of kfold {}'.format(i,kf.n\_splits))

xtr,xvl = X.loc[train\_index],X.loc[test\_index]

ytr,yvl = y[train\_index],y[test\_index]

model = XGBClassifier(n\_estimators=50, max\_depth=4)

model.fit(xtr, ytr)

pred\_test = model.predict(xvl)

score = accuracy\_score(yvl,pred\_test)

print('accuracy\_score',score)

i+=1

pred\_test = model.predict(test)

pred3=model.predict\_proba(test)[:,1]

1 of kfold 5

accuracy\_score 0.782258064516129

2 of kfold 5

accuracy\_score 0.8145161290322581

3 of kfold 5

accuracy\_score 0.7704918032786885

4 of kfold 5

accuracy\_score 0.7704918032786885

5 of kfold 5

accuracy\_score 0.7950819672131147

The mean validation accuracy for this model is 0.79

After trying and testing 4 different algorithms, the best accuracy on the public leaderboard is achieved by Logistic Regression (0.7847), followed by RandomForest (0.7638).