## Problem Statement

A 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. Here they have provided a partial data set.

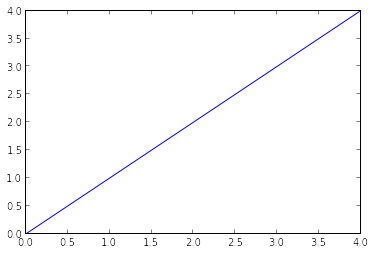
## Data

|  |  |
| --- | --- |
| **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) |

**Let’s begin with exploration**

This opens up iPython notebook in pylab environment, which has a few useful libraries already imported. Also, you will be able to plot your data inline, which makes this a really good environment for interactive data analysis. You can check whether the environment has loaded correctly, by typing the following command (and getting the output as seen in the figure below):

plot(arange(5))



**Importing libraries and the data set:**

Following are the libraries :

* numpy
* matplotlib
* pandas

Please note that you do not need to import matplotlib and numpy because of Pylab environment. I have still kept them in the code, in case you use the code in a different environment.

After importing the library, you read the dataset using function read\_csv(). This is how the code looks like till this stage:

import pandas as pd

import numpy as np

import matplotlib as plt

import seaborn as sns

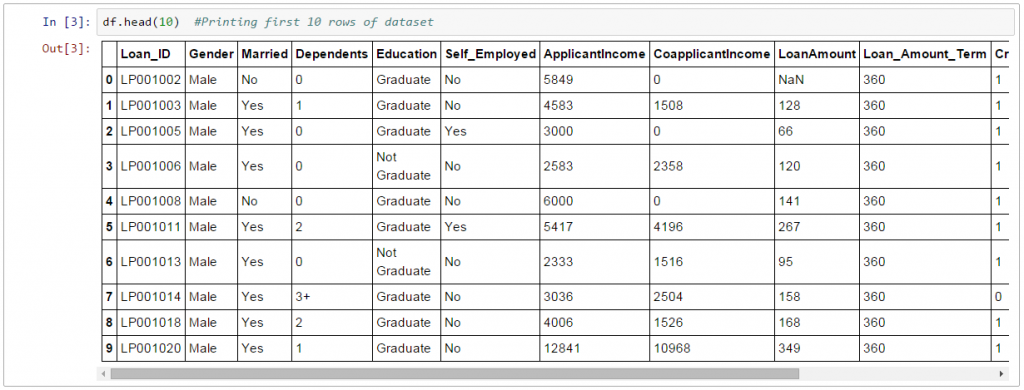
%matplotlib inline

df = pd.read\_csv("/home/nielit/Downloads/Loan\_Prediction/train.csv") #Reading the dataset in a dataframe using Pandas

**Quick Data Exploration**

Once you have read the dataset, you can have a look at few top rows by using the function head()

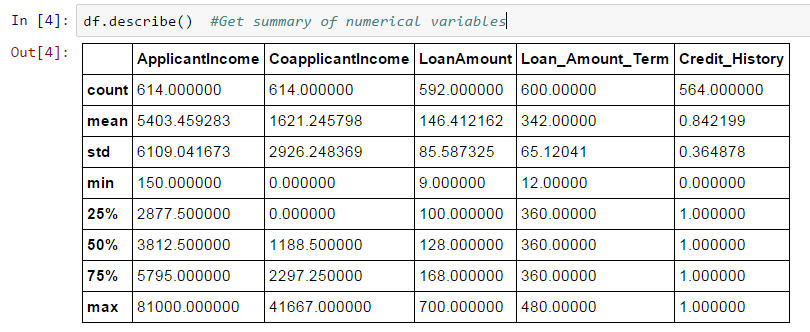
df.head(10)



This should print 10 rows. Alternately, you can also look at more rows by printing the dataset.

Next, you can look at summary of numerical fields by using describe() function

df.describe()



describe() function would provide count, mean, standard deviation (std), min, quartiles and max in its output.

Here are a few inferences, you can draw by looking at the output of describe() function:

1. LoanAmount has (614 – 592) 22 missing values.
2. Loan\_Amount\_Term has (614 – 600) 14 missing values.
3. Credit\_History has (614 – 564) 50 missing values.
4. We can also look that about 84% applicants have a credit\_history. How? The mean of Credit\_History field is 0.84 (Remember, Credit\_History has value 1 for those who have a credit history and 0 otherwise)
5. The ApplicantIncome distribution seems to be in line with expectation. Same with CoapplicantIncome

Please note that we can get an idea of a possible skew in the data by comparing the mean to the median, i.e. the 50% figure.

For the non-numerical values (e.g. Property\_Area, Credit\_History etc.), we can look at frequency distribution to understand whether they make sense or not. The frequency table can be printed by following command:

df['Property\_Area'].value\_counts()

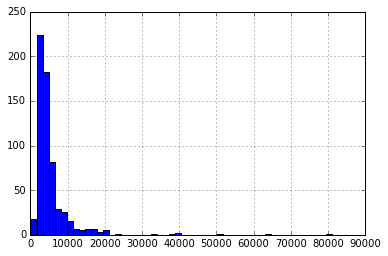
Similarly, we can look at unique values of port of credit history. Note that dfname[‘column\_name’] is a basic indexing technique to acess a particular column of the dataframe. It can be a list of columns as well. For more information, refer to the “10 Minutes to Pandas” resource shared above.

**Distribution analysis**

Now that we are familiar with basic data characteristics, let us study distribution of various variables. Let us start with numeric variables – namely ApplicantIncome and LoanAmount

Lets start by plotting the histogram of ApplicantIncome using the following commands:

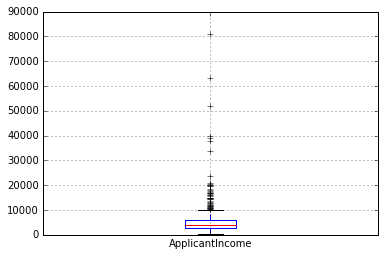
df['ApplicantIncome'].hist(bins=50)



Here we observe that there are few extreme values. This is also the reason why 50 bins are required to depict the distribution clearly.

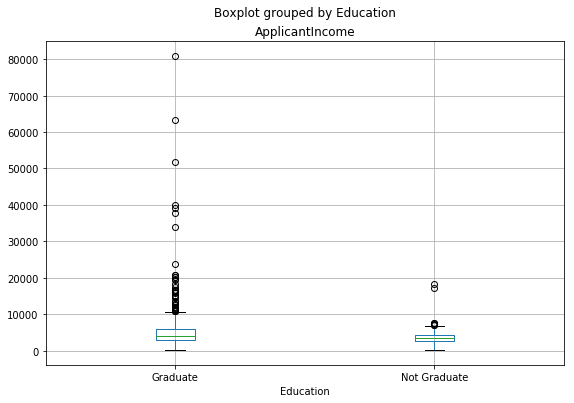
Next, we look at box plots to understand the distributions. Box plot for fare can be plotted by:

df.boxplot(column='ApplicantIncome')



This 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:

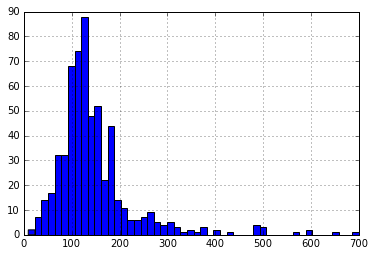
df.boxplot(column='ApplicantIncome', by = 'Education')



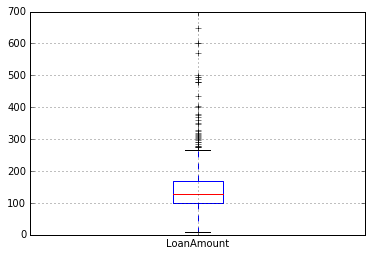
We can see that there is no substantial different between the mean income of graduate and non-graduates. But there are a higher number of graduates with very high incomes, which are appearing to be the outliers.

Now, Let’s look at the histogram and boxplot of LoanAmount..

df['LoanAmount'].hist(bins=50)



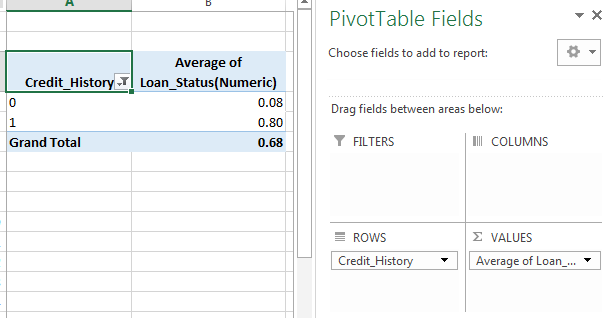
df.boxplot(column='LoanAmount')



Again, there are some extreme values. Clearly, both ApplicantIncome and LoanAmount require some amount of data munging. LoanAmount has missing and well as extreme values values, while ApplicantIncome has a few extreme values, which demand deeper understanding. We will take this up in coming sections.

**Categorical variable analysis**

Now that we understand distributions for ApplicantIncome and LoanIncome, let us understand categorical variables in more details. We will use Excel style pivot table and cross-tabulation. For instance, let us look at the chances of getting a loan based on credit history. This can be achieved in MS Excel using a pivot table as:



Note: here loan status has been coded as 1 for Yes and 0 for No. So the mean represents the probability of getting loan.

Now we will look at the steps required to generate a similar insight using Python.

temp1 = df['Credit\_History'].value\_counts(ascending=True)

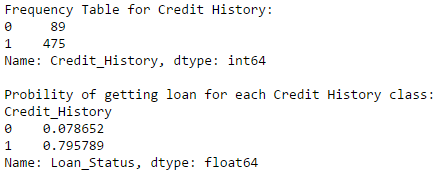
temp2 = df.pivot\_table(values='Loan\_Status',index=['Credit\_History'],aggfunc=lambda x: x.map({'Y':1,'N':0}).mean())

print ('Frequency Table for Credit History:')

print (temp1)

print ('\nProbility of getting loan for each Credit History class:')

print (temp2)



Now we can observe that we get a similar pivot\_table like the MS Excel one. This can be plotted as a bar chart using the “matplotlib” library with following code:

import matplotlib.pyplot as plt

fig = plt.figure(figsize=(8,4))

ax1 = fig.add\_subplot(121)

ax1.set\_xlabel('Credit\_History')

ax1.set\_ylabel('Count of Applicants')

ax1.set\_title("Applicants by Credit\_History")

temp1.plot(kind='bar')

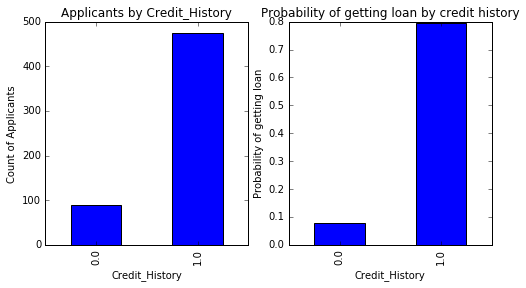
ax2 = fig.add\_subplot(122)

temp2.plot(kind = 'bar')

ax2.set\_xlabel('Credit\_History')

ax2.set\_ylabel('Probability of getting loan')

ax2.set\_title("Probability of getting loan by credit history")

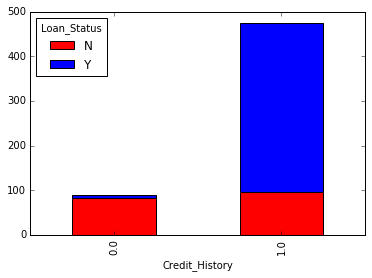


This shows that the chances of getting a loan are eight-fold if the applicant has a valid credit history. You can plot similar graphs by Married, Self-Employed, Property\_Area, etc.

Alternately, these two plots can also be visualized by combining them in a stacked chart::

temp3 = pd.crosstab(df['Credit\_History'], df['Loan\_Status'])

temp3.plot(kind='bar', stacked=True, color=['red','blue'], grid=False)



Next let’s explore ApplicantIncome and LoanStatus variables further, [perform data munging](https://www.analyticsvidhya.com/blog/2014/09/data-munging-python-using-pandas-baby-steps-python/) and create a dataset for applying various modeling techniques.

**4. Data Munging in Python : Using Pandas**

For those, who have been following, here are your must wear shoes to start running.

**Data munging – recap of the need**

While our exploration of the data, we found a few problems in the data set, which needs to be solved before the data is ready for a good model. This exercise is typically referred as “Data Munging”. Here are the problems, we are already aware of:

1. There are missing values in some variables. We should estimate those values wisely depending on the amount of missing values and the expected importance of variables.
2. While looking at the distributions, we saw that ApplicantIncome and LoanAmount seemed to contain extreme values at either end. Though they might make intuitive sense, but should be treated appropriately.

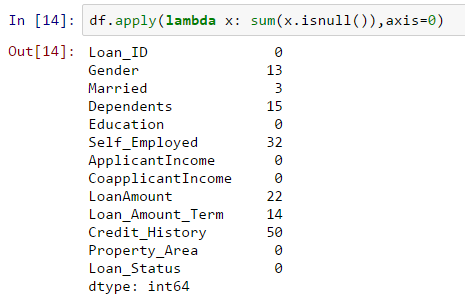
In addition to these problems with numerical fields, we should also look at the non-numerical fields i.e. Gender, Property\_Area, Married, Education and Dependents to see, if they contain any useful information.

**Check missing values in the dataset**

Let us look at missing values in all the variables because most of the models don’t work with missing data and even if they do, imputing them helps more often than not. So, let us check the number of nulls / NaNs in the dataset

df.apply(lambda x: sum(x.isnull()),axis=0)

This command should tell us the number of missing values in each column as isnull() returns 1, if the value is null.



Though the missing values are not very high in number, but many variables have them and each one of these should be estimated and added in the data.

Note: Remember that missing values may not always be NaNs. For instance, if the Loan\_Amount\_Term is 0, does it makes sense or would you consider that missing? I suppose your answer is missing and you’re right. So we should check for values which are unpractical.

**How to fill missing values in LoanAmount?**

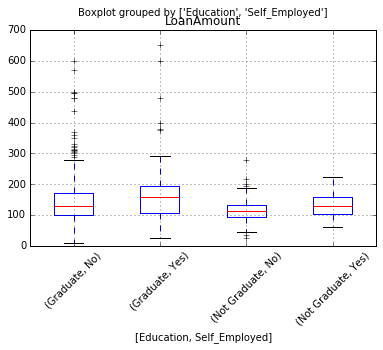
There are numerous ways to fill the missing values of loan amount – the simplest being replacement by mean, which can be done by following code:

df['LoanAmount'].fillna(df['LoanAmount'].mean(), inplace=True)

The other extreme could be to build a supervised learning model to predict loan amount on the basis of other variables and then use age along with other variables to predict survival.

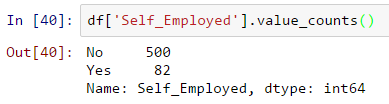
Since, the purpose now is to bring out the steps in data munging, I’ll rather take an approach, which lies some where in between these 2 extremes. A key hypothesis is that the whether a person is educated or self-employed can combine to give a good estimate of loan amount.

First, let’s look at the boxplot to see if a trend exists:



Thus we see some variations in the median of loan amount for each group and this can be used to impute the values. But first, we have to ensure that each of Self\_Employed and Education variables should not have a missing values.

As we say earlier, Self\_Employed has some missing values. Let’s look at the frequency table:



Since ~86% values are “No”, it is safe to impute the missing values as “No” as there is a high probability of success. This can be done using the following code:

df['Self\_Employed'].fillna('No',inplace=True)

Now, we will create a Pivot table, which provides us median values for all the groups of unique values of Self\_Employed and Education features. Next, we define a function, which returns the values of these cells and apply it to fill the missing values of loan amount:

table = df.pivot\_table(values='LoanAmount', index='Self\_Employed' ,columns='Education', aggfunc=np.median)

# Define function to return value of this pivot\_table

def fage(x):

return table.loc[x['Self\_Employed'],x['Education']]

# Replace missing values

df['LoanAmount'].fillna(df[df['LoanAmount'].isnull()].apply(fage, axis=1), inplace=True)

This should provide you a good way to impute missing values of loan amount.

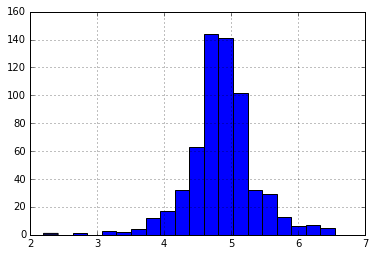
NOTE : This method will work only if you have not filled the missing values in Loan\_Amount variable using the previous approach, i.e. using mean.

**How to treat for extreme values in distribution of LoanAmount and ApplicantIncome?**

Let’s analyze LoanAmount first. Since the extreme values are practically possible, i.e. some people might apply for high value loans due to specific needs. So instead of treating them as outliers, let’s try a log transformation to nullify their effect:

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

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

Looking at the histogram again:  


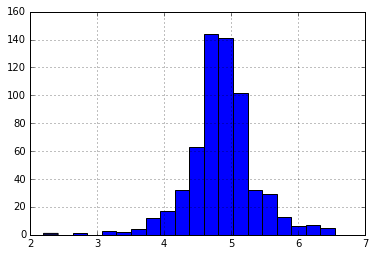
Now the distribution looks much closer to normal and effect of extreme values has been significantly subsided.

Coming to ApplicantIncome. One intuition can be that some applicants have lower income but strong support Co-applicants. So it might be a good idea to combine both incomes as total income and take a log transformation of the same.

df['TotalIncome'] = df['ApplicantIncome'] + df['CoapplicantIncome']

df['TotalIncome\_log'] = np.log(df['TotalIncome'])

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



Now we see that the distribution is much better than before. I will leave it upto you to impute the missing values for Gender, Married, Dependents, Loan\_Amount\_Term, Credit\_History. Also, I encourage you to think about possible additional information which can be derived from the data. For example, creating a column for LoanAmount/TotalIncome might make sense as it gives an idea of how well the applicant is suited to pay back his loan.

Next, we will look at making predictive models.

**5. Building a Predictive Model in Python**

After, we have made the data useful for modeling, let’s now look at the python code to create a predictive model on our data set. Skicit-Learn (sklearn) is the most commonly used library in Python for this purpose and we will follow the trail.

Since, sklearn requires all inputs to be numeric, we should convert all our categorical variables into numeric by encoding the categories. Before that we will fill all the missing values in the dataset. This can be done using the following code:

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

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

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

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

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

from sklearn.preprocessing import LabelEncoder

var\_mod = ['Gender','Married','Dependents','Education','Self\_Employed','Property\_Area','Loan\_Status']

le = LabelEncoder()

for i in var\_mod:

df[i] = le.fit\_transform(df[i])

df.dtypes

Next, we will import the required modules. Then we will define a generic classification function, which takes a model as input and determines the Accuracy and Cross-Validation scores.

#Import models from scikit learn module:

from sklearn.linear\_model import LogisticRegression

from sklearn.cross\_validation import KFold #For K-fold cross validation

from sklearn.ensemble import RandomForestClassifier

from sklearn.tree import DecisionTreeClassifier, export\_graphviz

from sklearn import metrics

#Generic function for making a classification model and accessing performance:

def classification\_model(model, data, predictors, outcome):

#Fit the model:

model.fit(data[predictors],data[outcome])

#Make predictions on training set:

predictions = model.predict(data[predictors])

#Print accuracy

accuracy = metrics.accuracy\_score(predictions,data[outcome])

print ("Accuracy : %s" % "{0:.3%}".format(accuracy))

#Perform k-fold cross-validation with 5 folds

kf = KFold(data.shape[0], n\_folds=5)

error = []

for train, test in kf:

# Filter training data

train\_predictors = (data[predictors].iloc[train,:])

# The target we're using to train the algorithm.

train\_target = data[outcome].iloc[train]

# Training the algorithm using the predictors and target.

   model.fit(train\_predictors, train\_target)

#Record error from each cross-validation run

   error.append(model.score(data[predictors].iloc[test,:], data[outcome].iloc[test]))

print ("Cross-Validation Score : %s" % "{0:.3%}".format(np.mean(error)))

#Fit the model again so that it can be refered outside the function:

model.fit(data[predictors],data[outcome])

**Logistic Regression**

Let’s make our first Logistic Regression model. One way would be to take all the variables into the model but this might result in overfitting (don’t worry if you’re unaware of this terminology yet). In simple words, taking all variables might result in the model understanding complex relations specific to the data and will not generalize well. Read more about [Logistic Regression](https://www.analyticsvidhya.com/blog/2015/11/beginners-guide-on-logistic-regression-in-r/).

We can easily make some intuitive hypothesis to set the ball rolling. The chances of getting a loan will be higher for:

1. Applicants having a credit history (remember we observed this in exploration?)
2. Applicants with higher applicant and co-applicant incomes
3. Applicants with higher education level
4. Properties in urban areas with high growth perspectives

So let’s make our first model with ‘Credit\_History’.

outcome\_var = 'Loan\_Status'

model = LogisticRegression()

predictor\_var = ['Credit\_History']

classification\_model(model, df,predictor\_var,outcome\_var)

Accuracy : 80.945% Cross-Validation Score : 80.946%

#We can try different combination of variables:

predictor\_var = ['Credit\_History','Education','Married','Self\_Employed','Property\_Area']

classification\_model(model, df,predictor\_var,outcome\_var)

Accuracy : 80.945% Cross-Validation Score : 80.946%

Generally we expect the accuracy to increase on adding variables. But this is a more challenging case. The accuracy and cross-validation score are not getting impacted by less important variables. Credit\_History is dominating the mode. We have two options now:

1. Feature Engineering: dereive new information and try to predict those. I will leave this to your creativity.
2. Better modeling techniques. Let’s explore this next.

**Decision Tree**

Decision tree is another method for making a predictive model. It is known to provide higher accuracy than logistic regression model. Read more about [Decision Trees](https://www.analyticsvidhya.com/blog/2015/01/decision-tree-simplified/).

model = DecisionTreeClassifier()

predictor\_var = ['Credit\_History','Gender','Married','Education']

classification\_model(model, df,predictor\_var,outcome\_var)

Accuracy : 81.930% Cross-Validation Score : 76.656%

Here the model based on categorical variables is unable to have an impact because Credit History is dominating over them. Let’s try a few numerical variables:

#We can try different combination of variables:

predictor\_var = ['Credit\_History','Loan\_Amount\_Term','LoanAmount\_log']

classification\_model(model, df,predictor\_var,outcome\_var)

Accuracy : 92.345% Cross-Validation Score : 71.009%

Here we observed that although the accuracy went up on adding variables, the cross-validation error went down. This is the result of model over-fitting the data. Let’s try an even more sophisticated algorithm and see if it helps:

**Random Forest**

Random forest is another algorithm for solving the classification problem.

An advantage with Random Forest is that we can make it work with all the features and it returns a feature importance matrix which can be used to select features.

model = RandomForestClassifier(n\_estimators=100)

predictor\_var = ['Gender', 'Married', 'Dependents', 'Education',

'Self\_Employed', 'Loan\_Amount\_Term', 'Credit\_History', 'Property\_Area',

'LoanAmount\_log','TotalIncome\_log']

classification\_model(model, df,predictor\_var,outcome\_var)

Accuracy : 100.000% Cross-Validation Score : 78.179%

Here we see that the accuracy is 100% for the training set. This is the ultimate case of overfitting and can be resolved in two ways:

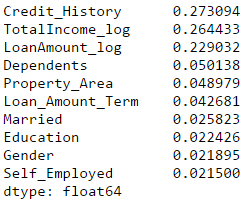
1. Reducing the number of predictors
2. Tuning the model parameters

Let’s try both of these. First we see the feature importance matrix from which we’ll take the most important features.

#Create a series with feature importances:

featimp = pd.Series(model.feature\_importances\_, index=predictor\_var).sort\_values(ascending=False)

print (featimp)



Let’s use the top 5 variables for creating a model. Also, we will modify the parameters of random forest model a little bit:

model = RandomForestClassifier(n\_estimators=25, min\_samples\_split=25, max\_depth=7, max\_features=1)

predictor\_var = ['TotalIncome\_log','LoanAmount\_log','Credit\_History','Dependents','Property\_Area']

classification\_model(model, df,predictor\_var,outcome\_var)

Accuracy : 82.899% Cross-Validation Score : 81.461%

Notice that although accuracy reduced, but the cross-validation score is improving showing that the model is generalizing well. Remember that random forest models are not exactly repeatable. Different runs will result in slight variations because of randomization. But the output should stay in the ballpark.

You would have noticed that even after some basic parameter tuning on random forest, we have reached a cross-validation accuracy only slightly better than the original logistic regression model. This exercise gives us some very interesting and unique learning:

1. Using a more sophisticated model does not guarantee better results.
2. Avoid using complex modeling techniques as a black box without understanding the underlying concepts. Doing so would increase the tendency of overfitting thus making your models less interpretable
3. [Feature Engineering](https://www.analyticsvidhya.com/blog/2015/03/feature-engineering-variable-transformation-creation/) is the key to success. Everyone can use an Xgboost models but the real art and creativity lies in enhancing your features to better suit the model.