Introduction

Company want to automate the loan eligibility process based on customer details like Marital Status, Job Status and Year at present employment etc.

The main objective of the project is to design a new predictive model (using various machine learning algorithms/methods) in the existing Loan Origination System that automates and manages the company's credit assessment functions for personal loans through instant credit decision.

Model Objectives

- Identify the factors that lead to default.
- Build a model that predicts customer's probability to default and based on the outcome take the instant decision whether the loan could be given to a new customer (who have applied).

Exploratory data analysis in Python using Panda

Panda's is one of the most useful data analysis libraries in Python. They have been instrumental in increasing the use of Python in data science community. I will now use Pandas to read a data set from and Loan Data Analysis, perform exploratory analysis.

Importing libraries and the data set:

Following are the libraries I will use during this analysis:

- Numpy
- Matplotlib
- Pandas

NumPy: It stands for Numerical Python. The most powerful feature of NumPy is n-dimensional array. This library also contains basic linear algebra functions, Fourier transforms,advanced random number capabilities and tools for integration with other low level languages like Fortran, C and C++.

Matplotlib: It use for plotting vast variety of graphs, starting from histograms to line plots to heat plots.

Pandas : It use for structured data operations and manipulations. It is extensively used for data mugging and preparation. Pandas were added relatively recently to Python and have been instrumental in boosting Python's usage in data scientist community.

After importing the library, you read the dataset using function read_csv().

Data Exploration

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

Data.head( )

Printing the first 10 rows of the data set.

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

You can look at summary of numerical fields by using describe () function.

Data.describe( )

Get summary of the numerical variable.

```
data.describe()
```

	Credit_Amount	Duration_in_Months	Current_Address_Yrs	Age	Num_Credits	Num_Dependents	Default_On_Payment
count	5000.000000	5000.000000	5000.000000	5000.000000	5000.000000	5000.000000	5000.000000
mean	3271.258000	20.903000	2.845000	35.550800	1.407000	1.155000	0.299000
std	2821.607329	12.053989	1.103276	11.386718	0.577423	0.361941	0.457866
min	250.000000	4.000000	1.000000	19.000000	1.000000	1.000000	0.000000
25%	1365.500000	12.000000	2.000000	27.000000	1.000000	1.000000	0.000000
50%	2319.500000	18.000000	3.000000	33.000000	1.000000	1.000000	0.000000
75%	3972.250000	24.000000	4.000000	42.000000	2.000000	1.000000	1.000000
max	18424.000000	72.000000	4.000000	85.000000	4.000000	2.000000	1.000000

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

Feature Scaling

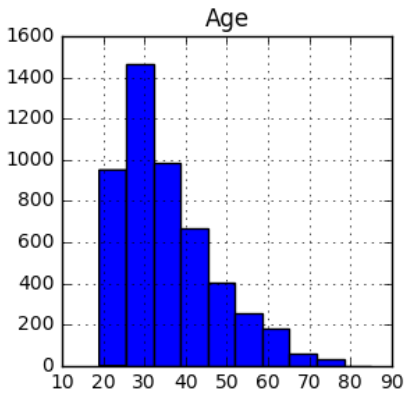
Feature scaling is the method to limit the range of variables so that they can be compared on common grounds. It is performed on continuous variables. Let’s plot the distribution of all the continuous variables in the data set.

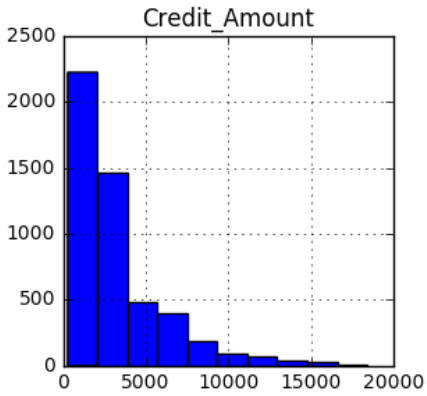
Continuous Variable

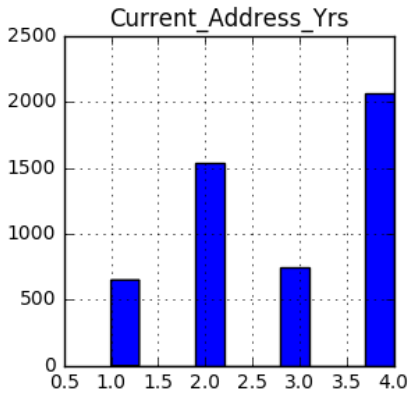
A variable is a quantity that has a changing value; the value can vary from one example to the next. A continuous variable is a variable that has an infinite **number of possible values**. In other words, any value is possible for the variable.

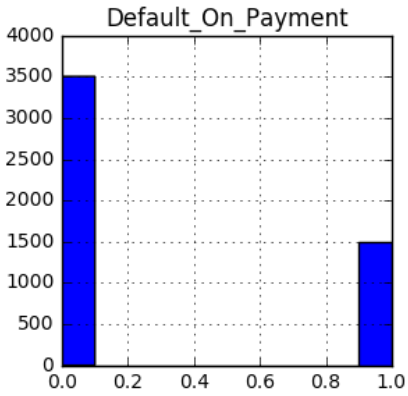
Example of Continius variable is.

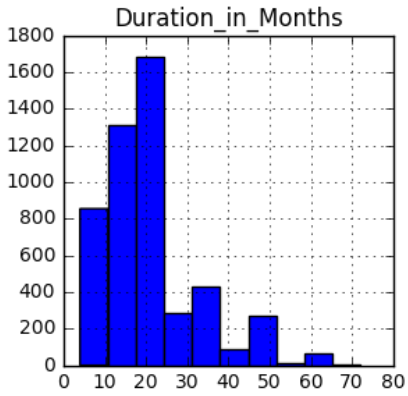
```
import matplotlib.pyplot as plt

data[data.dtypes[(data.dtypes=="float64")|
(data.dtypes=="int64")].index.values].hist(figsize=[11,11])
```

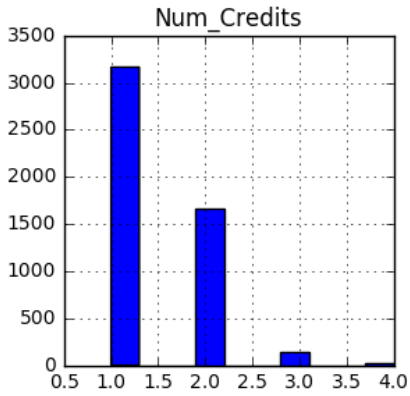


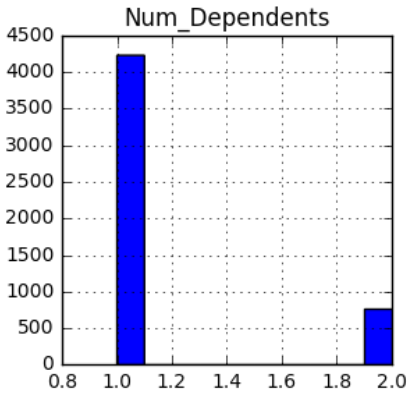








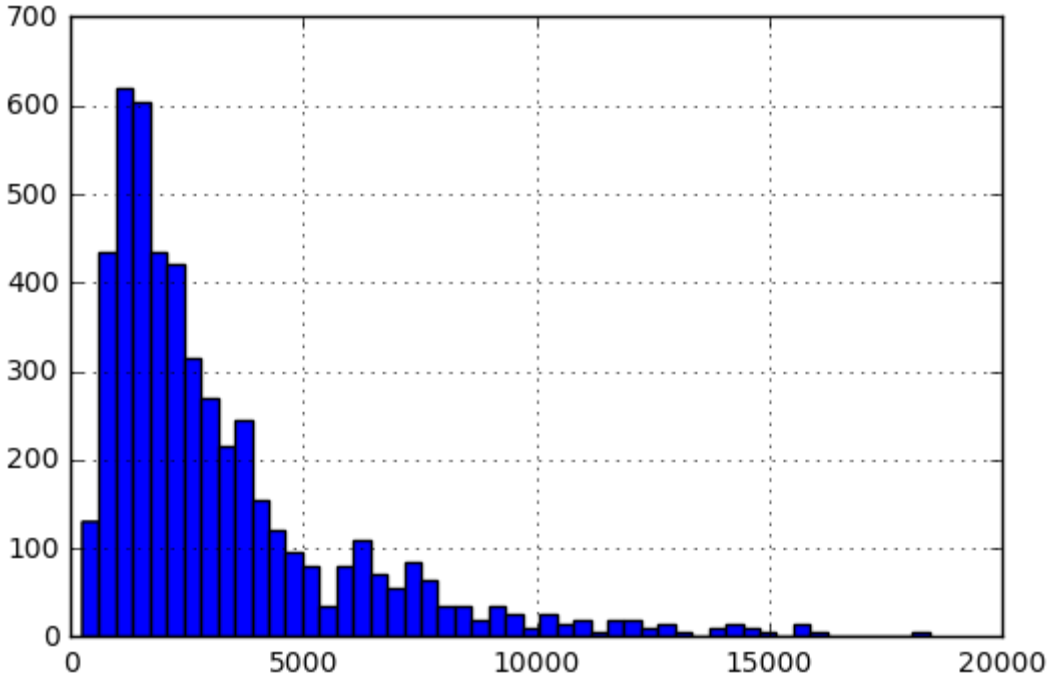




Distribution analysis

Distribution of various variables. Let's start with numeric variables – credit_Amount.
Plotting the histogram of Applicant Income using the following commands

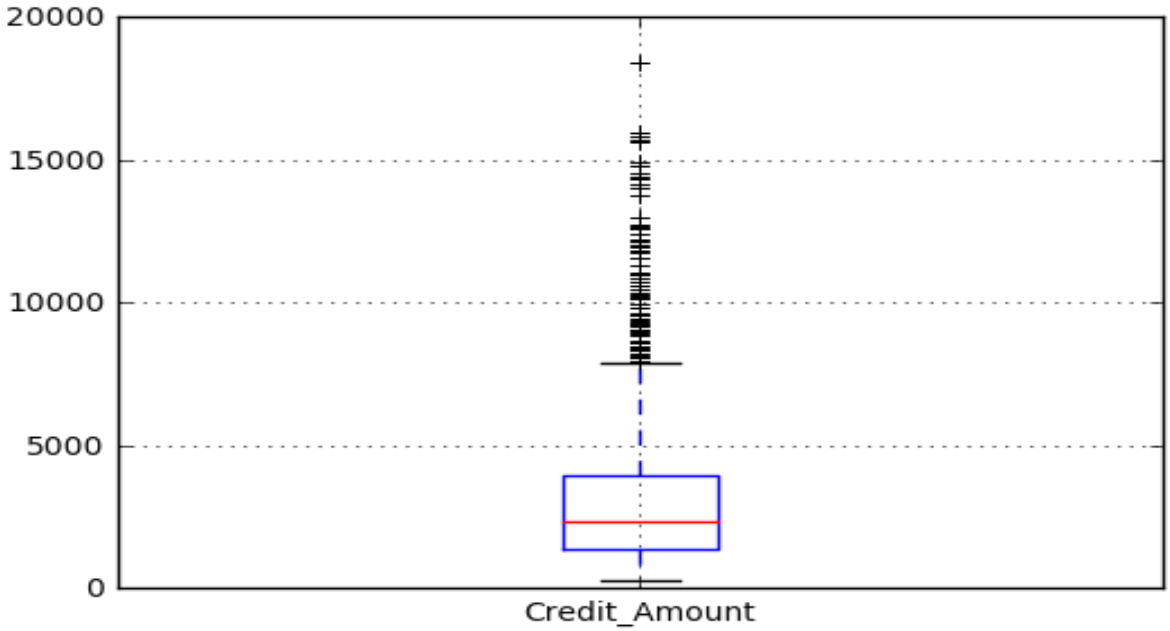
Data ['Credit_Amount'].hist(bins=50)



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

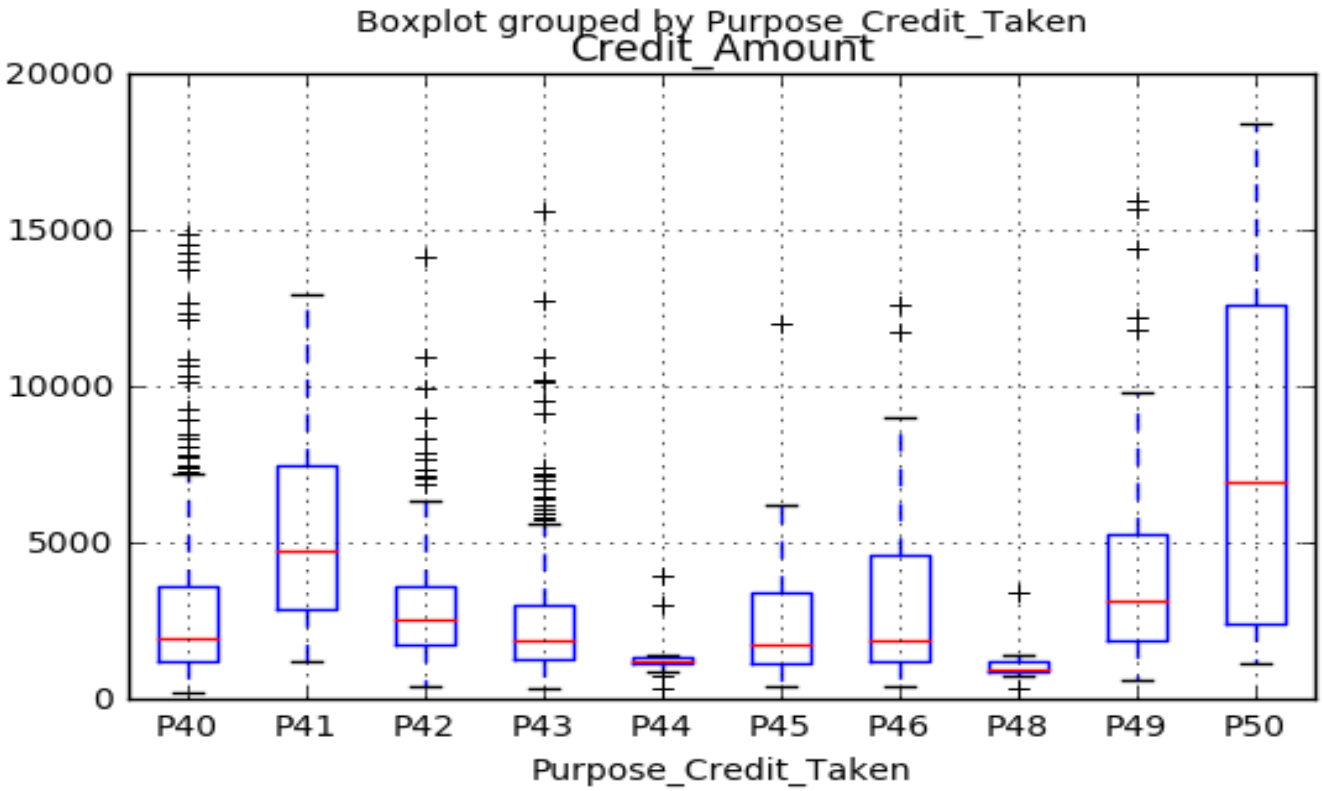
Boxplot

Next, we look at box plots to understand the distributions. Box plot for fare can be plotted by
data.boxplot(column='Credit_Amount')

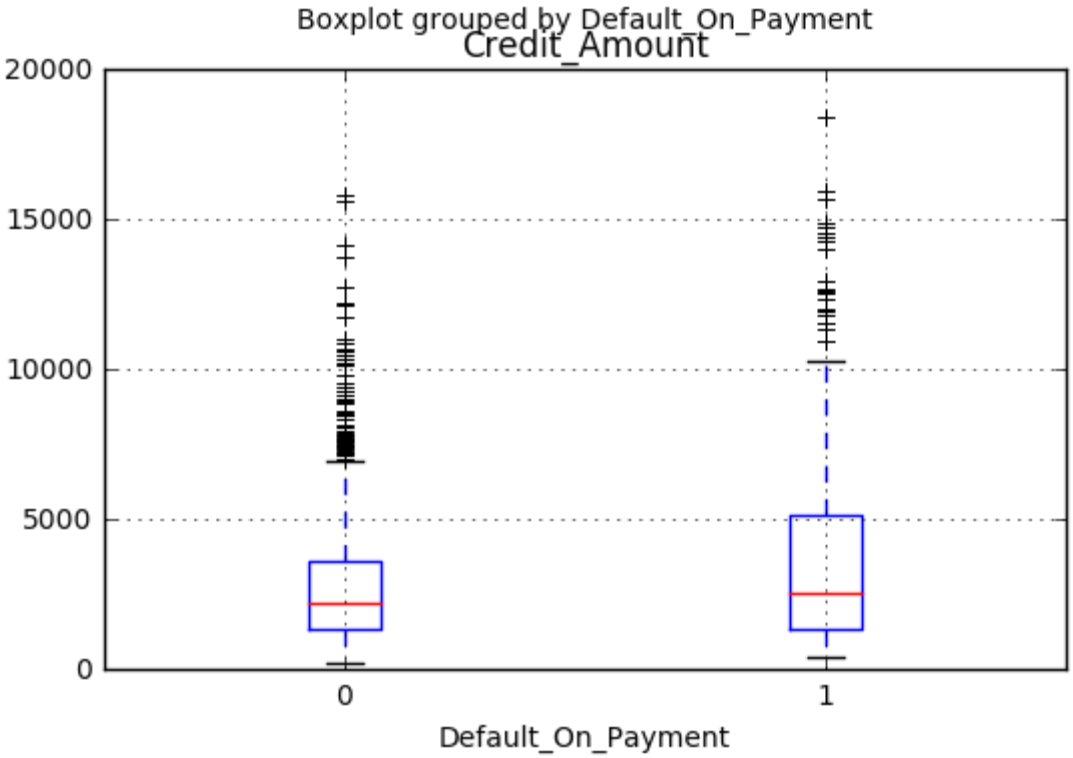


This confirms the presence of a lot of outliers/extreme values. This can be attributed to the loan disparity in the society. Part of this can be driven by the fact that we are looking at people with different purpose loan taken levels. Let us segregate them by Education:

data.boxplot(column='Credit_Amount', by = 'Purpose_Credit_Taken')



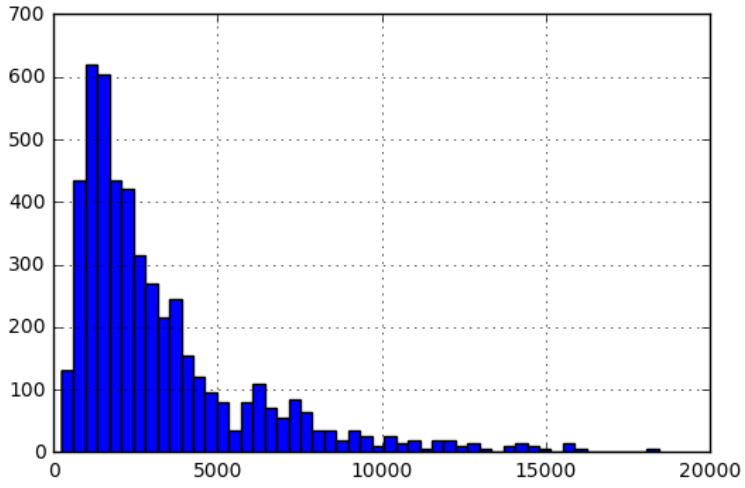
Here we can see that credit taken for Business, Vacation and others are very high which are appearing to be outliers.
Taking some other variable to explore more .
data.boxplot(column='Credit_Amount', by = 'Default_On_Payment')



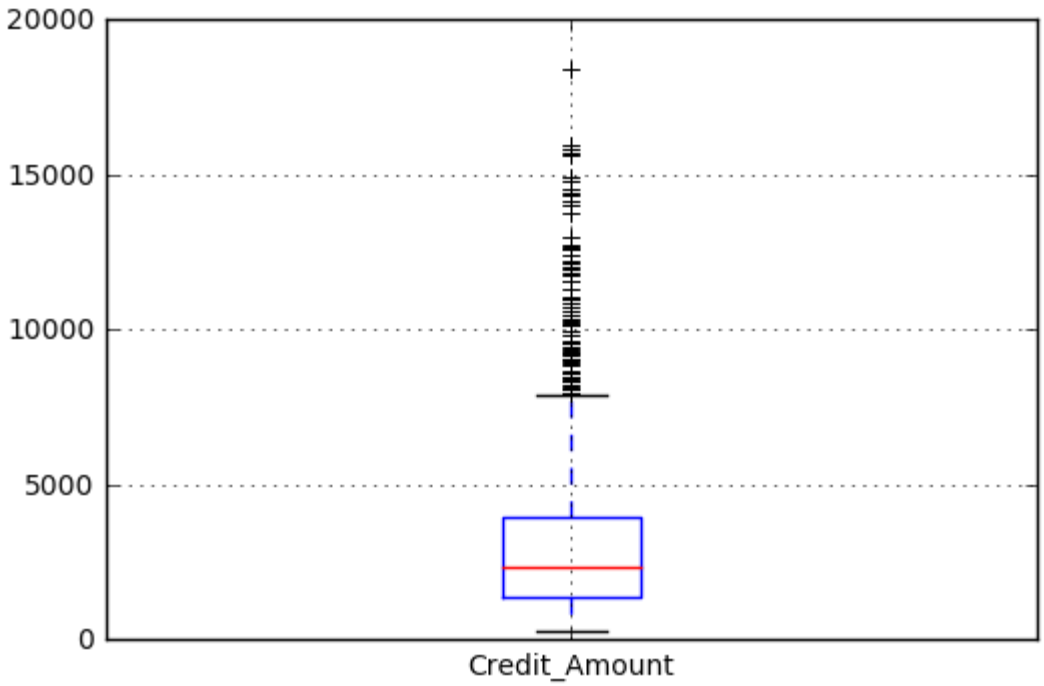
Now here we can see that numbers of credit taken defaulted are very high more than the no default.
Number of Defaulted which are appearing to be outliers.

Now let’s look at the histogram and boxplot of Credit Amount using the following command.

data['Credit_Amount'].hist(bins=50)



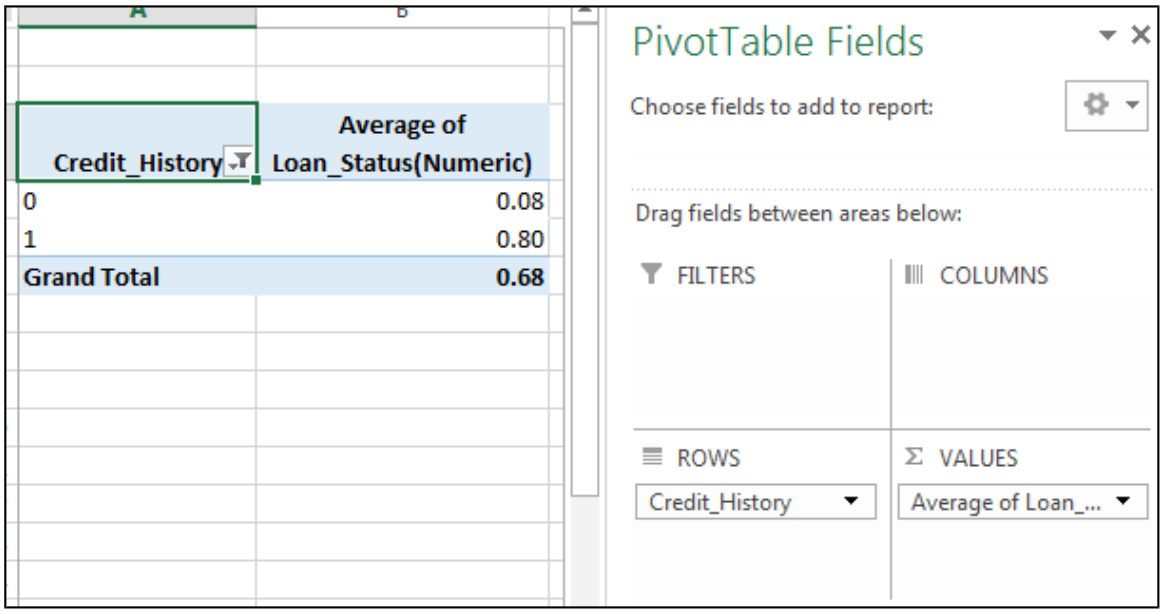
data.boxplot(column='Credit_Amount')



Again, there are some extreme values. Clearly, Loan Amount require some amount of data mugging. We will take this up in coming sections.

Categorical variable analysis

Let's 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:



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

temp1 = data['Credit_History'].value_counts(ascending=True)

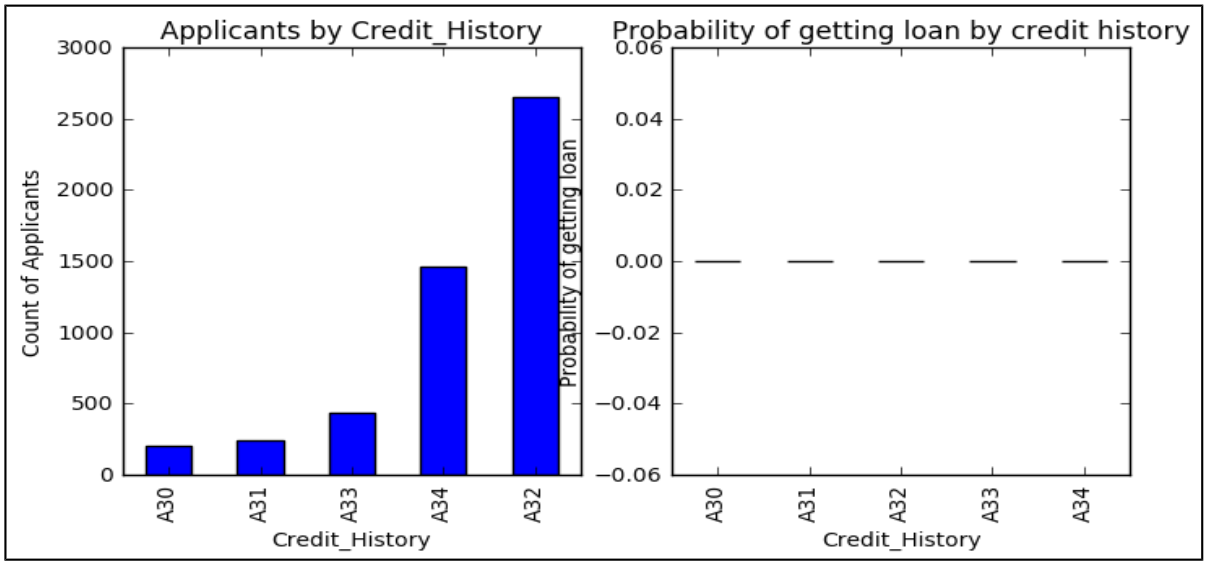
```
temp2 = data.pivot_table(values='Default_On_Payment',index=['Credit_History'],aggfunc=lambda x: x.map({'Y':1,'N':0}).mean())

print 'Frequency Table for Credit History:'

print temp1

Print '\n Probability of getting loan for each Credit History class:'
Print temp2
```

```
Frequency Table for Credit History:
A30     200
A31     245
A33     440
A34    1465
A32    2650
Name: Credit_History, dtype: int64
```

```
Probability of getting loan for each Credit History class:
Credit_History
A30   NaN
A31   NaN
A32   NaN
A33   NaN
A34   NaN
Name: Default_On_Payment, dtype: float64
```

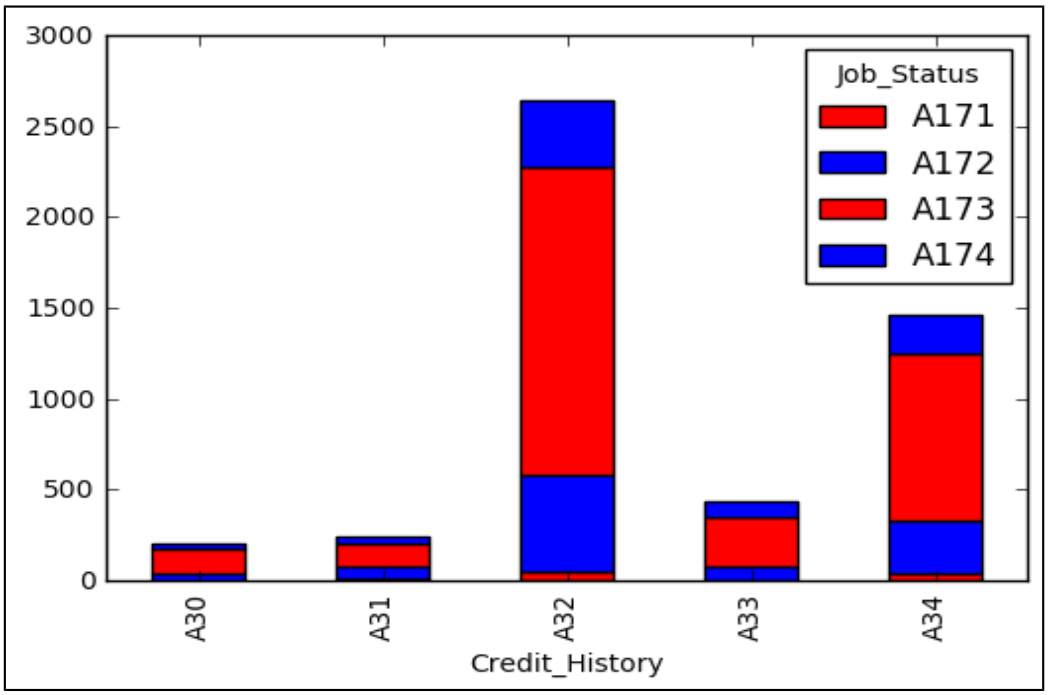
Now we can observe that we get a similar pivot_table like the MS Excel one. This can be plotted as a bar chart using the “matplotlib” library with following code:

```
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 six-fold if the applicant has a valid credit history.
Alternately, these two plots can also be visualized by combining them in a stacked chart

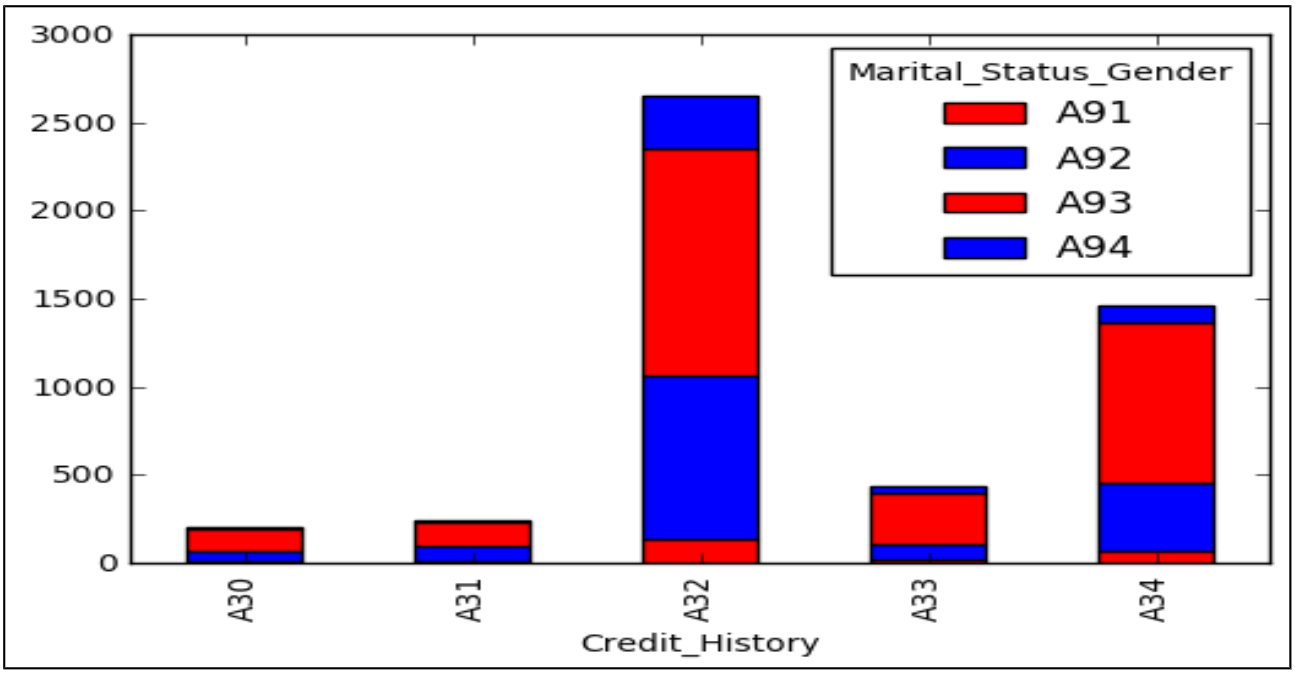
```
temp3 = pd.crosstab(data['Credit_History'], data['Job_Status'])
temp3.plot(kind='bar', stacked=True, color=['red','blue'], grid=False)
```



Here we can see that A171: Unemployed/ Unskilled - non-resident are very high and they are taking personal loan and delay to pay loan amount and A174: Management/ Self-employed/ Highly Qualified Employee paid loan on timely.

Now we are taking gender base variable to find out that which gender paid loan on time or not.

```
temp3 = pd.crosstab(data['Credit_History'], data['Marital_Status_Gender'])
temp3.plot(kind='bar', stacked=True, color=['red','blue'], grid=False)
```



Data Mugging: Using Pandas

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. Here are the problems, we are already aware of:

- 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.
- While looking at the distributions, we saw that Applicant Income and Loan Amount seemed to contain extreme values at either end. Though they might make intuitive sense, but should be treated appropriately.

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.

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

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

Customer_ID	0
Credit_Amount	0
Purpose_Credit_Taken	0
Duration_in_Months	0
Status_Checking_Accnt	0
Credit_History	0

Job_Status	12
Years_At_Present_Employment	0
Marital_Status_Gender	0
Other_Debtors_Guarantors	0
Current_Address_Yrs	0
Age	0
Housing	9
Num_Credits	0
Num_Dependents	0
Foreign_Worker	0
Default_On_Payment	0
Dtype:	int64

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. Get a detailed view on different imputation techniques.

How to fill missing values in Loan Amount

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:

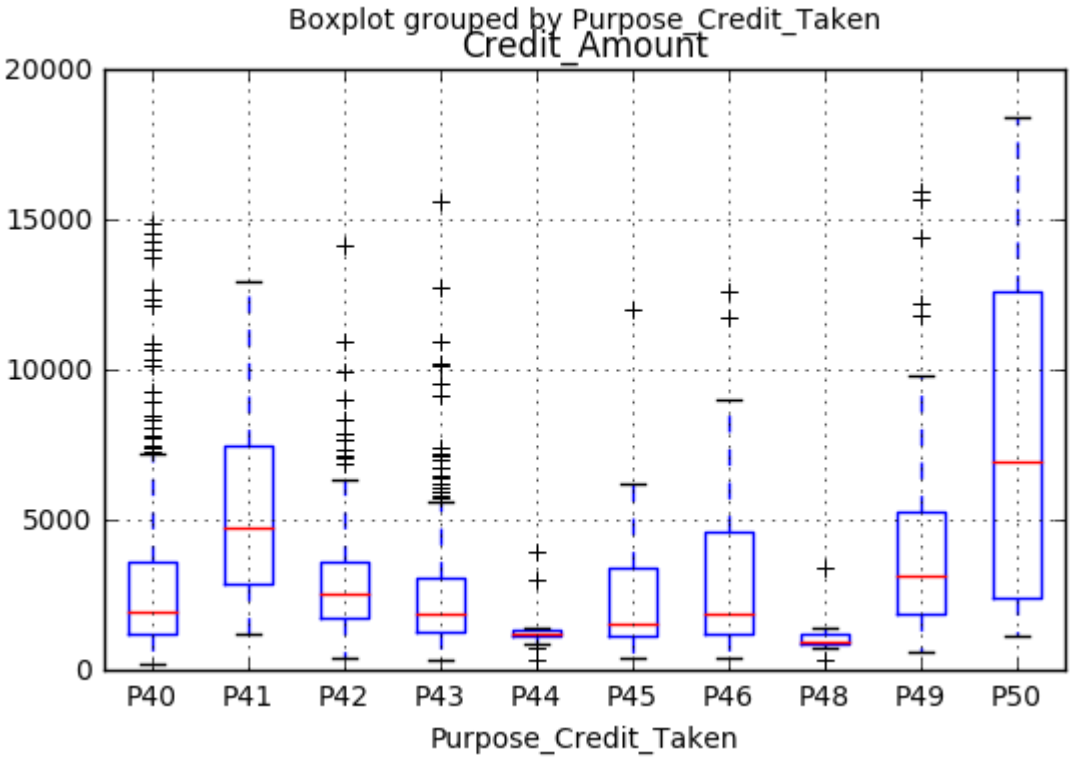
```
Data = data.dropna(subset=["Job_Status","Housing"])
```

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 mugging, I'll rather take an approach, which lies somewhere 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:

```
data.boxplot(column='Credit_Amount', by = 'Purpose_Credit_Taken')
```


Boxplot grouped by Purpose_Credit_Taken
Credit_Amount
20000
15000
10000
5000
0
P40
P41
P42
P43
P44
P45
P46
P48
P49
P50
Purpose_Credit_Taken

Building a Predictive Model

```
#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
```

1.Logistic Regression:

Logistic regression is a statistical method for analysing a dataset in which there are one or more independent variables that determine an outcome. The outcome is measured with a dichotomous variable (in which there are only two possible outcomes).

In logistic regression, the dependent variable is binary or dichotomous, i.e. it only contains data coded as 1 (TRUE, success, pregnant, etc.) or 0 (FALSE, failure, non-pregnant, etc.).

The goal of logistic regression is to find the best fitting model to describe the relationship between the dichotomous characteristic of interest and a set of independent variables.

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 over fitting. In simple words, taking all variables might result in the model understanding complex relations specific to the data and will not generalize well. .

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

- Applicants having a credit history
- Applicants with purpose of loan taken
- .Applicants with higher job status
- Properties in Number of existing credits at this bank

So let's make our first model with 'Credit_History'.

Importing libraries

```
import numpy as np
import matplotlib as plt
from sklearn import datasets
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
import graphlab
from __future__ import division
import numpy as np
graphlab.canvas.set_target('ipynb')
```

Split data into training and test sets

We split the data into a 80-20 split where 80% is in the training set and 20% is in the test set.

```
train_data, test_data = Data.random_split(.8, seed=1)
```

Logistic regression classifier

We will now train a logistic regression classifier with Default_On_Payment as the target and Credit_Amount as the features.

Implement logistic regression, I shall use GraphLab Create for its efficiency at processing this Loan dataset in its entirety.

model=graphlab.logistic_classifier.create(train_data,target='Default_On_Payment',features=['Credit_History'],)

Output

Logistic regression:

Number of examples: 3757

Number of classes: 2

Number of feature columns : 1

Number of unpacked features: 1

Number of coefficients : 5

Starting Newton Method

+-----------+----------+--------------+------------------+--------------------+

| Iteration | Passes | Elapsed Time | Training-accuracy | Validation-accuracy |

+-----------+----------+--------------+------------------+--------------------+

| 1 | 2 | 1.006459 | 0.717328 | 0.718447 |

| 2 | 3 | 1.124163 | 0.717328 | 0.718447 |

| 3 | 4 | 1.134996 | 0.717328 | 0.718447 |

| 4 | 5 | 1.140888 | 0.717328 | 0.718447 |

+-----------+----------+--------------+------------------+--------------------+

SUCCESS: Optimal solution found.

Try different combination of variables:

model1=graphlab.logistic_classifier.create(train_data,
target='Default_On_Payment',features=['Credit_Amount','Duration_in_Months','Years_At_Present_Employment','Num_Credits','Num_Dependents'])

Output:

Logistic regression:

Number of examples : 3763

Number of classes: 2

Number of feature columns: 5

Number of unpacked features: 5

Number of coefficients: 9

Starting Newton Method

+-----------+----------+--------------+-------------------+---------------------+

| Iteration | Passes | Elapsed Time | Training-accuracy | Validation-accuracy |

+-----------+----------+--------------+-------------------+---------------------+

| 1 | 2 | 0.020649 | 0.708743 | 0.705000 |

| 2 | 3 | 0.040265 | 0.706351 | 0.700000 |

| 3 | 4 | 0.055254 | 0.706351 | 0.700000 |

| 4 | 5 | 0.069619 | 0.706351 | 0.700000 |

+-----------+----------+--------------+-------------------+---------------------+

SUCCESS: Optimal solution found.

Model Evaluation

Accuracy

One performance metric I will use for our more advanced exploration is accuracy. Recall that the accuracy is given by

$$Accuracy = \frac{correctly\,classified\,data\,points}{Total\,Data\,point}$$

Confusion Matrix

In the case of binary classification, the confusion matrix is a 2-by-2 matrix laying out correct and incorrect predictions made in each label as follows:

		Predicted label	
		(+1)	(-1)
True label	(+1)	# of true positives	# of false negatives
	(-1)	# of false positives	# of true negatives

Precision and Recall

you may simply prefer to reduce the percentage of false positives to be less than, all positive predictions. This is where precision comes in:

$$Precision = \frac{[¿\,of\ positive\,data\,points\,with\,posotive\,prediction]}{[¿\,all\,points\,with\,posotive\,prediction]} + \frac{[¿\,true\ posotive]}{[¿\,true\,positive] + [¿\,false\,positives]}$$

First, let us compute the precision of the logistic regression classifier on the test_data.

A complementary metric is recall, which measures the ratio between the number of true positives and that of (ground-truth) positive reviews:

Let us compute the recall on the test_data.

$$Recall = \frac{[¿ of\ positive\ data\ points\ with\ posotive\ prediction]}{[¿ all\ posotive\ data\ points]} + \frac{[¿ true\ posotive]}{[¿ true\ positive] + [¿ false\ negatives]}$$

recall = predicted positive / (predicted positive + false negative)

Since we predict positive for all, there is no false negative. Our formula becomes

recall = predicted positive / predicted positive = 1

Predict the Model.

predictions = model.predict(test_data)

Evaluate the model and save the results into a dictionary

results = model1.evaluate(test_data)

results

Output

{'accuracy': 0.7107039537126326,
'auc': 0.6227776728626558,
'confusion_matrix': Columns:
target_label int
predicted_label int
count int

Rows: 4

Data:
+--------------+-----------------+-------+
| target_label | predicted_label | count |
+--------------+-----------------+-------+
0	1	33
0	0	701
1	1	36
1	0	267
+--------------+-----------------+-------+
[4 rows x 3 columns],
'f1_score': 0.1935483870967742,
'log_loss': 0.5853139988930907,
'precision': 0.5217391304347826,
'recall': 0.1188118811881188,

'roc_curve': Columns:
	threshold	float
	fpr	float
	tpr	float
	p	int
	n	int

Rows: 100001

Data:

threshold	fpr	tpr	p	n
0.0	1.0	1.0	303	734
1e-05	1.0	1.0	303	734
2e-05	1.0	1.0	303	734
3e-05	1.0	1.0	303	734
4e-05	1.0	1.0	303	734
5e-05	1.0	1.0	303	734
6e-05	1.0	1.0	303	734
7e-05	1.0	1.0	303	734
8e-05	1.0	1.0	303	734
9e-05	1.0	1.0	303	734

[100001 rows x 5 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.}

2. Decision Tree (Decision Tree Classifier)

Decision tree is another method for making a predictive model. It is known to provide higher accuracy than logistic regression model.

We will now train a Decision Tree Classifier with Default_On_Payment as the target and train_data as the features.

Implement logistic regression, I shall use GraphLab Create for its efficiency at processing this Loan dataset in its entirety.

model2 = graphlab.decision_tree_classifier.create(train_data, target='Default_On_Payment', max_depth = 3)

Decision tree classifier:

Number of examples : 3762

Number of classes : 2

Number of feature columns : 16

Number of unpacked features : 16

+-----------+--------------+------------------+--------------------+------------------+--------------------+

| Iteration | Elapsed Time | Training-accuracy | Validation-accuracy | Training-log_loss | Validation-log_loss |

+-----------+--------------+------------------+--------------------+------------------+--------------------+

| 1 | 0.021875 | 0.738969 | 0.766169 | 0.615033 | 0.613803 |

+-----------+--------------+------------------+--------------------+------------------+--------------------+

Predict the model

predictions = model2.predict(test_data)

Evaluate the model

results = model2.evaluate(test_data)

Results

Output:

{'accuracy': 0.7290260366441659,
'auc': 0.7470818607746333,
'confusion_matrix': Columns:
target_label int
predicted_label int
count int

Rows: 4

Data:
+--------------+-----------------+-------+
| target_label | predicted_label | count |
+--------------+-----------------+-------+
0	1	119
0	0	615
1	1	141
1	0	162
+--------------+-----------------+-------+
[4 rows x 3 columns],
'f1_score': 0.5008880994671403,
'log_loss': 0.6142756322733596,
'precision': 0.5423076923076923,
'recall': 0.46534653465346537,
'roc_curve': Columns:
threshold float
fpr float
tpr float
p int
n int

Rows: 100001

Data:

threshold	fpr	tpr	p	n
0.0	1.0	1.0	303	734
1e-05	1.0	1.0	303	734
2e-05	1.0	1.0	303	734
3e-05	1.0	1.0	303	734
4e-05	1.0	1.0	303	734
5e-05	1.0	1.0	303	734
6e-05	1.0	1.0	303	734
7e-05	1.0	1.0	303	734
8e-05	1.0	1.0	303	734
9e-05	1.0	1.0	303	734

[100001 rows x 5 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.}

3. 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.

model3 = graphlab.random_forest_classifier.create(train_data, target='Default_On_Payment',max_depth = 3)

Output

Random forest classifier:

Number of examples : 3782

Number of classes : 2

Number of feature columns : 16

Number of unpacked features : 16

+-----------+--------------+-------------------+---------------------+-------------------+---------------------+

| Iteration | Elapsed Time | Training-accuracy | Validation-accuracy | Training-log_loss | Validation-log_loss |

+-----------+--------------+-------------------+---------------------+-------------------+---------------------+

| 1 | 0.018006 | 0.731095 | 0.707182 | 0.530826 | 0.554171 |

| 2 | 0.028110 | 0.741407 | 0.723757 | 0.524921 | 0.549061 |

| 3 | 0.036003 | 0.745902 | 0.712707 | 0.519975 | 0.547513 |

| 4 | 0.041380 | 0.755685 | 0.718232 | 0.518041 | 0.542317 |

| 5 | 0.048110 | 0.752776 | 0.734807 | 0.518467 | 0.540688 |

| 6 | 0.057939 | 0.751454 | 0.729282 | 0.518810 | 0.540496 |

+-----------+--------------+-------------------+---------------------+-------------------+---------------------+

Predict the Model

predictions = model3.predict(test_data)

Evaluate the model

Results = model3.evaluate(test_data)

Results

{'accuracy': 0.7473481195756991,
 'auc': 0.7743770289835524,
 'confusion_matrix': Columns:
        target_label   int
        predicted_label   int
        count   int

 Rows: 4

 Data:
 +--------------+-----------------+-------+
| target_label | predicted_label | count |
 +--------------+-----------------+-------+
0	1	91
0	0	643
1	1	132
1	0	171
 +--------------+-----------------+-------+
[4 rows x 3 columns],
'f1_score': 0.5019011406844107,
'log_loss': 0.5183184430515253,
'precision': 0.5919282511210763,
'recall': 0.43564356435643564,
'roc_curve': Columns:
        threshold   float
        fpr   float
        tpr   float
        p   int
        n   int

 Rows: 100001

Data:

threshold	fpr	tpr	p	n
0.0	1.0	1.0	303	734
1e-05	1.0	1.0	303	734
2e-05	1.0	1.0	303	734
3e-05	1.0	1.0	303	734
4e-05	1.0	1.0	303	734
5e-05	1.0	1.0	303	734
6e-05	1.0	1.0	303	734
7e-05	1.0	1.0	303	734
8e-05	1.0	1.0	303	734
9e-05	1.0	1.0	303	734

[100001 rows x 5 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.}

4. K-Nearest Neighbours

The nearest neighbour classifier is one of the simplest classification models, but it often performs nearly as well as more sophisticated methods.

The nearest neighbour's classifier predicts the class of a data point to be the most common class among that point's neighbour's. Suppose we have n training data points, where the i'th point has both a vector of features xi and class label yi. For a new point x∗, the nearest neighbour classifier first finds the set of neighbours of x∗, denoted N(x∗). The class label for x∗ is then predicted to be

$$y* = \max_c \sum_{i \in N(x*)} I(y_i = c)$$

where the indicator function I() is 1 if the argument is true, and 0 otherwise. The simplicity of this approach makes the model relatively straightforward to understand and communicate to others, and naturally lends itself to multi-class classification.

```
model4 = graphlab.nearest_neighbor_classifier.create(train_data,
target='Default_On_Payment',
features=['Credit_Amount','Duration_in_Months','Years_At_Present_Employment','
Num_Credits','Num_Dependents']
```

Predict the model

```
predictions = model4.classify(test_data, max_neighbors=20, radius=None)

print predictions
```

Output:

```
Starting pairwise querying.
+--------------+---------+-------------+--------------+

| Query points | # Pairs | % Complete. | Elapsed Time |

+--------------+---------+-------------+--------------+

| 0            | 1037    | 0.0252334   | 6.699ms      |

| 888          | 3521652 | 85.6927     | 1.00s        |

| Done         |         | 100         | 1.21s        |

+--------------+---------+-------------+--------------+

+-------+-------------+
| class | probability |
+-------+-------------+
|   0   |     0.8     |
|   1   |     0.6     |
|   0   |     0.6     |
|   1   |     0.7     |
|   0   |     0.8     |
|   0   |     0.7     |
|   0   |     0.8     |
|   0   |     0.65    |
|   0   |     1.0     |
|   0   |     0.85    |
+-------+-------------+
[1037 rows x 2 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.
```

Advance Usage

topk = model4.predict_topk(test_data[:5], max_neighbors=20, k=3)

Print took

```
+--------+-------+-------------+---------------------+
| row_id | class | probability |
+--------+-------+-------------+
|   2    |   0   |     0.6     |
|   2    |   1   |     0.4     |
|   0    |   0   |     0.8     |
|   0    |   1   |     0.2     |
|   4    |   0   |     0.8     |
|   4    |   1   |     0.2     |
|   3    |   1   |     0.7     |
|   3    |   0   |     0.3     |
|   1    |   1   |     0.6     |
|   1    |   0   |     0.4     |
+--------+-------+-------------+
[10 rows x 3 columns]
```

To get a sense of the model validity, pass the test data to the evaluate method.

```
evals = model4.evaluate(test_data[:3000])

print evals['accuracy']
```

```
Starting pairwise querying.
+--------------+---------+-------------+--------------+

| Query points | # Pairs | % Complete. | Elapsed Time |

+--------------+---------+-------------+--------------+

| 0            | 1037    | 0.0252334   | 3.038ms      |

| 888          | 3521652 | 85.6927     | 1.00s        |

| Done         |         | 100         | 1.18s        |

+--------------+---------+-------------+--------------+

Starting pairwise querying.

+--------------+---------+-------------+--------------+

| Query points | # Pairs | % Complete. | Elapsed Time |

+--------------+---------+-------------+--------------+

| 0            | 1037    | 0.0252334   | 4.999ms      |

| 808          | 3203293 | 77.946      | 1.00s        |

| Done         |         | 100         | 1.28s        |

+--------------+---------+-------------+--------------+

0.772420443587
```

Confusion Matrix

```
conf_matrix = evals['confusion_matrix']
conf_matrix['within_one'] = conf_matrix.apply(
    lambda x: abs(x['target_label'] - x['predicted_label']) <= 1)
num_within_one = conf_matrix[conf_matrix['within_one']]['count'].sum()
print float(num_within_one) / len(test_data)
```

Output

1.0

Evaluate the Model

results = model4.evaluate(test_data)

results

Starting pairwise querying.

+--------------+---------+-------------+--------------+

| Query points | # Pairs | % Complete. | Elapsed Time |

+--------------+---------+-------------+--------------+

| 0 | 1037 | 0.0252334 | 9.646ms |

| 919 | 3644018 | 88.6702 | 1.00s |

| Done | | 100 | 1.16s |

+--------------+---------+-------------+--------------+

Starting pairwise querying.

+--------------+---------+-------------+--------------+

| Query points | # Pairs | % Complete. | Elapsed Time |

+--------------+---------+-------------+--------------+

| 0 | 1037 | 0.0252334 | 13.47ms |

| 854 | 3385805 | 82.3871 | 1.01s |

| Done | | 100 | 1.23s |

{'accuracy': 0.7830279652844745, 'confusion_matrix': Columns:
target_label int
predicted_label int
count int

Rows: 4

Data:

```
+--------------+-----------------+-------+
| target_label | predicted_label | count |
+--------------+-----------------+-------+
|      0       |        1        |   70  |
|      1       |        1        |  148  |
|      1       |        0        |  155  |
|      0       |        0        |  664  |
+--------------+-----------------+-------+
```

[4 rows x 3 columns], 'roc_curve': Columns:

```
threshold       float
fpr     float
tpr     float
p       int
n       int
```

Rows: 100001

Data:

```
+-----------+-----+-----+-----+-----+
| threshold | fpr | tpr |  p  |  n  |
+-----------+-----+-----+-----+-----+
|    0.0    | 1.0 | 1.0 | 303 | 734 |
|   1e-05   | 1.0 | 1.0 | 303 | 734 |
|   2e-05   | 1.0 | 1.0 | 303 | 734 |
|   3e-05   | 1.0 | 1.0 | 303 | 734 |
|   4e-05   | 1.0 | 1.0 | 303 | 734 |
|   5e-05   | 1.0 | 1.0 | 303 | 734 |
|   6e-05   | 1.0 | 1.0 | 303 | 734 |
|   7e-05   | 1.0 | 1.0 | 303 | 734 |
|   8e-05   | 1.0 | 1.0 | 303 | 734 |
|   9e-05   | 1.0 | 1.0 | 303 | 734 |
+-----------+-----+-----+-----+-----+
[100001 rows x 5 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.}
```

5. SVM (Support Vector Machines)

Support Vector Machines (SVM) is another popular model used for classification tasks.

In this model, given a set of features xi, and a label yi∈{0,1} the linear SVM minimizes the loss function:

fi(θ)=max(1−θTx,0)

model5 = graphlab.svm_classifier.create(train_data,
target='Default_On_Payment',
features=['Credit_Amount','Duration_in_Months','Years_At_Present_Employment','Num_Credits','Num_Dependents'])

SVM:

Number of examples : 3752

Number of classes : 2

Number of feature columns : 5

Number of unpacked features : 5

Number of coefficients : 9

Starting L-BFGS

+-----------+----------+-----------+-------------+------------------+--------------------+

| Iteration | Passes | Step size | Elapsed Time | Training-accuracy | Validation-accuracy |

+-----------+----------+-----------+-------------+------------------+--------------------+

| 1 | 4 | 0.000133 | 0.026253 | 0.699627 | 0.691943 |

| 2 | 8 | 0.250000 | 0.044625 | 0.699627 | 0.691943 |

| 3 | 9 | 0.250000 | 0.052400 | 0.699627 | 0.691943 |

| 4 | 10 | 0.250000 | 0.059741 | 0.699627 | 0.691943 |

| 5 | 11 | 0.250000 | 0.071019 | 0.699627 | 0.691943 |

| 6 | 13 | 1.000000 | 0.089934 | 0.699627 | 0.691943 |

+-----------+----------+-----------+-------------+-----------------+--------------------+

TERMINATED: Iteration limit reached.

This model may not be optimal. To improve it, consider increasing `max_iterations`.

Predict the Model

predictions = model5.predict(test_data)

predictions

Evaluate the Model

results = model.evaluate(test_data)

results

{'accuracy': 0.7116682738669238,
 'auc': 0.5569891457810652,
 'confusion_matrix': Columns:
	target_label	int
	predicted_label	int
	count	int

Rows: 4

Data:
+--------------+-----------------+-------+
| target_label | predicted_label | count |
+--------------+-----------------+-------+
0	1	9
0	0	725
1	1	13
1	0	290
+--------------+-----------------+-------+
[4 rows x 3 columns],
 'f1_score': 0.08,
 'log_loss': 0.5927744942796308,
 'precision': 0.5909090909090909,
 'recall': 0.0429042904290429,
 'roc_curve': Columns:
	threshold	float
	fpr	float
	tpr	float
	p	int
	n	int

Rows: 100001

Data:

threshold	fpr	tpr	p	n
0.0	1.0	1.0	303	734
1e-05	1.0	1.0	303	734
2e-05	1.0	1.0	303	734
3e-05	1.0	1.0	303	734
4e-05	1.0	1.0	303	734
5e-05	1.0	1.0	303	734
6e-05	1.0	1.0	303	734
7e-05	1.0	1.0	303	734
8e-05	1.0	1.0	303	734
9e-05	1.0	1.0	303	734

[100001 rows x 5 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.}

6. Gradient Boosted Regression Trees

The Gradient Boosted Regression Trees (GBRT) model (also called Gradient Boosted Machine or GBM), is one of the most effective machine learning models for predictive analytics.

```
model6 = graphlab.boosted_trees_classifier.create(train_data,
target='Default_On_Payment',max_iterations=2,max_depth = 3)

Boosted trees classifier:
--------------------------------------------------------

Number of examples          : 3752

Number of classes           : 2

Number of feature columns   : 16

Number of unpacked features : 16

+-----------+--------------+------------------+--------------------+------------------+--------------------+

| Iteration | Elapsed Time | Training-accuracy | Validation-accuracy | Training-log_loss | Validation-log_loss |

+-----------+--------------+------------------+--------------------+------------------+--------------------+

| 1         | 0.014779     | 0.739339         | 0.758294           | 0.615791         | 0.604876           |

| 2         | 0.021827     | 0.738806         | 0.763033           | 0.571385         | 0.557868           |

+-----------+--------------+------------------+--------------------+------------------+--------------------+
```

Predicts the Model

```
predictions = model6.classify(test_data)
```

Evaluate the Model

```
results = model6.evaluate(test_data)

results

{'accuracy': 0.7348119575699132,
 'auc': 0.7720119423386479,
 'confusion_matrix': Columns:
        target_label    int
        predicted_label int
        count   int
```

Rows: 4

Data:

```
+--------------+-----------------+-------+
| target_label | predicted_label | count |
+--------------+-----------------+-------+
|      0       |        1        |  111  |
|      0       |        0        |  623  |
|      1       |        1        |  139  |
|      1       |        0        |  164  |
+--------------+-----------------+-------+
[4 rows x 3 columns],
'f1_score': 0.5027124773960217,
'log_loss': 0.5697159856132781,
'precision': 0.556,
'recall': 0.45874587458745875,
'roc_curve': Columns:
        threshold       float
        fpr     float
        tpr     float
        p       int
        n       int
```

Rows: 100001

Data:

```
+-----------+-----+-----+-----+-----+
| threshold | fpr | tpr |  p  |  n  |
+-----------+-----+-----+-----+-----+
|    0.0    | 1.0 | 1.0 | 303 | 734 |
|   1e-05   | 1.0 | 1.0 | 303 | 734 |
|   2e-05   | 1.0 | 1.0 | 303 | 734 |
|   3e-05   | 1.0 | 1.0 | 303 | 734 |
|   4e-05   | 1.0 | 1.0 | 303 | 734 |
|   5e-05   | 1.0 | 1.0 | 303 | 734 |
|   6e-05   | 1.0 | 1.0 | 303 | 734 |
|   7e-05   | 1.0 | 1.0 | 303 | 734 |
|   8e-05   | 1.0 | 1.0 | 303 | 734 |
|   9e-05   | 1.0 | 1.0 | 303 | 734 |
+-----------+-----+-----+-----+-----+

[100001 rows x 5 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.}
```

Final Output Of the Model

Machine Learning Algorithm Model

Algorithms	Accuracy	AUC	f1_score	log_loss	precision	recall
Logistic Regression	71%	62%	19%	58%	52%	11%
Decision Tree	73%	74%	51%	61%	54%	46%
Random Forest	74%	77%	50%	51%	59%	43%
K-Nearest Neighbours	78%					
SVM	71%	56%	8%	59%	59%	4%
Gradient Boosted Regression Trees	73%	77%	50%	56%	55%	45%