Loan Approval Prediction

FEBRUARY 2021

Submitted by:

HRISAV BHOWMICK

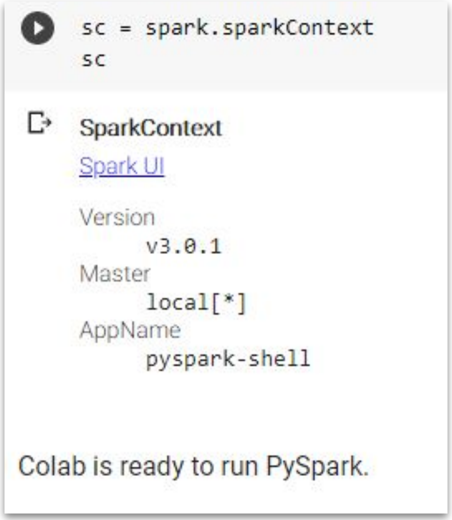
PGPDS
PRAXIS BUSINESS SCHOOL
KOLKATA, INDIA

Understanding Problem

- *Dream Housing Finance Company* deals in all home loans.
- They have presence across all urban, semi urban and rural areas.
- Initially the customers apply for a home loan and then the company validates the customer if he/she is eligible for loan.
- Company wants to automate the loan eligibility process.
- To automate this process, they have given a problem to identify the customer segments that are eligible for loan, so that they can specifically target these customers.

Setting up Spark environment

- Google Colab is being used for the project.
- Install all the dependencies in Colab environment such as Apache Spark 3.0.1 with Hadoop 3.2, Java 8
- Use Findspark in order to locate Spark in the system.
- Set the environment path that enables us to run PySpark in our Colab environment.
- Run a local spark session to test our installation.
- Colab is ready to run PySpark.



Understanding Data

- Gender, Married, Dependents, Education, Self Employed, Property Area, Loan Status are categorical data.
- Applicant Income, Co-applicant Income, Loan Amount, Loan Amount Term, Credit History are numerical data.
- We have 12 independent variables and 1 target variable.
- In total, we have 614 rows and 13 columns in the dataset.

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)

Understanding Data

- Loan Status is the target variable (Y = loan approved, N = loan not approved).
- Showing the top 5 rows of the dataset.
- Loan_ID is a unique identifier, it can be dropped off.

```
df.show(5)
```

```
+--------+------+-------+----------+------------+-------------+---------------+-----------------+----------+----------------+--------------+-------------+-----------+
| Loan_ID|Gender|Married|Dependents|   Education|Self_Employed|ApplicantIncome|CoapplicantIncome|LoanAmount|Loan_Amount_Term|Credit_History|Property_Area|Loan_Status|
+--------+------+-------+----------+------------+-------------+---------------+-----------------+----------+----------------+--------------+-------------+-----------+
|LP001002|  Male|     No|         0|    Graduate|           No|           5849|              0.0|      null|             360|             1|        Urban|          Y|
|LP001003|  Male|    Yes|         1|    Graduate|           No|           4583|           1508.0|       128|             360|             1|        Rural|          N|
|LP001005|  Male|    Yes|         0|    Graduate|          Yes|           3000|              0.0|        66|             360|             1|        Urban|          Y|
|LP001006|  Male|    Yes|         0|Not Graduate|           No|           2583|           2358.0|       120|             360|             1|        Urban|          Y|
|LP001008|  Male|     No|         0|    Graduate|           No|           6000|              0.0|       141|             360|             1|        Urban|          Y|
+--------+------+-------+----------+------------+-------------+---------------+-----------------+----------+----------------+--------------+-------------+-----------+
only showing top 5 rows
```

Handling Missing Values

```
from pyspark.sql.functions import col,sum

df.select(*(sum(col(c).isNull().cast("int")).alias(c) for c in df.columns)).show()
```

```
+------+-------+----------+---------+-------------+---------------+-----------------+----------+----------------+--------------+-------------+-----------+
|Gender|Married|Dependents|Education|Self_Employed|ApplicantIncome|CoapplicantIncome|LoanAmount|Loan_Amount_Term|Credit_History|Property_Area|Loan_Status|
+------+-------+----------+---------+-------------+---------------+-----------------+----------+----------------+--------------+-------------+-----------+
|    13|      3|        15|        0|           32|              0|                0|        22|              14|            50|            0|          0|
+------+-------+----------+---------+-------------+---------------+-----------------+----------+----------------+--------------+-------------+-----------+
```

- Missing values are present in both categorical and numerical columns.
- No missing value is present in Target variable, else we could have dropped those rows.
- We will be using mean or median (for numerical) and mode (for categorical) to impute the missing values.

Handling Missing Values

- Here we are imputing missing values in Loan Amount Term with its median = 360.

- There were 14 records with missing Loan Amount Term. After imputing number of missing records became 0.

```
from pyspark.sql.functions import isnan, when, count, col

df.select([count(when(col('Loan_Amount_Term').isNull(),True)).alias('Loan_Amount_Term')]).show()
```

```
+----------------+
|Loan_Amount_Term|
+----------------+
|              14|
+----------------+
```

```
median_loan_term = df.approxQuantile("Loan_Amount_Term", [0.5], 0.25)
median_loan_term = int(median_loan_term[0])

print('Median value of Loan_Amount_Term', median_loan_term)
```

```
Median value of Loan_Amount_Term 360
```

```
# using median_loan_term value to fill the nulls in Loan_Amount_Term column

df = df.na.fill(median_loan_term, subset=['Loan_Amount_Term'])

df.select([count(when(col('Loan_Amount_Term').isNull(),True)).alias('Loan_Amount_Term')]).show()
```

```
+----------------+
|Loan_Amount_Term|
+----------------+
|               0|
+----------------+
```

Handling Missing Values

```
from pyspark.sql.functions import mean

mean_val = df.select(mean(df.LoanAmount)).collect()

print('Mean value of Loan Amount', mean_val[0][0])
```

```
Mean value of Loan Amount 146.41216216216216
```

```
mean_loan_amount = mean_val[0][0]

# using mean_loan_amount value to fill the nulls in LoanAmount column

df = df.na.fill(mean_loan_amount, subset=['LoanAmount'])

mean_val = df.select(mean(df.LoanAmount)).collect()
print('New Mean value of Loan Amount', mean_val[0][0])
```

```
New Mean value of Loan Amount 146.3973941368078
```

- Here we are imputing missing values in Loan Amount with its mean.
- There were 22 missing values in Loan Amount. After imputing them, the mean for that column turned out to be 146.39.

Handling Missing Values

```
df.groupBy("Gender").count().show()
```

```
+------+-----+
|Gender|count|
+------+-----+
|  null|   13|
|Female|  112|
|  Male|  489|
+------+-----+
```

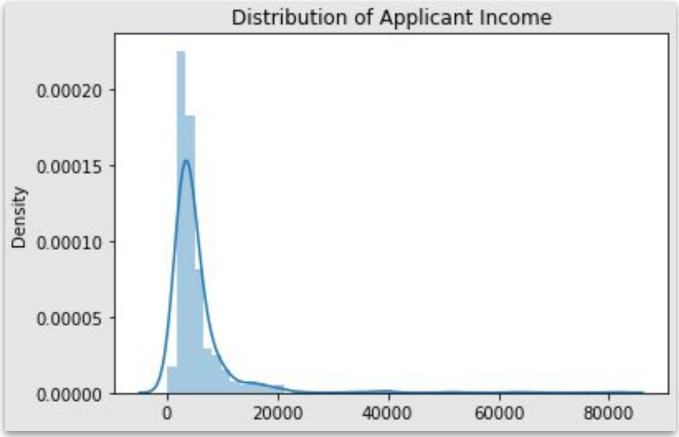
```
df = df.na.fill('Male',subset=['Gender'])
df.groupBy("Gender").count().show()
```

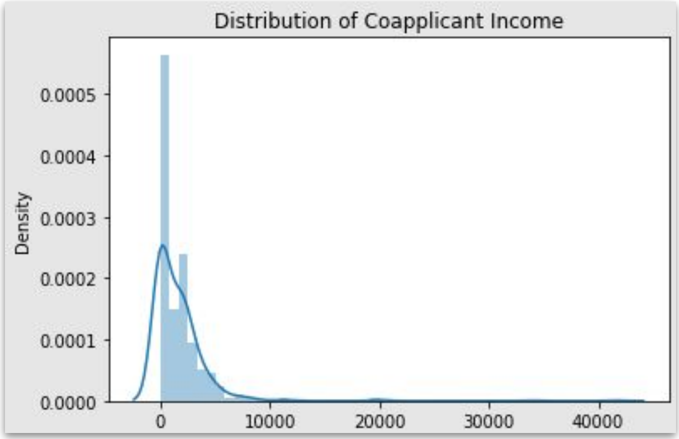
```
+------+-----+
|Gender|count|
+------+-----+
|Female|  112|
|  Male|  502|
+------+-----+
```

- Here we are imputing missing values in Gender with its mode.
- We will do the same imputation for rest of the categorical columns.

```
+------+-------+----------+---------+-------------+---------------+-----------------+----------+----------------+--------------+-------------+-----------+
|Gender|Married|Dependents|Education|Self_Employed|ApplicantIncome|CoapplicantIncome|LoanAmount|Loan_Amount_Term|Credit_History|Property_Area|Loan_Status|
+------+-------+----------+---------+-------------+---------------+-----------------+----------+----------------+--------------+-------------+-----------+
|     0|      0|         0|        0|            0|              0|                0|         0|               0|             0|            0|          0|
+------+-------+----------+---------+-------------+---------------+-----------------+----------+----------------+--------------+-------------+-----------+
```

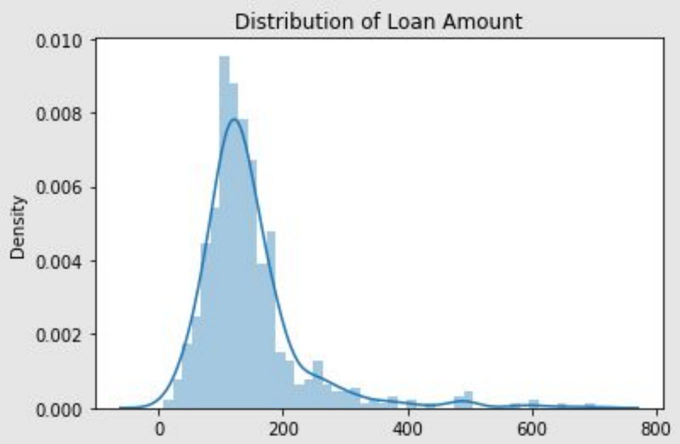
Exploratory Analysis

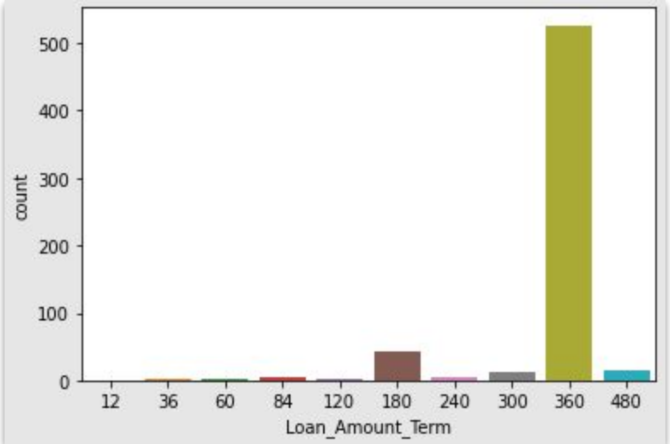




- Both applicant and co-applicant income is right/positively skewed.
- That is mean of each of them is greater than their median values.
- Majority of the Applicant Income falls below 15,000.

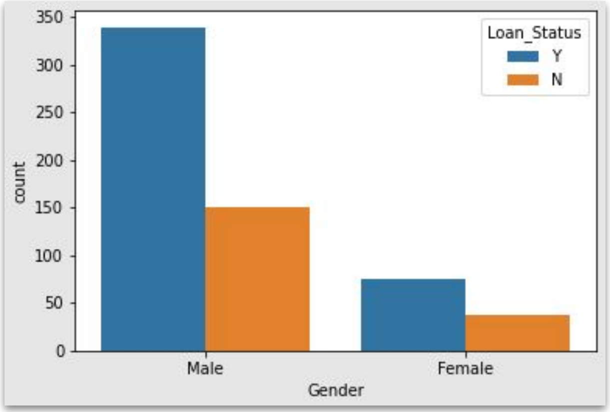
Exploratory Analysis

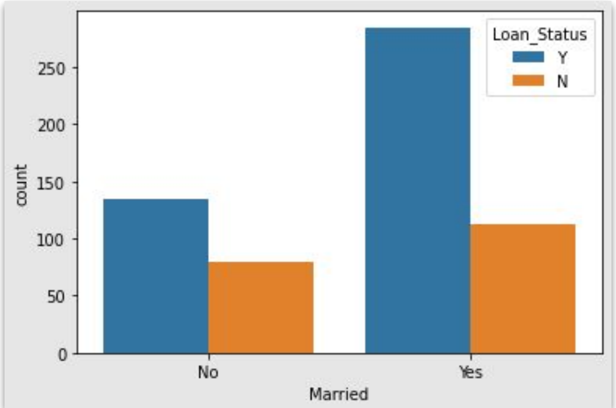


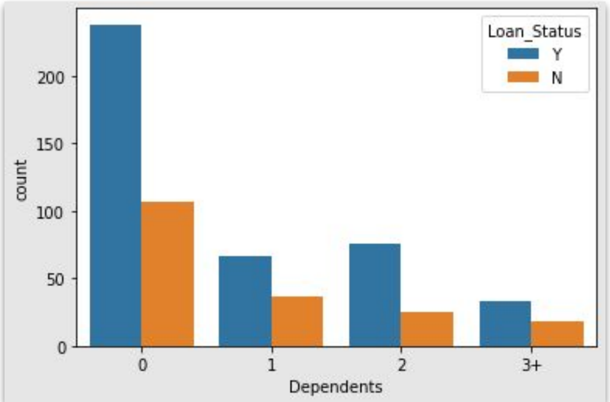


- Majority of the Loan Amount is between 100,000 to 200,000.
- Most of the loan term is applied for 360 months (ie. 30 years) followed by 180 months (ie. 15 years).

Exploratory Analysis

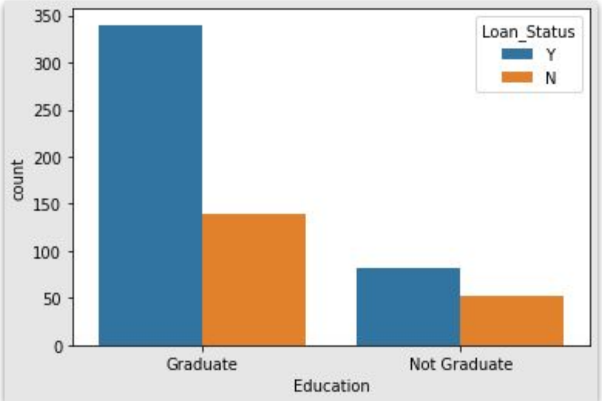


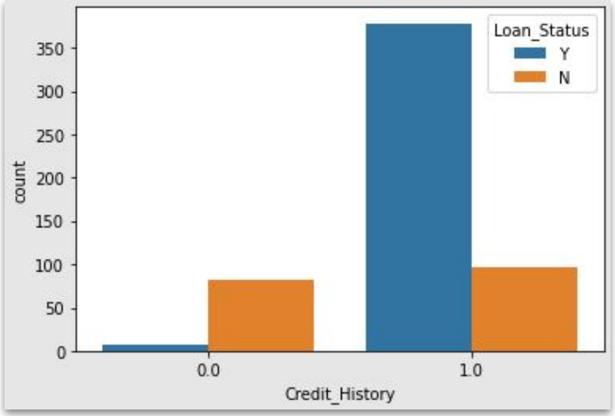


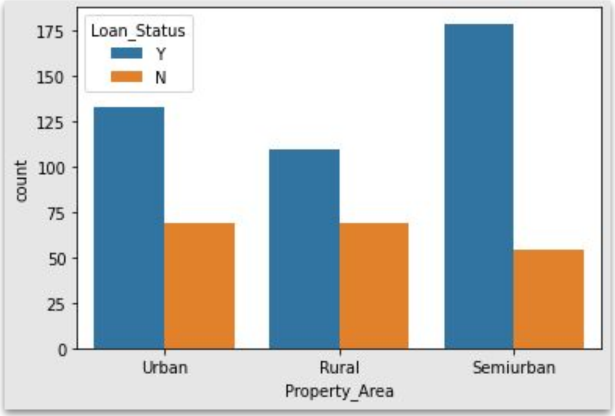


- Chances for getting Loan approved for Male or Female is almost equal.
- Married people have higher chance of getting loan approved.
- When the number of dependents is 2 there is more chance of getting loan approval compared to when the number of dependents is 1, even though both has similar data count.

Exploratory Analysis







- Graduates have more chances of getting their loan approved.
- There is very rare chance of getting loan if the person has no credit history.
- Most of the person is from Semi-urban area, and highest Loan acceptance is from that area.

Feature Engineering

```python
from pyspark.sql.functions import col

df = df.withColumn("TotalIncome", col("ApplicantIncome")+col("CoapplicantIncome"))
df = df.withColumn("EMI", col("LoanAmount")/col("Loan_Amount_Term"))
df.show(5)
```

Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History	Property_Area	Loan_Status	TotalIncome	EMI
Male	No	0	Graduate	No	5849	0.0	146	360	1	Urban	Y	5849.0	0.40555555555555556
Male	Yes	1	Graduate	No	4583	1508.0	128	360	1	Rural	N	6091.0	0.35555555555555557
Male	Yes	0	Graduate	Yes	3000	0.0	66	360	1	Urban	Y	3000.0	0.18333333333333332
Male	Yes	0	Not Graduate	No	2583	2358.0	120	360	1	Urban	Y	4941.0	0.3333333333333333
Male	No	0	Graduate	No	6000	0.0	141	360	1	Urban	Y	6000.0	0.39166666666666666

- Adding two more features to our data.
- **Total Income** is the collective income of Applicant and Co-Applicant.
- **EMI** is the equated monthly installment. It is calculated by dividing Loan Amount by number of months required to repay the loan.

Preprocessing

```
from pyspark.ml.feature import StringIndexer, OneHotEncoder

# create object of StringIndexer class and specify input and output column
SI_gender = StringIndexer(inputCol='Gender',outputCol='gender_Index')
SI_married = StringIndexer(inputCol='Married',outputCol='married_Index')
SI_dependents = StringIndexer(inputCol='Dependents',outputCol='dependents_Index')
SI_education = StringIndexer(inputCol='Education',outputCol='education_Index')
SI_selfemp = StringIndexer(inputCol='Self_Employed',outputCol='selfemp_Index')
SI_credit = StringIndexer(inputCol='Credit_History',outputCol='credit_Index')
SI_property = StringIndexer(inputCol='Property_Area',outputCol='property_Index')
SI_loanstatus = StringIndexer(inputCol='Loan_Status',outputCol='loanstatus_Index')

# transform the data
df = SI_gender.fit(df).transform(df)
df = SI_married.fit(df).transform(df)
df = SI_dependents.fit(df).transform(df)
df = SI_education.fit(df).transform(df)
df = SI_selfemp.fit(df).transform(df)
df = SI_credit.fit(df).transform(df)
df = SI_property.fit(df).transform(df)
df = SI_loanstatus.fit(df).transform(df)
```

```
OHE = OneHotEncoder(inputCols=['gender_Index', 'married_Index','dependents_Index','education_Index',
                               'selfemp_Index','credit_Index','property_Index','loanstatus_Index'],
                    outputCols=['gender_OHE', 'married_OHE','dependents_OHE','education_OHE',
                                'selfemp_OHE','credit_OHE','property_OHE','loanstatus_OHE'])

# transform the data
df = OHE.fit(df).transform(df)
```

- The **StringIndexer** encodes a string column of labels to a column of label indices.
- **One Hot Encoding** is used for converting categorical attributes into a numeric vector that machine learning models can understand.
- Each of the categorical features are converted to its Index and OHE columns.

Preprocessing

```
df.select('Married','married_Index', 'married_OHE',
          'Property_Area','property_Index','property_OHE').show(10)
```

```
+-------+-------------+-------------+-------------+--------------+-------------+
|Married|married_Index|  married_OHE|Property_Area|property_Index| property_OHE|
+-------+-------------+-------------+-------------+--------------+-------------+
|     No|          1.0|    (1,[],[])|        Urban|           1.0|(2,[1],[1.0])|
|    Yes|          0.0|(1,[0],[1.0])|        Rural|           2.0|    (2,[],[])|
|    Yes|          0.0|(1,[0],[1.0])|        Urban|           1.0|(2,[1],[1.0])|
|    Yes|          0.0|(1,[0],[1.0])|        Urban|           1.0|(2,[1],[1.0])|
|     No|          1.0|    (1,[],[])|        Urban|           1.0|(2,[1],[1.0])|
|    Yes|          0.0|(1,[0],[1.0])|        Urban|           1.0|(2,[1],[1.0])|
|    Yes|          0.0|(1,[0],[1.0])|        Urban|           1.0|(2,[1],[1.0])|
|    Yes|          0.0|(1,[0],[1.0])|    Semiurban|           0.0|(2,[0],[1.0])|
|    Yes|          0.0|(1,[0],[1.0])|        Urban|           1.0|(2,[1],[1.0])|
|    Yes|          0.0|(1,[0],[1.0])|    Semiurban|           0.0|(2,[0],[1.0])|
+-------+-------------+-------------+-------------+--------------+-------------+
```

- Showing results for Married and Property_Area only.
- For Married = No, Index has 1.0 and OHE has (1,[],[]).
- For Property_Area = Rural, Index has 2.0 and OHE has (2,[],[]).

Preprocessing

	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History	TotalIncome	EMI	gender_Index	married_Index	dependents_Index	education_Index	selfemp_Index	credit_Index	property_Index
ApplicantIncome	1.00	-0.12	0.57	-0.05	-0.02	0.89	0.32	-0.06	-0.05	0.12	-0.14	0.13	0.02	0.02
CoapplicantIncome	-0.12	1.00	0.19	-0.06	0.01	0.34	0.14	-0.08	-0.08	0.03	-0.06	-0.02	-0.01	0.02
LoanAmount	0.57	0.19	1.00	0.04	-0.00	0.62	0.49	-0.11	-0.15	0.16	-0.17	0.12	0.00	0.03
Loan_Amount_Term	-0.05	-0.06	0.04	1.00	-0.00	-0.07	-0.50	0.07	0.10	-0.10	-0.07	-0.03	0.00	-0.02
Credit_History	-0.02	0.01	-0.00	-0.00	1.00	-0.01	0.02	-0.01	-0.01	-0.04	-0.07	-0.00	-1.00	-0.03
TotalIncome	0.89	0.34	0.62	-0.07	-0.01	1.00	0.36	-0.09	-0.08	0.13	-0.16	0.11	0.01	0.03
EMI	0.32	0.14	0.49	-0.50	0.02	0.36	1.00	-0.06	-0.09	0.10	-0.08	0.05	-0.02	0.01
gender_Index	-0.06	-0.08	-0.11	0.07	-0.01	-0.09	-0.06	1.00	0.36	-0.17	-0.05	0.00	0.01	-0.11
married_Index	-0.05	-0.08	-0.15	0.10	-0.01	-0.08	-0.09	0.36	1.00	-0.33	-0.01	-0.00	0.01	0.01
dependents_Index	0.12	0.03	0.16	-0.10	-0.04	0.13	0.10	-0.17	-0.33	1.00	0.06	0.06	0.04	-0.00
education_Index	-0.14	-0.06	-0.17	-0.07	-0.07	-0.16	-0.08	-0.05	-0.01	0.06	1.00	-0.01	0.07	0.07
selfemp_Index	0.13	-0.02	0.12	-0.03	-0.00	0.11	0.05	0.00	-0.00	0.06	-0.01	1.00	0.00	0.01
credit_Index	0.02	-0.01	0.00	0.00	-1.00	0.01	-0.02	0.01	0.01	0.04	0.07	0.00	1.00	0.03
property_Index	0.02	0.02	0.03	-0.02	-0.03	0.03	0.01	-0.11	0.01	-0.00	0.07	0.01	0.03	1.00
loanstatus_Index	0.00	0.06	0.04	0.02	-0.54	0.03	0.01	0.02	0.09	-0.01	0.09	0.00	0.54	0.14

- Using the correlation matrix we can understand which features are highly correlated and which are not.
- Applicant Income and Total Income is highly correlated, so we will not consider Total Income for further analysis.

Preprocessing

```python
from pyspark.ml.feature import VectorAssembler

# specify the input and output columns of the vector assembler
assembler = VectorAssembler(inputCols=['gender_Index','married_Index','dependents_Index',
                                       'education_Index','selfemp_Index','ApplicantIncome',
                                       'CoapplicantIncome','EMI','LoanAmount','Loan_Amount_Term',
                                       'credit_Index','property_Index','gender_OHE',
                                       'married_OHE','dependents_OHE','education_OHE',
                                       'selfemp_OHE','credit_OHE','property_OHE'],
                            outputCol='features')

# transform the data
df2 = assembler.transform(df)
```

```
df2.select('features','loanstatus_Index').show(5)

+--------------------+----------------+
|            features|loanstatus_Index|
+--------------------+----------------+
|(21,[1,5,7,8,10,1...|             0.0|
|(21,[2,5,6,7,8,10...|             1.0|
|(21,[4,5,7,8,10,1...|             0.0|
|(21,[3,5,6,7,8,10...|             0.0|
|(21,[1,5,7,8,10,1...|             0.0|
+--------------------+----------------+
```

- The **Vector Assembler** is used to concatenate all the features into a single vector which can be further passed to the estimator or ML algorithm.
- Here it merges all the columns into a single feature vector, named ‘features’.
- In the bottom part, the features are drawn against loanstatus_Index.

Model Building

```
from pyspark.ml.classification import DecisionTreeClassifier

dt = DecisionTreeClassifier(featuresCol = 'features',
                            labelCol = 'label', maxDepth = 3)
dtModel = dt.fit(train_data)
predictions = dtModel.transform(test_data)
predictions.show(10)
```

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator

evaluator = BinaryClassificationEvaluator()
print("Decision Tree - Test set AUC: " + str(evaluator.evaluate
                                        (predictions,
                                         {evaluator.metricName: "areaUnderROC"})))
```

```
Decision Tree - Test set AUC: 0.3090934371523915
```

- **ROC curve** is an evaluation metric for binary classification problems. **AUC** is the measure of the ability of a classifier to distinguish between classes and is used as a summary of the ROC curve.
- The higher the AUC, the better the performance of the model at distinguishing between the positive and negative classes.
- For **Decision Tree**, area under ROC for test set is **0.31**.
- Decision tree performed poorly because it is too weak given the range of different features.

Model Building

```
from pyspark.ml.classification import RandomForestClassifier

rf = RandomForestClassifier(featuresCol = 'features', labelCol = 'label')
rfModel = rf.fit(train_data)
predictions = rfModel.transform(test_data)
predictions.show(10)
```

```
evaluator = BinaryClassificationEvaluator()
print("Random Forest - Test set AUC: " + str(evaluator.evaluate
                                        (predictions,
                                         {evaluator.metricName: "areaUnderROC"})))
```

```
Random Forest - Test set AUC: 0.7291434927697449
```

- The prediction accuracy of decision trees has been improved by Ensemble method - **Random Forest.**
- Here certain number of weak learners (decision trees) are combined to make a powerful prediction model.
- For Random Forest, area under ROC for test set is **0.72**.
- When 0.5<AUC<1, there is a high chance that the classifier will be able to distinguish the positive class values from the negative class values.

Model Building

```
from pyspark.ml.classification import LogisticRegression

lr = LogisticRegression(featuresCol = 'features', labelCol = 'label', maxIter=10)
lrModel = lr.fit(train_data)
```

```
trainingSummary = lrModel.summary
print('Training set Area Under ROC: ' + str(trainingSummary.areaUnderROC))
print('Training set Accuracy: ' + str(trainingSummary.accuracy))
print('Training set Precision by label: ' + str(trainingSummary.precisionByLabel))
print('Training set Recall by label: ' + str(trainingSummary.recallByLabel))
```

```
Training set Area Under ROC: 0.7926975858960232
Training set Accuracy: 0.8078703703703703
Training set Precision by label: [0.7945205479452054, 0.8805970149253731]
Training set Recall by label: [0.9731543624161074, 0.44029850746268656]
```

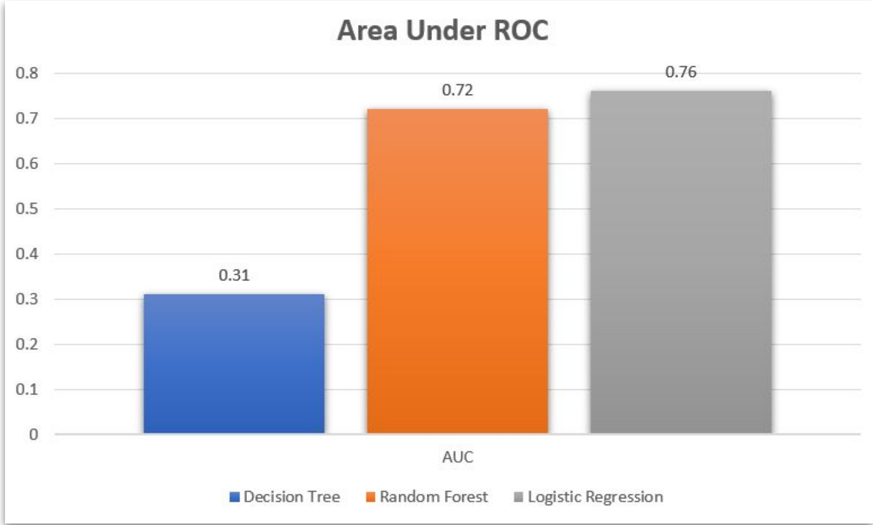
```
from pyspark.ml.evaluation import BinaryClassificationEvaluator

evaluator = BinaryClassificationEvaluator()
print('Logistic Regression - Test set AUC: ', evaluator.evaluate(predictions))
```

```
Logistic Regression - Test set AUC:  0.762931034482759
```

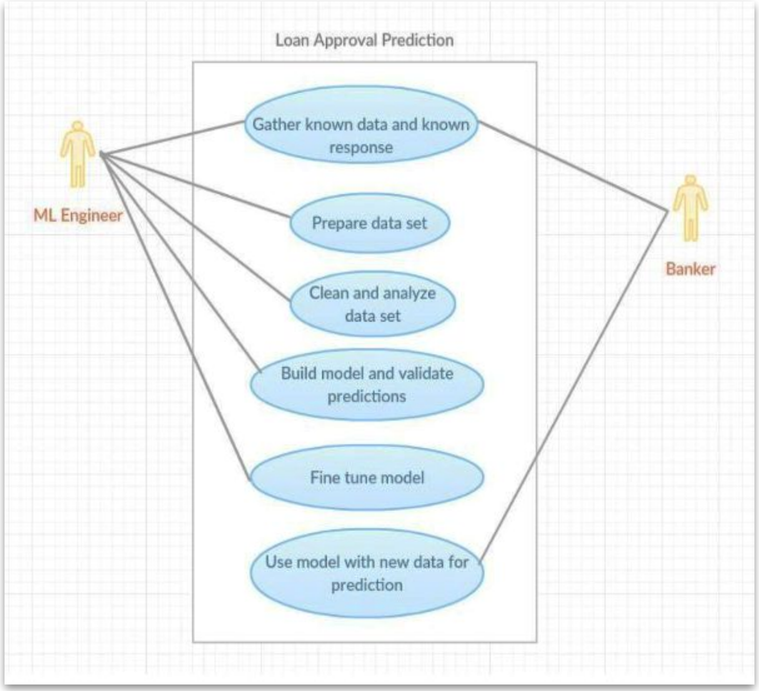
- In the third case, a **Logistic Regression** model was built on the training set. Logistic regression is an estimation of Logit function. The logit function is simply a log of odds in favor of the event.
- Training set accuracy is 0.81 and area under ROC is 0.79.
- For Logistic Regression, area under ROC for test set is **0.76**.
- This model gives us the highest value of area under the curve.

Conclusion



- From the Exploratory Data Analysis, we could generate insight from the data - how each of the features are related to the target.
- Also, it can be seen from the evaluation of three models that Logistic Regression performed better than others, and Random Forest did better than Decision Tree.
- Logistic Regression gives us the highest value of AUC. Thus we will consider this for model building.

Conclusion



- So we have built a robust model which can predict whether to provide loan or not to an applicant.
- In future, this module of prediction can be integrated with the module of automated processing system.
- Once loan is approved, analysis can be done to predict interest rates based on applicant’s profile.
- Time series analysis can be also be done using loan data of several years, for prediction of time when the applicant can default.