CS570 Big Data Processing and Analytics

Northwestern Polytechnic University, Fremont, CA

PROJECT REPORT

Fall 2021

Prof. Nooshin Nabizadeh

Lending Club Data analysis and Default Loan/Rating Prediction

13th November 2021

1. Project Background and Description

This is a Course project for CS570 Big Data Processing and Analytics. Under the scope of the course work project, we are required to present implementation of one of the taught concepts to satisfy the course project requirements.

I used "*Lending Club historical dataset*" for analysis and modeling. This is an open-source dataset from lending club, which contains complete loan data for all the loan issued through 2007-2015.

The peer-to-peer lending industry has grown significantly since its inception in 2007. With billions in annual loans, there are significant opportunities to capitalize on this alternative investment instrument. I have developed a sophisticated learning model to bolster investment strategy that utilizes Lending Club Corporation's massive historical datasets to understand which features best predict someone's probability of defaulting loan. To model the default loan prediction, I have used many machine learning algorithms/techniques and implemented them using big data distributed computing frameworks I have learnt during my course work.

Summary of results

For this project, I have performed the explanatory data analysis using apache spark framework to gather business insights from the historical loan data. I have built four machine learning classifiers to identify the borrowers who are more likely to default on their loan. The machine learning algorithms used are Logistic Regression, Naïve Bayes, Random Forest, and Gradient Boosting Classifier.

Spark's ML pipeline with parameter grid to find the best hyper parameters for each classifier was implemented. Cross validation, confusion matrix and ROC are used to evaluate each classifier's performance.

Among these classifiers, Gradient Boosting and Random Forest are predicting the nearly perfect scores.

Technology Specs: findSpark 3.2.1, pySpark 3.2.1, Spark 3.2.1, Spark MLlib, Anaconda, Jupyter Notebook, Python programming, Seaborn, pyplot libraries, sklearn

2. Project Scope

The project scope is to run the exploratory data analysis using apache spark framework to find the business insights from Lending Club loan data, and to build a learning model using data mining techniques / machine leaning algorithms that will use the historic loan data to learn and helps to identify loans/borrowers which are likely to default.
As per the recent studies, 3-4% of the total loans defaults every year. This is a huge risk for the investors who is funding the loans. Investors require more comprehensive assessment of these borrowers than what is presented by Lending Club to make a smart business decision. Data mining techniques and Machine Learning model/analysis could help predicting the loan default likelihood which may allow investors to avoid loan defaults thus limiting the risk of their investments.

3. Importing necessary Libraries

First, we need to import the necessary libraries required for the project.

```
import findspark
findspark.init("/Users/hemanthharshinee/Downloads/spark-3.1.2-bin-hadoop3.2")

from pyspark import SparkConf
from pyspark.sql import SparkSession
from pyspark.sql import SQLContext
from pyspark.sql import functions as F
from pyspark.sql.functions import isnan, when, count, col, year, quarter, lit, to_date, to_timestamp, concat, avg
from pyspark.sql.types import DateType, TimestampType
from pyspark import SparkContext
from pyspark import SparkConf
from pyspark.ml.feature import Imputer
from pyspark.sql import DataFrameStatFunctions as statFunc
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import IndexToString
from pyspark.mllib.tree import RandomForest, RandomForestModel
from pyspark.ml.classification import GBTClassifier
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.classification import NaiveBayes
from pyspark.ml.classification import LinearSVC
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.mllib.evaluation import BinaryClassificationMetrics
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.feature import PCA


from pyspark.ml.classification import LogisticRegression
from pyspark.mllib.classification import LogisticRegressionWithLBFGS

from pyspark.ml import Pipeline
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

from pyspark.mllib.classification import SVMWithSGD, SVMModel
from pyspark.mllib.regression import LabeledPoint

from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

from sklearn.metrics import roc_curve, auc


%matplotlib inline
```

```
import datetime
import numpy as np
import pandas as pd
from pandas import DataFrame as df
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(color_codes=True)
from scipy import stats

from chart_studio import plotly as py

import plotly.graph_objs as go
from plotly.offline import init_notebook_mode,iplot
init_notebook_mode(connected=True)

import os
memory = '4g'
pyspark_submit_args = ' --driver-memory ' + memory + ' pyspark-shell'
os.environ["PYSPARK_SUBMIT_ARGS"] = pyspark_submit_args

SparkContext.setSystemProperty('spark.executor.memory', '4g')
SparkContext.setSystemProperty('spark.driver.memory', '4g')

spark_conf = SparkConf().setAll(pairs = [('spark.executor.memory', '4g'), ('spark.executor.cores', '3'),
                                         ('spark.cores.max', '3'), ('spark.driver.memory','4g')])

spark = SparkSession.builder.master("local[*]").config(conf = spark_conf)
.appName("Lending-Club Loan Analysis using Pyspark").getOrCreate()
sqlContext = SQLContext(spark)

spark.sparkContext.setLogLevel('ERROR')

import warnings
warnings.filterwarnings('ignore')
```

4. Loading the data to Spark DataFrame

Load Data to Spark DataFrame

In [2]:
```
loanDF = spark.read.csv("loan.csv", header=True, mode="DROPMALFORMED")

loanDF.printSchema()

loanDF_Pandas = pd.read_csv("loan.csv", low_memory=False)
```

```
root
 |-- id: string (nullable = true)
 |-- member_id: string (nullable = true)
 |-- loan_amnt: string (nullable = true)
 |-- funded_amnt: string (nullable = true)
 |-- funded_amnt_inv: string (nullable = true)
 |-- term: string (nullable = true)
 |-- int_rate: string (nullable = true)
 |-- installment: string (nullable = true)
 |-- grade: string (nullable = true)
 |-- sub_grade: string (nullable = true)
 |-- emp_title: string (nullable = true)
 |-- emp_length: string (nullable = true)
 |-- home_ownership: string (nullable = true)
 |-- annual_inc: string (nullable = true)
 |-- verification_status: string (nullable = true)
 |-- issue_d: string (nullable = true)
 |-- loan_status: string (nullable = true)
 |-- pymnt_plan: string (nullable = true)
 |-- url: string (nullable = true)
 |-- desc: string (nullable = true)
 |-- purpose: string (nullable = true)
 |-- title: string (nullable = true)
 |-- zip_code: string (nullable = true)
 |-- addr_state: string (nullable = true)
 |-- dti: string (nullable = true)
 |-- delinq_2yrs: string (nullable = true)
 |-- earliest_cr_line: string (nullable = true)
 |-- inq_last_6mths: string (nullable = true)
 |-- mths_since_last_delinq: string (nullable = true)
 |-- mths_since_last_record: string (nullable = true)
 |-- open_acc: string (nullable = true)
 |-- pub_rec: string (nullable = true)
 |-- revol_bal: string (nullable = true)
 |-- revol_util: string (nullable = true)
 |-- total_acc: string (nullable = true)
 |-- initial_list_status: string (nullable = true)
 |-- out_prncp: string (nullable = true)
 |-- out_prncp_inv: string (nullable = true)
 |-- total_pymnt: string (nullable = true)
 |-- total_pymnt_inv: string (nullable = true)
 |-- total_rec_prncp: string (nullable = true)
 |-- total_rec_int: string (nullable = true)
 |-- total_rec_late_fee: string (nullable = true)
 |-- recoveries: string (nullable = true)
 |-- collection_recovery_fee: string (nullable = true)
 |-- last_pymnt_d: string (nullable = true)
 |-- last_pymnt_amnt: string (nullable = true)
 |-- next_pymnt_d: string (nullable = true)
 |-- last_credit_pull_d: string (nullable = true)
 |-- collections_12_mths_ex_med: string (nullable = true)
 |-- mths_since_last_major_derog: string (nullable = true)
 |-- policy_code: string (nullable = true)
 |-- application_type: string (nullable = true)
 |-- annual_inc_joint: string (nullable = true)
 |-- dti_joint: string (nullable = true)
 |-- verification_status_joint: string (nullable = true)
 |-- acc_now_delinq: string (nullable = true)
 |-- tot_coll_amt: string (nullable = true)
 |-- tot_cur_bal: string (nullable = true)
 |-- open_acc_6m: string (nullable = true)
 |-- open_il_6m: string (nullable = true)
 |-- open_il_12m: string (nullable = true)
 |-- open_il_24m: string (nullable = true)
 |-- mths_since_rcnt_il: string (nullable = true)
 |-- total_bal_il: string (nullable = true)
 |-- il_util: string (nullable = true)
 |-- open_rv_12m: string (nullable = true)
 |-- open_rv_24m: string (nullable = true)
 |-- max_bal_bc: string (nullable = true)
 |-- all_util: string (nullable = true)
 |-- total_rev_hi_lim: string (nullable = true)
 |-- inq_fi: string (nullable = true)
 |-- total_cu_tl: string (nullable = true)
 |-- inq_last_12m: string (nullable = true)
```

5. Exploratory Data Analysis

The data used for this project is the structured data with few missing/null values. This data consists of 80+ features of three distinct types: continuous, categorical, ordinal. I have gathered business domain knowledge about the data to deal with data cleaning and missing data imputation. During the preliminary exploration of data, it was noticed that some of the features with more than 50% of the missing data, I have planned to drop these features while data processing for learning model.

For data analysis phase, I have converted some of the feature type to relevant primitive types. I have handled data cleaning and imputation part while preparing data for learning model. For preliminary data cleanup, spark data frame transformation APIs was used to convert the feature data types.

As part of exploratory data analysis, I have tried to find any interesting fact and findings about the loans from the historical loan data. This analysis helped to develop my understanding about data and its distribution patterns. In addition, this assisted to select the most effecting features and develop business rules for missing data imputation.

Find the Feature columns which has more than 50% empty data

```
def findMissingValueCols(df):
    missingValueColumns = []
    for column in df.columns:
        nullRows = df.where(col(column).isNull()).count()
        print(column, "--", nullRows)
        if nullRows > loanDFRows*0.5 : # i.e. if ALL values are NULL
            missingValueColumns.append(column)
    return missingValueColumns
```

5.1 Analyzing Loan amount and Interest rates:

By using Spark's transformation and action API's, I have aggregated the data by loan amount and loan interest rate and plotted the distribution plot and box plot.

```
%matplotlib inline

loanDF = loanDF.withColumn("loan_amnt",loanDF["loan_amnt"].cast('float'))
loanDF = loanDF.withColumn("int_rate",loanDF["int_rate"].cast('float'))

tmp = loanDF.select("loan_amnt", "int_rate").toPandas()

fig, ax =plt.subplots(2,2, figsize=(18,12))
plt.subplots_adjust(hspace = 0.4, top = 0.8)

# Loan amount distribution plots
sns.distplot(tmp.loan_amnt, fit=stats.gamma, axlabel="Loan Amount", label="Loan Amount Frequency distribution",
             ax=ax[0][0])
sns.boxplot(x=tmp.loan_amnt, ax=ax[0][1])

# Interest rates distribution plots
sns.distplot(tmp.int_rate, fit=stats.gamma, axlabel="Interest Rate", label="Interest Frequency distribution",
             ax=ax[1][0])
sns.boxplot(x=tmp.int_rate, ax=ax[1][1])

fig.show()

fig.savefig("LoanDistribution.pdf")
```

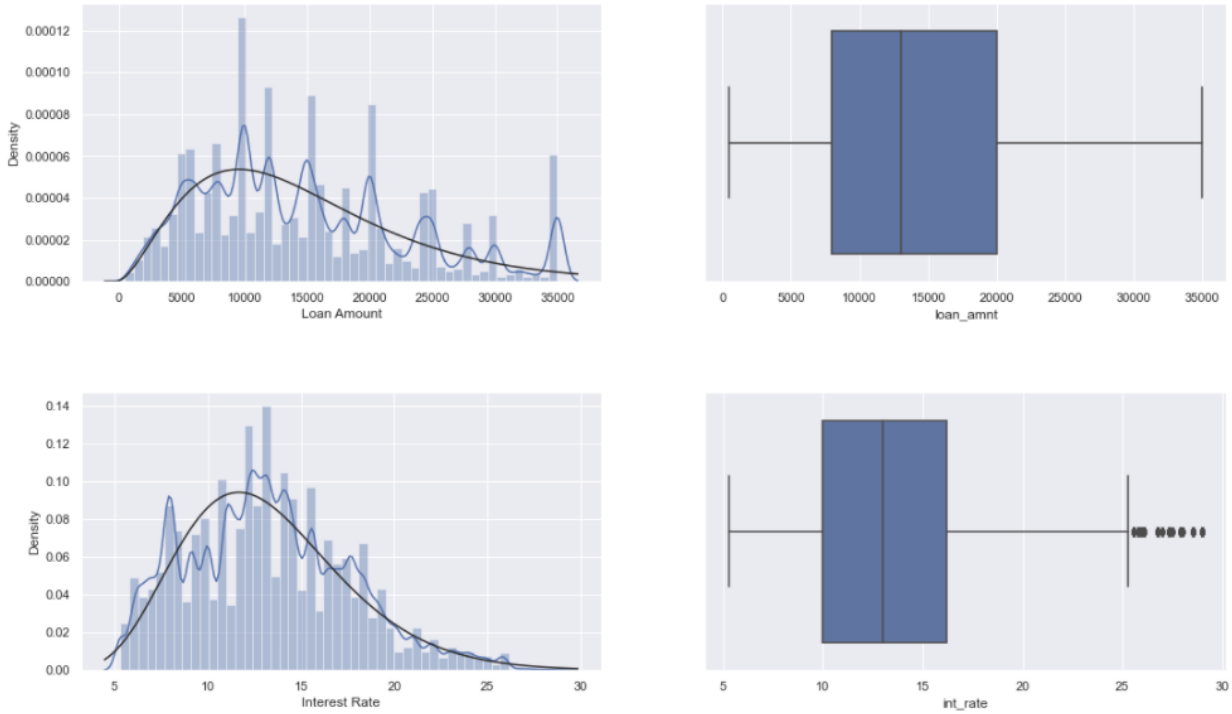


Figure 1: Distribution of Loan and Interest

From above Figure 1, we can conclude that both Interest Rate & Loan Amount is normally distributed, slightly right skewed. i.e., most of the customers are seeking loan amount ranging 5-15K and their interest rate is ranging from 10-15%. The loan amount shows no outliers point, since it fits perfectly within the five-point summary. However, Interest rates shows few outliers falling outside the 1.5IQR range. Those customers may have **poor credit ratings**, therefore **high interest rate.**

5.2 Analyzing Loans Interest rates over time:

To analyze how are loan book is changing over time in terms of loan amount and interest rate, I first created a new column for quarterly date range. Spark's *withColumn* and *to_date* APIs is used to extract this information from the loan issue date column. After which the data is aggregated over quarterly date range and plotted in the point plot. To get the total loan amounts over the time we grouped by on the date variable and aggregate all the loans amount to get the total loans. Customers loan requirements are calculated over the time (Median loan amount and median interest) by using *percentile_approx* function.
This function is faster way to compute the quantile range in very large set of data where data stored in multiple partition. The underline algorithm usages

approximation technique to find the quantile range instead of running the computation in all the partitions.

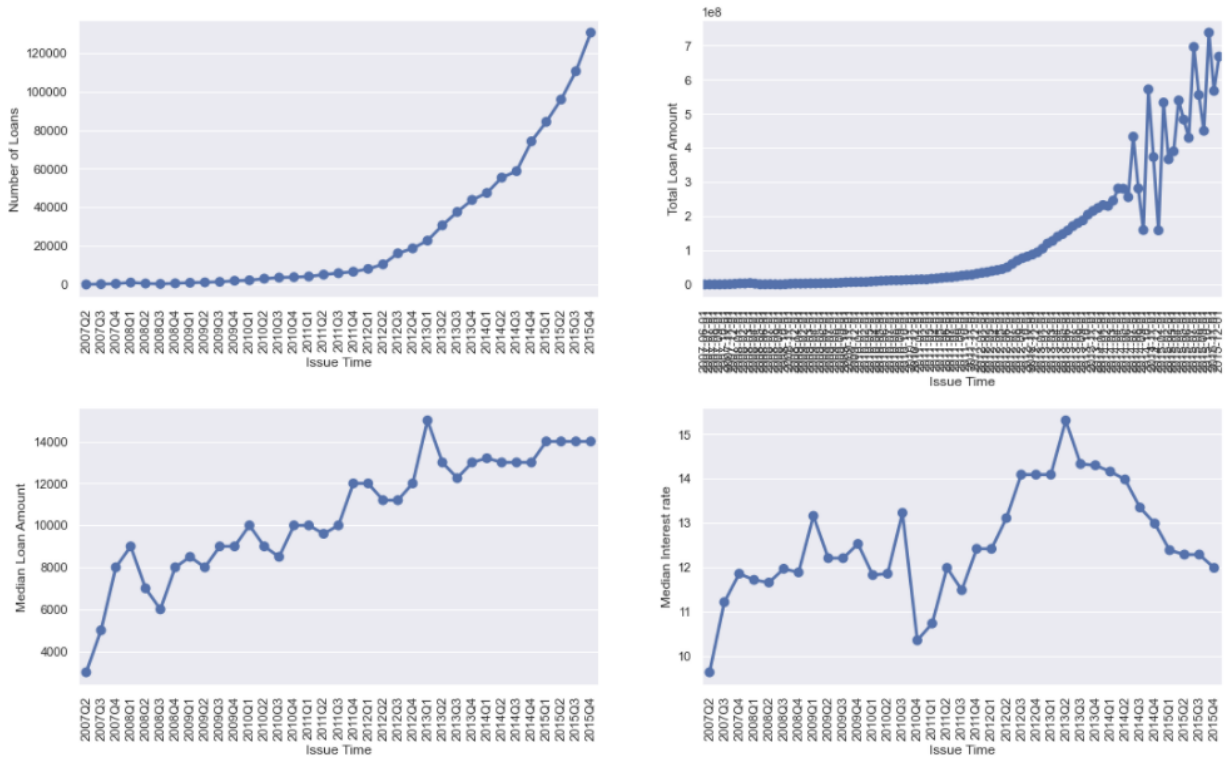


Figure 2:Analyzing Loans Interest rates over time

The first two plots show how Number of Loans and Total Loan Amount is growing over Issue Time, this gives an indication of growing borrowers at lending club platform. The next two plots are between Median Loan Amount & Median Interest rate over Issue Time. This indicates that the borrower's loan requirements are increasing over time so as the median interest rates. However, there is a sharp decline in the interest rate during 2010, which might be due to 2009 fiscal crisis.

5.3 Analyzing Loans over loan status

We analyze the loans according to their current status. Complex spark data frame operations are used to know the distribution of total accounts for each loan status and distribution of loan amount with the probability density for each loan value over the loan status. The density distribution is used to analyze any anomalies/outliers specific to the status of loans.

To show the distribution of loan amount, the violin plot was used (figure 5). A *Violin Plot* is used to visualize the distribution of the data and its probability

density. This chart is a combination of a Box Plot and a Density Plot. The density plot is rotated and placed on both sides of the box plot, to show the distribution shape of the data.

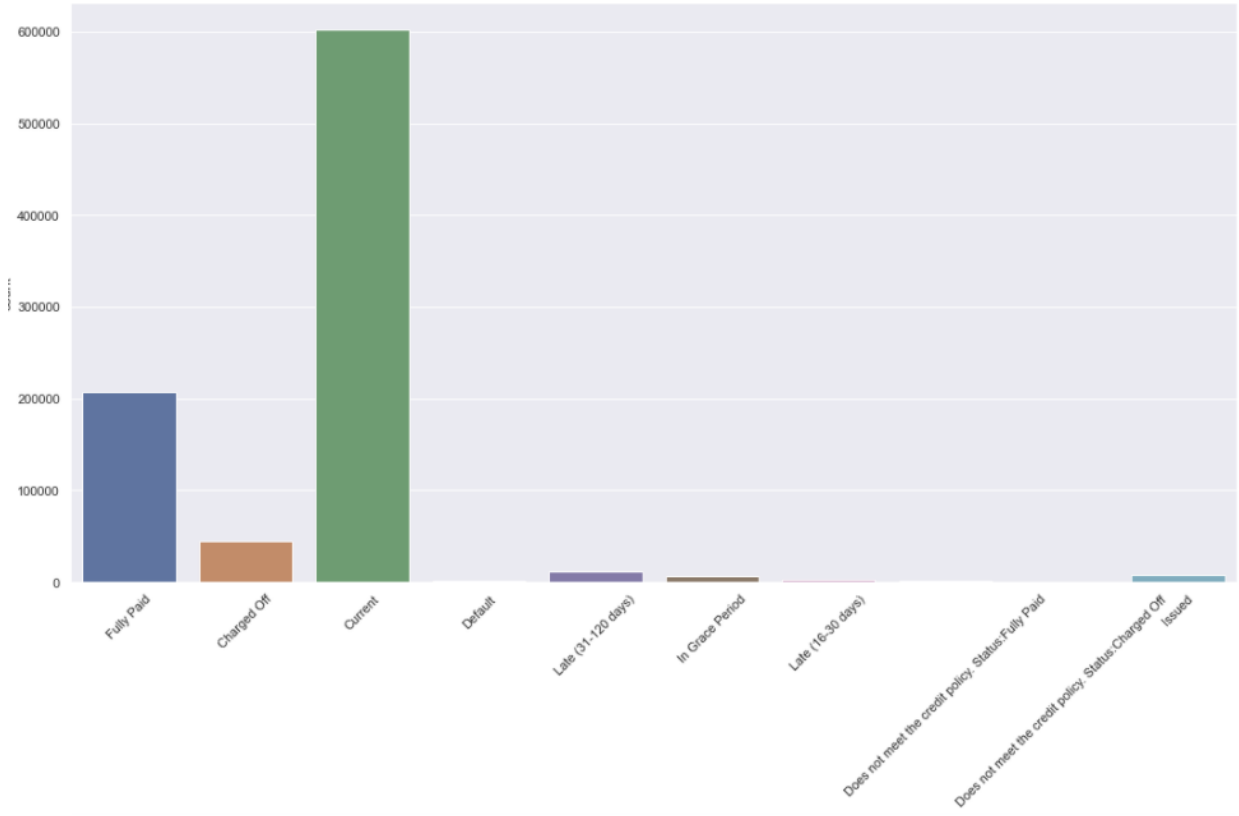


Figure 3: Bar plot for Analyzing Loans over loan status

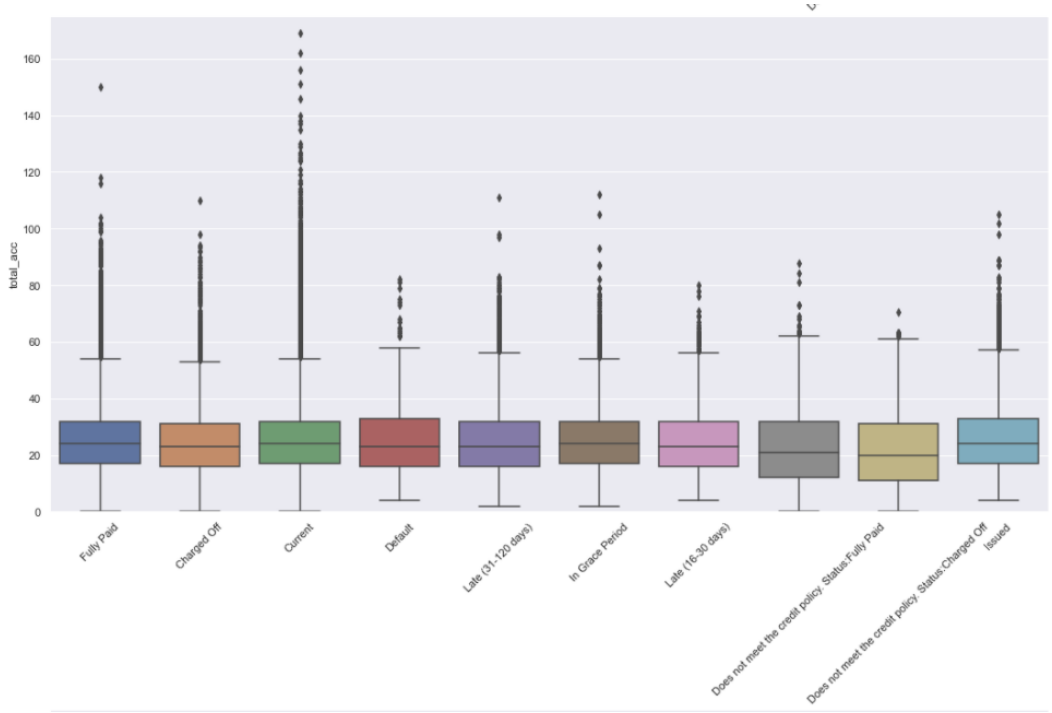


Figure 4: Box plot for Analyzing Loans over loan status

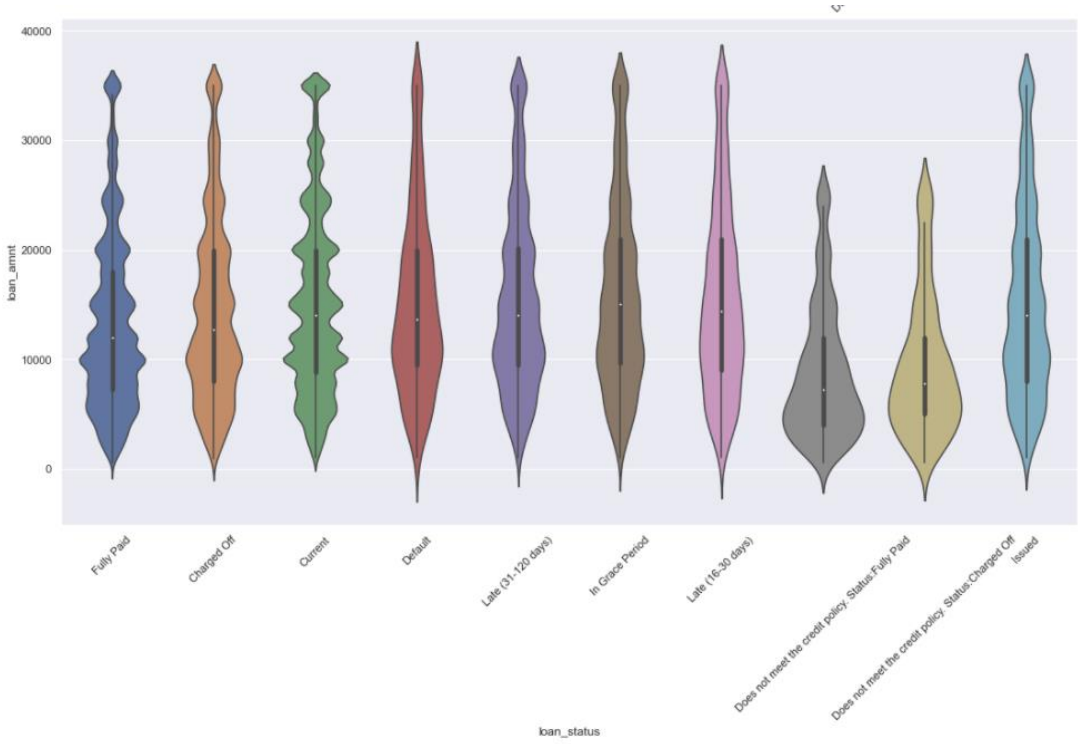


Figure 5: Violin Plot for Analyzing Loans over loan status

By running a group by query, we found out the number of loans for each loan status. The results are listed in following table.

Loan Counts group by loan_status

```
loanDF.groupBy('loan_status').count().sort("count", ascending=False).show()
```

```
+--------------------+------+
|         loan_status| count|
+--------------------+------+
|             Current|601778|
|          Fully Paid|207720|
|         Charged Off| 45248|
|  Late (31-120 days)| 11591|
|              Issued|  8460|
|     In Grace Period|  6253|
|   Late (16-30 days)|  2357|
|Does not meet the...|  1987|
|             Default|  1219|
|Does not meet the...|   761|
+--------------------+------+
```

We also analyzed about the verification status of the loan which were defaulted. We achieve this result by running a group by aggregations on *verification_status* and default *loan status*.

Default Loan Count group by varification_status

```
loanDF.filter("loan_status == 'Default'").groupBy('verification_status').count().show()
```

```
+-------------------+-----+
|verification_status|count|
+-------------------+-----+
|           Verified|  479|
|    Source Verified|  462|
|       Not Verified|  278|
+-------------------+-----+
```

5.4 Analysis of Grades and Subgrades: Analyzing loan amount and interest rate quantile summary for each grade, factored over sub grade

Based on borrower's credit score, credit history, desired loan amount and the borrower's debt-to-income ratio, Lending Club determines whether the borrower is credit worthy. After that they assign a credit grade that determines payable interest rate and fees to their approved loans. These grades are assigned within an alphabetical range from A to G. Each of these letter grades has five finer-grain sub-grades, numbered 1 to 5, with 1 being the highest category with-in the grade. Loan interest rates is inverse proportionate to the credit grade. 'A' being the highest grade, therefore low interest rate and vice-a-versa.

In figure 6, we can see a linear relationship between loan amount and customer credit ratings, notice here that requested loan amount is slightly higher for the low rating customers. This indicates that higher rating borrowers are financially stronger and require lesser loan amount in general.

In figure 7, we can see that the median interest rates increase for low credit rating customers.

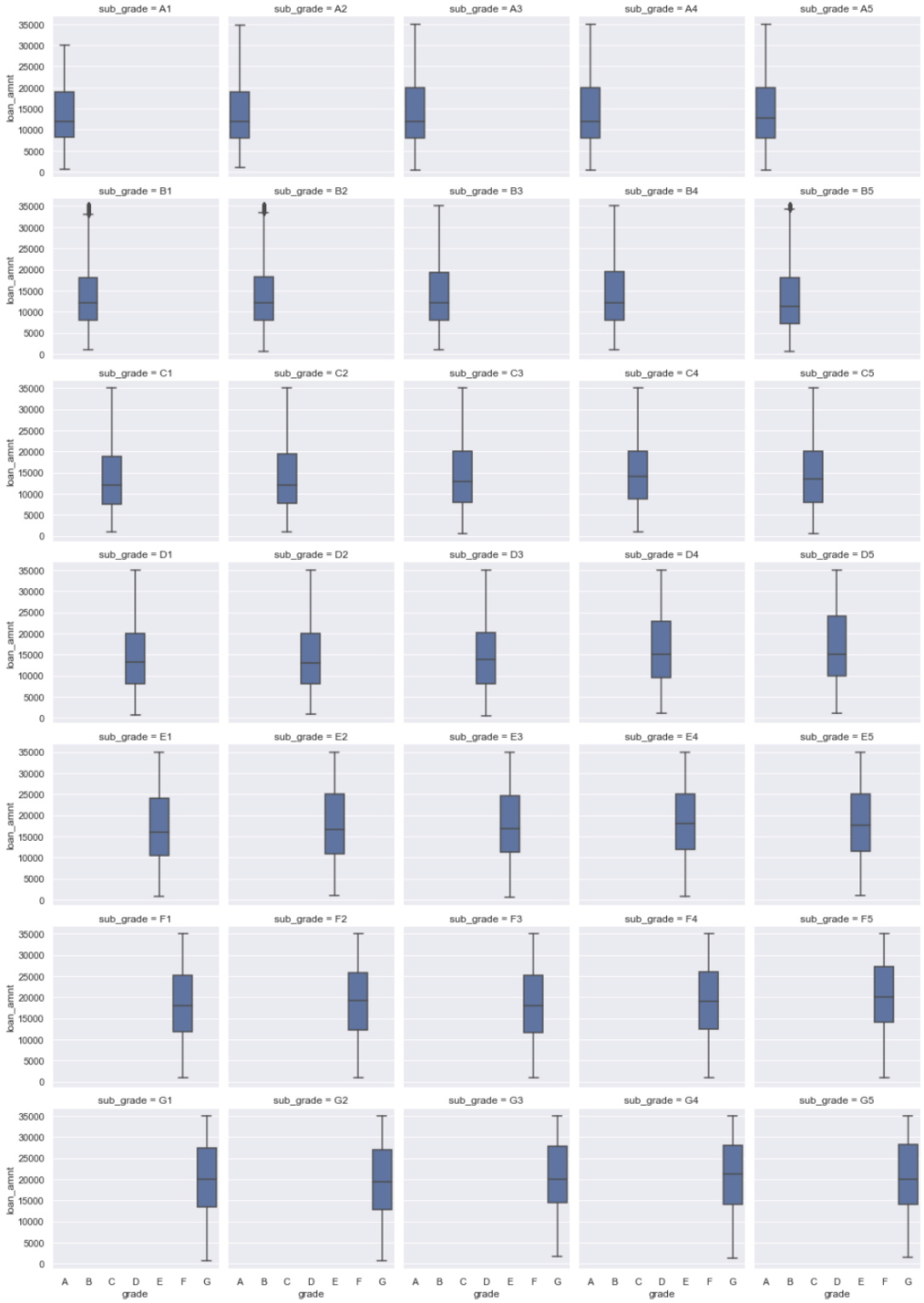


Figure 6: Analyzing loan amount distribution for each grade, factored over sub grade.

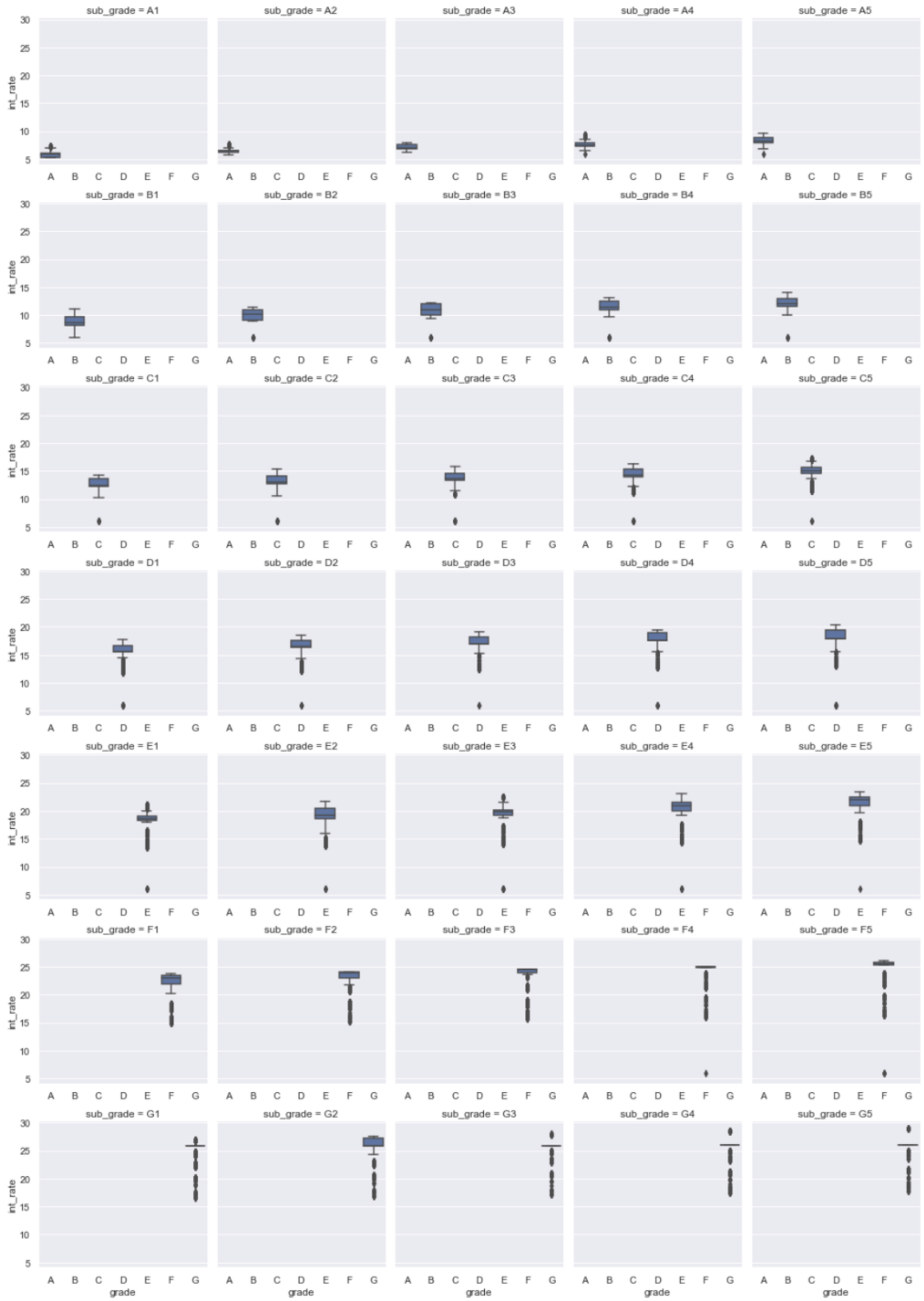


Figure 7: Analyzing interest rate distribution for each grade, factored over sub grade.

5.5 US States map with the total loan amount

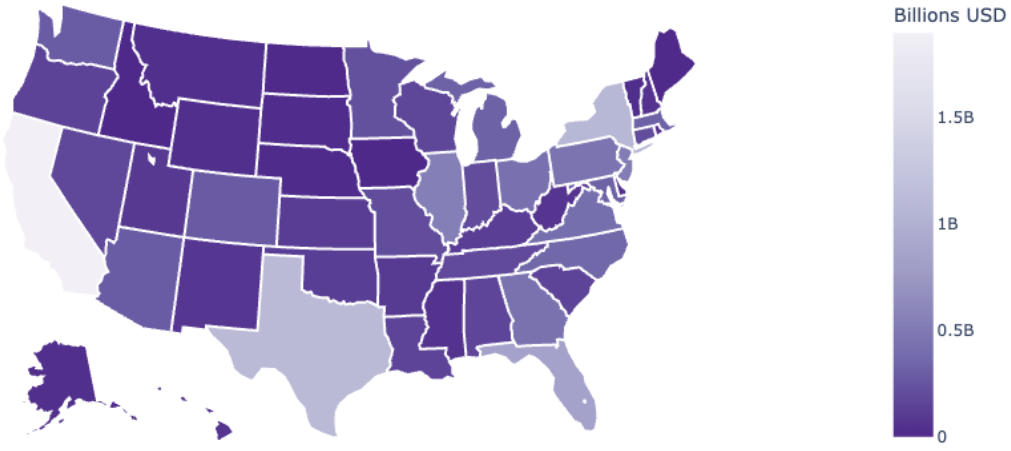


Figure 8:US states map with the total loan amount

5.6 US States map with median interest rates

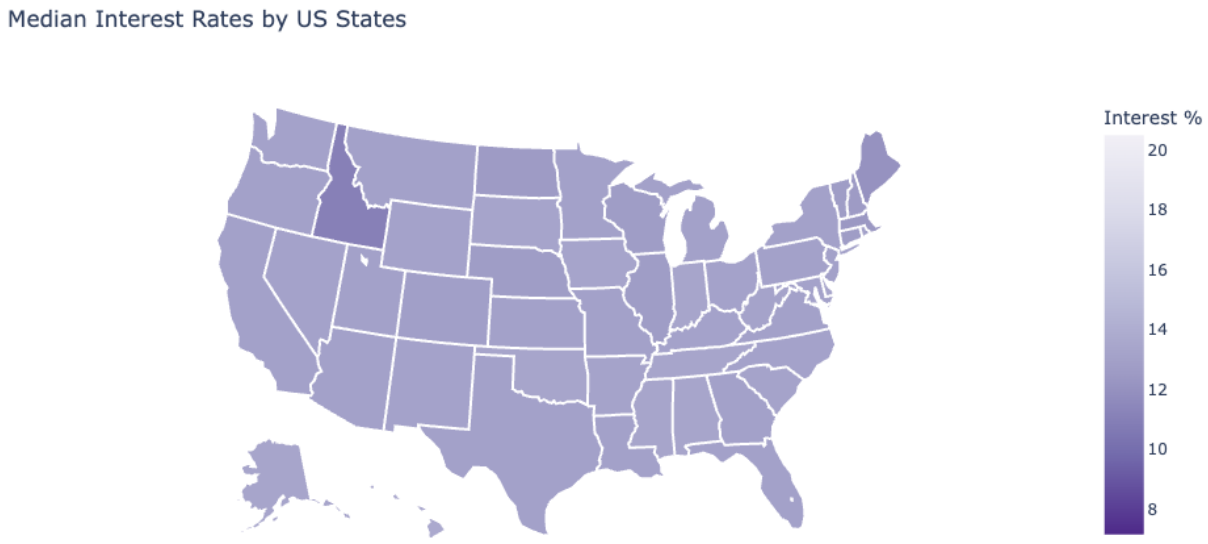


Figure 9:US states map with the median interest rates

Note: - To further analyze the state wise effect on loan default, we tried to merge the census data for median house-hold income and inflation rate for each state. But the data we collected was not cleaned and had lot of missing states.

5.7 Total loan amount by income range and loan status

Bins are created for continuous variables to define the income range to enhance interpretation of results. To perform the analysis of income range, 15 bins for income range were created. To create these bins, we used spark's ***Bucketizer*** API to create the bins. Bucketizer transforms a column of continuous features to a column of feature buckets, where the buckets are specified by users. However, this method was adding an extra computation time in our processing, so we create these bins by writing our own customize code.
The income range bins were created in a way that our data can fit in to the original distribution (Gaussian) of continuous feature. Using the new range column, the following were calculated:

- Total number of loans grouped by income range and loan status
- Total loan amount by income range and loan status

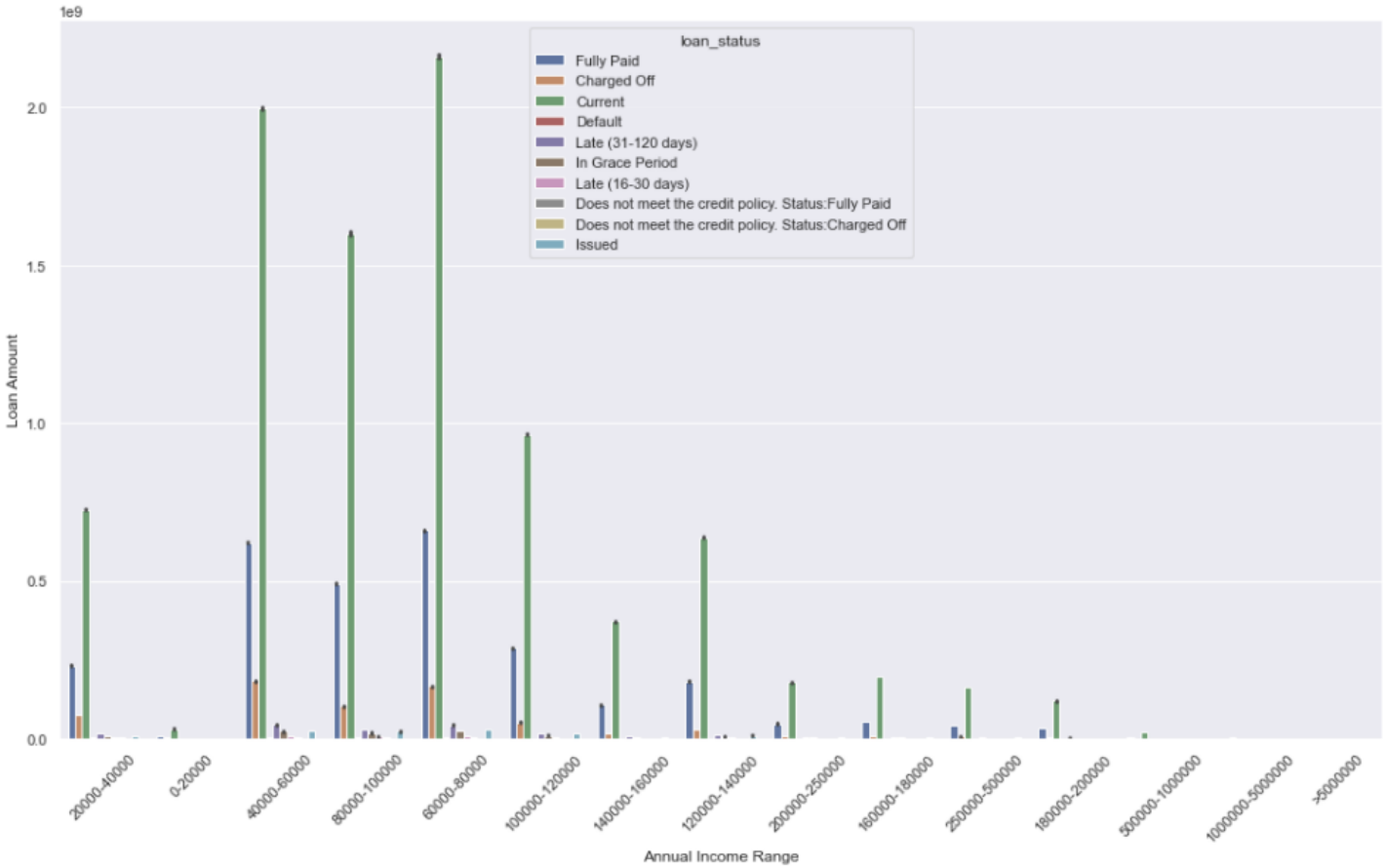


Figure 10:Total loan amount by income range and loan status

5.8 Analyzing Loan Amount and interest rate over customers employment length (With the loan term).

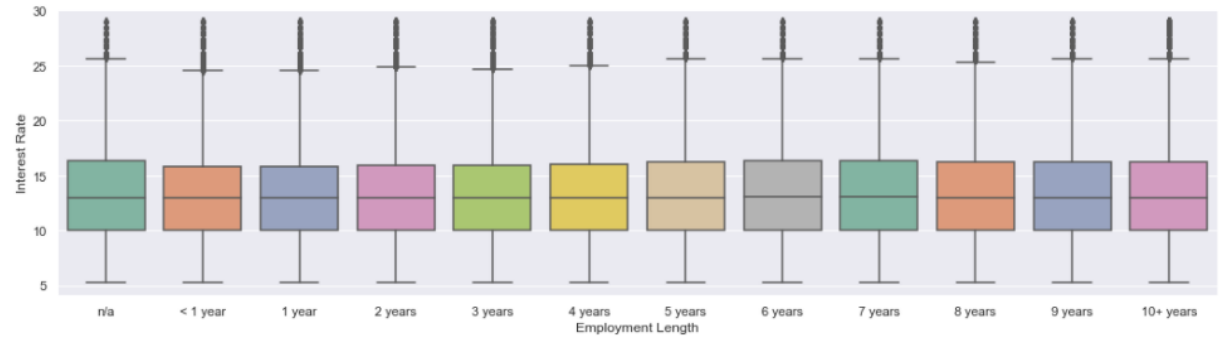


Figure 11: Analyzing Loan Amount and interest rate over customers employment length

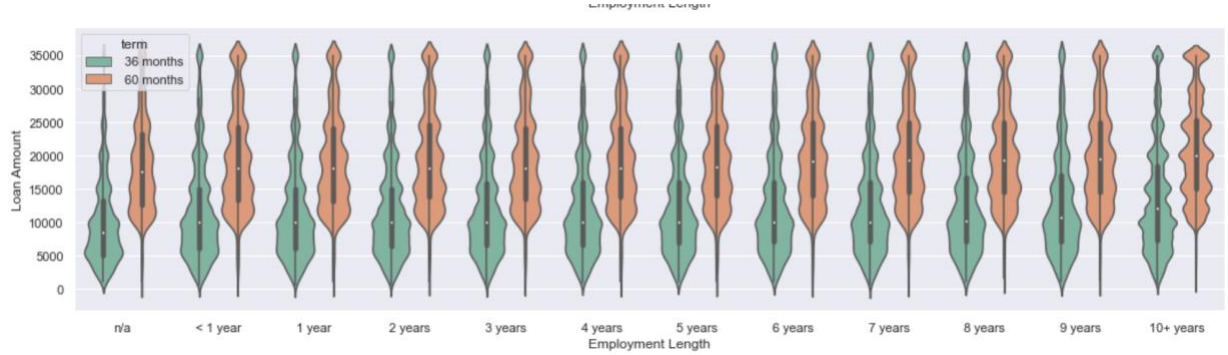
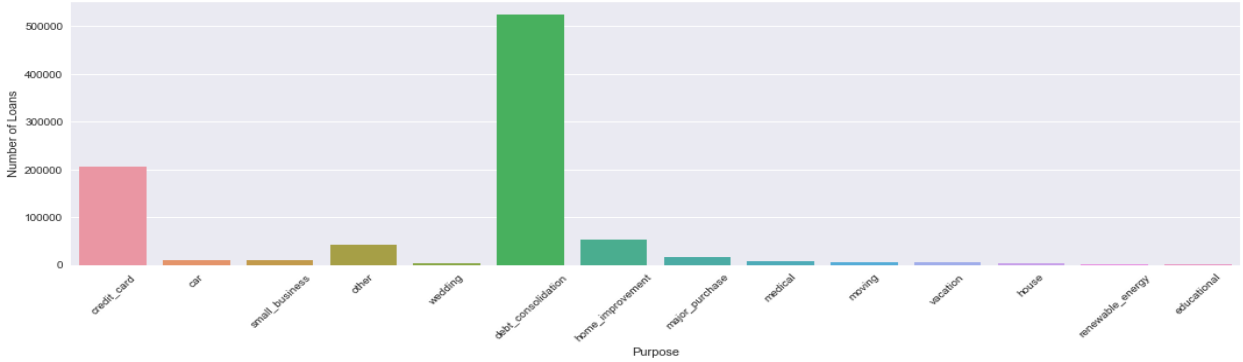


Figure 12: Analyzing Loan Amount and interest rate over customers employment length

5.9Analyzing loans by its purpose

In figure 13 below, the first plot shows number of loans by its purposes. The second plot shows the Loan amount with its distribution pattern by purpose and the third plot shows interest rate with its distribution pattern by purpose.



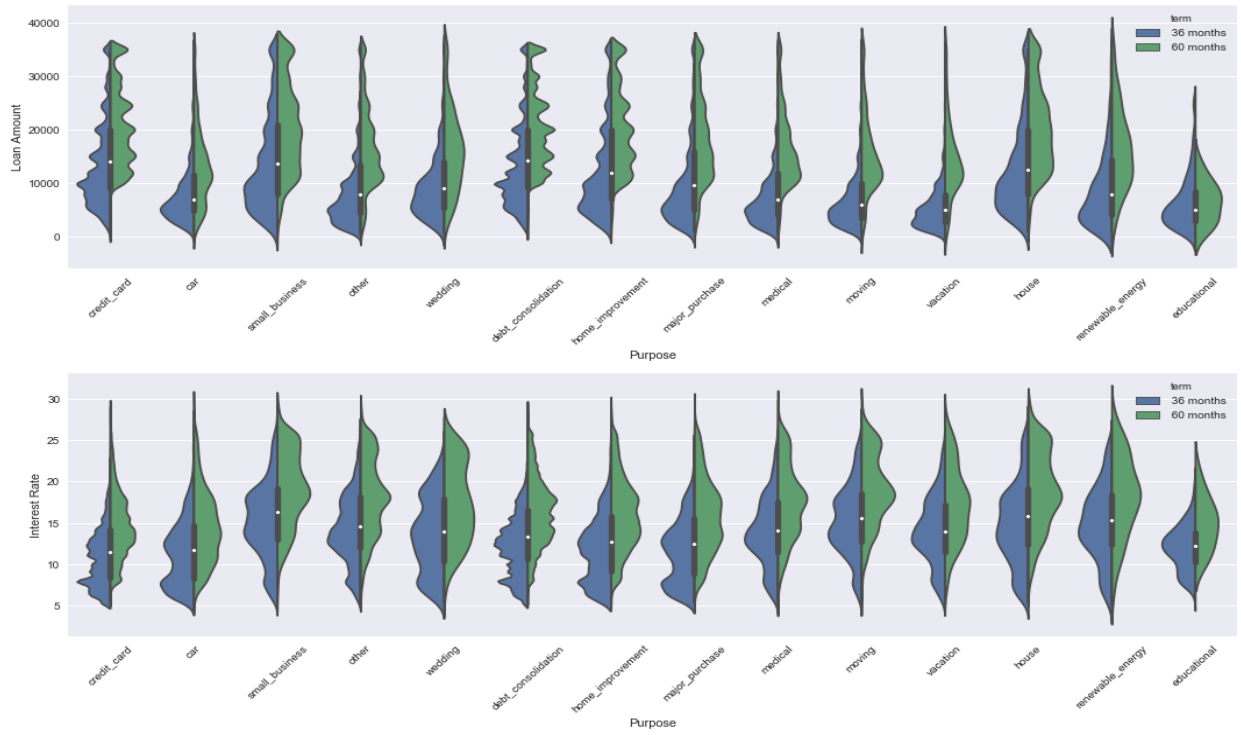
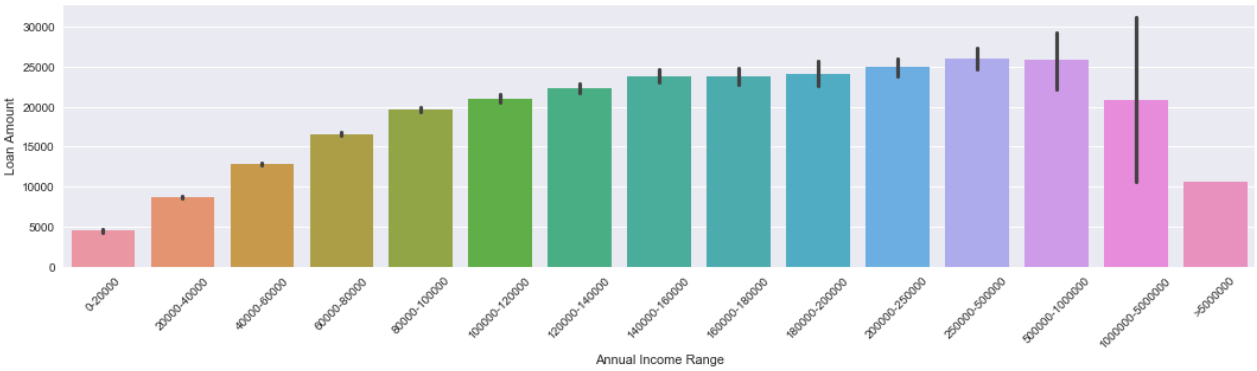


Figure 13: Analyzing loans by its purpose

5.10 Analyzing Default Loans

Bins for interest rates are created as well in the same way as mentioned as above. Loan status which are in following status are considered as defaulted –

- Default, Late (31-120 days),
- In Grace Period, Late (16-30 days),
- Does not meet the credit policy.
- Status: Charged Off



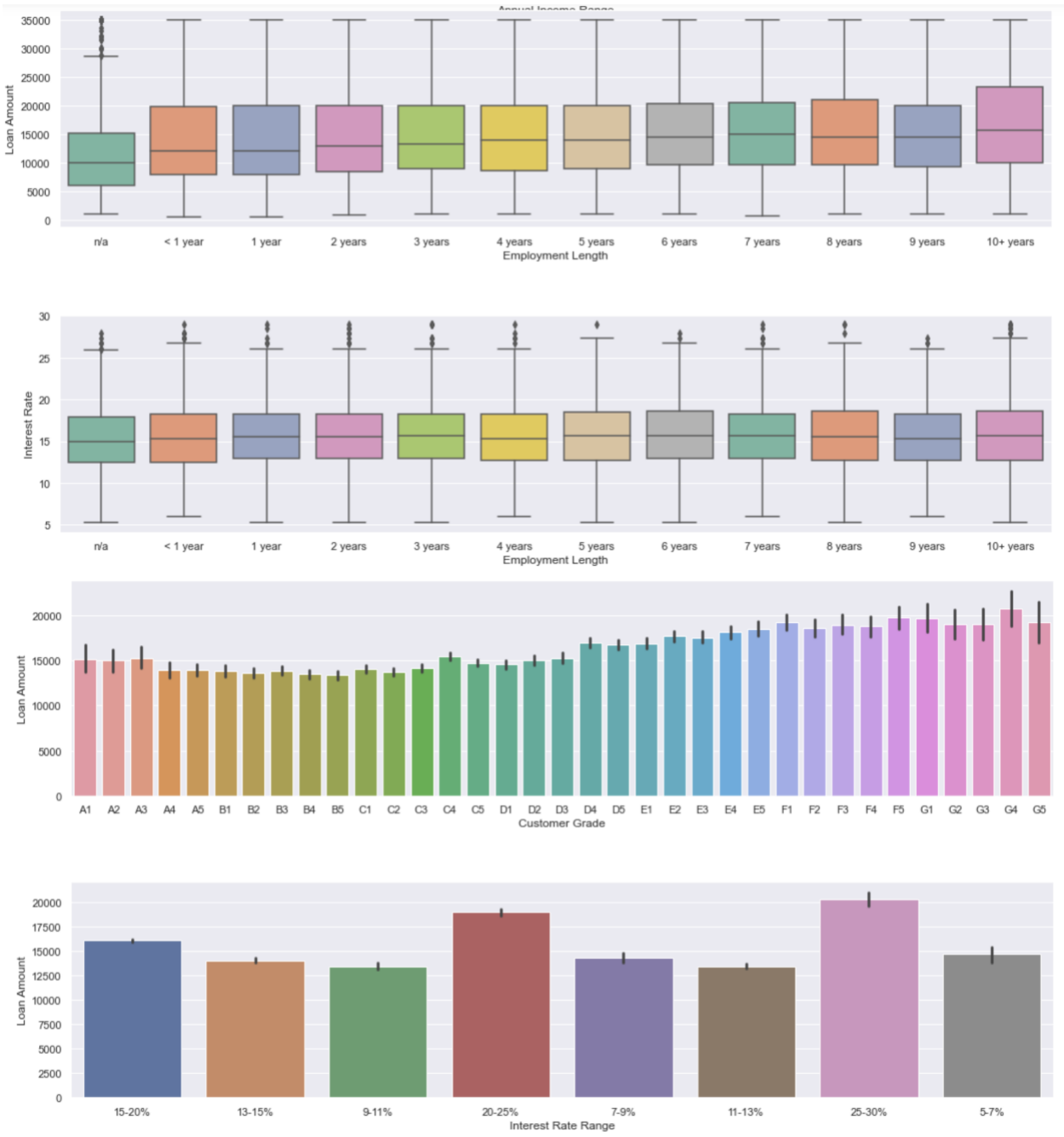


Figure 14:Analyzing Default loans

In above figure 14, the first plot shows the loan amount for each income range that we have created. We can see that the requirement for loan amount is increasing for income range from 120K to 500K, after which it decreases. The second plot shows loan amount according to the employment length. It is observed that people who have larger employer length usually borrow large amount of loan, which is mostly for mortgage purpose. The third plot shows the interest rate for each employment

length. The fourth plot shows the loan amount according to subgrade. And the final 5th plot shows the loan amount for each interest rate range.

6. Data Cleaning, Encoding and Missing Data Imputation

Features present within the dataset provided an ample amount of information which we could use to identify relationships and gauge their effect upon the success or failure of a borrower fulfilling the terms of their loan agreement. We required only the variables that had a direct or indirect response to a borrower's potential to default. To achieve this, the data is prepared by choosing select variables that would best fit these criteria.

6.1 Data Cleaning:

As a first step, we removed all unique id fields which is to represent a loan request, since it does not contribute in data analysis or model building. Next, we removed all the features which has more than 50% missing data. After which removed few categorical features which had only one category in the data.

There were few features in our dataset which were specific to the defaulted loans, we removed all the features since they possibly can create leakage in our model.

In [22]:
```
# Cleaning up the data

######### 1. Removing all the features which has more than 50% of the data empty ##########
# Temporary setting these hard coded values. (Above section takes lot of time to run)
missingValueColList = ['desc', 'mths_since_last_delinq', 'mths_since_last_record', 'mths_since_last_major_derog', 'annu
loanDFForModel = loanDF.drop(*missingValueColList)

######### 2. Removing unique ID columns ##########
# Dropping ID & date columns (Unique id's, Don't help much in data analysis/modelling)
loanDFForModel = loanDFForModel.drop("id", "member_id", "issue_d")

######### 3. Removing Other insignificant columns ##########
# application_type has only INDIVIDUAL, can be removed.
# pymnt_plan & initial_list_status has only one category "n" & "f". Keeping state feature instead of zip_code.
# removing date fileds as well. policy_code has only one category "1"
loanDFForModel = loanDFForModel.drop("emp_title", "url", "title", "zip_code", "earliest_cr_line", "last_pymnt_d",
                                     "next_pymnt_d", "last_credit_pull_d", "policy_code" )


######### 4. Missing data imputation for tot_cur_bal ##########
# 90% of the missing data in "tot_cur_bal", "tot_coll_amt" column can be filled with 0 cause their loan status is "Full
loanDFForModel = loanDFForModel.withColumn("tot_cur_bal", when((col("tot_cur_bal").isNull() &
                                                    col("loan_status").isin("Fully Paid", "Charged Off")), 1
                                           .otherwise(col("tot_cur_bal")))

loanDFForModel = loanDFForModel.withColumn("tot_coll_amt", when((col("tot_coll_amt").isNull() &
                                                    col("loan_status").isin("Fully Paid", "Charged Off")), 1
                                           .otherwise(col("tot_coll_amt")))

# Inputing mean value for "total_rev_hi_lim"
mean = int(loanDFForModel.select(avg("total_rev_hi_lim")).take(1)[0][0])
loanDFForModel = loanDFForModel.withColumn("total_rev_hi_lim", when(col("total_rev_hi_lim").isNull(), lit(mean))
                                           .otherwise(col("total_rev_hi_lim")))

######### 5. Removing loan observations which still have missing data. (~ 0.8% records) ##########
loanDFForModel = loanDFForModel.dropna(how="any")
```

```
######### 6. Adding the lable column to dataframe. 1- defalut and 0-paid/current ##########
loanDFForModel = loanDFForModel.withColumn("isDefault", when(col("loan_status").isin("Default","Charged Off", "Late (31
                                                                "Does not meet the credit policy. Status:Charged O
                           .otherwise(0))

######### 7. Changing the feature datatype from string to numeric ##########
loanDFForModel = loanDFForModel.withColumn("loan_amnt",loanDFForModel["loan_amnt"].cast('float'))
loanDFForModel = loanDFForModel.withColumn("funded_amnt",loanDFForModel["funded_amnt"].cast('float'))
loanDFForModel = loanDFForModel.withColumn("funded_amnt_inv",loanDFForModel["funded_amnt_inv"].cast('float'))
loanDFForModel = loanDFForModel.withColumn("int_rate",loanDFForModel["int_rate"].cast('float'))
loanDFForModel = loanDFForModel.withColumn("installment",loanDFForModel["installment"].cast('float'))
loanDFForModel = loanDFForModel.withColumn("annual_inc",loanDFForModel["annual_inc"].cast('float'))
loanDFForModel = loanDFForModel.withColumn("dti",loanDFForModel["dti"].cast('float'))
loanDFForModel = loanDFForModel.withColumn("delinq_2yrs",loanDFForModel["delinq_2yrs"].cast('float'))
loanDFForModel = loanDFForModel.withColumn("inq_last_6mths",loanDFForModel["inq_last_6mths"].cast('float'))
loanDFForModel = loanDFForModel.withColumn("open_acc",loanDFForModel["open_acc"].cast('float'))
loanDFForModel = loanDFForModel.withColumn("pub_rec",loanDFForModel["pub_rec"].cast('float'))
loanDFForModel = loanDFForModel.withColumn("revol_bal",loanDFForModel["revol_bal"].cast('float'))
loanDFForModel = loanDFForModel.withColumn("revol_util",loanDFForModel["revol_util"].cast('float'))
loanDFForModel = loanDFForModel.withColumn("total_acc",loanDFForModel["total_acc"].cast('float'))

######### 8. Finally removing loan_status ##########
#loan_status is used to create the class lable, removing it to avoid data leakage.
loanDFForModel = loanDFForModel.drop("loan_status")


######### 8. Removing the fileds which are related to the current loan ##########
loanDFForModel = loanDFForModel.drop("out_prncp", "out_prncp_inv", "total_pymnt", "total_pymnt_inv", "total_rec_prncp",
                                     "total_rec_int", "total_rec_late_fee", "recoveries", "collection_recovery_fee",
                                    "last_pymnt_amnt", "collections_12_mths_ex_med", "acc_now_delinq", "tot_coll_amt",
                                    "tot_cur_bal", "total_rev_hi_lim")
```

6.2 Encoding:

After data cleaning, we encode all the categorical columns using in-built spark API's. Our data consist of these types of categorical features: ordinal, nominal and binary. We use *StringIndexer API* to convert the ordinal features such as borrowers experience level and *VectorAssembler API* to convert the rest nominal features. Following which, we binary encoded the variables with binary values.

Binary Encoding for Categorical Feature - "term", "initial_list_status", "application_type", "pymnt_plan"

These features have only two categories

```
indexer = StringIndexer(inputCol="term", outputCol="termIndex", handleInvalid="keep")
loanDFForModel = indexer.fit(loanDFForModel).transform(loanDFForModel)

indexer = StringIndexer(inputCol="initial_list_status", outputCol="initial_list_statusIndex", handleInvalid="keep")
loanDFForModel = indexer.fit(loanDFForModel).transform(loanDFForModel)

indexer = StringIndexer(inputCol="application_type", outputCol="application_typeIndex", handleInvalid="keep")
loanDFForModel = indexer.fit(loanDFForModel).transform(loanDFForModel)

indexer = StringIndexer(inputCol="pymnt_plan", outputCol="pymnt_planIndex", handleInvalid="keep")
loanDFForModel = indexer.fit(loanDFForModel).transform(loanDFForModel)
```

6.3 Missing data imputation:

While doing exploratory data analysis, we developed some business rules which helped us to handle the missing data imputation such as 90% of the missing data in "*tot_cur_bal*", "*tot_coll_amt*" column can be filled with 0 cause their loan status is either "Fully Paid" or "Charged Off". Median value can be imputed for missing values in *total_rev_hi_lim*, since 86% of the data has "0".
After implying all imputation rules, we removed the observations from data set which still has missing data (which is approximately 0.8% of total records).

6.4 Label creation:

Our data doesn't have the label column by itself but there is combination of the features which represent the default loan. For our classification model we needed to add a class label variable to our data set. We added a new column "*classlabel*" and populate the values based in the combinations of features such as *loan_status, tot_curr_bal, delig_month* etc. The class count for each class is shown in following table.

```
loanDFForModel.groupby("ClassLable").count().show()
```

```
+----------+------+
|ClassLable| count|
+----------+------+
|         1| 60153|
|         0|819919|
+----------+------+
```

Additionally, we classified any loan that defaulted, were charged off, or were late on payments was classified as negative examples (defaulted), while we classified any loan that was fully paid or current was classified as positive examples (non-defaulted). After labelling our data, we dropped the features which were used for labelling at the first place.

7. Learning Algorithm Implementation

For this project, four different classifier learning models namely Logistic Regression, Naïve Bayes, Random Forest, and Gradient Boosting Classifier are built and tested. Each of the learning models was comprised by a different combination of the hyper parameters. Each of these classifiers are implemented in Apache Spark's machine learning framework (MLlib) which uses the computational power from spark-core, we used Pyspark's libraries for these implementations. Pyspark is an API developed in python for spark programming and writing spark applications in Python style, but underlying execution model is the distributed in-memory computing framework provided by spark-core.

Furthermore, to tune/optimize the learning model's hyper parameters, Spark's machine learning pipeline using parameter grid search to optimize the learning hyper parameters was implemented. A Spark ML Pipeline is specified as a sequence of stages, and each stage is either a transformer (Feature Vectorization, Selection, Feature Encoding etc.) or an Estimator (Learning classifier, Model Evaluator). These stages are run in order, and the input spark DataFrame is transformed as it passes through each stage. This pipeline internally creates the DAG (directed acyclic graph) of stages consist of transformers and estimators. To ensure faster processing speed, ML Pipeline employs spark's parallel computation power to run each stage on individual partition of the data and pass-through processed data to next stage instead of computing a stage on the whole input data. In case of a node failure, Spark usages the DAG (Logical execution plans – Lineage) to recover the data and re-commute the stages.

To choose best hyper parameter, cross validation with spark ML pipeline is implemented and each model is evaluated using the binary model evaluator. Binary model evaluator is the built-in implementation in spark ml package, which helps to validate the cross-validation results.
CrossValidator class provides a function "*bestmodel*" which returned the "*pipelinemodel*" with the best selected grid parameters (Hyper parameters). This pipeline model can then be cast to the original learning classifier, which was added in the pipeline as estimator. Please note, that this functionality has a different implementation in pyspark (than Scala/java), where one must call the stage array

on the pipeline model object based on the index of our estimator in stage array. Once, we get our estimator object for best model, we can call underline "_java_obj" to access the best values for our individual hyper parameters, which were added to the parameter grid. (This implementation is specific to pyspark, other languages such as Java/Scala has separate way to do this).

Following is the snap shot of the code for better explanation –

Logistic Regression Classifier with ML Pipeline to find the best hyper parameters Using Cross Validation

```
In [44]: paramGrid = ParamGridBuilder().addGrid(lr_classifier.regParam, [0.01, 0.1, 1.0]).addGrid(lr_classifier.elasticNetParam,

         pipeline = Pipeline(stages=[ lr_classifier ])

         evaluator = MulticlassClassificationEvaluator( labelCol = "label" )

         crossval_lr = CrossValidator( estimator = pipeline, estimatorParamMaps = paramGrid, evaluator = evaluator, numFolds = 1

         # Run cross-validation, and choose the best set of parameters.
         cvModel_lr = crossval_lr.fit( trainingSetDF )

         cvLR_predictions = cvModel_lr.transform(testSetDF)
         cvLR_accuracy = evaluator.evaluate(cvLR_predictions)

         bestModel = cvModel_lr.bestModel
         print(cvModel_lr.avgMetrics)
         print(list(zip(cvModel_lr.avgMetrics, paramGrid)))

         print(bestModel.stages[0]._java_obj.getRegParam())
         print(bestModel.stages[0]._java_obj.getElasticNetParam())
         print(bestModel.stages[0]._java_obj.getMaxIter())

         print(cvLR_accuracy)

         getEvaluationMatrix(cvLR_predictions)
```

Note: *When we use ML pipeline with cross validation to select the best hyper parameter, spark internally runs multiple different copies of model with combinations of given hyper parameters. For example, if we are adding 3 parameters in the parameter grid; each with 3 distinct values, spark will compare 27(3^3) different model. Furthermore, we are running 10 fold cross validation to evaluate the best model which increases our computations 10 fold. Therefore, we ran the pipeline only once to find the best params and using them in the individual classifier. We have commented this code in the final code submission to save the computational time. However, you can uncomment and run if you want to test this code. This usually takes 30-45 mins to run with 2 executor core assigned with 4GB memory.*

The data is split into the training data frame and testing data frame. (70 : 30)

Train/Test split based on the hardcoded seed value (70 : 30)

```
:  # Creating Training and Test set (70% , 30%)
   (trainingSetDF, testSetDF) = loanDFTransformed_2.randomSplit([0.7, 0.3], 1395)

   trainingSetDF.cache()
   testSetDF.cache()
```

The following method is used to compute the evaluation matrix

Method to compute the model evaluation matrix

```
n [39]: def getEvaluationMatrix(predicDF):
            lablePrediction = predicDF.select( "label", "prediction")
            lablePrediction.cache()
            totalCount = lablePrediction.count()
            correctCount = lablePrediction.filter(col("label") == col("prediction")).count()
            wrongCount = lablePrediction.filter(~(col("label") == col("prediction"))).count()
            trueP = lablePrediction.filter(col("label") == 0.0).filter(col("label") == col("prediction")).count()
            trueN = lablePrediction.filter(col("label") == 1.0).filter(col("label") == col("prediction")).count()
            falseN = lablePrediction.filter(col("label") == 1.0).filter(~(col("label") == col("prediction"))).count()
            falseP = lablePrediction.filter(col("label") == 0.0).filter(~(col("label") == col("prediction"))).count()

            ratioWrong = float(wrongCount) / float(totalCount)
            ratioCorrect = float(correctCount)/ float(totalCount)

            print("totalCount   - ", totalCount)
            print("correctCount - ", correctCount)
            print("wrongCount   - ", wrongCount)
            print("trueP        - ", trueP)
            print("trueN        - ", trueN)
            print("falseN       - ", falseN)
            print("falseP       - ", falseP)
            print("ratioWrong   - ", ratioWrong)
            print("ratioCorrect - ", ratioCorrect)

            precision = ((float(trueP) / (float(trueP) + float(falseP))) * 100 )
            recall = ((float(trueP) / (float(trueP) + float(falseN))) * 100 )
            print("Accuracy     - ", (trueP + trueN) / totalCount)
            print("Precision    - ", precision)
            print("Recall       - ", recall)
            print("F-1 Score    - ", ((2* ( (precision*recall) / (precision + recall))) ))
            print("Sensitivity  - ", ((float(trueP) / (float(trueP) + float(falseN))) * 100 ))
            print("Specificity  - ", ((float(trueN) / (float(trueN) + float(falseP))) * 100 ))

            createROC(predictions)
```

The following method is used to compute the ROC curve.

Method to compute the ROC Curve

```
def createROC(predictions):
    results = predictions.select(['probability', 'label'])

    ## prepare score-label set
    results_collect = results.collect()
    results_list = [(float(i[0][0]), 1.0-float(i[1])) for i in results_collect]
    scoreAndLabels = spark.sparkContext.parallelize(results_list)

    bcMetrics = BinaryClassificationMetrics(scoreAndLabels)
    print("ROC score is - ", bcMetrics.areaUnderROC)

    fpr = dict()
    tpr = dict()
    roc_auc = dict()

    y_test = [i[1] for i in results_list]
    y_score = [i[0] for i in results_list]

    fpr, tpr, _ = roc_curve(y_test, y_score)
    roc_auc = auc(fpr, tpr)

    plt.figure()
    plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic example')
    plt.legend(loc="lower right")
    plt.show()
```

7.1 Logistic Regression: **-** Logistic regression is a popular method to predict a categorical/binary response. It is a specialized case of "Generalized Linear Models" that predicts the probability of the outcomes. The main intention behind using this learning algorithm is to handle the class imbalance. Logistic regression is one of the implementations of linear models, for which spark provides ability to handle the class imbalance by adding a class weight. Apparently, spark doesn't provide any built-in implementation to handle class imbalancing for other nonlinear machine learning models.

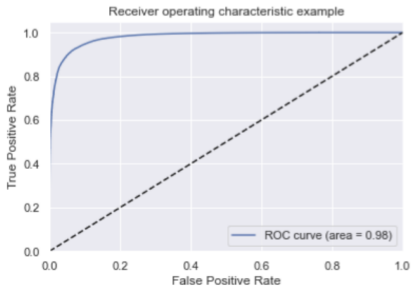
pyspark.ml.classifierpackage is used to implement the model with optimized hyper parameter. "Elasticnet" regularization is also implemented since it solves the limitations of both L1 and L2 regularization.

Following are the hyper parameters that are tested and added in the param grid search.
regParam – [0.01, 0.1, 1.0] ,*elasticNetParam* - [0.0, 0.5, 1.0], *maxIter* - [1, 5, 10]

Logistic Regression Classifier

```
print("**** Running Logistic Regression Classifier with best parameter found using ML pipeline **** ")

# Create initial LogisticRegression model
lr_classifier = LogisticRegression(labelCol="label", featuresCol="features", maxIter=3, weightCol="weightColumn")

# Train model with Training Data
lrModel = lr_classifier.fit(trainingSetDF)

# Make predictions on test data using the transform() method.
# LogisticRegression.transform() will only use the 'features' column.
predictions = lrModel.transform(testSetDF)

# Evaluate model
evaluator = MulticlassClassificationEvaluator( labelCol="label")
lr_accuracy = evaluator.evaluate(predictions)

getEvaluationMatrix(predictions)
```

```
**** Running Logistic Regression Classifier with best parameter found using ML pipeline ****
totalCount    -  263654
correctCount  -  251846
wrongCount    -  11808
trueP         -  235938
trueN         -  15908
falseN        -  2226
falseP        -  9582
ratioWrong    -  0.04478596949031685
ratioCorrect  -  0.9552140305096831
Accuracy      -  0.9552140305096831
Precision     -  96.09726295210166
Recall        -  99.06534992694111
F-1 Score     -  97.5587366958593
Sensitivity   -  99.06534992694111
Specificity   -  62.40878775990585
ROC score is  -  0.9800881762583983
```

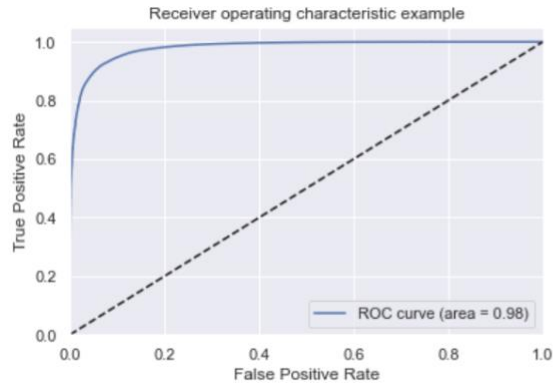


Logistic Regression Classifier with best hyper parameters.

Logistic Regression Classifier with ML Pipeline to find the best hyper parameters Using Cross Validation

```
paramGrid = ParamGridBuilder().addGrid(lr_classifier.regParam, [0.01, 0.1, 1.0])
.addGrid(lr_classifier.elasticNetParam, [0.0, 0.5, 1.0]).addGrid(lr_classifier.maxIter, [1, 3, 10]).build()

pipeline = Pipeline(stages=[ lr_classifier ])

evaluator = MulticlassClassificationEvaluator( labelCol = "label" )

crossval_lr = CrossValidator( estimator = pipeline, estimatorParamMaps = paramGrid,
                              evaluator = evaluator, numFolds = 10)

# Run cross-validation, and choose the best set of parameters.
cvModel_lr = crossval_lr.fit( trainingSetDF )

cvLR_predictions = cvModel_lr.transform(testSetDF)
cvLR_accuracy = evaluator.evaluate(cvLR_predictions)

bestModel = cvModel_lr.bestModel
print(cvModel_lr.avgMetrics)
print(list(zip(cvModel_lr.avgMetrics, paramGrid)))

print(bestModel.stages[0]._java_obj.getRegParam())
print(bestModel.stages[0]._java_obj.getElasticNetParam())
print(bestModel.stages[0]._java_obj.getMaxIter())

print(cvLR_accuracy)

getEvaluationMatrix(cvLR_predictions)
```

```
totalCount   -  263654
correctCount -  263653
wrongCount   -  1
trueP        -  245520
trueN        -  18133
falseN       -  1
falseP       -  0
ratioWrong   -  3.792849719708406e-06
ratioCorrect -  0.9999962071502803
Accuracy     -  0.9999962071502803
Precision    -  100.0
Recall       -  99.99959270286453
F-1 Score    -  99.99979635101754
Sensitivity  -  99.99959270286453
Specificity  -  100.0
ROC score is -  0.9800872919883237
```



7.2 Naïve Bayes: To start with the model building, I wanted to include a generative machine learning model (probabilistic classifier) in the trails. Naïve Bayes is a simple probabilistic classifier based on Bayes theorem with strong independence assumptions between the features. The intention behind using this model was to handle the class imbalance issue by adding the default prior belief (beta distribution function) to calculate the max posterior probability.

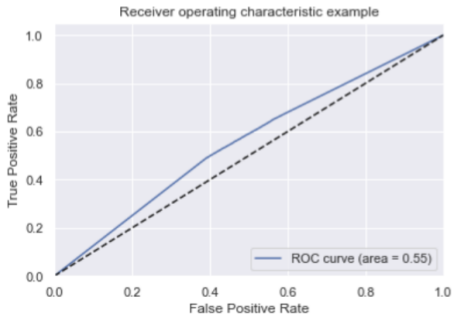
pyspark.ml.classifier package was used to implement this algorithm with only one optimized hyper parameter for Laplace smoothing.
Following are the hyper parameters we tested and added in the param grid search.
smoothing– [1.0, 2.0, 3.0]

Note: *- As we notice from the results****, this algorithm is not performing*** *well as compared to others. I tried to drill down the root cause for the issue, but the spark provides very abstract level API for the algorithm implementation and doesn't provide much control on its implementation.*

NaiveBayes Classifier

```
print("**** Running NaiveBayes Classifier with best parameter found using ML pipeline **** ")
# Create initial NaiveBayes model
nb_classifier = NaiveBayes(labelCol="label", featuresCol="features", smoothing=50, weightCol="weightColumn")

# Train model with Training Data
nbModel = nb_classifier.fit(trainingSetDF)

# Make predictions on test data using the transform() method.
# NaiveBayes.transform() will only use the 'features' column.
predictions = nbModel.transform(testSetDF)

# Evaluate model
evaluator = MulticlassClassificationEvaluator( labelCol="label", predictionCol="prediction", metricName="accuracy")
nb_accuracy = evaluator.evaluate(predictions)

getEvaluationMatrix(predictions)
```

```
**** Running NaiveBayes Classifier with best parameter found using ML pipeline ****
totalCount    -  263654
correctCount -  133092
wrongCount    -  130562
trueP         -  122204
trueN         -  10888
falseN        -  7246
falseP        -  123316
ratioWrong    -  0.4952020451045689
ratioCorrect -  0.5047979548954311
Accuracy      -  0.5047979548954311
Precision     -  49.773541870316066
Recall        -  94.40247199691001
F-1 Score     -  65.18068112115637
Sensitivity   -  94.40247199691001
Specificity   -  8.113021966558374
ROC score is -  0.5541658747092667
```

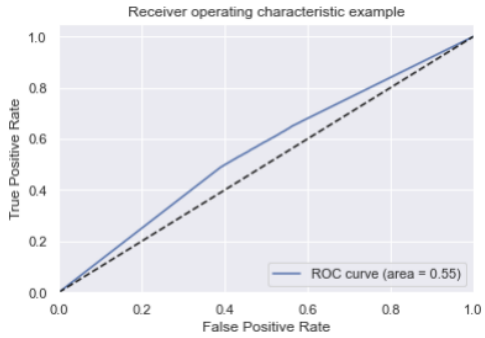


Naïve Bayes Classifier with best hyper parameters.

NaiveBayes Classifier with ML Pipeline to find the best hyper parameters Using Cross Validation¶

```
paramGrid = ParamGridBuilder().addGrid(nb_classifier.smoothing, [1.0, 2.0, 3.0]).build()

pipeline = Pipeline(stages=[ nb_classifier ])

evaluator = MulticlassClassificationEvaluator( labelCol="label", predictionCol="prediction", metricName="accuracy")

crossval_nb = CrossValidator( estimator = pipeline, estimatorParamMaps = paramGrid,
                              evaluator = evaluator, numFolds = 10)

# Run cross-validation, and choose the best set of parameters.
cvModel_nb = crossval_nb.fit( trainingSetDF )

cvNB_predictions = cvModel_nb.transform(testSetDF)
cvNB_accuracy = evaluator.evaluate(cvNB_predictions)

bestModel = cvModel_nb.bestModel
print(cvModel_nb.avgMetrics)
print(list(zip(cvModel_nb.avgMetrics, paramGrid)))

print(bestModel.stages[0]._java_obj.getSmoothing())

print(cvNB_accuracy)

getEvaluationMatrix(cvNB_predictions)
```

```
[0.50656081382053, 0.50656081382053, 0.50656081382053]
[(0.50656081382053, {Param(parent='NaiveBayes_db803d943002', name='smoothing', doc='The smoothing parameter, should b
e >= 0, default is 1.0'): 1.0}), (0.50656081382053, {Param(parent='NaiveBayes_db803d943002', name='smoothing', doc='T
he smoothing parameter, should be >= 0, default is 1.0'): 2.0}), (0.50656081382053, {Param(parent='NaiveBayes_db803d9
43002', name='smoothing', doc='The smoothing parameter, should be >= 0, default is 1.0'): 3.0})]
1.0
0.5047941620457114
totalCount    -  263654
correctCount  -  133091
wrongCount    -  130563
trueP         -  122203
trueN         -  10888
falseN        -  7246
falseP        -  123317
ratioWrong    -  0.4952058379542886
ratioCorrect  -  0.5047941620457114
Accuracy      -  0.5047941620457114
Precision     -  49.773134571521666
Recall        -  94.40242875572619
F-1 Score     -  65.18032157325005
Sensitivity   -  94.40242875572619
Specificity   -  8.11296151410156
ROC score is  -  0.554159276711371
```



7.3 Random Forest Classifier

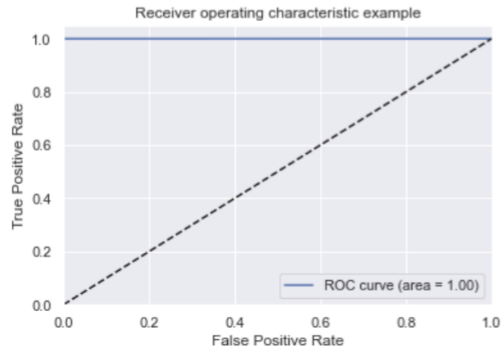
Random Forest (the ensembles of decision trees) is used to further deal with the data imbalance issue. Random forest is the aggregation of multiple decision trees which uses entropy/Gini to find the impurities, which makes it less sensitive to the class imbalance. The Spark implementation supports random forest for binary as well as multiclass classifications.

pyspark.ml.classifierpackage is used to implement this algorithm with optimized hyper parameters. The model is also tested with different bin sizes and tree depths. Following are the hyper parameters we tested and added in the param grid search. *maxBins*– [25, 28, 31, 34],*maxDepth*- [4, 6, 8, 10], *impurity*- ["entropy", "gini"]

Random Forest Classifier

```
# Create initial Random Forest Classifier model
print("**** Running Random Forest Classifier with best parameter found using ML pipeline **** ")
rf_classifier = RandomForestClassifier( impurity="gini", maxDepth=12,
                                        numTrees=10, featureSubsetStrategy="auto", seed=1395)

# Train model with Training Data
rf_model = rf_classifier.fit(trainingSetDF)

# Print the Forest tree rules.
#rf_model.toDebugString

# Make predictions on test data using the transform() method.
# RandomForest.transform() will only use the 'features' column.
predictions = rf_model.transform(testSetDF)

evaluator = MulticlassClassificationEvaluator( labelCol = "label" )
rf_accuracy = evaluator.evaluate(predictions)

getEvaluationMatrix(predictions)
```

```
**** Running Random Forest Classifier with best parameter found using ML pipeline ****
totalCount    -  263654
correctCount -  257868
wrongCount    -  5786
trueP         -  245520
trueN         -  12348
falseN        -  5786
falseP        -  0
ratioWrong    -  0.021945428478232835
ratioCorrect -  0.9780545715217671
Accuracy      -  0.9780545715217671
Precision     -  100.0
Recall        -  97.69762759345181
F-1 Score     -  98.8354071646814
Sensitivity   -  97.69762759345181
Specificity   -  100.0
ROC score is -  0.9999137780525865
```

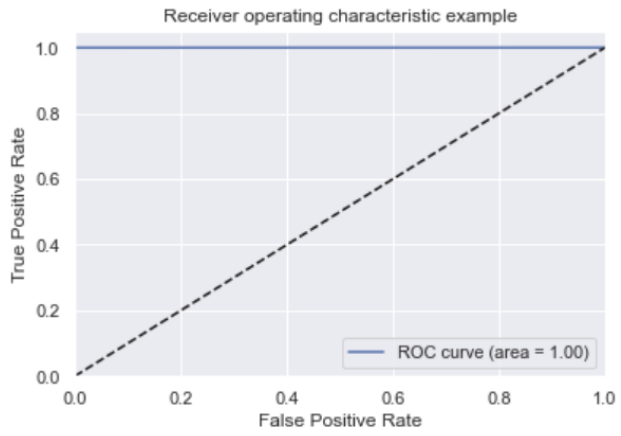


Random Forest Classifier with best hyper parameters.

Random Forest Classifier with ML Pipeline to find the best hyper parameters Using Cross Validation

```
paramGrid = ParamGridBuilder().addGrid(rf_classifier.maxBins,
            [25, 28, 31, 34]).addGrid(rf_classifier.maxDepth, [4, 6, 8, 12]).addGrid(rf_classifier.impurity,
                                                                                     ["entropy", "gini"]).build()

pipeline = Pipeline(stages=[ rf_classifier ])

evaluator = MulticlassClassificationEvaluator( labelCol = "label" )

crossval = CrossValidator( estimator = pipeline, estimatorParamMaps = paramGrid, evaluator = evaluator, numFolds = 10)

# Run cross-validation, and choose the best set of parameters.
cvModel = crossval.fit( trainingSetDF )

cv_predictions = cvModel.transform(testSetDF)
cv_accuracy = evaluator.evaluate(cv_predictions)

bestModel = cvModel.bestModel
print(cvModel.avgMetrics)
print(list(zip(cvModel.avgMetrics, paramGrid)))

print(bestModel.stages[0]._java_obj.getMaxBins())
print(bestModel.stages[0]._java_obj.getMaxDepth())
print(bestModel.stages[0]._java_obj.getImpurity())

print(cv_accuracy)

getEvaluationMatrix(cv_predictions)
```

```
totalCount    -  263654
correctCount  -  261772
wrongCount    -  1882
trueP         -  245520
trueN         -  16252
falseN        -  1882
falseP        -  0
ratioWrong    -  0.00713814317249122
ratioCorrect  -  0.9928618568275088
Accuracy      -  0.9928618568275088
Precision     -  100.0
Recall        -  99.23929475105294
F-1 Score     -  99.61819517083838
Sensitivity   -  99.23929475105294
Specificity   -  100.0
ROC score is  -  0.9999136312731876
```



7.4 Gradient Boosted Trees: - In contrary to random forest, which tried to minimize the error by reducing the variance, I tested the opposite way by reducing the bias. Boosting reduces error mainly by reducing bias and to some extent variance, by aggregating the output from many models. GBM is a boosting method, which builds on weak classifiers. The idea is to add a classifier at a time, so that the next classifier is trained to improve the already trained ensemble in sequential order. On the other hand, for RF each iteration the classifier is trained independently from the rest in parallel.

pyspark.ml.classifier package is used to implement this algorithm with optimized hyper parameter. The model is also tested with different step sizes for gradient descent and tree debts.

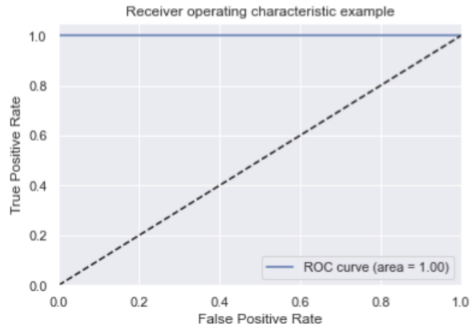
Following are the hyper parameters we tested and added in the param grid search.

stepSize– [0.01, 0.1, 1.0],*maxIter*- [15, 20, 25], *maxDepth*- [5, 10, 15]

Gradient Boosting Classifier

```
print("**** Running Gradient Boosting Classifier with best parameter found using ML pipeline **** ")

# Create initial Gradient Boosting Classifier model
gb_classifier = GBTClassifier(labelCol="label", featuresCol="features", maxDepth=5,
                              maxBins=5, lossType="logistic", maxIter=10, stepSize=.00000001)

# Train model with Training Data
gbModel = gb_classifier.fit(trainingSetDF)

# Make predictions on test data using the transform() method.
# NaiveBayes.transform() will only use the 'features' column.
predictions = gbModel.transform(testSetDF)

# Evaluate model
evaluator = MulticlassClassificationEvaluator( labelCol="label", predictionCol="prediction", metricName="accuracy")
gb_accuracy = evaluator.evaluate(predictions)

getEvaluationMatrix(predictions)
```

```
**** Running Gradient Boosting Classifier with best parameter found using ML pipeline ****
totalCount   -  263654
correctCount -  263654
wrongCount   -  0
trueP        -  245520
trueN        -  18134
falseN       -  0
falseP       -  0
ratioWrong   -  0.0
ratioCorrect -  1.0
Accuracy     -  1.0
Precision    -  100.0
Recall       -  100.0
F-1 Score    -  100.0
Sensitivity  -  100.0
Specificity  -  100.0
ROC score is -  1.0
```

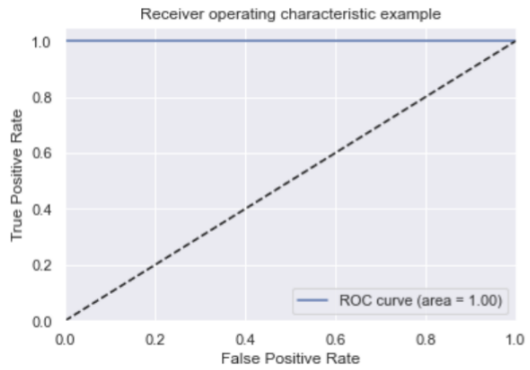


Gradient Boosting Classifier with best hyper parameters.

Gradient Boosting Classifier with ML Pipeline to find the best hyper parameters Using Cross Validation

```
paramGrid = ParamGridBuilder().addGrid(gb_classifier.maxDepth,
            [3, 5, 10]).addGrid(gb_classifier.maxIter, [5, 10, 15]).
            addGrid(gb_classifier.stepSize, [0.01, 0.1, 1.0]).build()

pipeline = Pipeline(stages=[ gb_classifier ])

evaluator = MulticlassClassificationEvaluator( labelCol="label", predictionCol="prediction", metricName="accuracy")

crossval_gb = CrossValidator( estimator = pipeline, estimatorParamMaps = paramGrid, evaluator = evaluator,
                              numFolds = 10)

# Run cross-validation, and choose the best set of parameters.
cvModel_gb = crossval_gb.fit( trainingSetDF )

cvGB_predictions = cvModel_gb.transform(testSetDF)
cvGB_accuracy = evaluator.evaluate(cvGB_predictions)

bestModel = cvModel_gb.bestModel
print(cvModel_gb.avgMetrics)
print(list(zip(cvModel_gb.avgMetrics, paramGrid)))

print(bestModel.stages[0]._java_obj.getMaxDepth())
print(bestModel.stages[0]._java_obj.getMaxIter())
print(bestModel.stages[0]._java_obj.getStepSize())

print(cvGB_accuracy)

getEvaluationMatrix(cvGB_predictions)
```

```
totalCount   -  263654
correctCount -  263654
wrongCount   -  0
trueP        -  245520
trueN        -  18134
falseN       -  0
falseP       -  0
ratioWrong   -  0.0
ratioCorrect -  1.0
Accuracy     -  1.0
Precision    -  100.0
Recall       -  100.0
F-1 Score    -  100.0
Sensitivity  -  100.0
Specificity  -  100.0
ROC score is -  1.0
```



8. Usefulness of the model

As part of the Lending Club's revenue generation model, it charges an origination fee to the borrowers and service fee to the investors. This model can be useful to provide comprehensive analysis of the historical data as well as a smart prediction about the investor's money to lower their risk. By developing a nearly perfect prediction model, we would hope to reduce the number of delinquencies in the investment and helps genuine borrowers to maintain their credit ratings. This would help Lending Club to engage more investors and borrowers in their platform, hence increasing the revenue growth.

Furthermore, I would like to highlight few recommendations noticed from the exploratory data analysis:

1. The higher the loan amount, the higher the likelihood of default. Investors should invest in loans that are approximation $9000 or less.
2. Loans with term of 36 months tended to be defaulted a lot more than loans with term of 60 months. Investor should invest in long terms.
3. Certain sub-grades were almost likely to default compared to other sub-grades. Selecting loans of subgrade B5 and higher will result in a 90% chance of repayment.

9. Summary

If Lending Club must provide the more precise information to the investors to reduce their investment risk, they need to provide the default likeliness of borrower more accurately. Should this model be used in real life, the goal of this project is to retain more investors for Lending Club. Therefore, more focus is put on precision and specificity of the model.

	Accuracy (%)	**Precision**	**Recall**	**F1 Score**
Naïve Bayes	50	49.96	94.31	65.32
Logistic Regression	95	96.05	99.0	97.50
Random Forest	**99**	**100.0**	**99.9**	**99.99**
Gradient Boosting Classifier	**100**	**100.0**	**100.0**	**100.00**

10. References

https://www.lendingclub.com/
https://www.kaggle.com/wendykan
https://data.world/lpetrocelli/lendingclub-loan-data-2017-q-1
https://spark.apache.org/docs/2.2.0/ml-guide.html
https://spark.apache.org/docs/2.3.0/api/python/pyspark.html