Credit Risk Analysis

For the Coursera course

Supervised Machine Learning: Classification by IBM

By Adithya Narayan Samal

this dataset has been take from kaggle, [note as the dataset is very large we only consider a small part of the data to train models]

dataset by RAMESH MEHTA,you can download the dataset from [https://www.kaggle.com/datasets/rameshmehta/credit-risk-analysis (https://www.kaggle.com/datasets/rameshmehta/credit-risk-analysis)](https://www.kaggle.com/datasets/rameshmehta/credit-risk-analysis)

Main objective of the analysis that specifies whether your model will be focused on prediction or interpretation and the benefits that your analysis provides to the business or stakeholders of this data.

The Main Aim of this analysis is to build a model that predicts has accurate prediction and explain the importance of the features by explaining the model(we use simple models as they easier to understand) (we try to predict if a person will default on their loan payments)

Brief description of the data set you chose, a summary of its attributes, and an outline of what you are trying to accomplish with this analysis.

The dataset contains loan data for loans issued through the year 2007 - 2015 as the loan has a lot of attributes, we only consider ones which have some Correlation with the default_ind(default indicator) of the loan which is our Target Variable,

some of its attributes are,

default_ind--> this shows if the particular person has defaulted the payment of the instalment

loan_amnt --> Amount of money requested by the borrower.

int_rate --> Interest rate of the loan.

grade --> Loan grade with categories A, B, C, D, E, F, G

sub_grade --> sub categories ranging from 1-5 for the grade

annual_inc --> Borrowers annual income

purpose --> The primary purpose of borrowing.

installments —-> Monthly amount payments for opted loan term —> duration of the loan until it’s paid off

as this dataset contains about 73 features we ignore most of them on the basis of correlation and null/missing values, this data is heavily unbalanced about 5.42% of all loans considered as defaulted

Brief summary of data exploration and actions taken for data cleaning and feature engineering

for data cleaning,we first take the drop all columns which have more than 1% of null values, then we drop all null/missing value this removes most of the columns so we work with that is left, we then transform rows as such as Grade,sub_grade for Grade who's values can be A,B,C etc we use dummy encoding and for sub grade which is whatever the grade attribute is with an additional value from 1-5 (eg A1,B3,D4) for subgrade we simply just consider the number (eg for A5 we only consider 5)

We apply binary encoding for categorical variables with two possible values,and apply dummy encoding for the other categorical attributes and we also convert the Date type attributes to 3 new attributes that being day,month,year

we also try to look at the correlation between the remaining attributes and default_ind but we dont eliminate any attributes, we try to explain how the attributes which has comparitively high correlation with default_indm but we notice that most of the attributes are pretty similar for both defaulters and non-defaulters,

Summary of training at least three different classifier models, preferably of different nature in explainability and predictability. For example, you can start with a simple logistic regression as a baseline, adding other models or ensemble models. Preferably, all your models use the same training and test splits, or the same cross-validation method

Here is a summary of the diffrent models Trained along with the hyperparameters and performance metrics

Model	Hyperparameters	Perfo
Gaussian Naive Bayes	GaussianNB(priors= [0.94575, 0.05425])	0.864 0.934 0.276 0.427 0.89

Logistic Regression	LogisticRegression(class_weight={0: 0.05425,1:0.94575},C=0.01,penalty='l2')	0.993 0.967 0.920 0.943 0.98
Decision Tree	DecisionTreeClassifier(random_state=123)	0.953 0.242 0.715 0.361 0.618
Random Forest	RandomForestClassifier(n_estimators=69,random_state=123)	0.997 0.956 0.99 0.977 0.978
KNN	KNeighborsClassifier(n_neighbors=3)	0.953 0.242 0.715

A paragraph explaining which of your classifier models you recommend as a final model that best fits your needs in terms of accuracy and explainability.

Although Both Logistic Regression and Random Forest have similar Accuracy,F1 Score and AUC, although Random Forest holds a slightly better performance it is better to use logistic Regression as the final model, as it is less resource intensive and not as likely to overfit, and mainly it is easier to understand by looking at the weights of the Coeff of each feature where as to understand feature importance in Randomforest we will have to go through a large and Complex tree,

Which is why i would suggest Logistic Regression as the final model

Summary Key Findings and Insights, which walks your reader through the main drivers of your model and insights from your data derived from your classifier model.

out_prncp,out_prncp_inv,total_rec_prncp,loan_amnt,funded_amnt,funded_amnt_inv,total_pymnt_i are some of the features which have the most impact on if a person defaults on their loan or not,(according to Logistic Regression)

Some of the features considered by Randomforest are--> recoveries,collection_recovery_fee, total_rec_prncp,out_prncp,last_pymnt_amnt,total_pymnt_inv,out_prncp_inv,total_pymnt ,funded_amnt,loan_amnt

some othere key findings are

- most of the features like int_rate are similar for both defaulters,non-defaulters although it has similar range for both, it is marginally higher for defaulters

- some features like collection_recovery_fee which are related with recovery fee can have more significance as they are mostly zero for non-defaulters , making it a effective feature when deciding if someone is likely to default, this is probably because you only pay recovery related fees when you have defaulted atleast once, and maybe if someone had defaulted once they might default again

- We can see that most of the features considered important has relation to recoveries, and most of the important features are similar for both Logistic Regression, and RandomForest

Suggestions for next steps in analyzing this data, which may include suggesting revisiting this model after adding specific data features that may help you achieve a better explanation or a better prediction.

Some models like SVM were not used as they take a lot of time to train if dataset has a lot of features, so we can try a SVM and as we have a huge quantity of data we can also try using Deep Learning,Neural Networks for better performance,

we can also try including more features as in this case i removed/discarded a lot of potentially usefull features because of stroage and processing power constraints, if we get better resources we can try being less selective and include more of the features

In [1]:
```
import sklearn
import pandas as pd
import matplotlib.pyplot as plt
```

In [2]:
```
#ignore warnings
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
```

In [3]:
```
df=pd.read_csv('data.csv',low_memory=False)
```

In [4]:
```
size=df.shape[0]
df
```

Out[4]:

	id	member_id	loan_amnt	funded_amnt	funded_amnt_inv	term	int_rate	insta
0	1077501	1296599	5000	5000	4975.0	36 months	10.65	
1	1077430	1314167	2500	2500	2500.0	60 months	15.27	
2	1077175	1313524	2400	2400	2400.0	36 months	15.96	
3	1076863	1277178	10000	10000	10000.0	36 months	13.49	
4	1075358	1311748	3000	3000	3000.0	60 months	12.69	
...	...	...	...	...	...	...	...	
855964	36371250	39102635	10000	10000	10000.0	36 months	11.99	
855965	36441262	39152692	24000	24000	24000.0	36 months	11.99	
855966	36271333	38982739	13000	13000	13000.0	60 months	15.99	
855967	36490806	39222577	12000	12000	12000.0	60 months	19.99	
855968	36271262	38982659	20000	20000	20000.0	36 months	11.99	

855969 rows × 73 columns

In [5]:
```
dropped_list=[]
for x in df:
    if (df[x].count()<(size*0.99)) :
        df.drop([x],axis=1,inplace=True)
        dropped_list.append(x)
```

In [6]:
```
dropped_list
```

Out[6]:
```
['emp_title',
 'emp_length',
 'desc',
 'mths_since_last_delinq',
 'mths_since_last_record',
 'last_pymnt_d',
 'next_pymnt_d',
 'mths_since_last_major_derog',
 'annual_inc_joint',
 'dti_joint',
 'verification_status_joint',
 'tot_coll_amt',
 'tot_cur_bal',
 'open_acc_6m',
 'open_il_6m',
 'open_il_12m',
 'open_il_24m',
 'mths_since_rcnt_il',
 'total_bal_il',
 'il_util',
 'open_rv_12m',
 'open_rv_24m',
 'max_bal_bc',
 'all_util',
 'total_rev_hi_lim',
 'inq_fi',
 'total_cu_tl',
 'inq_last_12m']
```

In [7]:
```
#now we deal with the non-numeric attributes
df.dropna(inplace=True)
df.isna().value_counts()
```

Out[7]:
```
id     member_id  loan_amnt  funded_amnt  funded_amnt_inv  term   int_rate  i
nstallment  grade  sub_grade  home_ownership  annual_inc  verification_status
issue_d  pymnt_plan  purpose  title  zip_code  addr_state  dti    delinq_2yrs
earliest_cr_line  inq_last_6mths  open_acc  pub_rec  revol_bal  revol_util  t
otal_acc  initial_list_status  out_prncp  out_prncp_inv  total_pymnt  total_p
ymnt_inv  total_rec_prncp  total_rec_int  total_rec_late_fee  recoveries  col
lection_recovery_fee  last_pymnt_amnt  last_credit_pull_d  collections_12_mth
s_ex_med  policy_code  application_type  acc_now_delinq  default_ind
False  False      False      False        False            False  False     F
alse        False  False      False           False       False
False    False       False    False  False     False        False  False
False            False            False     False     False       False       F
alse      False                False      False          False        False
False            False          False                False       False
False            False               False                        False
False             False            False         855385
dtype: int64
```

In [8]:
```
#as sub_grade and grade are related, we can assume if grade is like A, subgrade
df[['sub_grade','grade']].head(10)
```

Out[8]:

	sub_grade	grade
0	B2	B
1	C4	C
2	C5	C
3	C1	C
4	B5	B
5	A4	A
6	C5	C
7	E1	E
8	F2	F
9	B5	B

In [9]:
```
df['home_ownership'].value_counts()
```

Out[9]:
```
MORTGAGE    428822
RENT        342312
OWN          84063
OTHER          142
NONE            43
ANY              3
Name: home_ownership, dtype: int64
```

In [10]:
```
df['term']=df['term'].apply(lambda x : int(x.split(" ")[1]))
#for sub grade, we do ordinal encoding for grade column and only consider the s
df['sub_grade']=df['sub_grade'].apply(lambda x : int(x[1]))
```

In [11]:
```
df['initial_list_status']=df['initial_list_status'].apply(lambda x:0 if x == 'f
df['pymnt_plan']=df['pymnt_plan'].apply(lambda x:0 if x == 'n' else 1)
df['application_type']=df['application_type'].apply(lambda x:0 if x == 'INDIVID
```

In [12]:
```
#dealing with date based columns,dropping columns
date_list=['issue_d','earliest_cr_line','last_credit_pull_d']
drop_list=['title','zip_code','addr_state','id']
for x in  date_list:
    df[x]=pd.to_datetime(df[x])
    df[x+"_day"]=df[x].dt.day
    df[x+"_month"]=df[x].dt.month
    df[x+"_year"]=df[x].dt.year
    df.drop(labels=[x],inplace=True,axis=1)
df.drop(labels=drop_list,axis=1,inplace=True)
#ordinal encoding of the remaining columns
df=pd.get_dummies(df)
```

In [13]:
```
df_cr=df.corr()
df_cols=df.columns
```

In [14]:
```
high_cor_list=[]
for x in df_cols:
    if (df_cr['default_ind'][x]>0.10 and df_cr['default_ind'][x]>0) or (df_cr['
        high_cor_list.append(x)
print(high_cor_list)
for x in high_cor_list:
    print(x+" : ",df_cr['default_ind'][x])
#as we have very low correlation among the columns, we have to consider every c
low_cor_list=[]
for x in df_cols:
    if (df_cr['default_ind'][x]==0):
        low_cor_list.append(x)
for x in low_cor_list:
    print(x+" : ",df_cr['default_ind'][x])
# as all columns have some non-zero correlation we dont drop any,
# we can see that 'last_credit_pull_d_day ' has somewhat high correlation, howe
```

```
['int_rate', 'total_rec_late_fee', 'recoveries', 'collection_recovery_fee',
'default_ind', 'last_credit_pull_d_day']
int_rate :  0.15500366390293305
total_rec_late_fee :  0.14071868635751128
recoveries :  0.4757988426510833
collection_recovery_fee :  0.3307443207798686
default_ind :  1.0
last_credit_pull_d_day :  0.17658276317654667
```

In [15]:
```
# as the dataset is too large, we use  to take a sample and well use it as our
from sklearn.model_selection import train_test_split
df_maj,df_min=train_test_split( df,test_size=0.2, random_state=42, stratify=df[
```

In [16]:
```
df_maj['default_ind'].value_counts(normalize=True)
```

Out[16]:
```
0    0.94575
1    0.05425
Name: default_ind, dtype: float64
```

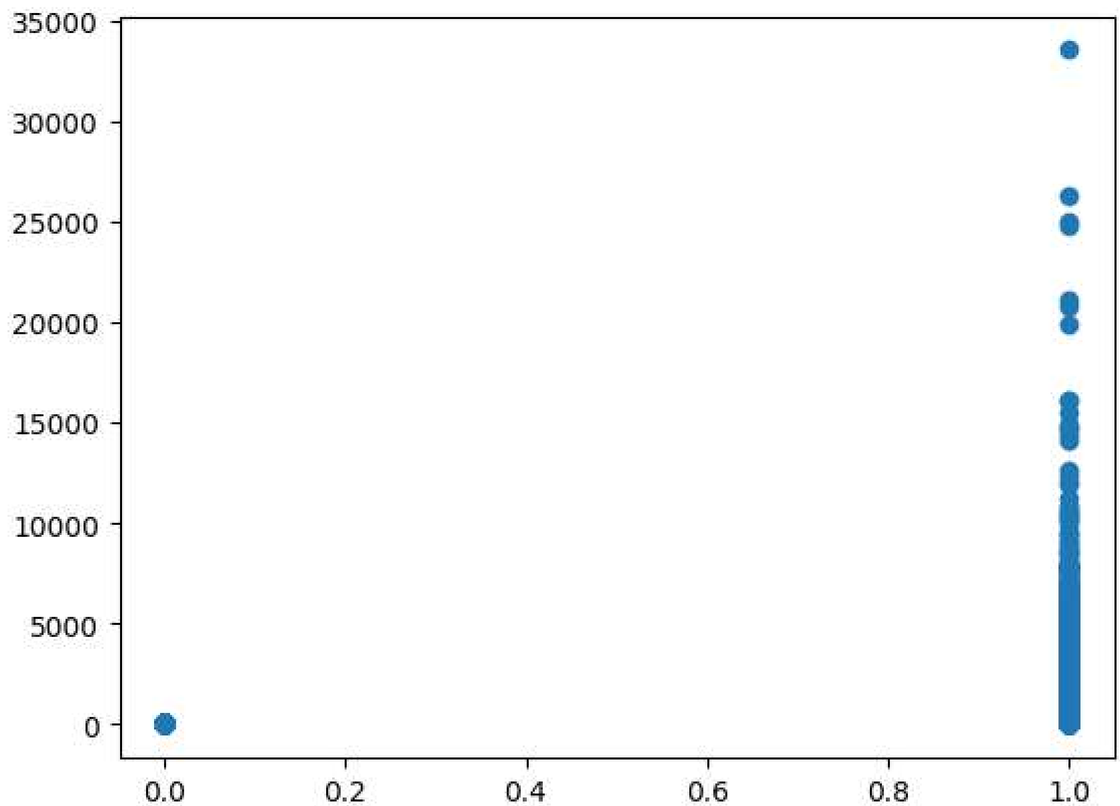
In [17]:
```
df_min['default_ind'].value_counts(normalize=True)
```

Out[17]:
```
0    0.94575
1    0.05425
Name: default_ind, dtype: float64
```

In [18]:
```
#we see that both have the same ratio of Defaults, we take df_min, as our main
old_df=df.copy()
df=df_min.copy()
```

In [19]:
```
plt.scatter(df['default_ind'],df['recoveries'])
```

Out[19]: <matplotlib.collections.PathCollection at 0x1faea886e50>



In [20]:
```
df_group=df.groupby('default_ind')
```

In [21]:
```
df_group['recoveries'].value_counts(normalize=True)
# we can see that people who did not default have 0 recoveries, so there is a t
#the relationship might be if you are a defaulty you might have to pay 'recover
```

Out[21]:
```
default_ind  recoveries
0            0.00          1.000000
1            0.00          0.480228
             10.40         0.000323
             10.70         0.000323
             11.29         0.000323
                             ...
             21096.30      0.000108
             24833.68      0.000108
             25000.29      0.000108
             26308.47      0.000108
             33520.27      0.000108
Name: recoveries, Length: 4736, dtype: float64
```

In [22]:
```
df_group['grade_A'].value_counts(normalize=True)
```

Out[22]:
```
default_ind  grade_A
0            0          0.824711
             1          0.175289
1            0          0.943325
             1          0.056675
Name: grade_A, dtype: float64
```

In [23]:
```
df_group['grade_B'].value_counts(normalize=True)
```

Out[23]:
```
default_ind  grade_B
0            0          0.704412
             1          0.295588
1            0          0.789570
             1          0.210430
Name: grade_B, dtype: float64
```

In [24]:
```
df_group['grade_C'].value_counts(normalize=True)
```

Out[24]:
```
default_ind  grade_C
0            0          0.723856
             1          0.276144
1            0          0.716087
             1          0.283913
Name: grade_C, dtype: float64
```

In [25]:
```
df_group['grade_D'].value_counts(normalize=True)
```

Out[25]:
```
default_ind  grade_D
0            0          0.849026
             1          0.150974
1            0          0.769852
             1          0.230148
Name: grade_D, dtype: float64
```

In [26]:
```
df_group['grade_E'].value_counts(normalize=True)
```

Out[26]:
```
default_ind  grade_E
0            0          0.925740
             1          0.074260
1            0          0.864778
             1          0.135222
Name: grade_E, dtype: float64
```

In [27]:
```
df_group['grade_F'].value_counts(normalize=True)
```

Out[27]:
```
default_ind  grade_F
0            0          0.977187
             1          0.022813
1            0          0.935136
             1          0.064864
Name: grade_F, dtype: float64
```

In [28]:
```
df_group['grade_G'].value_counts(normalize=True)
# we see that the grade they belong to doesnt have much impact to if they defau
```

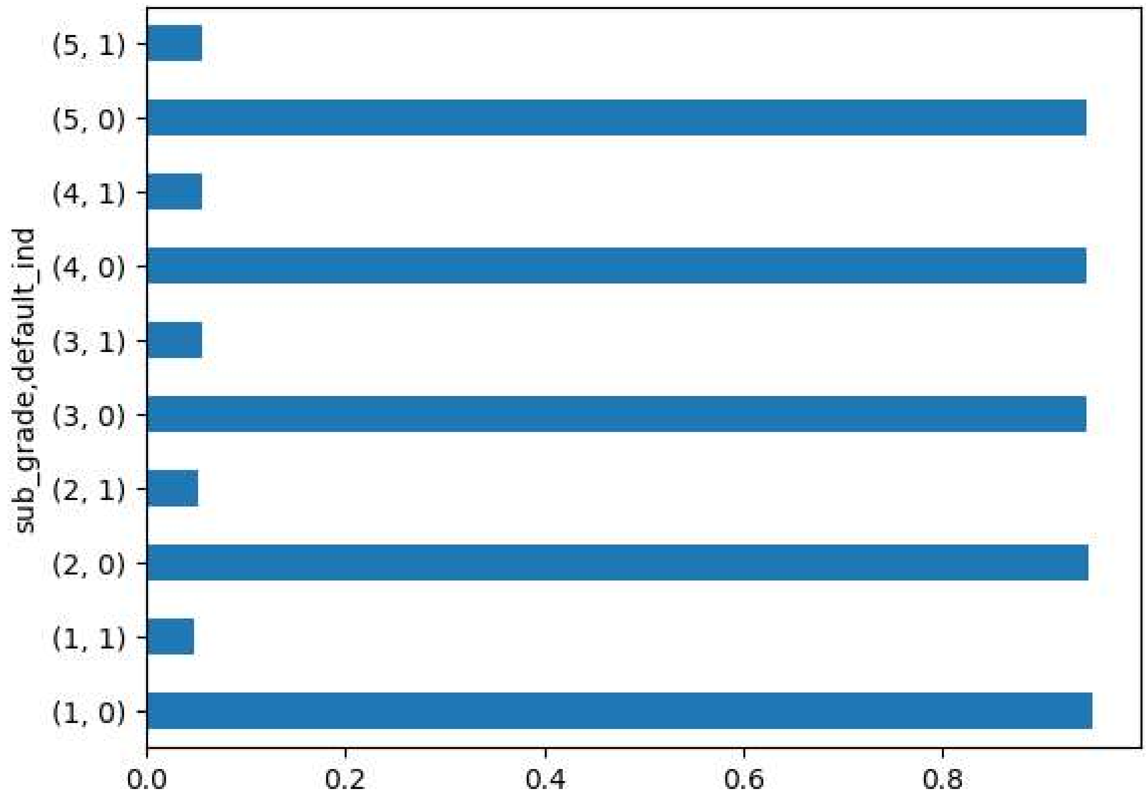
Out[28]:
```
default_ind  grade_G
0            0          0.995068
             1          0.004932
1            0          0.981252
             1          0.018748
Name: grade_G, dtype: float64
```

In [29]:
```
import seaborn as sns

# we know all the sub_grades(1-5) have similar presence(proportions/counts)
df_group=df.groupby('sub_grade')
df_group['default_ind'].value_counts(normalize=True)
```

Out[29]:
```
sub_grade  default_ind
1          0              0.950567
           1              0.049433
2          0              0.946477
           1              0.053523
3          0              0.943320
           1              0.056680
4          0              0.944057
           1              0.055943
5          0              0.944063
           1              0.055937
Name: default_ind, dtype: float64
```

In [30]:
```
df_group['default_ind'].value_counts(normalize=True).plot(kind='barh')
#we see something similar here,
```

Out[30]: <Axes: ylabel='sub_grade,default_ind'>



In [31]:
```
df2=df.copy()
df2=df2.sort_values(['int_rate'])
df2.reset_index(inplace=True)
default_ind_0=df2['default_ind']==0
default_ind_1=df2['default_ind']==1
df3=df2[default_ind_1]
df4=df2[default_ind_0]
df3.reset_index(inplace=True)
df4.reset_index(inplace=True)
```

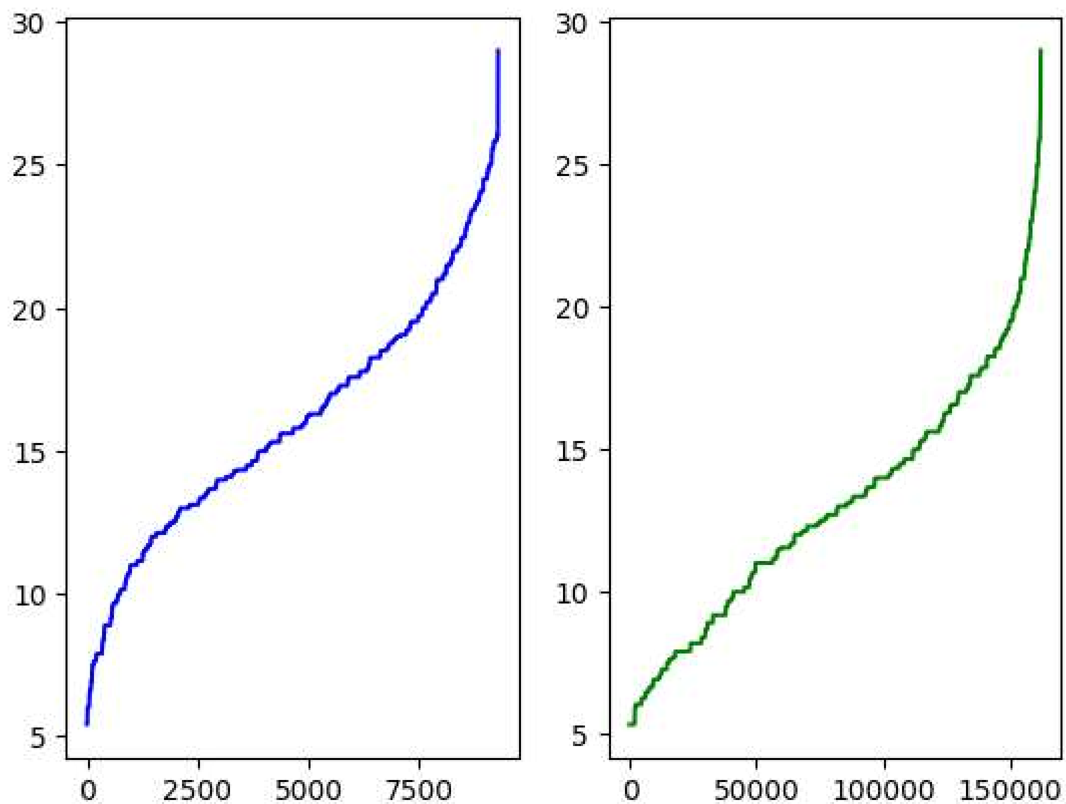
In [32]:
```
#default_ind=1
df3['int_rate'].describe()
```

Out[32]:
```
count    9281.000000
mean       16.006910
std         4.291598
min         5.420000
25%        13.050000
50%        15.650000
75%        18.920000
max        28.990000
Name: int_rate, dtype: float64
```

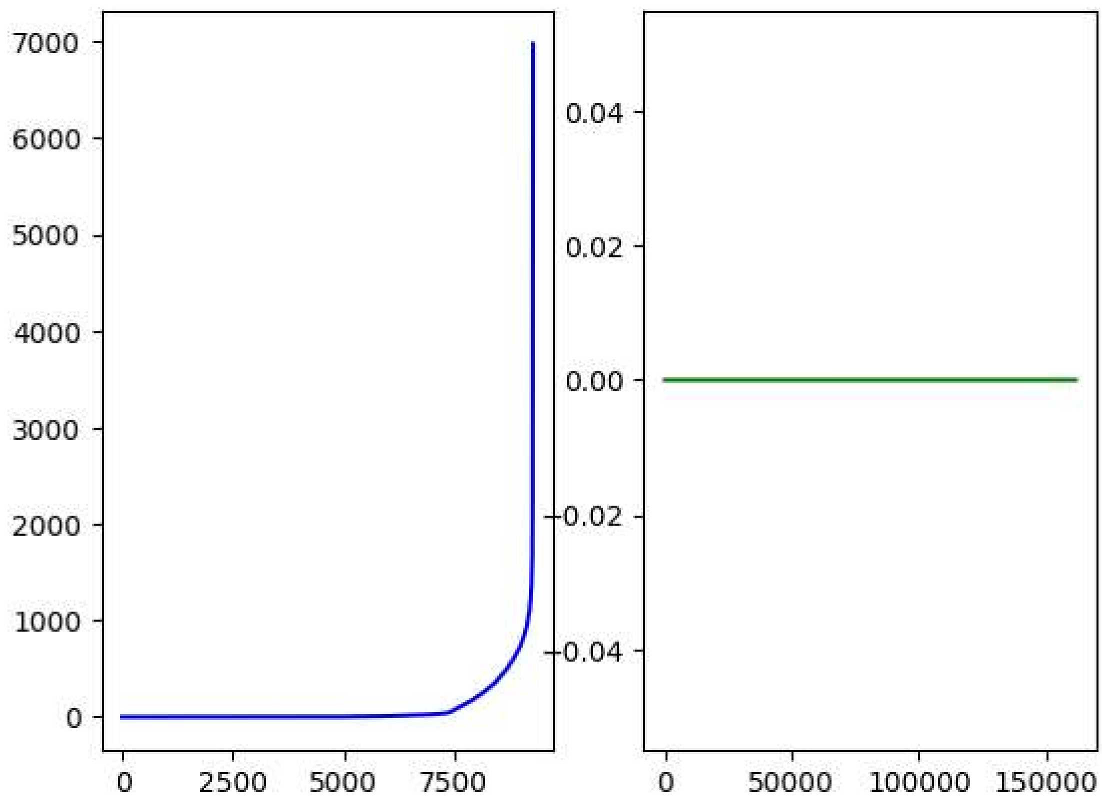
In [33]:
```
#default_ind=0
df4['int_rate'].describe()
```

Out[33]:
```
count    161796.000000
mean         13.038807
std           4.313886
min           5.320000
25%           9.760000
50%          12.690000
75%          15.610000
max          28.990000
Name: int_rate, dtype: float64
```

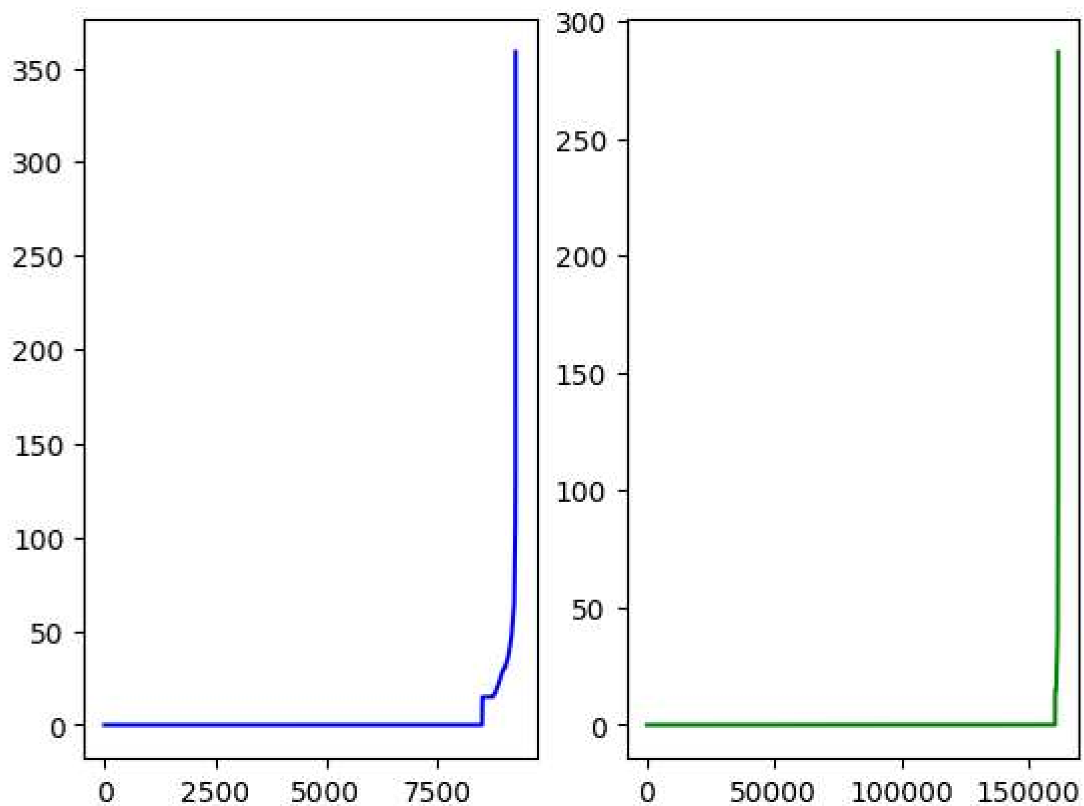
In [34]:
```
plt.subplot(1, 2, 1)
plt.plot(df3['int_rate'],c='blue')#default_ind=1
plt.subplot(1, 2, 2)
plt.plot(df4['int_rate'],c='green')#default_ind=0
plt.show()

#both defaulty and non-defaulty have similar intrest rate min,max and average,
```



In [35]:
```
df2=df.copy()
df2=df2.sort_values(['collection_recovery_fee'])
df2.reset_index(inplace=True)
default_ind_0=df2['default_ind']==0
default_ind_1=df2['default_ind']==1
df3=df2[default_ind_1]
df4=df2[default_ind_0]
df3.reset_index(inplace=True)
df4.reset_index(inplace=True)
plt.subplot(1, 2, 1)
plt.plot(df3['collection_recovery_fee'],c='blue')#default_ind=1
plt.subplot(1, 2, 2)
plt.plot(df4['collection_recovery_fee'],c='green')#default_ind=0
plt.show()
# we see that 'collection_recovery_fee' is 0 for all defaultys, so this feature
```



In [36]:
```
df2=df.copy()
df2=df2.sort_values(['total_rec_late_fee'])
df2.reset_index(inplace=True)
default_ind_0=df2['default_ind']==0
default_ind_1=df2['default_ind']==1
df3=df2[default_ind_1]
df4=df2[default_ind_0]
df3.reset_index(inplace=True)
df4.reset_index(inplace=True)
plt.subplot(1, 2, 1)
plt.plot(df3['total_rec_late_fee'],c='blue')#default_ind=1
plt.subplot(1, 2, 2)
plt.plot(df4['total_rec_late_fee'],c='green')#default_ind=0
plt.show()
#again we got similar graphs and the are quite similar,
```



In [37]:
```
df3['total_rec_late_fee'].describe()
```

Out[37]:
```
count    9281.000000
mean        2.360642
std        10.090214
min         0.000000
25%         0.000000
50%         0.000000
75%         0.000000
max       358.680000
Name: total_rec_late_fee, dtype: float64
```

In [38]:
```
df4['total_rec_late_fee'].describe()
# we can see that the mean, and the maximum is much lower for non-defaulty(defa
```

Out[38]:
```
count    161796.000000
mean          0.209447
std           2.990831
min           0.000000
25%           0.000000
50%           0.000000
75%           0.000000
max         286.747566
Name: total_rec_late_fee, dtype: float64
```

We Now Build Models,

In [39]:
```
from sklearn.metrics import recall_score,precision_score,accuracy_score,f1_scor
def performance_eval(y,ypred):
    print("Accuracy : ",accuracy_score(y,ypred))
    print("Recall :",recall_score(y,ypred))
    print("Precision :",precision_score(y,ypred))
    print("F1 Score:",f1_score(y,ypred))
    print("AUC :",roc_auc_score(y,ypred))
```

In [40]:
```
#Train_test split
from sklearn.model_selection import train_test_split

train_df,test_df=train_test_split(df,test_size=0.3, random_state=42, stratify=d
y_train=train_df['default_ind']
x_train=train_df.drop('default_ind',axis=1)

y_test=test_df['default_ind']
x_test=test_df.drop('default_ind',axis=1)
```

In [41]:
```

from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
params={'n_neighbors':[3,5,7,9,11]}
knn_mod=KNeighborsClassifier()
knn_grid=GridSearchCV(knn_mod,param_grid=params,cv=7,scoring='f1')
knn_grid.fit(x_train,y_train)
```

Out[41]:
```
▸ GridSearchCV
▸ estimator: KNeighborsClassifier
    ▸ KNeighborsClassifier
```

In [42]:
```
knn_grid.best_params_
```

Out[42]:
```
{'n_neighbors': 3}
```

In [43]:
```
#best knn, the model
knn_mod=KNeighborsClassifier(n_neighbors=3)
knn_mod.fit(x_train,y_train)
ypred=knn_mod.predict(x_test)
performance_eval(y_test,ypred)
# the model doesn't perform very well,
```

```
Accuracy :  0.9536669004754111
Recall : 0.24209770114942528
Precision : 0.7154989384288747
F1 Score: 0.3617820719269994
AUC : 0.6182882407683673
```

In [44]:
```
#decision tree
from sklearn.tree import DecisionTreeClassifier

dt=DecisionTreeClassifier(random_state=123)
dt_to_plot=dt.fit(x_train,y_train)
dt=dt.predict(x_test)
performance_eval(y_test,ypred)
```

```
Accuracy :  0.9536669004754111
Recall : 0.24209770114942528
Precision : 0.7154989384288747
F1 Score: 0.3617820719269994
AUC : 0.6182882407683673
```

In [45]:
```
from sklearn.tree import plot_tree
plot_tree(dt_to_plot)
```

In [46]:
```
#Random Forest

from sklearn.ensemble import RandomForestClassifier
rfc=RandomForestClassifier(n_estimators=69,random_state=123)
rfc.fit(x_train,y_train)
ypred=rfc.predict(x_test)
performance_eval(y_test,ypred)

# we see that the results are pretty good, it has high accuracy,AUC, and a good
```

```
Accuracy :  0.9976424284934923
Recall : 0.9568965517241379
Precision : 0.999624765478424
F1 Score: 0.9777940906588365
AUC : 0.9784379750792094
```

In [47]:
```
from sklearn.linear_model import LogisticRegression
lg=LogisticRegression(class_weight={0: 0.05425, 1: 0.94575})
params={'C':[0.01,0.1,0,1,10],'penalty':['l1', 'l2', 'elasticnet', None]}
lg_grid=GridSearchCV(lg,param_grid=params,cv=7,scoring='f1')
lg_grid.fit(x_train,y_train)
```

Out[47]:
```
▸ GridSearchCV
▸ estimator: LogisticRegression
  ▸ LogisticRegression
```

In [48]:
```
lg_grid.best_params_
```

Out[48]:
```
{'C': 0.01, 'penalty': 'l2'}
```

In [49]:
```
#best logistic regression
lr=LogisticRegression(class_weight={0: 0.05425, 1: 0.94575},C=0.01,penalty='l2'
lr.fit(x_train,y_train)
ypred=lr.predict(x_test)
performance_eval(y_test,ypred)
```

```
Accuracy :  0.9936676798378926
Recall : 0.9673132183908046
Precision : 0.9200546634779638
F1 Score: 0.9430922780598845
AUC : 0.98124622600628
```

In [50]:
```
from sklearn.naive_bayes import GaussianNB
nb = GaussianNB()
params={'priors':[[0.94575,0.05425],None]}
nb_grid=GridSearchCV(nb,param_grid=params,cv=7,scoring='f1')
nb_grid.fit(x_train,y_train)
nb_grid.best_params_
```

Out[50]:
```
{'priors': [0.94575, 0.05425]}
```

In [51]:
```
nb = GaussianNB(priors= [0.94575, 0.05425])
nb.fit(x_train,y_train)
ypred=nb.predict(x_test)
performance_eval(y_test,ypred)
```

```
Accuracy :  0.8640791832281194
Recall : 0.9342672413793104
Precision : 0.27687885884607194
F1 Score: 0.4271637378879947
AUC : 0.89716040272509
```

In [52]:
```
logistic_param_weights=pd.DataFrame(zip([x_train.columns.to_list()][0],lr.coef_
```

In [53]:
```
logistic_param_weights
```

Out[53]:

	0	1
0	member_id	-1.062964e-07
1	loan_amnt	7.406546e-04
2	funded_amnt	7.375541e-04
3	funded_amnt_inv	7.261827e-04
4	term	3.487377e-06
...	...	...
67	purpose_other	7.070247e-09
68	purpose_renewable_energy	2.023633e-10
69	purpose_small_business	3.869925e-09
70	purpose_vacation	1.281302e-09
71	purpose_wedding	3.545767e-10

72 rows × 2 columns

In [54]:
```
import math
logistic_param_weights['absoulute_value']=logistic_param_weights[1].apply(lambd
logistic_param_weights.describe()
```

Out[54]:

	1	absoulute_value
count	7.200000e+01	7.200000e+01
mean	-2.316220e-05	1.047809e-04
std	2.762482e-04	2.563658e-04
min	-9.923144e-04	0.000000e+00
25%	1.003067e-10	3.408619e-09
50%	1.087537e-08	5.295893e-08
75%	4.773216e-07	6.192984e-06
max	7.406546e-04	9.923144e-04

In [55]:
```
logistic_param_weights.sort_values(by=['absoulute_value'],ascending=False).head
```

Out[55]:

	0	1	absoulute_value
19	out_prncp	-0.000992	0.000992
20	out_prncp_inv	-0.000992	0.000992
23	total_rec_prncp	-0.000873	0.000873
1	loan_amnt	0.000741	0.000741
2	funded_amnt	0.000738	0.000738
3	funded_amnt_inv	0.000726	0.000726
22	total_pymnt_inv	-0.000625	0.000625
21	total_pymnt	-0.000625	0.000625
28	last_pymnt_amnt	-0.000471	0.000471
26	recoveries	0.000194	0.000194

In [56]:
```
rfc_feature_imp=pd.DataFrame(zip([x_train.columns.to_list()][0],rfc.feature_imp
```

In [57]:
```
import math
rfc_feature_imp['absoulute_value']=rfc_feature_imp[1].apply(lambda x:math.sqrt(
rfc_feature_imp.describe()
```

Out[57]:

	1	absoulute_value
count	72.000000	72.000000
mean	0.013889	0.013889
std	0.034031	0.034031
min	0.000000	0.000000
25%	0.000135	0.000135
50%	0.000764	0.000764
75%	0.006189	0.006189
max	0.210248	0.210248

In [58]:
```
rfc_feature_imp.sort_values(by=['absoulute_value'],ascending=False).head(10)
```

Out[58]:

	0	1	absoulute_value
26	recoveries	0.210248	0.210248
27	collection_recovery_fee	0.131294	0.131294
23	total_rec_prncp	0.113195	0.113195
19	out_prncp	0.070504	0.070504
28	last_pymnt_amnt	0.068871	0.068871
22	total_pymnt_inv	0.053084	0.053084
20	out_prncp_inv	0.048432	0.048432
21	total_pymnt	0.044764	0.044764
2	funded_amnt	0.036633	0.036633
1	loan_amnt	0.028127	0.028127