LoanDefaultProbability_STUDENT_V2

November 14, 2021

- Marcel Santos de Carvalho 79083
- Loris Baudry 79794
- Alejandro Palacios García 73713

Based on the data for loans from Lending Club available on Kaggle through 2007-2017Q3 we will try to build a machine learning model able to predict the probability that a loan will be charged-off. At the end of our analysis we will try to identify the best model for the task along with its hyperparameters. The case study will be structured in the following manner:

Index

1 Problem Definition

In the banking sector, one of the most relevant subjects for managers is how to better assess the probability of a person or company defaulting on its debt. It is hihgly important to have a good quality credit proftolio in ordert to have a healthy business. Machine Learning can provide powerful tools to approximate the probability of a loan, given its characteristics, to be charged-off in the future. A loan is charged-off by the bank after several months of missed payments. Based on the information available on Kaggle about some loans from 2007 to 2017Q3 we will try to come up with a model to accomplish exactly this task.

2 Getting Started- Loading the data and python packages

2.1 Load libraries

Given the nature of the problem presented here, we will be using pandas and numpy library to perform some exploratory analysis on the proposed dataset. We will be using the matplot lib and the seaborn methods in order to produce good looking plots which will help us to identify important characteristics embedded in the data.

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
!pip install scikit-plot
from sklearn.preprocessing import MinMaxScaler
import warnings
warnings.filterwarnings('ignore')
```

```
Requirement already satisfied: scikit-plot in c:\users\apala\anaconda3\lib\site-
packages (0.3.7)
Requirement already satisfied: joblib>=0.10 in
c:\users\apala\anaconda3\lib\site-packages (from scikit-plot) (1.0.1)
Requirement already satisfied: scipy>=0.9 in c:\users\apala\anaconda3\lib\site-
packages (from scikit-plot) (1.6.2)
Requirement already satisfied: matplotlib>=1.4.0 in
c:\users\apala\anaconda3\lib\site-packages (from scikit-plot) (3.3.4)
Requirement already satisfied: scikit-learn>=0.18 in
c:\users\apala\anaconda3\lib\site-packages (from scikit-plot) (0.24.1)
Requirement already satisfied: kiwisolver>=1.0.1 in
c:\users\apala\anaconda3\lib\site-packages (from matplotlib>=1.4.0->scikit-plot)
(1.3.1)
Requirement already satisfied: cycler>=0.10 in
c:\users\apala\anaconda3\lib\site-packages (from matplotlib>=1.4.0->scikit-plot)
(0.10.0)
```

```
Requirement already satisfied: python-dateutil>=2.1 in
c:\users\apala\anaconda3\lib\site-packages (from matplotlib>=1.4.0->scikit-plot)
(2.8.1)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.3 in
c:\users\apala\anaconda3\lib\site-packages (from matplotlib>=1.4.0->scikit-plot)
(2.4.7)
Requirement already satisfied: numpy>=1.15 in c:\users\apala\anaconda3\lib\site-
packages (from matplotlib>=1.4.0->scikit-plot) (1.20.1)
Requirement already satisfied: pillow>=6.2.0 in
c:\users\apala\anaconda3\lib\site-packages (from matplotlib>=1.4.0->scikit-plot)
(8.2.0)
Requirement already satisfied: six in c:\users\apala\anaconda3\lib\site-packages
(from cycler>=0.10->matplotlib>=1.4.0->scikit-plot) (1.15.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in
c:\users\apala\anaconda3\lib\site-packages (from scikit-learn>=0.18->scikit-
plot) (2.1.0)
```

2.2 Loading the Data

The data file provided is a zipped csv file which requires an additional argument in the function 'read_csv' in order to work well. This option is 'compression' which needs to be set equal to'gzip'. Other than that, we only set the first column of our dataset to be set as the index. The resulting dataframe from the file import has 150 variables and 100,000 observations. From the 150 variables, the "charge-off" variable will be the label that our model will be looking to predict.

[2]:
```
dataset = pd.read_csv('LoansData_sample.csv.gz', compression='gzip',␣
↪encoding='utf-8', index_col=0)
```

[3]:
```
dataset.tail()
```

[3]:
```
             id  member_id  loan_amnt  funded_amnt  funded_amnt_inv  \
99995  22454240        NaN     8400.0       8400.0           8400.0
99996  11396920        NaN    10000.0      10000.0          10000.0
99997   8556176        NaN    30000.0      30000.0          30000.0
99998  24023408        NaN     8475.0       8475.0           8475.0
99999  24023398        NaN    25000.0      25000.0          25000.0

             term  int_rate  installment grade sub_grade  …  \
99995   36 months      9.17       267.79     B        B1  …
99996   36 months     12.99       336.90     C        C1  …
99997   60 months     20.99       811.44     E        E4  …
99998   36 months     24.99       336.92     F        F4  …
99999   36 months     10.15       808.45     B        B2  …

      hardship_payoff_balance_amount hardship_last_payment_amount  \
99995                            NaN                          NaN
99996                            NaN                          NaN
99997                            NaN                          NaN
```

```
99998                              NaN                             NaN
99999                              NaN                             NaN

      disbursement_method  debt_settlement_flag debt_settlement_flag_date  \
99995                Cash                     N                       NaN
99996                Cash                     N                       NaN
99997                Cash                     N                       NaN
99998                Cash                     N                       NaN
99999                Cash                     N                       NaN

      settlement_status settlement_date settlement_amount  \
99995               NaN             NaN               NaN
99996               NaN             NaN               NaN
99997               NaN             NaN               NaN
99998               NaN             NaN               NaN
99999               NaN             NaN               NaN

      settlement_percentage settlement_term
99995                   NaN             NaN
99996                   NaN             NaN
99997                   NaN             NaN
99998                   NaN             NaN
99999                   NaN             NaN

[5 rows x 150 columns]
```

Examine the properties of the data frame

As we mentioned before, the data sample is composed by 100,000 observations of 149 features and 1 label. The label, loan_status, is a categorical variable indicating if a loan is still open ('Current'), closed ('Fully Paid'), present late payments (separeated beteween a less than or more than 30 days late), is in grace period, defaulted or if it has been 'Charged Off'. On the other hand, the feature space is composed by 116 numerical features and 33 categorical ones.

[4]:
```python
dataset.shape
```

[4]:
```
(100000, 150)
```

[5]:
```python
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 100000 entries, 0 to 99999
Columns: 150 entries, id to settlement_term
dtypes: float64(115), int64(1), object(34)
memory usage: 115.2+ MB
```

[6]:
```python
dataset.describe()
```

```
[6]:                  id  member_id      loan_amnt    funded_amnt  funded_amnt_inv  \
count  1.000000e+05        0.0  100000.000000  100000.000000    100000.000000
mean   3.029995e+07        NaN   14886.930000   14886.930000     14883.910500
std    4.763500e+06        NaN    8504.432514    8504.432514      8502.519174
min    5.716700e+04        NaN    1000.000000    1000.000000      1000.000000
25%    2.737015e+07        NaN    8000.000000    8000.000000      8000.000000
50%    3.052556e+07        NaN   13050.000000   13050.000000     13050.000000
75%    3.438201e+07        NaN   20000.000000   20000.000000     20000.000000
max    3.809811e+07        NaN   35000.000000   35000.000000     35000.000000

            int_rate    installment    annual_inc            dti  \
count  100000.000000  100000.000000  1.000000e+05  100000.000000
mean       13.278073     437.331824  7.468924e+04      18.769787
std         4.390210     244.317648  5.809527e+04       8.539769
min         6.000000      30.420000  4.000000e+03       0.000000
25%        10.150000     261.640000  4.500000e+04      12.320000
50%        12.990000     380.180000  6.400000e+04      18.210000
75%        15.610000     573.320000  9.000000e+04      24.760000
max        26.060000    1408.130000  7.500000e+06      39.990000

         delinq_2yrs  …  deferral_term  hardship_amount  hardship_length  \
count  100000.000000  …          185.0       185.000000            185.0
mean        0.343920  …            3.0       110.335568              3.0
std         0.906525  …            0.0        89.266601              0.0
min         0.000000  …            3.0         1.470000              3.0
25%         0.000000  …            3.0        23.760000              3.0
50%         0.000000  …            3.0        96.580000              3.0
75%         0.000000  …            3.0       164.750000              3.0
max        22.000000  …            3.0       382.340000              3.0

       hardship_dpd  orig_projected_additional_accrued_interest  \
count    185.000000                                  152.000000
mean      14.037838                                  323.495132
std        9.657374                                  267.627244
min        0.000000                                    4.410000
25%        7.000000                                   63.885000
50%       15.000000                                  281.580000
75%       22.000000                                  481.492500
max       32.000000                                 1147.020000

       hardship_payoff_balance_amount  hardship_last_payment_amount  \
count                      185.000000                    185.000000
mean                      8046.616541                    186.563135
std                       5585.653253                    168.552986
min                        174.150000                      0.040000
25%                       2465.360000                     27.610000
50%                       8049.850000                    172.460000
```

```
75%                    11968.940000                       285.890000
max                    21750.750000                       757.420000

       settlement_amount  settlement_percentage  settlement_term
count        1290.000000            1290.000000      1290.000000
mean         4768.376357              47.720519         8.265116
std          3703.963945               7.046587         8.263566
min           233.160000               0.550000         0.000000
25%          1951.125000              45.000000         0.000000
50%          3881.120000              45.040000         6.000000
75%          6503.000000              50.000000        14.000000
max         26751.740000             100.000000        36.000000

[8 rows x 116 columns]
```

[7]:
```
dataset.loan_status
```

[7]:
```
0         Fully Paid
1        Charged Off
2         Fully Paid
3            Current
4        Charged Off
            …
99995     Fully Paid
99996     Fully Paid
99997        Current
99998    Charged Off
99999     Fully Paid
Name: loan_status, Length: 100000, dtype: object
```

3 Data Preparation and Feature Selection

3.1 Preparing the predicted variable

As we can see below, the 'loan_status' feature can take 7 different values but 'Fully Paid', 'Charged Off' and 'Current' are the most relevant ones as they represent almos 99% of the dataset.

[8]:
```
dataset.loan_status.unique()
```

[8]:
```
array(['Fully Paid', 'Charged Off', 'Current', 'Late (31-120 days)',
       'Late (16-30 days)', 'In Grace Period', 'Default'], dtype=object)
```

[9]:
```
dataset.loan_status.value_counts()
```

[9]:
```
Fully Paid            69982
Charged Off           16156
Current               12726
Late (31-120 days)      730
```

```
In Grace Period         264
Late (16-30 days)       139
Default                   3
Name: loan_status, dtype: int64
```

Given how unbalanced the dataset is, we deiced to keep only the obervations labeled as 'Fully Paid' and 'Charged Off' as the othrs labels could add noise to our models withoug adding any relevant informaation. The observations labeled as 'Current' do not have any ameeded information regarding the default probability of a loan, so we deleted it as well.

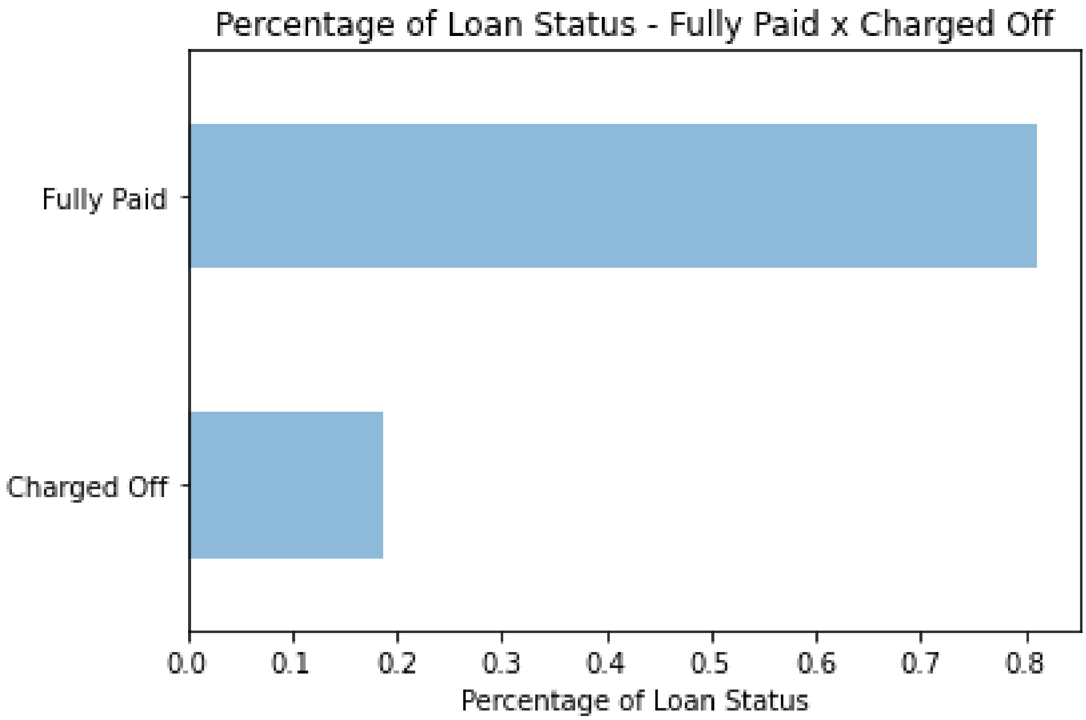
[10]:
```
# Deleting unconsidered loans
for _ in ['Current', 'Late (31-120 days)', 'In Grace Period', 'Late (16-30␣
↪days)', 'Default']:
    dataset.drop(dataset[dataset.loan_status == _].index, inplace=True)

dataset.loan_status.value_counts()
```

[10]:
```
Fully Paid     69982
Charged Off    16156
Name: loan_status, dtype: int64
```

Is this an unbalanced dataset ?

[11]:
```
# Creating graph for understanding weight of each loan status
plt.title("Percentage of Loan Status - Fully Paid x Charged Off")
numer = dataset.loan_status.value_counts().sort_index()
denom = dataset.loan_status.count()
(numer/denom).plot(kind='barh', alpha=0.5)
plt.xlabel("Percentage of Loan Status")
```

[11]:
```
Text(0.5, 0, 'Percentage of Loan Status')
```



As we can observe in the graphic above, the dataset is highly unbalanced in the sense that we have much more Fully Paid loans (~80%) than Charged Off (~20%). This is an important characteristic to have in midn when interpeting the results from our model as the conditional probability of a new datapoint being labeled as Fully Paid might be biased.

Set the labels to be 1 for Charged off else 0

In order to implement our Machine Learning classification models, we need to assing a numerical value to our categorical label. For this we will set the Fully Paid loans as 0 ans the ones Charged Off as 1.

[12]:
```
di = {"Fully Paid":0, "Charged Off":1}
dataset = dataset.replace({'loan_status': di})

dataset.loan_status.value_counts()
```

[12]:
```
0    69982
1    16156
Name: loan_status, dtype: int64
```

3.2 Feature Selection-Limit the Feature Space

The full dataset has 150 features for each loan. Toget a "cleaner" dataset we will next eliminate some features using three different approaches: * Eliminate features that have more than 30% missing values. * Eliminate features that are unintuitive based on subjective judgement. * Eliminate features with low correlation with the 'loan_status' label (less than 3%).

3.2.1 Features elimination by significant missing values

Calculating the percentage of missing data for each feature using isnull().mean()

[13]:
```
dataset.isnull().mean()
```

[13]:
```
id                      0.000000
member_id               1.000000
loan_amnt               0.000000
funded_amnt             0.000000
funded_amnt_inv         0.000000
                          …
settlement_status       0.985407
settlement_date         0.985407
settlement_amount       0.985407
settlement_percentage   0.985407
settlement_term         0.985407
Length: 150, dtype: float64
```

Drop the columns with more than 30% of missing data

[14]:
```
len(dataset.columns)
```

[14]:
```
150
```

[15]:
```
dataset.drop(dataset.columns[dataset.apply(lambda col: col.isnull().mean() > 0.
↪3)], axis=1, inplace=True)
```

[16]:
```
len(dataset.columns)
```

[16]:
```
92
```

As we can see above, 58 features were deleted in this step (150-92). It is important not to keep features with a high number of missing values as the information they can add will be too little and it can be misleading for our model.

How large is the remaining dataset ?

[17]:
```
dataset.shape
```

[17]:
```
(86138, 92)
```

3.2.2 Features elimination based on the intutiveness

Based on a check on the features description it is possible to reduce further the feature space by keeping only those features with a logic and relevant relation to wether a loan will be charged off or not. When performing a Machine Learning problem it is important to have a good understanding on the economic relationship between varuables to avoid the appearance of spurious correlations. This are the features we decided to keep. From the list of proposed features, we considered 'chargeoff_within_12_mths' instead of 'charged_off' as the latter did not exist in the data set.

[18]:
```
keep_list = ['chargeoff_within_12_mths','funded_amnt','addr_state',␣
↪'annual_inc', 'application_type',
             'dti', 'earliest_cr_line', 'emp_length', 'emp_title',␣
↪'fico_range_high',
             'fico_range_low', 'grade', 'home_ownership', 'id',␣
↪'initial_list_status',
             'installment', 'int_rate', 'loan_amnt', 'loan_status', 'mort_acc',␣
↪'open_acc',
             'pub_rec', 'pub_rec_bankruptcies', 'purpose', 'revol_bal',␣
↪'revol_util',
             'sub_grade', 'term', 'title', 'total_acc', 'verification_status',␣
↪'zip_code',
             'last_pymnt_amnt','num_actv_rev_tl',␣
↪'mo_sin_rcnt_rev_tl_op','mo_sin_old_rev_tl_op',
             "bc_util","bc_open_to_buy","avg_cur_bal","acc_open_past_24mths" ]
```

[19]:
```
for _ in dataset.columns:
    if _ not in keep_list:
        dataset.drop(_, axis=1, inplace=True)
```

How large is the remaining dataset ?

[20]:
```
dataset.shape
```

[20]:
```
(86138, 40)
```

3.2.3 Features elimination based on the correlation

Lastly, we will eliminate those features having a correlation of less than 3% with our label as that would mean that those features have a weak explaatory power for our label. To do this we will compute a correlation matrix to spot those features. To spot them easily we will, first, reorder the dataset columns, placing the label in the last column.

[21]:

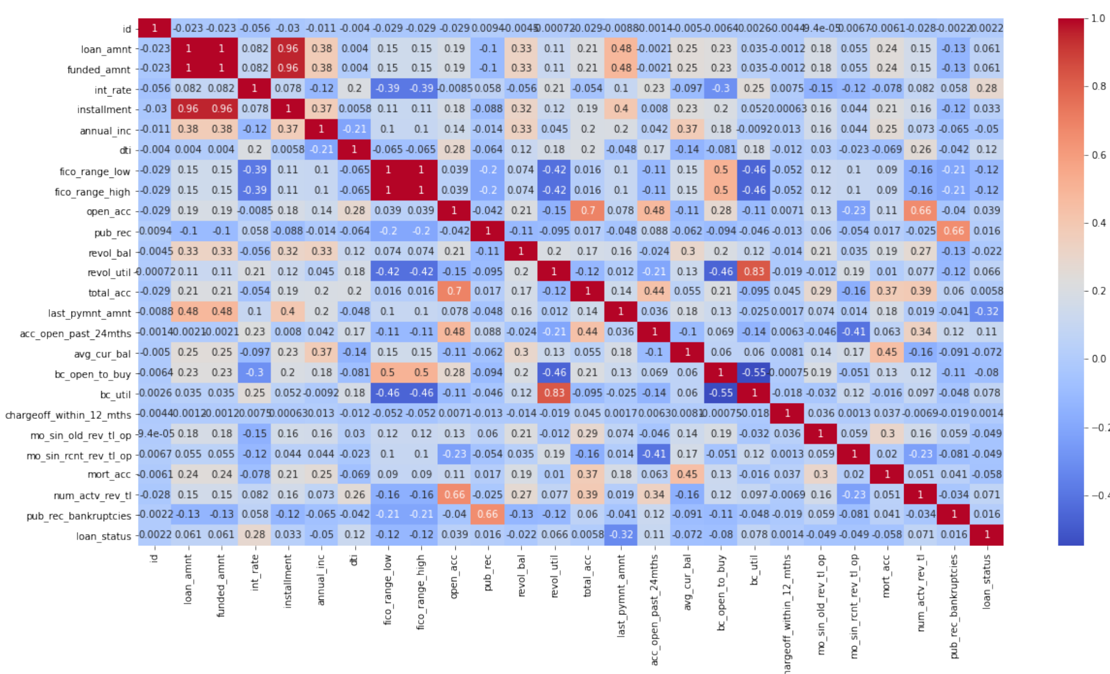
```
cols = ['id', 'loan_amnt', 'funded_amnt', 'term', 'int_rate', 'installment',␣
↪'grade', 'sub_grade', 'emp_title', 'emp_length', 'home_ownership',␣
↪'annual_inc', 'verification_status', 'purpose', 'title', 'zip_code',␣
↪'addr_state', 'dti', 'earliest_cr_line', 'fico_range_low',␣
↪'fico_range_high', 'open_acc', 'pub_rec', 'revol_bal', 'revol_util',␣
↪'total_acc', 'initial_list_status', 'last_pymnt_amnt', 'application_type',␣
↪'acc_open_past_24mths', 'avg_cur_bal', 'bc_open_to_buy', 'bc_util',␣
↪'chargeoff_within_12_mths', 'mo_sin_old_rev_tl_op', 'mo_sin_rcnt_rev_tl_op',␣
↪'mort_acc', 'num_actv_rev_tl', 'pub_rec_bankruptcies', 'loan_status']

dataset = dataset[cols]

dataset.shape
```

[21]: (86138, 40)

We will plot the correlation matrix to have a visual notion of which fetures have low label correlations. We will do this only for the numerical features as the correlation with the categorical features might be misleading.

[22]:
```
dataset_nums = dataset.select_dtypes(include=['int64','float64'])
corr = dataset_nums.corr()
plt.figure(figsize=(20,10))
sns.heatmap(corr, annot=True, cmap='coolwarm')
```

[22]: <AxesSubplot:>



[23]:
```
dataset['funded_amnt']
```

[23]:
```
0        15000.0
1        10400.0
2        21425.0
4         7650.0
5         9600.0
          …
99994    15000.0
99995     8400.0
99996    10000.0
99998     8475.0
99999    25000.0
Name: funded_amnt, Length: 86138, dtype: float64
```

[24]:
```
deleted_features =['funded_amnt','installment']

for k in deleted_features:
    dataset.drop(k, axis=1, inplace=True)

dataset_nums = dataset.select_dtypes(include=['int64','float64'])
for _ in dataset_nums.columns:
    if abs(dataset[_].corr(dataset['loan_status'])) < 0.03:
        dataset.drop(_, axis=1, inplace=True)
        deleted_features.append(_)
```

We will also delete the 'funded_amnt' feature as it is exactly correlated with 'loan_amnt' and 'installment' as its correlation is very close to 3% and is highly correlated with 'loan_amnt' as well. We did not treat the 'fico_range_low' and 'fico_range_high' in this step as they will be treated more specifically in the next section.

[25]:
```
deleted_features #This is the list of deleted features in this step
```

[25]:
```
['funded_amnt',
 'installment',
 'id',
 'pub_rec',
 'revol_bal',
 'total_acc',
 'chargeoff_within_12_mths',
 'pub_rec_bankruptcies']
```

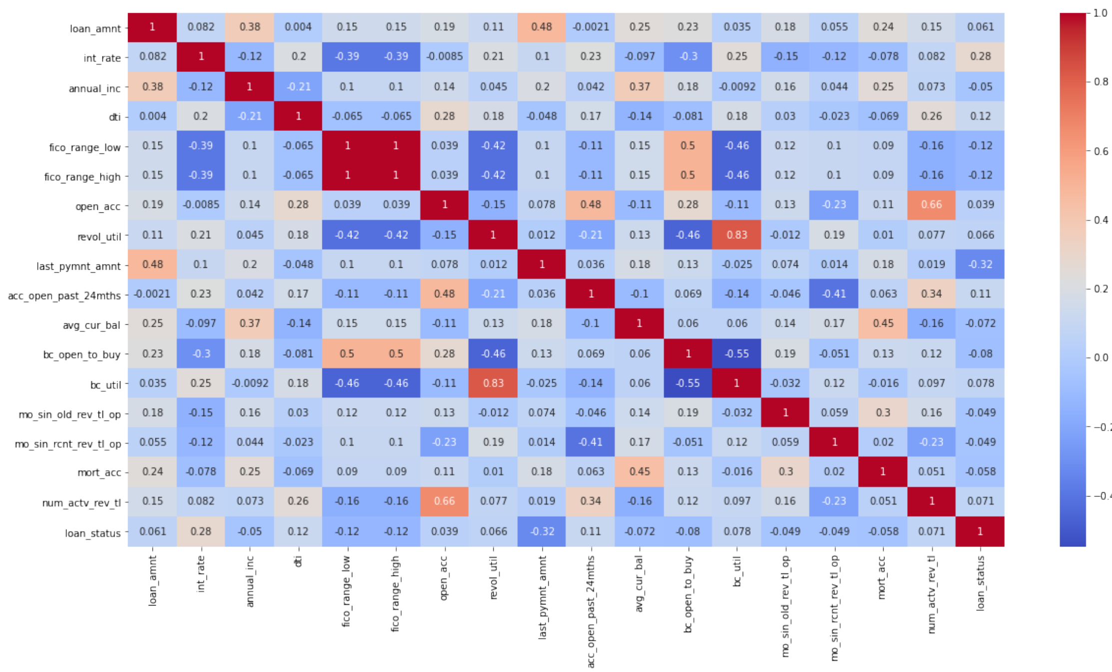
How large is the resulting dataset ?

[26]:
```
dataset.shape
```

[26]:
```
(86138, 32)
```

```
[27]: dataset_nums = dataset.select_dtypes(include=['int64','float64'])
      corr = dataset_nums.corr()
      plt.figure(figsize=(20,10))
      sns.heatmap(corr, annot=True, cmap='coolwarm')
```

```
[27]: <AxesSubplot:>
```



4 Feature Engineering and Exploratory Analysis

Examining the properties of the remaining features

The resulting dataframe has 86,138 observations of 31 explanatory variables and 1 label. From the explanatory variables we identified that there are 17 numerical ones and 14 categorical ones. The latter needs further processing in order to be used in our classification models so we will need to explore first which among them are worth keeping and which ones are not. Regarding the numerical features, it is easy to observe that the scale between some of them is very different, so we will scale the features with the MinMax Scaling method. We will exclude the 'annual_inc' feature from this process as it will be treated differently later because of a specific characteristic of this feature.

```
[28]: dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 86138 entries, 0 to 99999
Data columns (total 32 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
```

```
 0   loan_amnt             86138 non-null  float64
 1   term                  86138 non-null  object
 2   int_rate              86138 non-null  float64
 3   grade                 86138 non-null  object
 4   sub_grade             86138 non-null  object
 5   emp_title             81416 non-null  object
 6   emp_length            81421 non-null  object
 7   home_ownership        86138 non-null  object
 8   annual_inc            86138 non-null  float64
 9   verification_status   86138 non-null  object
 10  purpose               86138 non-null  object
 11  title                 86138 non-null  object
 12  zip_code              86138 non-null  object
 13  addr_state            86138 non-null  object
 14  dti                   86138 non-null  float64
 15  earliest_cr_line      86138 non-null  object
 16  fico_range_low        86138 non-null  float64
 17  fico_range_high       86138 non-null  float64
 18  open_acc              86138 non-null  float64
 19  revol_util            86094 non-null  float64
 20  initial_list_status   86138 non-null  object
 21  last_pymnt_amnt       86138 non-null  float64
 22  application_type      86138 non-null  object
 23  acc_open_past_24mths  86138 non-null  float64
 24  avg_cur_bal           86138 non-null  float64
 25  bc_open_to_buy        85142 non-null  float64
 26  bc_util               85089 non-null  float64
 27  mo_sin_old_rev_tl_op  86138 non-null  float64
 28  mo_sin_rcnt_rev_tl_op 86138 non-null  float64
 29  mort_acc              86138 non-null  float64
 30  num_actv_rev_tl       86138 non-null  float64
 31  loan_status           86138 non-null  int64
dtypes: float64(17), int64(1), object(14)
memory usage: 21.7+ MB
```

[29]:
```
dataset.describe()
```

[29]:
```
          loan_amnt      int_rate    annual_inc           dti  fico_range_low  \
count  86138.000000  86138.000000  8.613800e+04  86138.000000    86138.000000
mean   14106.526446     13.002360  7.384311e+04     18.532747      692.462966
std     8391.139221      4.397419  5.929352e+04      8.538247       29.731549
min     1000.000000      6.000000  4.000000e+03      0.000000      660.000000
25%     7800.000000      9.490000  4.500000e+04     12.070000      670.000000
50%    12000.000000     12.990000  6.247372e+04     17.950000      685.000000
75%    20000.000000     15.610000  9.000000e+04     24.500000      705.000000
max    35000.000000     26.060000  7.500000e+06     39.990000      845.000000
```

```
       fico_range_high      open_acc    revol_util  last_pymnt_amnt  \
count     86138.000000  86138.000000  86094.000000     86138.000000
mean        696.463024     11.746453     54.582777      4757.453184
std          29.731848      5.433122     23.515901      6466.767327
min         664.000000      1.000000      0.000000         0.000000
25%         674.000000      8.000000     37.200000       358.522500
50%         689.000000     11.000000     54.900000      1241.230000
75%         709.000000     14.000000     72.500000      7303.205000
max         850.000000     84.000000    180.300000     36234.440000

       acc_open_past_24mths    avg_cur_bal  bc_open_to_buy       bc_util  \
count          86138.000000   86138.000000    85142.000000  85089.000000
mean               4.594732   13066.638371     8942.506507     63.808959
std                3.070996   16232.739293    14100.186250     27.051347
min                0.000000       0.000000        0.000000      0.000000
25%                2.000000    3010.000000     1087.000000     44.100000
50%                4.000000    6994.500000     3823.000000     67.700000
75%                6.000000   17905.000000    10588.000000     87.500000
max               53.000000  447433.000000   249625.000000    255.200000

       mo_sin_old_rev_tl_op  mo_sin_rcnt_rev_tl_op      mort_acc  \
count          86138.000000           86138.000000  86138.000000
mean             183.524333              12.796896      1.748880
std               93.266430              16.224586      2.091488
min                3.000000               0.000000      0.000000
25%              118.000000               3.000000      0.000000
50%              167.000000               8.000000      1.000000
75%              232.000000              15.000000      3.000000
max              718.000000             372.000000     34.000000

       num_actv_rev_tl   loan_status
count     86138.000000  86138.000000
mean          5.762358      0.187559
std           3.224598      0.390362
min           0.000000      0.000000
25%           3.000000      0.000000
50%           5.000000      0.000000
75%           7.000000      0.000000
max          38.000000      1.000000
```

[30]:
```
dataset_numsc = dataset.select_dtypes(include=['int64','float64']).columns
dataset_numsc=dataset_numsc.drop('annual_inc')
dataset.loc[:,dataset_numsc] = MinMaxScaler().fit_transform(dataset.loc[:
 ↪,dataset_numsc])
dataset.describe()
```

[30]:

	loan_amnt	int_rate	annual_inc	dti	fico_range_low
count	86138.000000	86138.000000	8.613800e+04	86138.000000	86138.000000
mean	0.385486	0.349071	7.384311e+04	0.463435	0.175475
std	0.246798	0.219213	5.929352e+04	0.213510	0.160711
min	0.000000	0.000000	4.000000e+03	0.000000	0.000000
25%	0.200000	0.173978	4.500000e+04	0.301825	0.054054
50%	0.323529	0.348455	6.247372e+04	0.448862	0.135135
75%	0.558824	0.479063	9.000000e+04	0.612653	0.243243
max	1.000000	1.000000	7.500000e+06	1.000000	1.000000

	fico_range_high	open_acc	revol_util	last_pymnt_amnt
count	86138.000000	86138.000000	86094.000000	86138.000000
mean	0.174532	0.129475	0.302733	0.131296
std	0.159849	0.065459	0.130427	0.178470
min	0.000000	0.000000	0.000000	0.000000
25%	0.053763	0.084337	0.206323	0.009895
50%	0.134409	0.120482	0.304493	0.034256
75%	0.241935	0.156627	0.402108	0.201554
max	1.000000	1.000000	1.000000	1.000000

	acc_open_past_24mths	avg_cur_bal	bc_open_to_buy	bc_util
count	86138.000000	86138.000000	85142.000000	85089.000000
mean	0.086693	0.029204	0.035824	0.250035
std	0.057943	0.036280	0.056485	0.106001
min	0.000000	0.000000	0.000000	0.000000
25%	0.037736	0.006727	0.004355	0.172806
50%	0.075472	0.015633	0.015315	0.265282
75%	0.113208	0.040017	0.042416	0.342868
max	1.000000	1.000000	1.000000	1.000000

	mo_sin_old_rev_tl_op	mo_sin_rcnt_rev_tl_op	mort_acc
count	86138.000000	86138.000000	86138.000000
mean	0.252482	0.034400	0.051438
std	0.130443	0.043614	0.061514
min	0.000000	0.000000	0.000000
25%	0.160839	0.008065	0.000000
50%	0.229371	0.021505	0.029412
75%	0.320280	0.040323	0.088235
max	1.000000	1.000000	1.000000

	num_actv_rev_tl	loan_status
count	86138.000000	86138.000000
mean	0.151641	0.187559
std	0.084858	0.390362
min	0.000000	0.000000
25%	0.078947	0.000000
50%	0.131579	0.000000

```
75%            0.184211       0.000000
max            1.000000       1.000000
```

4.1 Feature Analysis and Exploration

4.1.1 Analysing the categorical features

After identifying the categorical features we will see the characteristics in each of them. We will drop those features having a high number of different levels as information embedded in those features is hard to capture by our models and they can add noise to our analysis.

[31]:
```python
categorical_feature = dataset.select_dtypes(include=object)
categorical_feature.columns
```

[31]:
```
Index(['term', 'grade', 'sub_grade', 'emp_title', 'emp_length',
       'home_ownership', 'verification_status', 'purpose', 'title', 'zip_code',
       'addr_state', 'earliest_cr_line', 'initial_list_status',
       'application_type'],
      dtype='object')
```

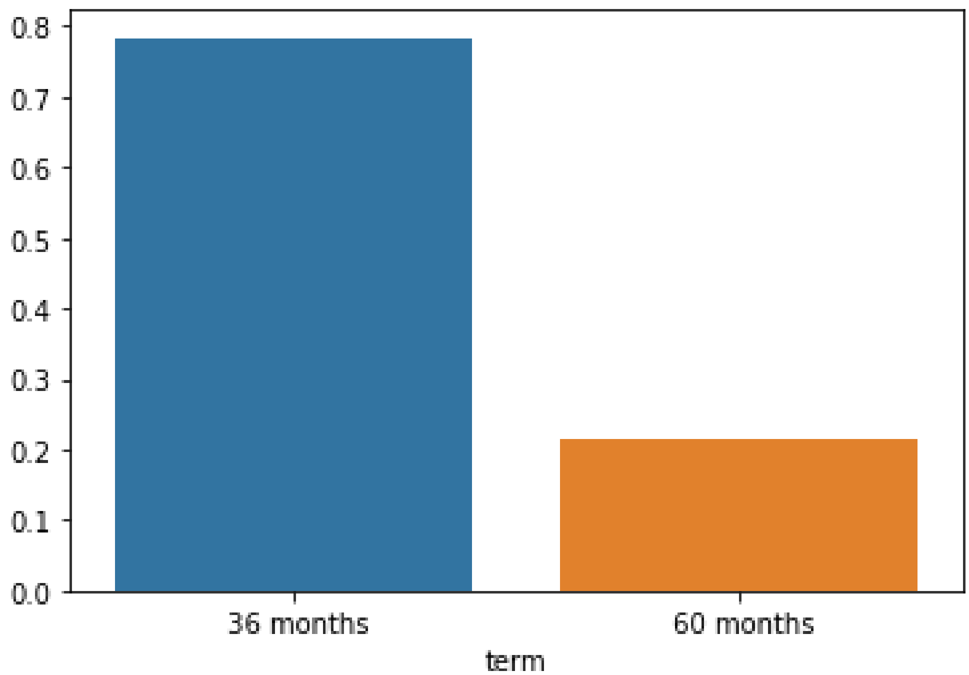
[32]:
```python
categorical_feature.head()
```

[32]:
```
         term grade sub_grade                        emp_title emp_length  \
0   60 months     C        C1                       MANAGEMENT  10+ years
1   36 months     A        A3   Truck Driver Delivery Personel    8 years
2   60 months     D        D1  Programming Analysis Supervisor    6 years
4   36 months     C        C3             Technical Specialist   < 1 year
5   36 months     C        C3                 Admin Specialist  10+ years

  home_ownership verification_status             purpose  \
0           RENT     Source Verified  debt_consolidation
1       MORTGAGE        Not Verified         credit_card
2           RENT     Source Verified         credit_card
4           RENT     Source Verified  debt_consolidation
5           RENT     Source Verified  debt_consolidation

                     title zip_code addr_state earliest_cr_line  \
0       Debt consolidation    235xx         VA         Aug-1994
1  Credit card refinancing    937xx         CA         Sep-1989
2  Credit card refinancing    658xx         MO         Aug-2003
4       Debt consolidation    850xx         AZ         Aug-2002
5       Debt consolidation    077xx         NJ         Nov-1992

  initial_list_status application_type
0                   w       Individual
1                   w       Individual
2                   w       Individual
4                   f       Individual
```

```
5                    f       Individual
```

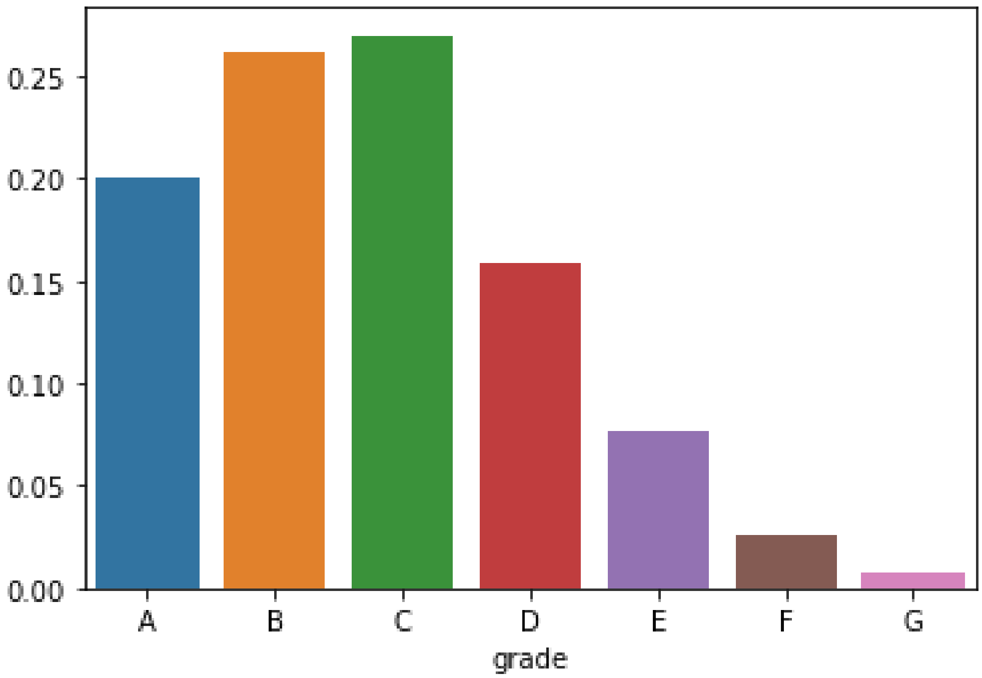
[33]:
```
loan_status_rates = dataset.groupby('term')['loan_status'].
 ↪value_counts(normalize=False).loc[:,1]+ dataset.
 ↪groupby('term')['loan_status'].value_counts(normalize=False).loc[:,0]
loan_status_rates = loan_status_rates/len(dataset.loan_status)
sns.barplot(x=loan_status_rates.index, y=loan_status_rates.values)
```

[33]: <AxesSubplot:xlabel='term'>



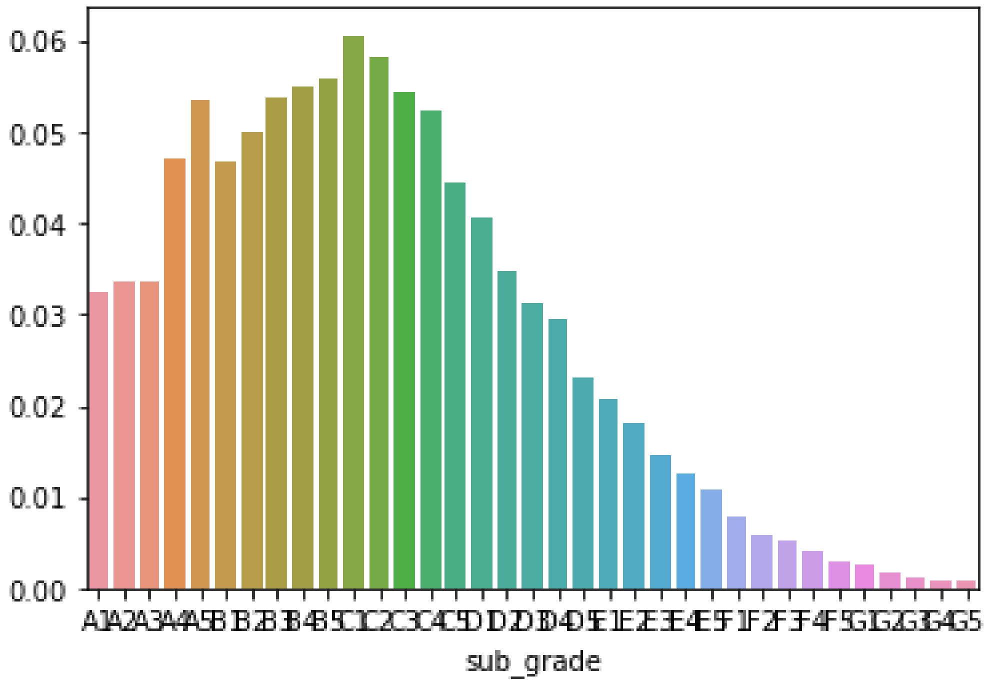
[34]:
```
loan_status_rates = dataset.groupby('grade')['loan_status'].
 ↪value_counts(normalize=False).loc[:,1]+dataset.
 ↪groupby('grade')['loan_status'].value_counts(normalize=False).loc[:,0]
loan_status_rates = loan_status_rates/len(dataset.loan_status)
sns.barplot(x=loan_status_rates.index, y=loan_status_rates.values)
```

[34]: <AxesSubplot:xlabel='grade'>



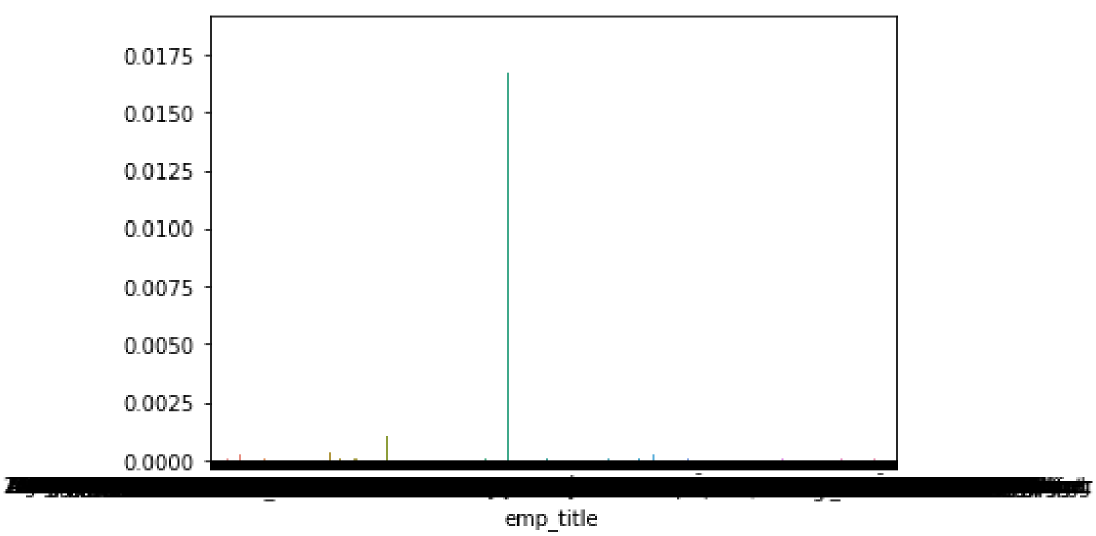
[35]:
```
loan_status_rates = dataset.groupby('sub_grade')['loan_status'].
 ↪value_counts(normalize=False).loc[:,1]+dataset.
 ↪groupby('sub_grade')['loan_status'].value_counts(normalize=False).loc[:,0]
loan_status_rates = loan_status_rates/len(dataset.loan_status)
sns.barplot(x=loan_status_rates.index, y=loan_status_rates.values)
```

[35]: <AxesSubplot:xlabel='sub_grade'>



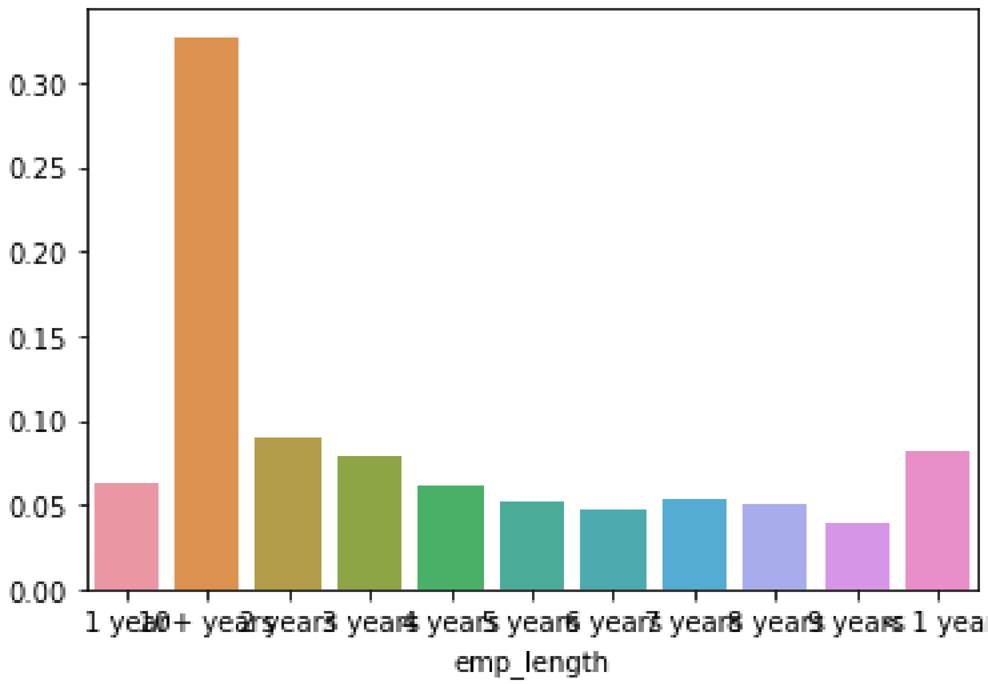

```
[36]: loan_status_rates = dataset.groupby('emp_title')['loan_status'].
       ↪value_counts(normalize=False).loc[:,1]+dataset.
       ↪groupby('emp_title')['loan_status'].value_counts(normalize=False).loc[:,0]
      loan_status_rates = loan_status_rates/len(dataset.loan_status)
      sns.barplot(x=loan_status_rates.index, y=loan_status_rates.values)
```

```
[36]: <AxesSubplot:xlabel='emp_title'>
```



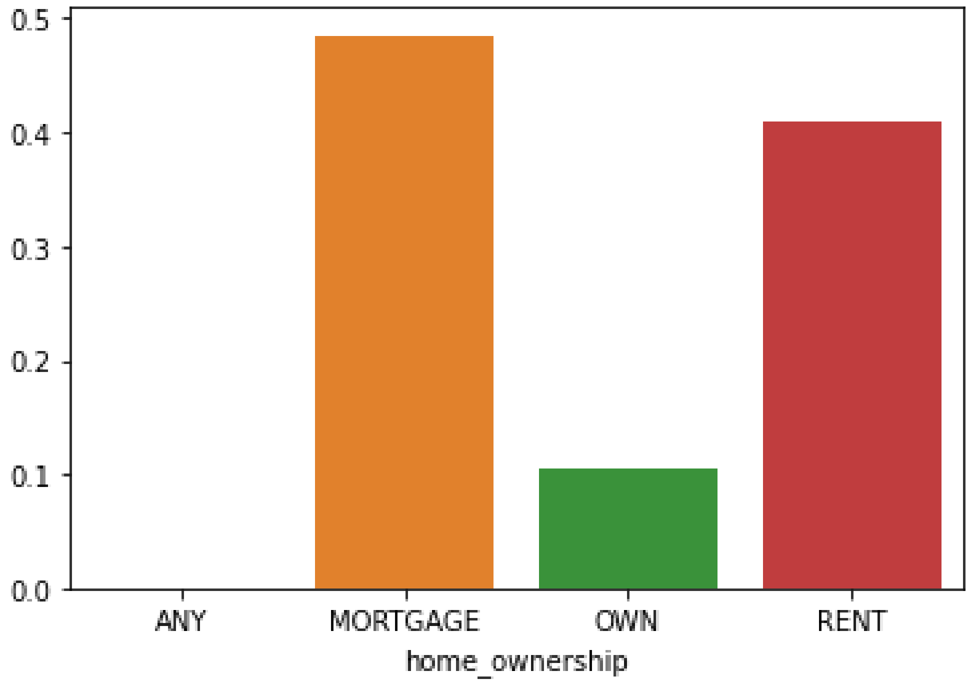
[37]:
```
loan_status_rates = dataset.groupby('emp_length')['loan_status'].
 ↪value_counts(normalize=False).loc[:,1]+dataset.
 ↪groupby('emp_length')['loan_status'].value_counts(normalize=False).loc[:,0]
loan_status_rates = loan_status_rates/len(dataset.loan_status)
sns.barplot(x=loan_status_rates.index, y=loan_status_rates.values)
```

[37]: <AxesSubplot:xlabel='emp_length'>



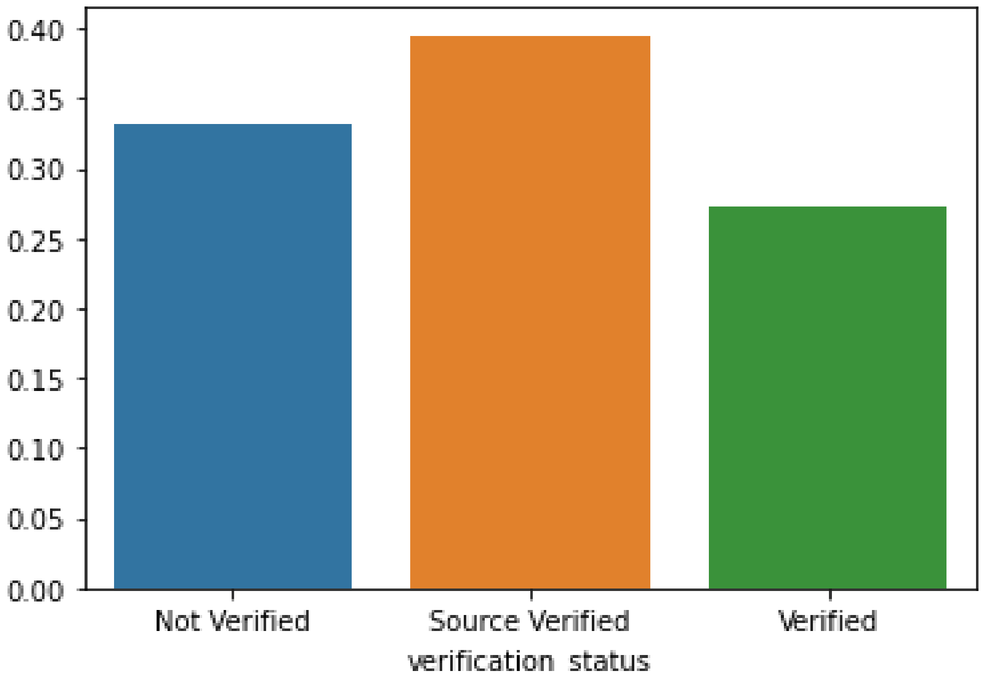
[38]:
```
loan_status_rates = dataset.groupby('home_ownership')['loan_status'].
 ↪value_counts(normalize=False).loc[:,1]+dataset.
 ↪groupby('home_ownership')['loan_status'].value_counts(normalize=False).loc[:
 ↪,0]
loan_status_rates = loan_status_rates/len(dataset.loan_status)
sns.barplot(x=loan_status_rates.index, y=loan_status_rates.values)
```

[38]: <AxesSubplot:xlabel='home_ownership'>



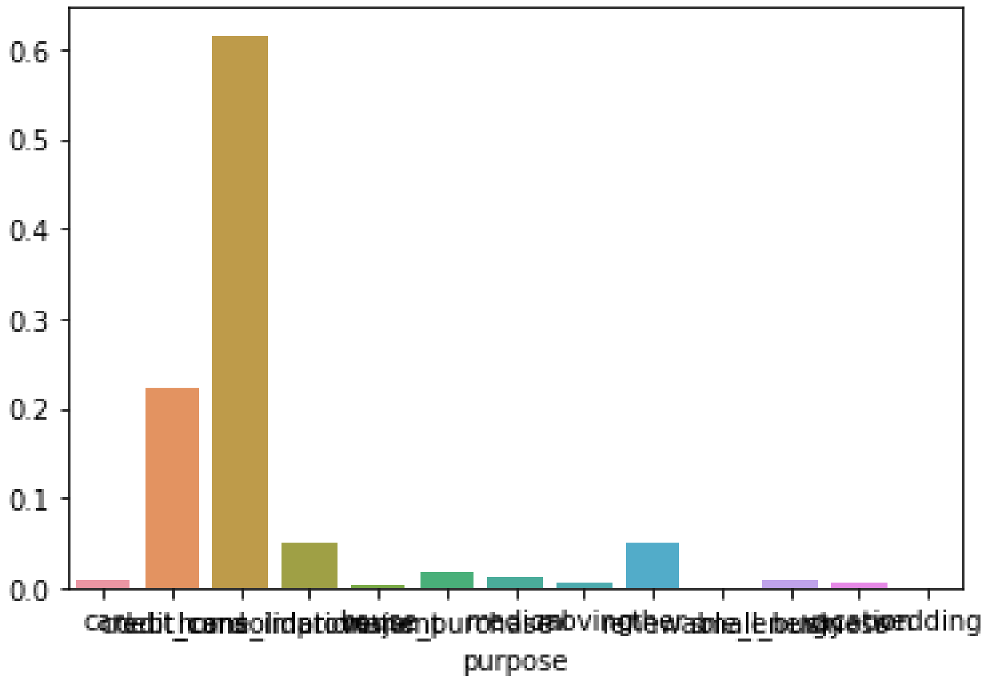
[39]:
```
loan_status_rates = dataset.groupby('verification_status')['loan_status'].
↪value_counts(normalize=False).loc[:,1]+dataset.
↪groupby('verification_status')['loan_status'].value_counts(normalize=False).
↪loc[:,0]
loan_status_rates = loan_status_rates/len(dataset.loan_status)
sns.barplot(x=loan_status_rates.index, y=loan_status_rates.values)
```

[39]:
```
<AxesSubplot:xlabel='verification_status'>
```



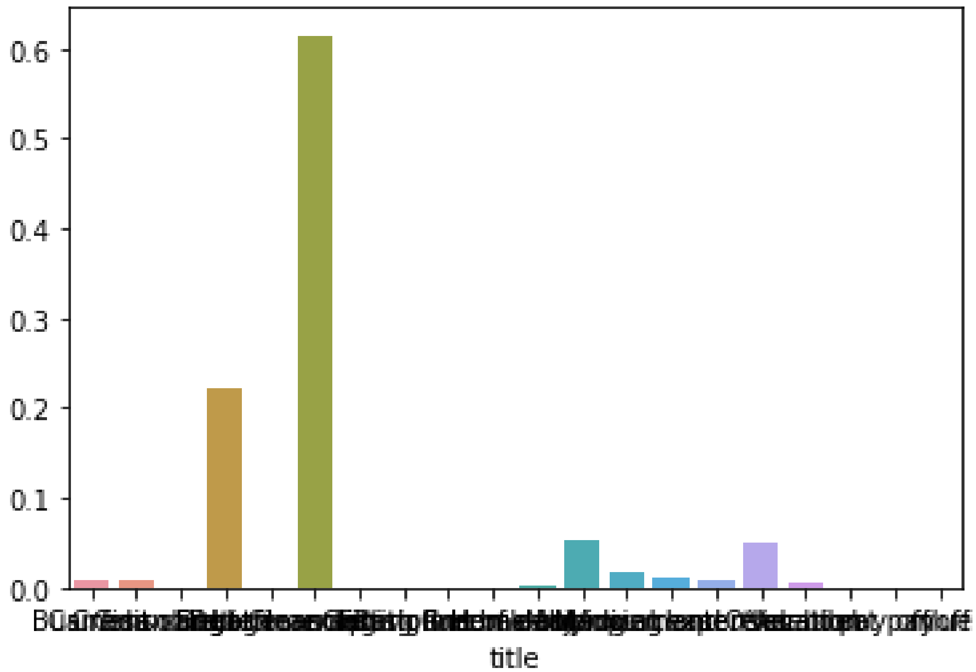
[40]:
```
loan_status_rates = dataset.groupby('purpose')['loan_status'].
 ↪value_counts(normalize=False).loc[:,1]+dataset.
 ↪groupby('purpose')['loan_status'].value_counts(normalize=False).loc[:,0]
loan_status_rates = loan_status_rates/len(dataset.loan_status)
sns.barplot(x=loan_status_rates.index, y=loan_status_rates.values)
```

[40]: <AxesSubplot:xlabel='purpose'>



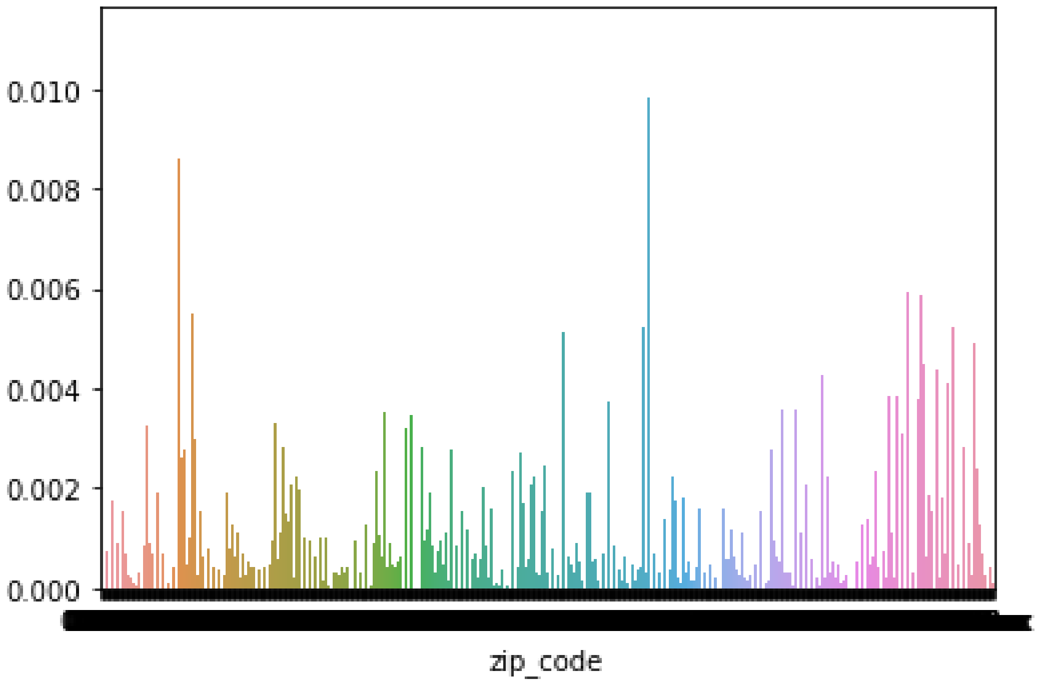
[41]:
```
loan_status_rates = dataset.groupby('title')['loan_status'].
 ↪value_counts(normalize=False).loc[:,1]+dataset.
 ↪groupby('title')['loan_status'].value_counts(normalize=False).loc[:,0]
loan_status_rates = loan_status_rates/len(dataset.loan_status)
sns.barplot(x=loan_status_rates.index, y=loan_status_rates.values)
```

[41]: <AxesSubplot:xlabel='title'>



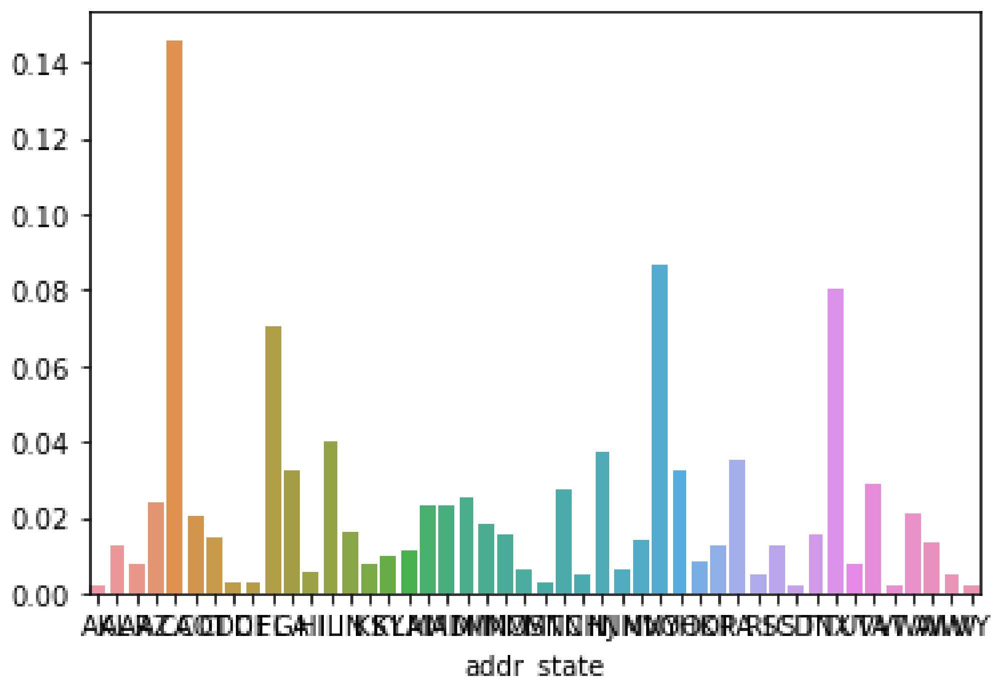
[42]:
```
loan_status_rates = dataset.groupby('zip_code')['loan_status'].
 ↪value_counts(normalize=False).loc[:,1]+dataset.
 ↪groupby('zip_code')['loan_status'].value_counts(normalize=False).loc[:,0]
loan_status_rates = loan_status_rates/len(dataset.loan_status)
sns.barplot(x=loan_status_rates.index, y=loan_status_rates.values)
```

[42]: <AxesSubplot:xlabel='zip_code'>



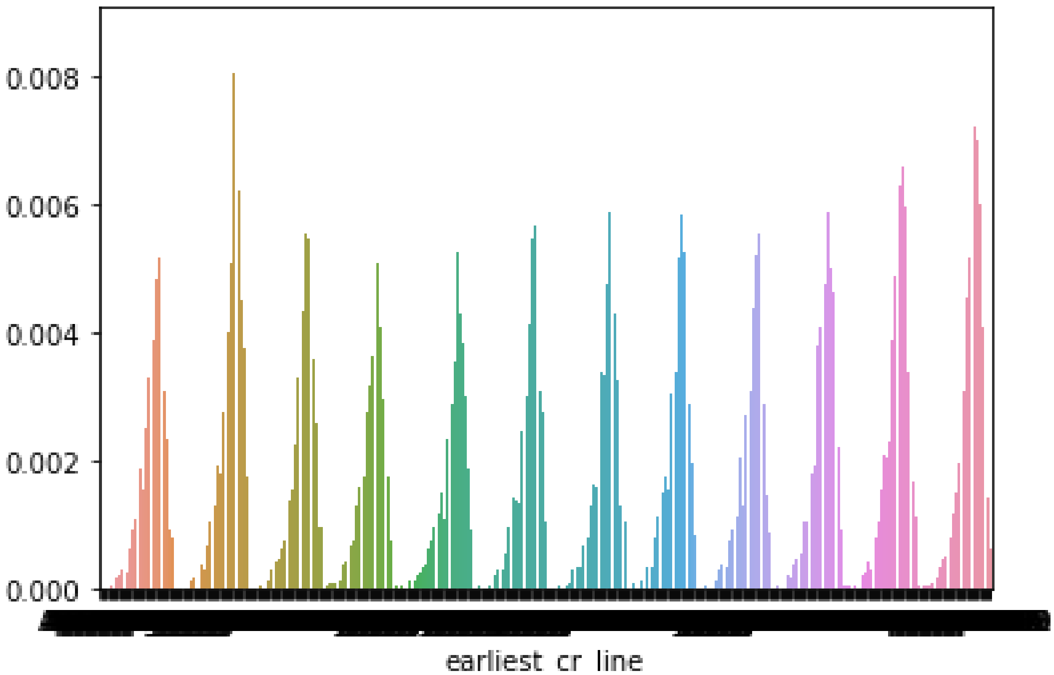
[43]:
```
loan_status_rates = dataset.groupby('addr_state')['loan_status'].
↪value_counts(normalize=False).loc[:,1]+dataset.
↪groupby('addr_state')['loan_status'].value_counts(normalize=False).loc[:,0]
loan_status_rates = loan_status_rates/len(dataset.loan_status)
sns.barplot(x=loan_status_rates.index, y=loan_status_rates.values)
```

[43]: <AxesSubplot:xlabel='addr_state'>



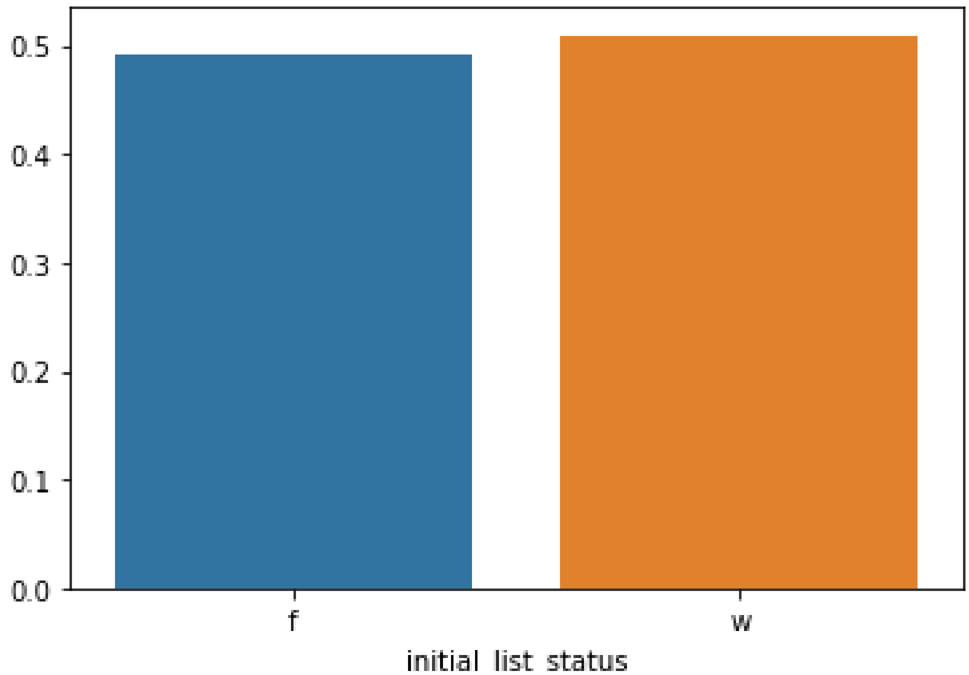

```
[44]: loan_status_rates = dataset.groupby('earliest_cr_line')['loan_status'].
      ↪value_counts(normalize=False).loc[:,1]+dataset.
      ↪groupby('earliest_cr_line')['loan_status'].value_counts(normalize=False).
      ↪loc[:,0]
      loan_status_rates = loan_status_rates/len(dataset.loan_status)
      sns.barplot(x=loan_status_rates.index, y=loan_status_rates.values)
```

```
[44]: <AxesSubplot:xlabel='earliest_cr_line'>
```



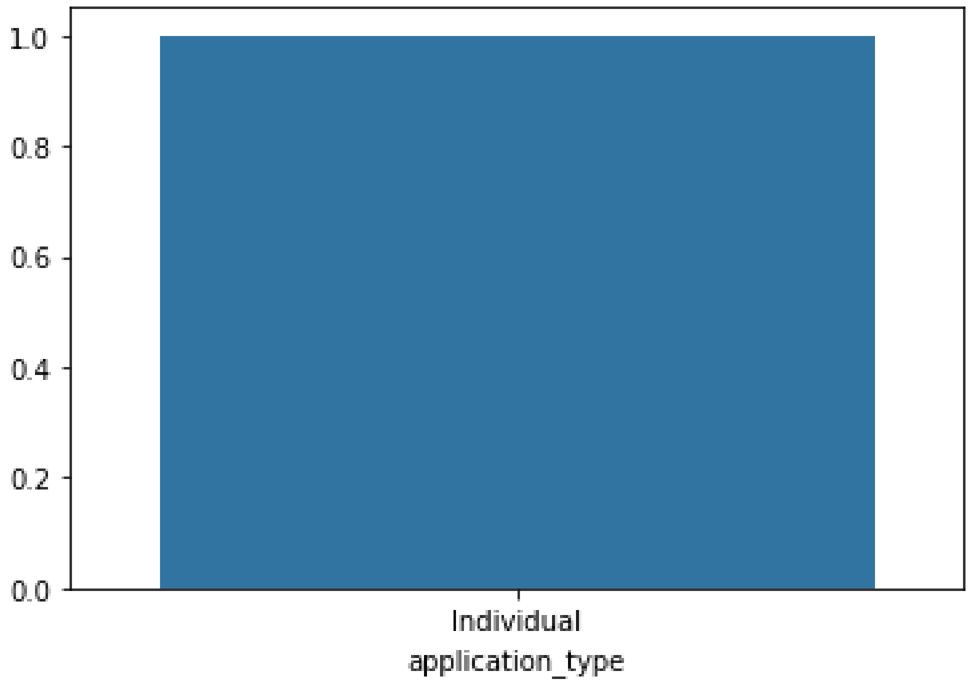
[45]:
```
loan_status_rates = dataset.groupby('initial_list_status')['loan_status'].
 ↪value_counts(normalize=False).loc[:,1]+dataset.
 ↪groupby('initial_list_status')['loan_status'].value_counts(normalize=False).
 ↪loc[:,0]
loan_status_rates = loan_status_rates/len(dataset.loan_status)
sns.barplot(x=loan_status_rates.index, y=loan_status_rates.values)
```

[45]: <AxesSubplot:xlabel='initial_list_status'>



[46]:
```
loan_status_rates = dataset.groupby('application_type')['loan_status'].
 ↪value_counts(normalize=False).loc[:,1]+dataset.
 ↪groupby('application_type')['loan_status'].value_counts(normalize=False).
 ↪loc[:,0]
loan_status_rates = loan_status_rates/len(dataset.loan_status)
sns.barplot(x=loan_status_rates.index, y=loan_status_rates.values)
```

[46]: <AxesSubplot:xlabel='application_type'>



[47]:
```
dataset['application_type'].describe()
```

[47]:
```
count           86138
unique              1
top        Individual
freq            86138
Name: application_type, dtype: object
```

It is easy to see that the application_type feature has a unique value 'Individual', and therefore has no explanatory power and we can drop it

[48]:
```
dataset.drop('application_type', axis=1, inplace=True)
```

We are dropping 'earliest_cr_line', 'zip_code' and 'emp_title' as well given the large number of levels in those features.

[49]:
```
dataset.drop(['earliest_cr_line', 'zip_code', 'emp_title'], axis=1,␣
 ↪inplace=True)
```

Convert 'term' to a numerical feature

As the models work better with numerical data instead of categorical data, we will now transform the feature 'term' to a numerical feature by removing the word months and keeping the number of months as the new feature.

[50]:
```
dataset['term'] = dataset['term'].apply(lambda s: np.int64(s.split()[0]))
u,c=np.unique(dataset['term'],return_counts=True)
loan_status_rates = dataset.groupby('term')['loan_status'].
 ↪value_counts(normalize=False)
loan_status_rates
```

[50]:
```
term  loan_status
36    0.0            57953
      1.0             9554
60    0.0            12029
      1.0             6602
Name: loan_status, dtype: int64
```

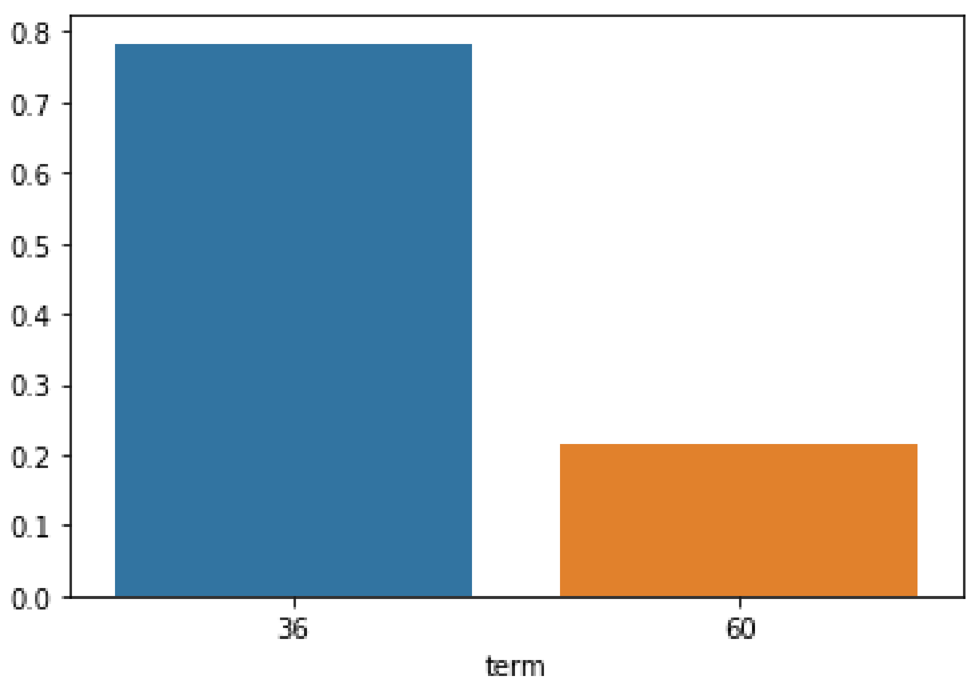
[51]:
```
dataset.groupby('term')['loan_status'].describe()
```

[51]:

term	count	mean	std	min	25%	50%	75%	max
36	67507.0	0.141526	0.348566	0.0	0.0	0.0	0.0	1.0
60	18631.0	0.354356	0.478330	0.0	0.0	0.0	1.0	1.0

[52]:
```
loan_status_rates = dataset.groupby('term')['loan_status'].
 ↪value_counts(normalize=False).loc[:,0]+ dataset.
 ↪groupby('term')['loan_status'].value_counts(normalize=False).loc[:,1]
loan_status_rates = loan_status_rates/len(dataset.loan_status)
sns.barplot(x=loan_status_rates.index, y=loan_status_rates.values)
```

[52]: <AxesSubplot:xlabel='term'>



[53]:
```
loan_status_rates
```

[53]:
```
term
36    0.783708
60    0.216292
Name: loan_status, dtype: float64
```

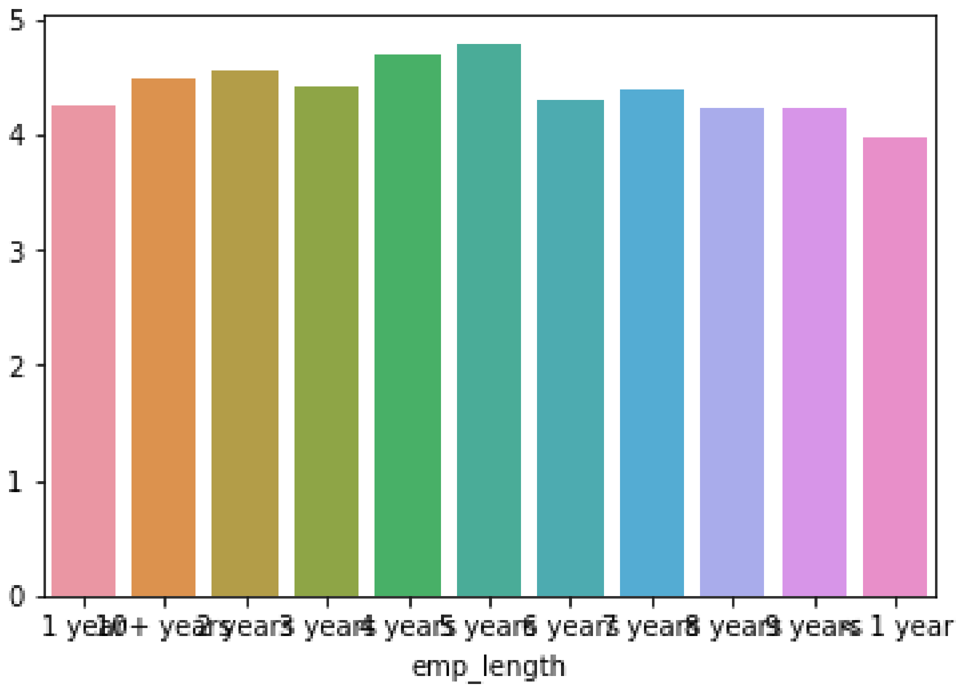
Decide if we need to do anything to emp_length

[54]:
```
loan_status_rates = dataset.groupby('emp_length')['loan_status'].
↪value_counts(normalize=False)
loan_status_rates
```

[54]:
```
emp_length  loan_status
1 year      0.0             4439
            1.0             1043
10+ years   0.0            23051
            1.0             5126
2 years     0.0             6316
            1.0             1383
3 years     0.0             5534
            1.0             1250
4 years     0.0             4334
            1.0              921
5 years     0.0             3739
            1.0              780
6 years     0.0             3329
            1.0              774
7 years     0.0             3731
            1.0              851
8 years     0.0             3494
            1.0              825
9 years     0.0             2784
            1.0              657
< 1 year    0.0             5644
            1.0             1416
Name: loan_status, dtype: int64
```

[55]:
```
loan_status_rates = dataset.groupby('emp_length')['loan_status'].
↪value_counts(normalize=False).loc[:,0]/dataset.
↪groupby('emp_length')['loan_status'].value_counts(normalize=False).loc[:,1]
sns.barplot(x=loan_status_rates.index, y=loan_status_rates.values)
```

[55]:
```
<AxesSubplot:xlabel='emp_length'>
```



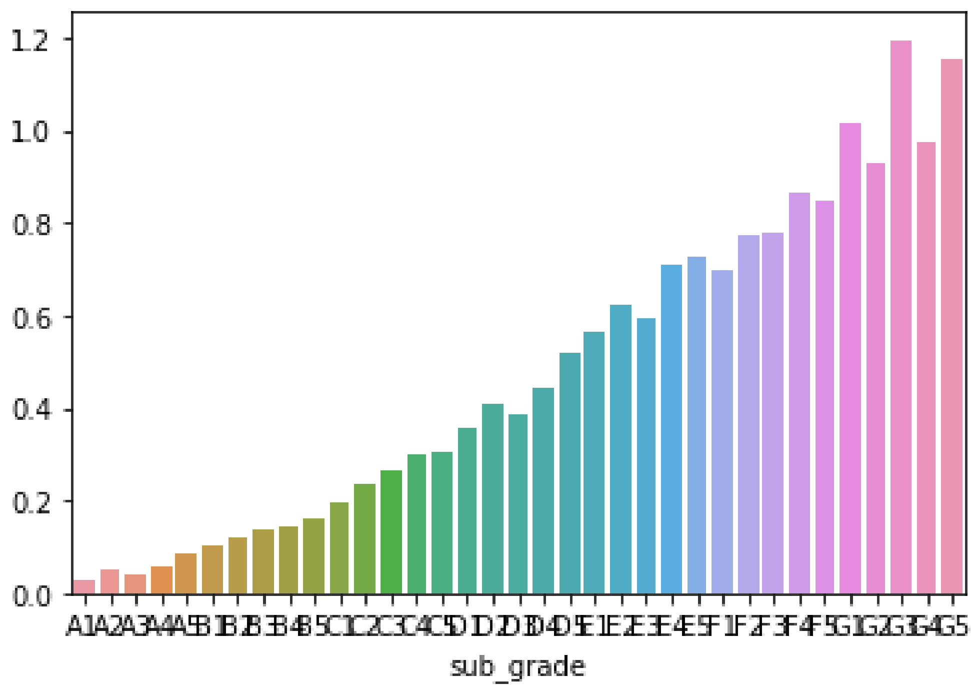
We will drop 'emp_length' since the proportion of loan status (Fully Paid/Charged Off) do not vary with emp_length values, which means that differentiating between the levels of 'empl_lenght' does not actually add much information.

[56]:
```
dataset.drop(['emp_length'], axis=1, inplace=True)
```

Is sub_grade worth keeping ?

[57]:
```
loan_status_rates = dataset.groupby('sub_grade')['loan_status'].
 ↪value_counts(normalize=True).loc[:,1]/ dataset.
 ↪groupby('sub_grade')['loan_status'].value_counts(normalize=True).loc[:,0]
sns.barplot(x=loan_status_rates.index, y=loan_status_rates.values)
```

[57]: <AxesSubplot:xlabel='sub_grade'>



In contrast of what happended with ‘emp_lenght’. Here, the trend of the proportion of loan status shows that ‘sub_grade’ is an explanatory feature of the loan status and, therefore, it is worth keeping it.

4.1.2 Analysing the continuous features

Do we need to do anything to Annual Income ?

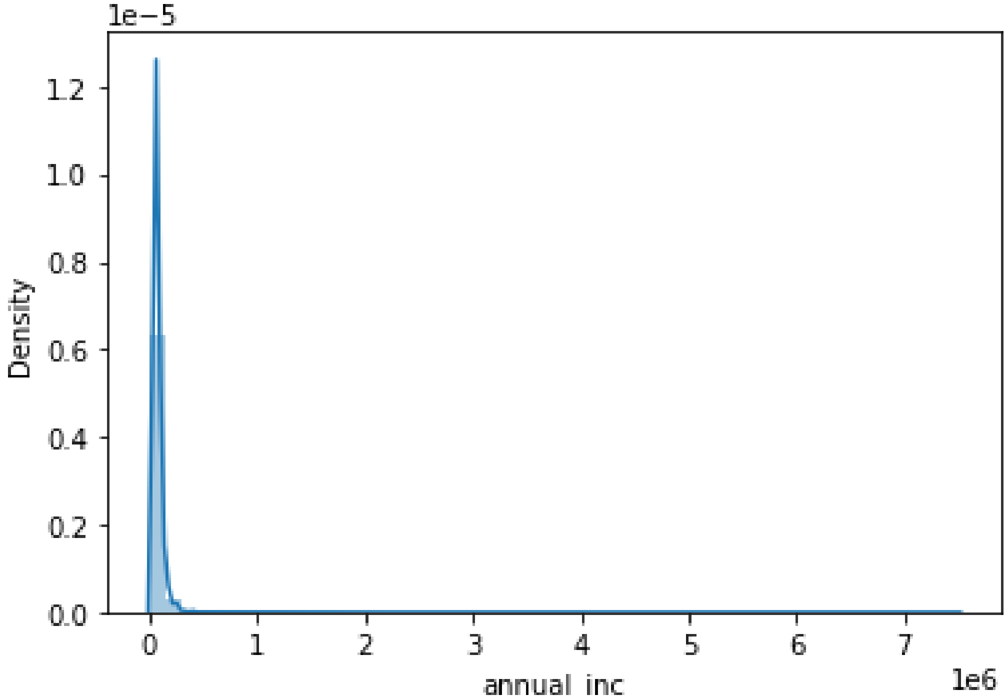
We are goint to explore if the Annual Income feature present some characteristics that suggest that further treatment of the feature is neccesary

Feature : Annual Income

[58]:
```
dataset.annual_inc.describe()
```

[58]:
```
count    8.613800e+04
mean     7.384311e+04
std      5.929352e+04
min      4.000000e+03
25%      4.500000e+04
50%      6.247372e+04
75%      9.000000e+04
max      7.500000e+06
Name: annual_inc, dtype: float64
```

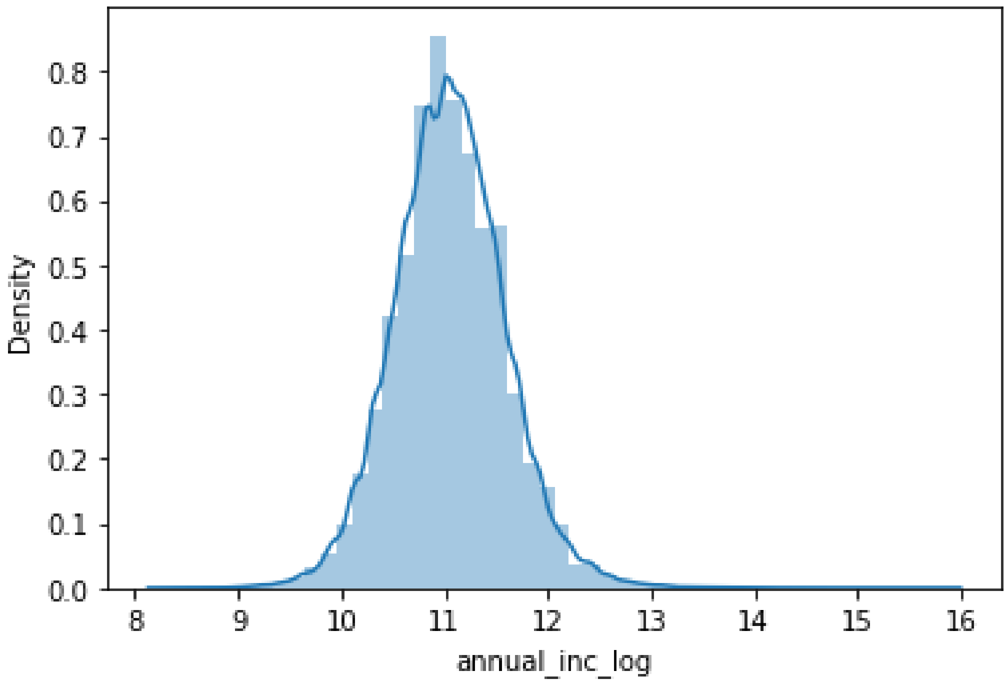
[59]:
```
sns.distplot(dataset['annual_inc'])
fig = plt.figure()
```




```
<Figure size 432x288 with 0 Axes>
```

From the graph above, we see that this feature shows skewness and we should, therefore, use a log transformation to avoid that the bais in the datasample to impact negatively our analysis.

[60]:
```
dataset['annual_inc_log'] = np.log(dataset['annual_inc'])
dataset.annual_inc_log.describe()
sns.distplot(dataset['annual_inc_log'])
fig = plt.figure()
```

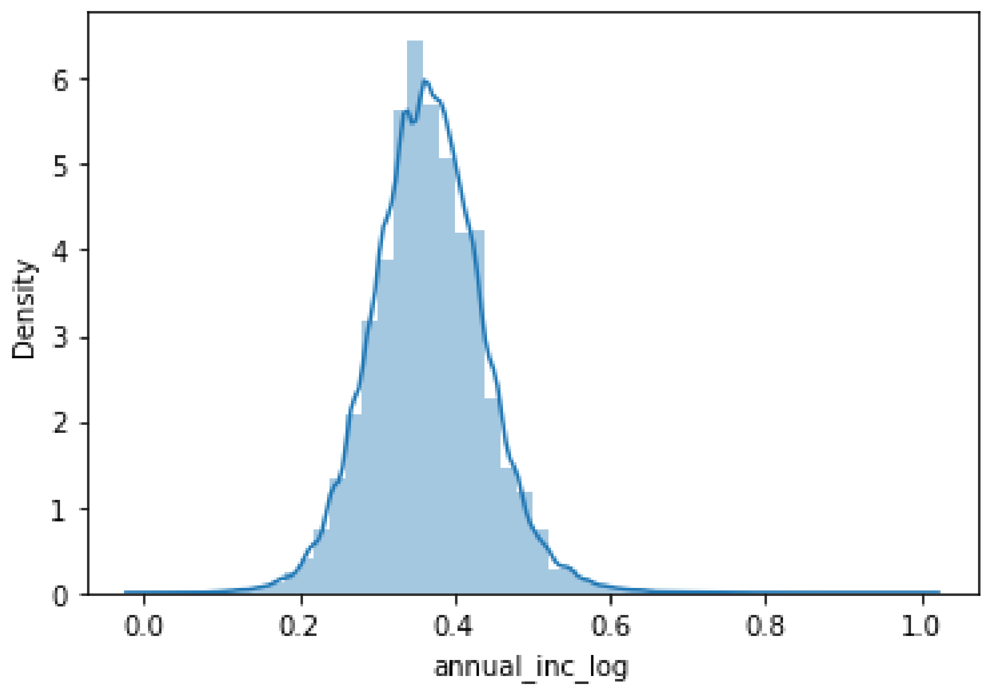


<Figure size 432x288 with 0 Axes>

[61]:
```
dataset['annual_inc_log'] = np.log(dataset['annual_inc'])
dataset.drop('annual_inc', axis=1, inplace=True)
scale=['annual_inc_log','loan_status']
dataset.loc[:,scale] = MinMaxScaler().fit_transform(dataset.loc[:,scale])
dataset.annual_inc_log.describe()
```

[61]:
```
count    86138.000000
mean         0.366314
std          0.071332
min          0.000000
25%          0.321159
50%          0.364692
75%          0.413132
max          1.000000
Name: annual_inc_log, dtype: float64
```

[62]:
```
sns.distplot(dataset['annual_inc_log'])
fig = plt.figure()
```




```
<Figure size 432x288 with 0 Axes>
```

[63]:
```
dataset.head()
```

[63]:
```
   loan_amnt  term  int_rate grade sub_grade home_ownership  \
0   0.411765    60  0.318544     C        C1           RENT
1   0.276471    36  0.049352     A        A3       MORTGAGE
2   0.600735    60  0.478066     D        D1           RENT
4   0.195588    36  0.381854     C        C3           RENT
5   0.252941    36  0.381854     C        C3           RENT

  verification_status             purpose                    title addr_state  \
0     Source Verified  debt_consolidation       Debt consolidation         VA
1        Not Verified         credit_card  Credit card refinancing         CA
2     Source Verified         credit_card  Credit card refinancing         MO
4     Source Verified  debt_consolidation       Debt consolidation         AZ
5     Source Verified  debt_consolidation       Debt consolidation         NJ

   …  acc_open_past_24mths  avg_cur_bal  bc_open_to_buy   bc_util  \
0  …              0.094340     0.066665        0.038157  0.018417
1  …              0.132075     0.021313        0.030442  0.162618
2  …              0.075472     0.009458        0.001298  0.383229
4  …              0.113208     0.013090        0.001330  0.365204
5  …              0.150943     0.007183        0.026015  0.271160
```

```
   mo_sin_old_rev_tl_op mo_sin_rcnt_rev_tl_op  mort_acc  num_actv_rev_tl  \
0              0.337063              0.002688  0.000000         0.105263
1              0.401399              0.002688  0.029412         0.236842
2              0.186014              0.018817  0.000000         0.105263
4              0.202797              0.021505  0.000000         0.105263
5              0.366434              0.061828  0.000000         0.184211

   loan_status  annual_inc_log
0          0.0        0.394144
1          1.0        0.354833
2          0.0        0.367479
4          1.0        0.335139
5          0.0        0.377876

[5 rows x 27 columns]
```

How do you want to treat the two FICO scores ? As we spotted earlier, the 'fico_range_low' and 'fico_range_high' features exhibited a correlation of 1. We are going to check if both features permit to explain the label. To do this we will analyze the correlation between the 2 FICO scores and between the FICO scores and the label

[64]:
```
dataset[['loan_status', 'fico_range_low', 'fico_range_high']].corr()
```

[64]:
```
                 loan_status  fico_range_low  fico_range_high
loan_status         1.000000       -0.121892        -0.121891
fico_range_low     -0.121892        1.000000         1.000000
fico_range_high    -0.121891        1.000000         1.000000
```

Given that there is a correlation of 1 between the 2 FICO scores and almost the same correlation with the label , we will take the average of the 2 scores and keep only 1 feature. This will be similar to just keeping any of the two features.

[65]:
```
dataset['FICO_Score']= 0.5*dataset['fico_range_low'] + 0.
 ↪5*dataset['fico_range_high']
dataset.drop(['fico_range_high', 'fico_range_low'], axis=1, inplace=True)
```

4.2 Encoding Categorical Data

We are going to process further the features 'grade', 'sub_grade', 'home_ownership', 'verification_status', 'purpose', 'addr_state' and 'initial_list_status' as wee need them to convert them to numerical features in order to use them in our classification models. To do this we will use the methdo Label Encoder from sklearn.preprocessing library.

[66]:
```
from sklearn.preprocessing import LabelEncoder
```

[67]:
```
categ = ['grade', 'sub_grade', 'home_ownership', 'verification_status',␣
 ↪'purpose','title', 'addr_state', 'initial_list_status']
# Encode Categorical Columns
```

```
le = LabelEncoder()
dataset[categ] = dataset[categ].apply(le.fit_transform)
```

[68]:
```
dataset.head()
```

[68]:
```
   loan_amnt  term  int_rate  grade  sub_grade  home_ownership  \
0   0.411765    60  0.318544      2         10               3
1   0.276471    36  0.049352      0          2               1
2   0.600735    60  0.478066      3         15               3
4   0.195588    36  0.381854      2         12               3
5   0.252941    36  0.381854      2         12               3

   verification_status  purpose  title  addr_state  …  avg_cur_bal  \
0                    1        2      5          40  …     0.066665
1                    0        1      3           4  …     0.021313
2                    1        1      3          21  …     0.009458
4                    1        2      5           3  …     0.013090
5                    1        2      5          26  …     0.007183

   bc_open_to_buy   bc_util  mo_sin_old_rev_tl_op  mo_sin_rcnt_rev_tl_op  \
0        0.038157  0.018417              0.337063               0.002688
1        0.030442  0.162618              0.401399               0.002688
2        0.001298  0.383229              0.186014               0.018817
4        0.001330  0.365204              0.202797               0.021505
5        0.026015  0.271160              0.366434               0.061828

   mort_acc  num_actv_rev_tl  loan_status  annual_inc_log  FICO_Score
0  0.000000         0.105263          0.0        0.394144    0.485179
1  0.029412         0.236842          1.0        0.354833    0.269544
2  0.000000         0.105263          0.0        0.367479    0.134772
4  0.000000         0.105263          1.0        0.335139    0.134772
5  0.000000         0.184211          0.0        0.377876    0.107817

[5 rows x 26 columns]
```

4.3 Sampling Data

Our final step to prepare the data will be to create a balanced dataset by drawing randomly 5500 rows from each of the two classes and combine them into a new dataframe.

[69]:
```
dataset.shape
```

[69]:
```
(86138, 26)
```

[70]:
```
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 86138 entries, 0 to 99999
```

```
Data columns (total 26 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   loan_amnt              86138 non-null  float64
 1   term                   86138 non-null  int64
 2   int_rate               86138 non-null  float64
 3   grade                  86138 non-null  int32
 4   sub_grade              86138 non-null  int32
 5   home_ownership         86138 non-null  int32
 6   verification_status    86138 non-null  int32
 7   purpose                86138 non-null  int32
 8   title                  86138 non-null  int32
 9   addr_state             86138 non-null  int32
 10  dti                    86138 non-null  float64
 11  open_acc               86138 non-null  float64
 12  revol_util             86094 non-null  float64
 13  initial_list_status    86138 non-null  int32
 14  last_pymnt_amnt        86138 non-null  float64
 15  acc_open_past_24mths   86138 non-null  float64
 16  avg_cur_bal            86138 non-null  float64
 17  bc_open_to_buy         85142 non-null  float64
 18  bc_util                85089 non-null  float64
 19  mo_sin_old_rev_tl_op   86138 non-null  float64
 20  mo_sin_rcnt_rev_tl_op  86138 non-null  float64
 21  mort_acc               86138 non-null  float64
 22  num_actv_rev_tl        86138 non-null  float64
 23  loan_status            86138 non-null  float64
 24  annual_inc_log         86138 non-null  float64
 25  FICO_Score             86138 non-null  float64
dtypes: float64(17), int32(8), int64(1)
memory usage: 15.1 MB
```

[71]:
```
dataset_o=dataset
Fully_Paid = dataset[dataset["loan_status"]==0]
Charged_Off = dataset[dataset["loan_status"]==1]
Fully_Paid_Subset = Fully_Paid.sample(n=5500, random_state=999)
Charged_Off_Subset = Charged_Off.sample(n=5500, random_state=999)
#dataset = dataset.sample(frac=1).reset_index(drop=True)
dataset = pd.concat([Charged_Off_Subset, Fully_Paid_Subset])
dataset.shape
```

[71]:
```
(11000, 26)
```

[72]:
```
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 11000 entries, 23614 to 76670
Data columns (total 26 columns):
```

```
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   loan_amnt             11000 non-null  float64
 1   term                  11000 non-null  int64
 2   int_rate              11000 non-null  float64
 3   grade                 11000 non-null  int32
 4   sub_grade             11000 non-null  int32
 5   home_ownership        11000 non-null  int32
 6   verification_status   11000 non-null  int32
 7   purpose               11000 non-null  int32
 8   title                 11000 non-null  int32
 9   addr_state            11000 non-null  int32
 10  dti                   11000 non-null  float64
 11  open_acc              11000 non-null  float64
 12  revol_util            10993 non-null  float64
 13  initial_list_status   11000 non-null  int32
 14  last_pymnt_amnt       11000 non-null  float64
 15  acc_open_past_24mths  11000 non-null  float64
 16  avg_cur_bal           11000 non-null  float64
 17  bc_open_to_buy        10859 non-null  float64
 18  bc_util               10851 non-null  float64
 19  mo_sin_old_rev_tl_op  11000 non-null  float64
 20  mo_sin_rcnt_rev_tl_op 11000 non-null  float64
 21  mort_acc              11000 non-null  float64
 22  num_actv_rev_tl       11000 non-null  float64
 23  loan_status           11000 non-null  float64
 24  annual_inc_log        11000 non-null  float64
 25  FICO_Score            11000 non-null  float64
dtypes: float64(17), int32(8), int64(1)
memory usage: 1.9 MB
```

It is easy to see that there are some Nas in both 'revol_util', 'bc_open_to_buy' and 'bc_until'. We will count the number of appearances that this has happended to see te best method to treat those cases.

[73]:
```
# Computing the number of NAs per column
dataset.isna().sum(axis=0)
```

[73]:
```
loan_amnt              0
term                   0
int_rate               0
grade                  0
sub_grade              0
home_ownership         0
verification_status    0
purpose                0
title                  0
addr_state             0
```

```
dti                       0
open_acc                  0
revol_util                7
initial_list_status       0
last_pymnt_amnt           0
acc_open_past_24mths      0
avg_cur_bal               0
bc_open_to_buy          141
bc_util                 149
mo_sin_old_rev_tl_op      0
mo_sin_rcnt_rev_tl_op     0
mort_acc                  0
num_actv_rev_tl           0
loan_status               0
annual_inc_log            0
FICO_Score                0
dtype: int64
```

[74]:
```
# Computing the number of lines that contain NAs
dataset.isna().sum(axis=1).astype(bool).sum()
```

[74]: 149

[75]:
```
# Since the number of lines with NA is irrelevant (~1.4% of total sample) we␣
↪can simply drop them
dataset.dropna(inplace=True)
dataset.shape
```

[75]: (10851, 26)

5 Evaluate Algorithms and Models

5.1 Train Test Split

Now, to fit the models to our data, we will separate out data into train test (80%) and test split(20%).

[76]:
```
X = dataset.loc[:, dataset.columns != "loan_status"]
y = dataset["loan_status"]
```

[77]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test  = train_test_split(X, y, test_size=0.2,␣
↪random_state = 999)
```

[78]:
```
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(8680, 25)
(2171, 25)
(8680,)
(2171,)
```

5.2 Test Options and Evaluation Metrics

The last step before fitting our models will be to determine the perfomance measure that we will be using. The fisrt measure that we will use is the K-Folds Cross Validation score. The idea behind this measure is to vary K times the chosen train and test sample in order to avoid overfitting. We will be doing this process 10 times and then averaging the accuracy measure of each fitting. The second measure will be the Area Under the Curve from the Receiver Operating Characteristic curve. This measure indicates the probability of a given model to correctly assinging a positive value to a new observation. To more easily compute this measures, we will be creating some functions only varying the threshold criteria in the case of the ROC curve.

[79]:
```
import sklearn
#from sklearn import metrics
from sklearn.metrics import make_scorer
from sklearn.model_selection import KFold, cross_val_score
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
from sklearn.metrics import accuracy_score
```

[80]:
```
#k_fold = KFold(n_splits=10)
#def cross(clsf, x,y,num_folds):
#    res = cross_val_score(clsf, x, y, cv=num_folds, n_jobs=-1)
#    res.mean()
```

[81]:
```
def plotROCCurve(fpr_v,tpr_v):
    plt.figure(figsize=(8, 8))
    plt.plot(fpr_v, tpr_v)
    plt.plot([0, 1], [0, 1], color='red', lw=2, linestyle='--')
    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')
```

[82]:
```
def aucs(clsf,X_train,X_test,y_train,y_test):
    clsf.fit(X_train,y_train)
    y_score = clsf.decision_function(X_test)
    fpr, tpr, thresholds = roc_curve(y_test, y_score)
    plotROCCurve(fpr,tpr)
    area = auc(fpr,tpr)
    print("AUC:", area)
    y_pred = clsf.predict(X_test)
    acc=accuracy_score(y_test,y_pred)
```

```
    print()
    return area,acc
```

[83]:
```
def aucs2(clsf,X_train,X_test,y_train,y_test):
    clsf.fit(X_train,y_train)
    y_score = clsf.predict_proba(X_test)
    fpr, tpr, thresholds = roc_curve(y_test, y_score[:,1])
    plotROCCurve(fpr,tpr)
    area = auc(fpr,tpr)
    print("AUC:", area)
    y_pred = clsf.predict(X_test)
    acc=accuracy_score(y_test,y_pred)
    print()
    return area,acc
```

5.3 Compare Models and Algorithms

We are ready now to start fitting our Machine Learning classification methods. We will be fitting a logistic regression model, a K-Nearest Neighbors model, a Decision Tree model, a Naive Bayes model, a Neural Network Model and two ensemble methods: the Random Forest and the Boosting based on AdaBoost. To select the best perfroming model we will be using the two different performance measure described above.

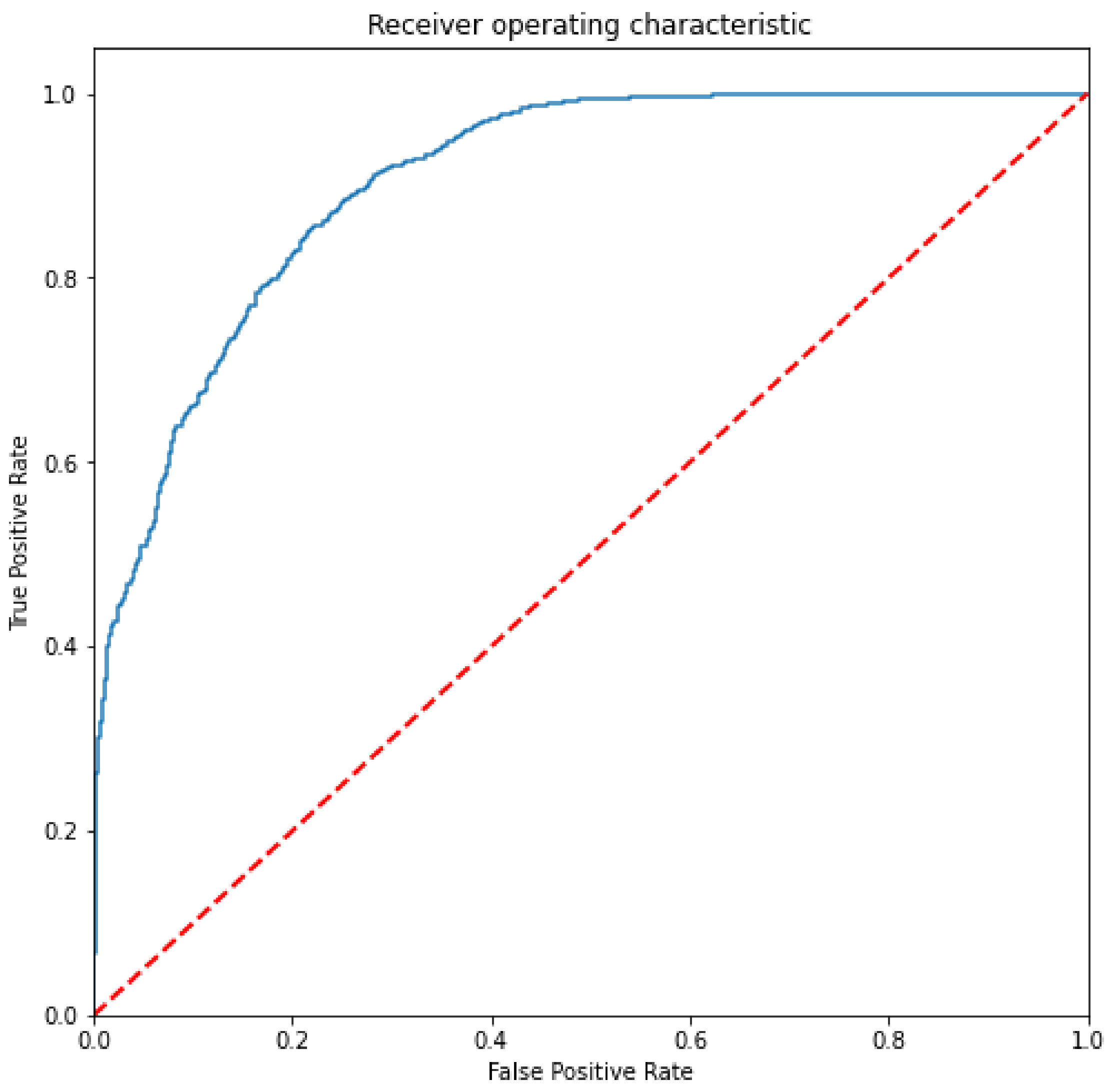
[84]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import RandomForestClassifier
```

[85]:
```
index=['LR','KNN','DT','NB','NN','AB','RF']
aucscore=pd.DataFrame({'AUC score':pd.Series(dtype='float'),'Accuracy':pd.
 ↪Series(dtype='float')},index=index)
aucscore
```

[85]:
```
     AUC score  Accuracy
LR         NaN       NaN
KNN        NaN       NaN
DT         NaN       NaN
NB         NaN       NaN
NN         NaN       NaN
AB         NaN       NaN
RF         NaN       NaN
```

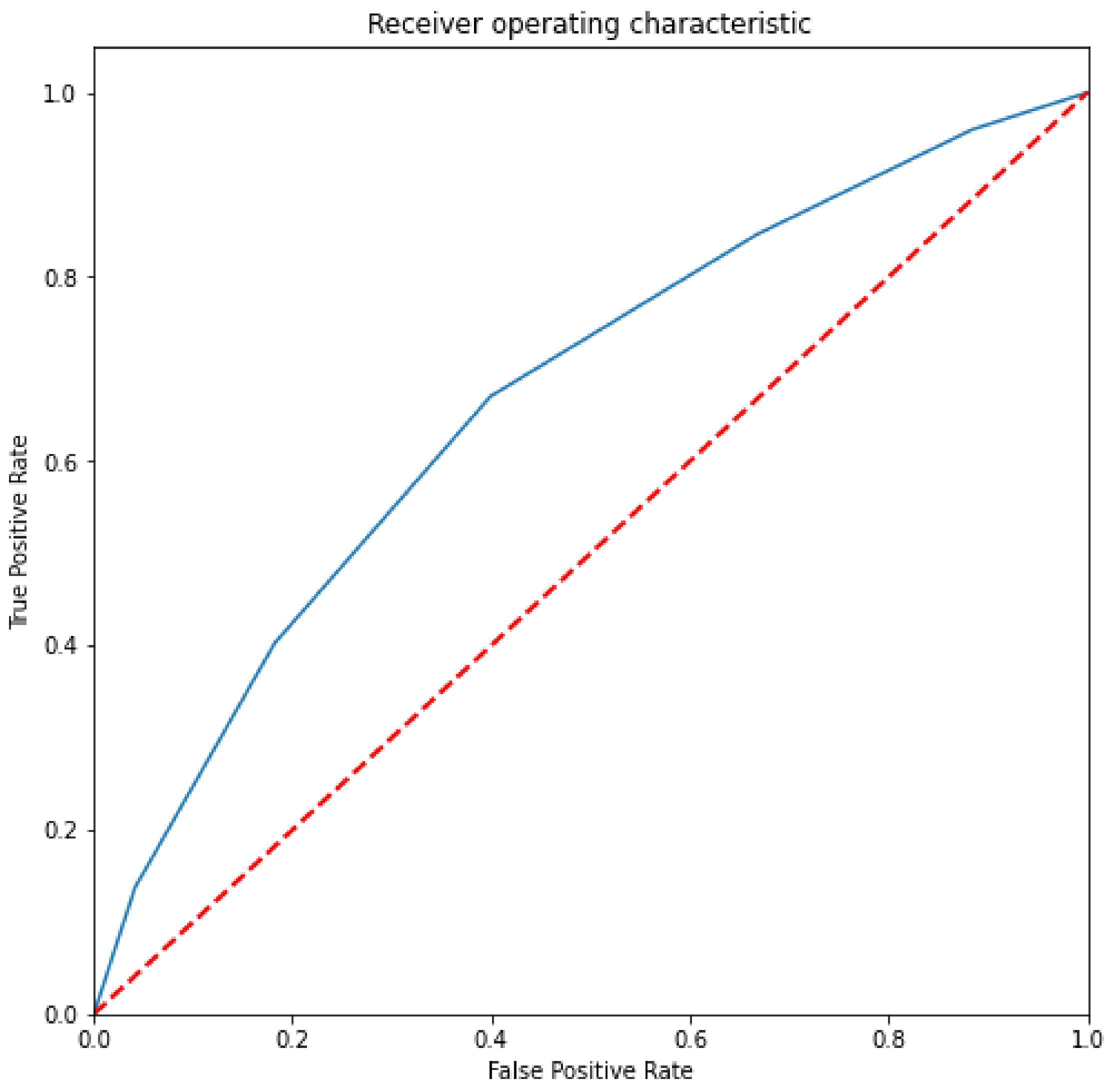
[86]:
```
clsf = LogisticRegression()
a=aucs(clsf,X_train,X_test,y_train,y_test)
aucscore.iloc[0]=a
```

AUC: 0.9056069434823357



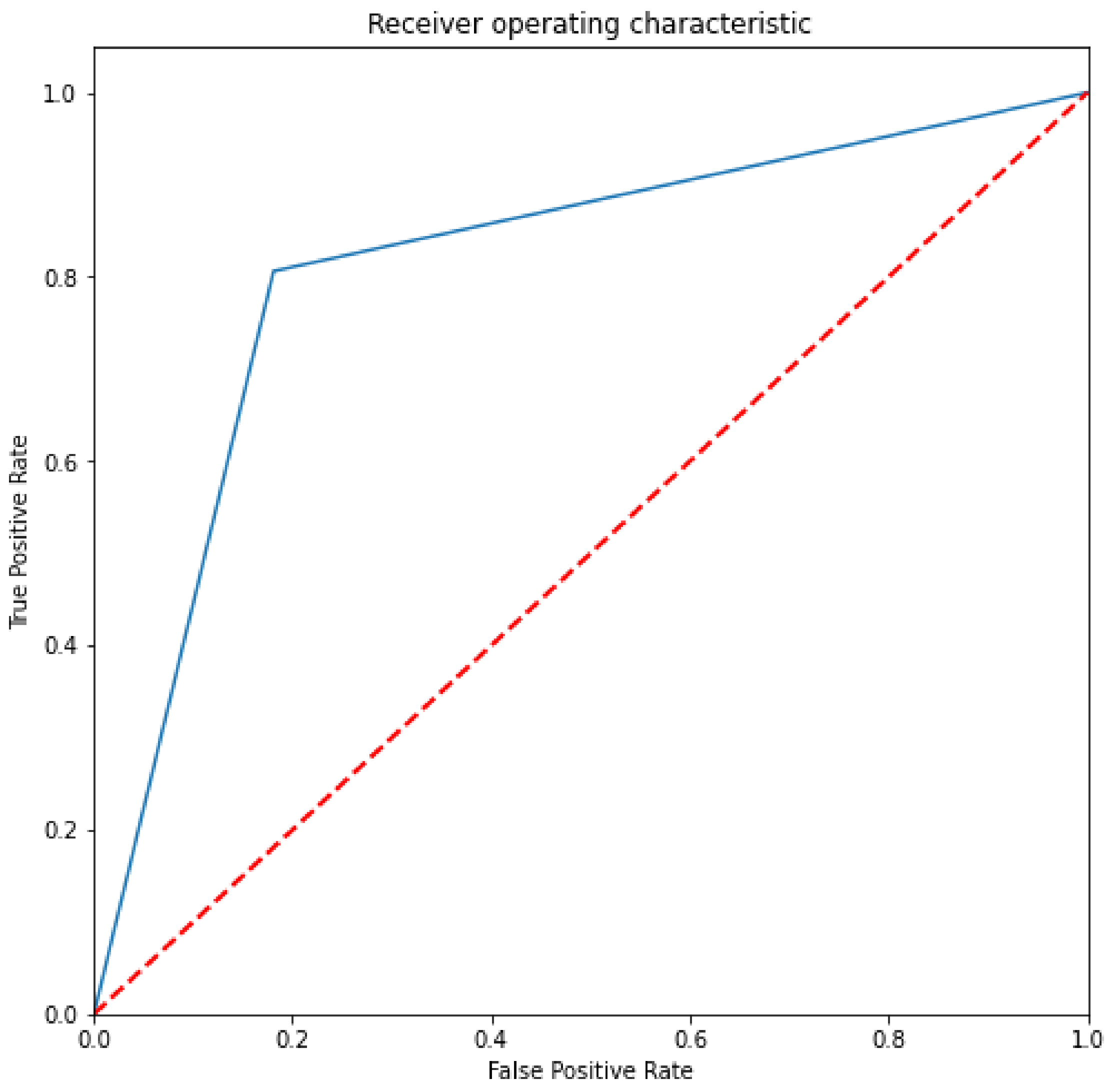
[87]:
```
clsf = KNeighborsClassifier()
a=aucs2(clsf,X_train,X_test,y_train,y_test)
aucscore.iloc[1]=a
```

AUC: 0.6698509636934109



[88]:
```
clsf = DecisionTreeClassifier()
a=aucs2(clsf,X_train,X_test,y_train,y_test)
aucscore.iloc[2]=a
```

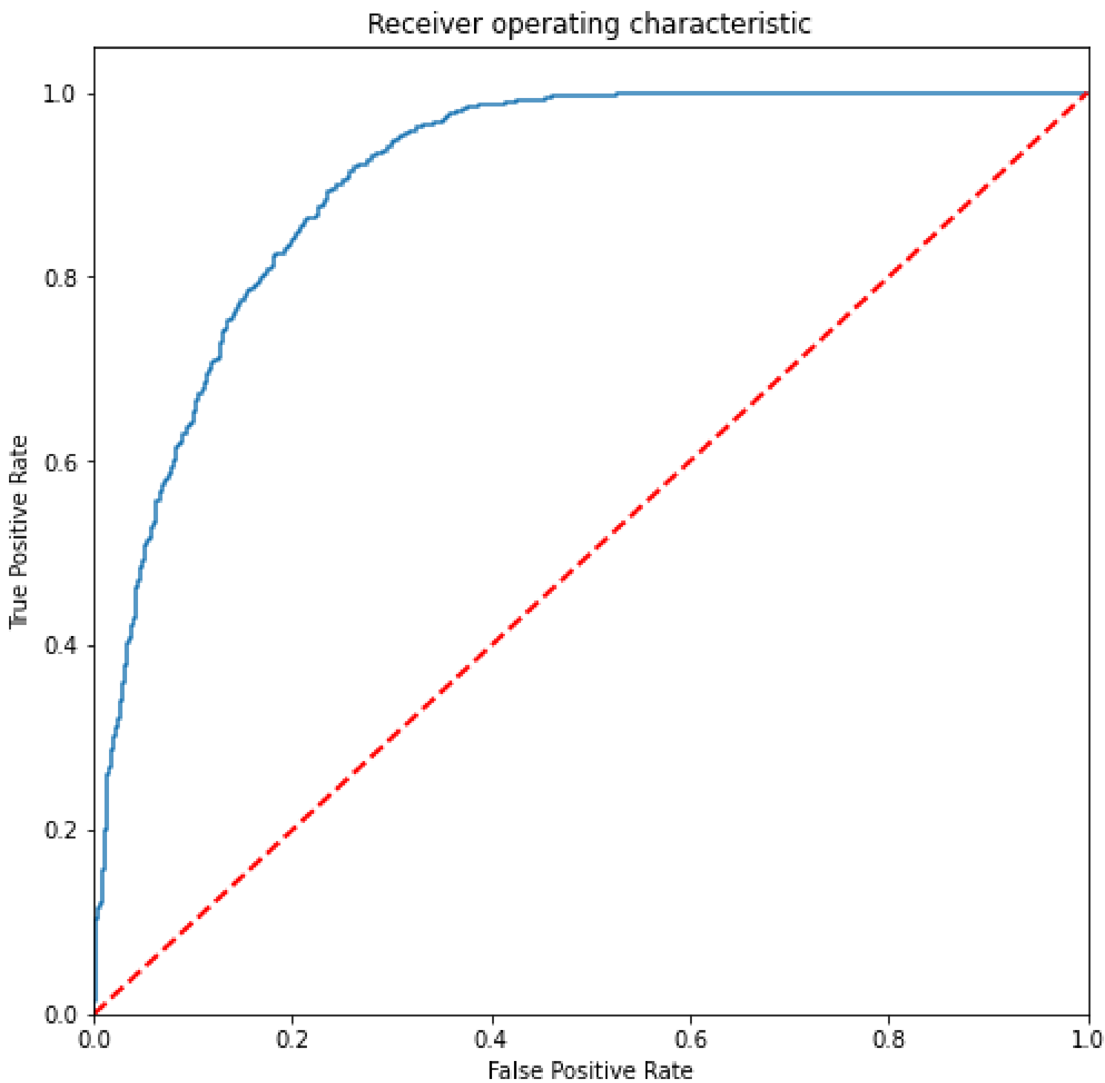
```
AUC: 0.8126392227972237
```



[89]:

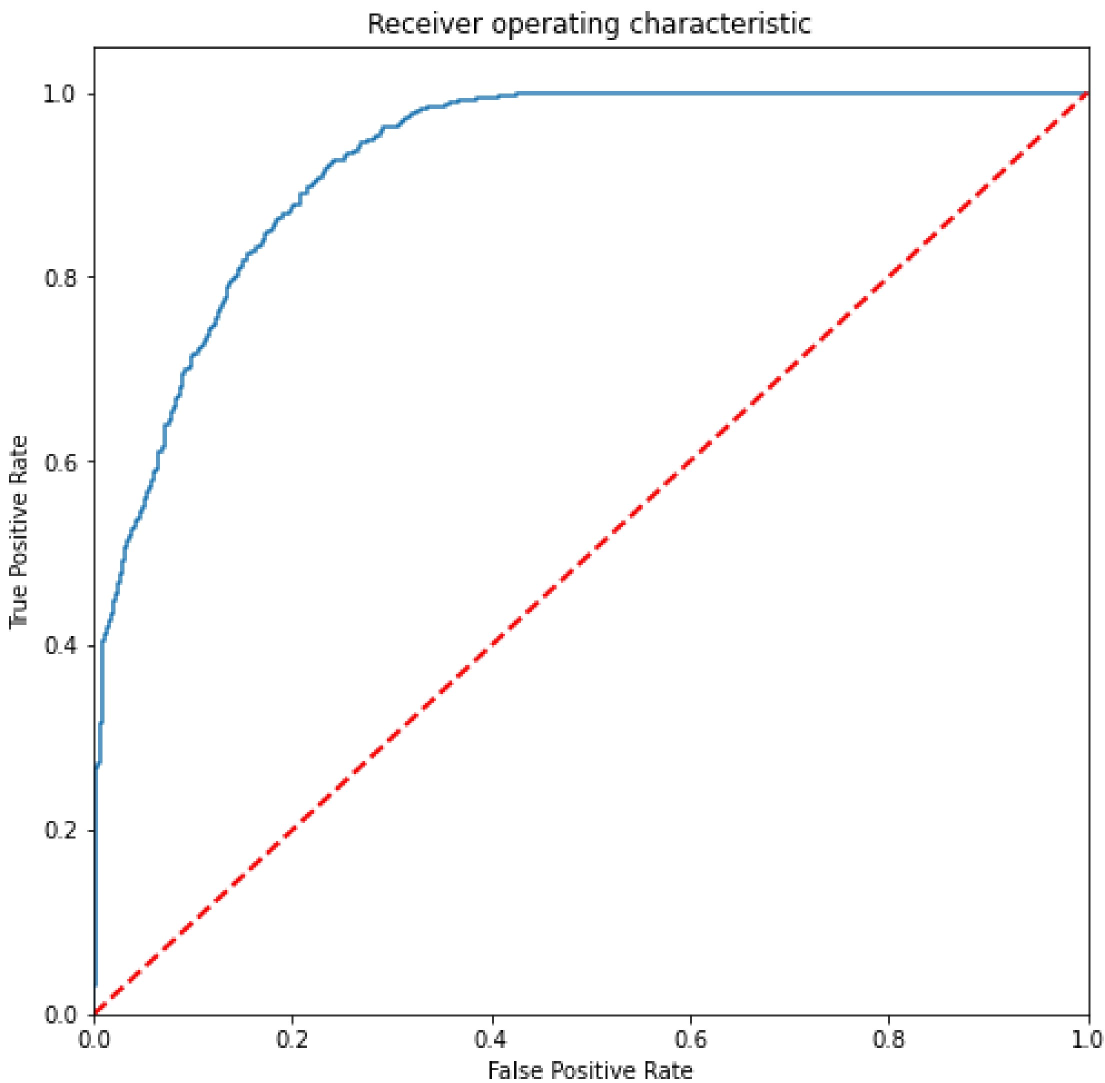
```
clsf = GaussianNB()
a=aucs2(clsf,X_train,X_test,y_train,y_test)
aucscore.iloc[3]=a
```

```
AUC: 0.9067767545468128
```



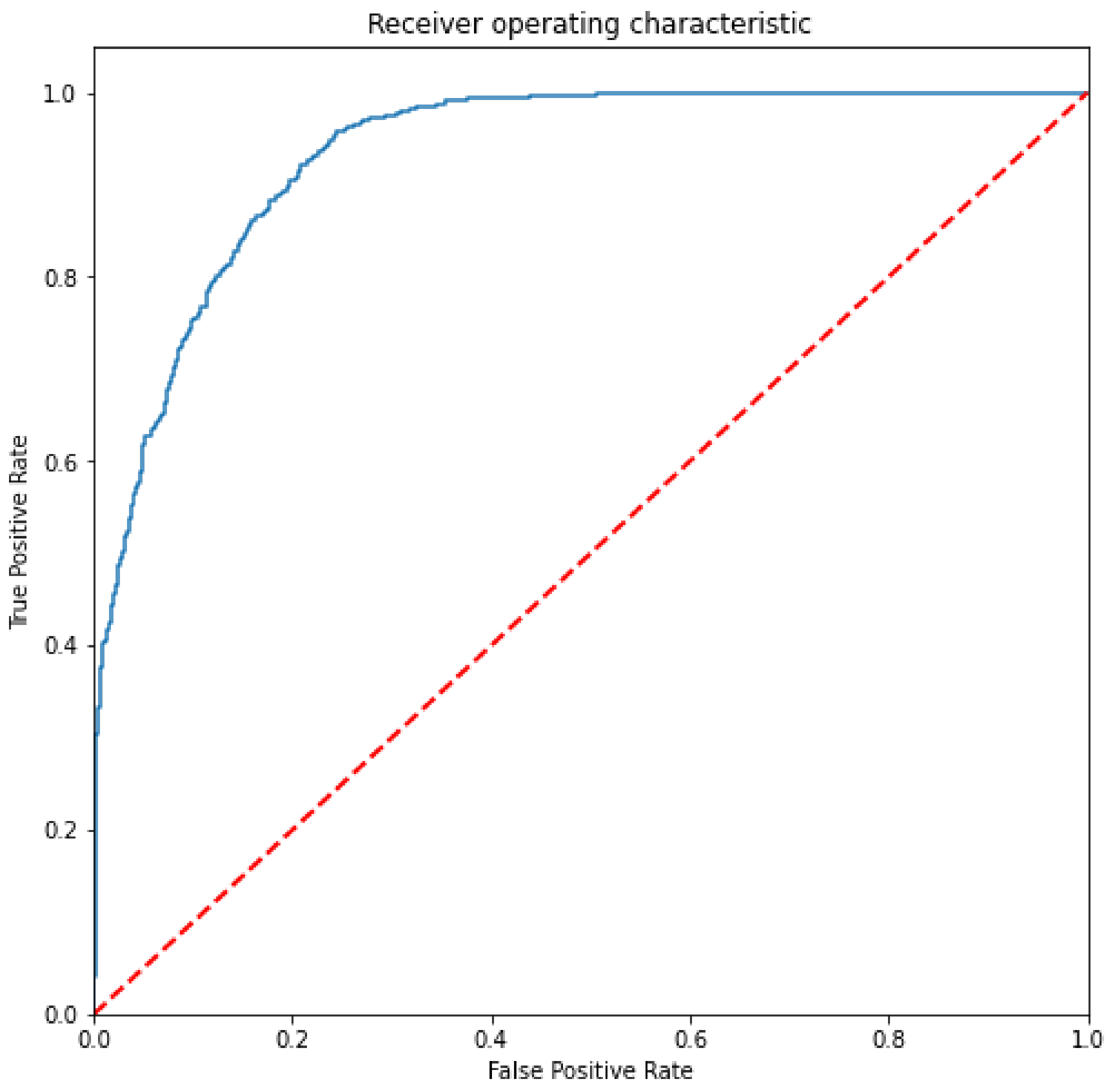
[90]:
```
clsf = MLPClassifier()
a=aucs2(clsf,X_train,X_test,y_train,y_test)
aucscore.iloc[4]=a
```

```
AUC: 0.9250862502207191
```



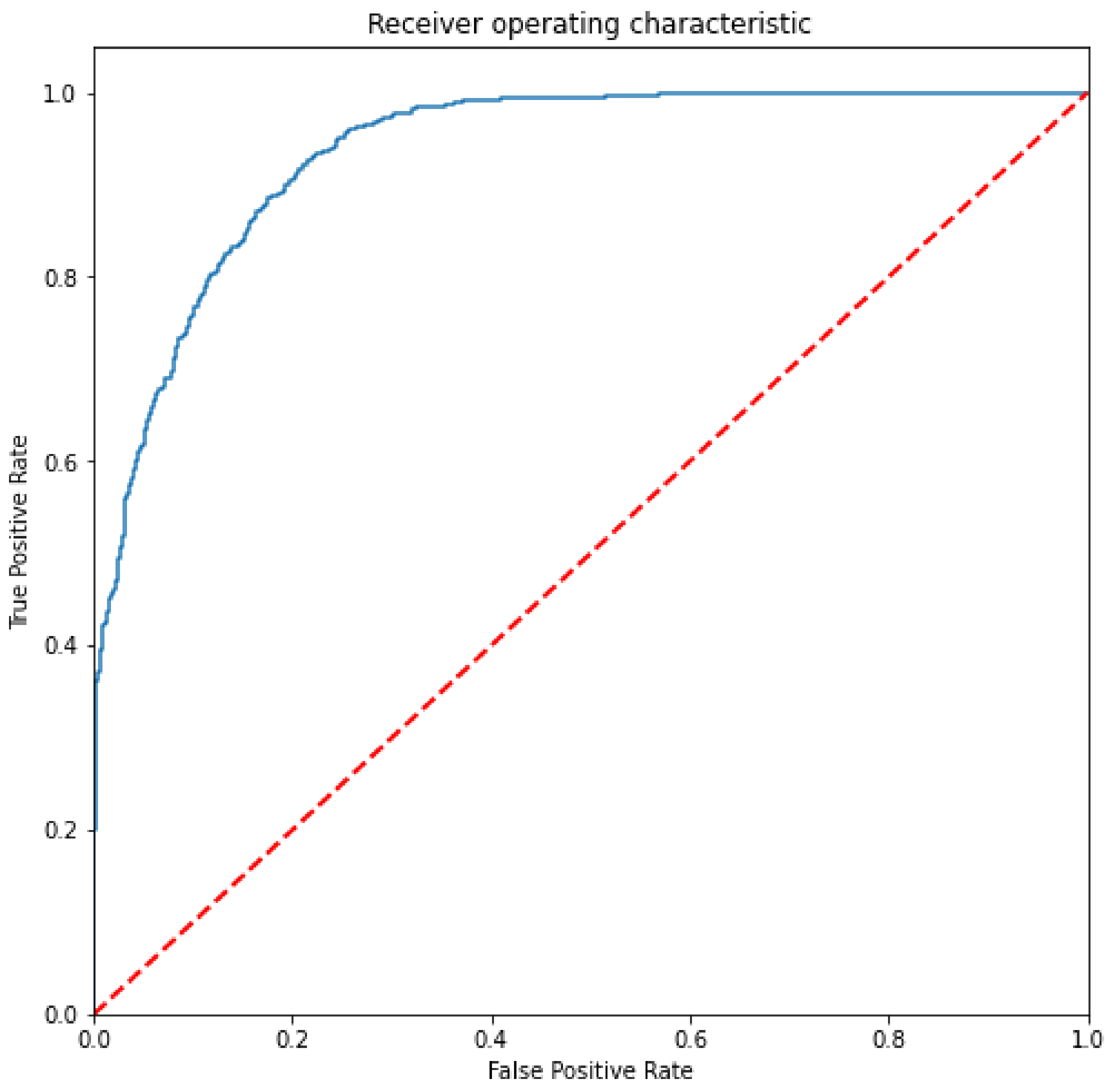
[91]:
```
clsf = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1), n_estimators =␣
↪200, algorithm = 'SAMME.R', learning_rate = 0.5)
a=aucs(clsf,X_train,X_test,y_train,y_test)
aucscore.iloc[5]=a
```

```
AUC: 0.9342435448161581
```



[92]:
```
clsf = RandomForestClassifier(n_estimators = 500, max_leaf_nodes = 64,␣
 ↪n_jobs=-1,random_state=8)
a=aucs2(clsf,X_train,X_test,y_train,y_test)
aucscore.iloc[6]=a
```

```
AUC: 0.9357266071743886
```



[93]:
```
aucscore.sort_values(by='AUC score')
```

[93]:

	AUC score	Accuracy
KNN	0.669851	0.636112
DT	0.812639	0.812529
LR	0.905607	0.811608
NB	0.906777	0.828190
NN	0.925086	0.834178
AB	0.934244	0.851681
RF	0.935727	0.852142

Given the result from the table above we could infere that the Ada Boosting Model based on a Decision Tree Classifier and a learning rate of 0.5 and the Random Forest Model are indeed the best performing methods among the ones proposed in this excersice because it yields the highest

AUC. We can also see that the accuracy score rank is consistent with the one rsulting from the AUC score. Let us verify this result by computing the K-Fold Cross Validation score.

K-folds cross validation

[101]:
```
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn import metrics
from numpy import mean
from numpy import std
```

[102]:
```
models = []
models.append(('LR', LogisticRegression()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('DT', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
# Neural Network
models.append(('NN', MLPClassifier()))
# Boosting methods
models.append(('AB', AdaBoostClassifier(DecisionTreeClassifier(max_depth=1),␣
↪n_estimators = 200, algorithm = 'SAMME.R', learning_rate = 0.5)))
# Bagging methods
models.append(('RF', RandomForestClassifier(n_estimators = 500, max_leaf_nodes␣
↪= 64, n_jobs=-1,random_state=8)))
```

[103]:
```
names = []
results = []
num_folds=10

for name, model in models:
    scores = cross_val_score(model, X, y, cv=KFold(n_splits=num_folds,␣
↪shuffle=True, random_state=7), n_jobs=-1)
    results.append(scores)
    names.append(name)

    Result = '%s: %f (%f)' % (name, scores.mean(), scores.std())
    print (Result)
```

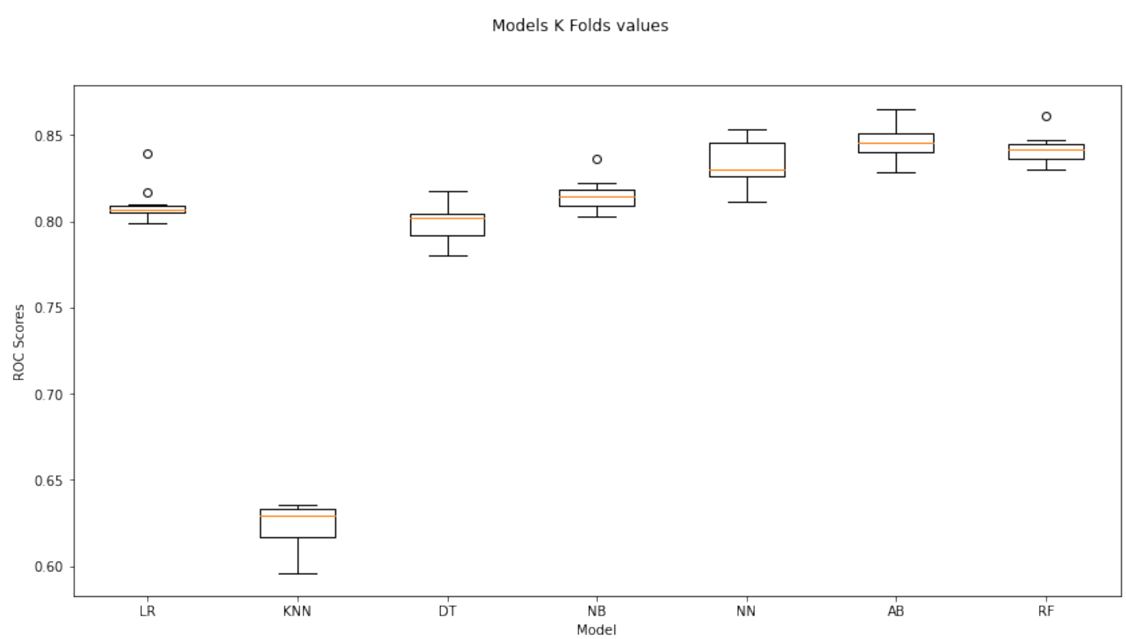
```
LR: 0.809972 (0.010771)
KNN: 0.623260 (0.012398)
DT: 0.799099 (0.010600)
NB: 0.814949 (0.009072)
NN: 0.832735 (0.013602)
AB: 0.845360 (0.010355)
RF: 0.841767 (0.008216)
```

[104]:
```
results
```

[104]: [array([0.80570902, 0.80645161, 0.81658986, 0.80737327, 0.80645161,
        0.81013825, 0.80368664, 0.79907834, 0.83963134, 0.80460829]),
 array([0.6335175 , 0.59631336, 0.6359447 , 0.63041475, 0.61658986,
        0.63317972, 0.62857143, 0.60921659, 0.61658986, 0.63225806]),
 array([0.77992634, 0.80460829, 0.80276498, 0.79078341, 0.80921659,
        0.79539171, 0.81751152, 0.78617512, 0.80184332, 0.80276498]),
 array([0.80847145, 0.80276498, 0.81198157, 0.8156682 , 0.80460829,
        0.81935484, 0.81290323, 0.8156682 , 0.8359447 , 0.82211982]),
 array([0.82596685, 0.8359447 , 0.83317972, 0.81105991, 0.82580645,
        0.84976959, 0.84884793, 0.82672811, 0.85345622, 0.81658986]),
 array([0.84162063, 0.84976959, 0.8516129 , 0.84700461, 0.85437788,
        0.83133641, 0.84423963, 0.82857143, 0.86543779, 0.83963134]),
 array([0.83241252, 0.840553 , 0.84700461, 0.84239631, 0.84239631,
        0.83502304, 0.84608295, 0.83041475, 0.86082949, 0.840553])]

Plot of model comparison using a BoxPlot to capture the range of values coming from the K Folds for each model

[105]:
```python
from matplotlib import pyplot
fig = pyplot.figure()
fig.suptitle('Models K Folds values')
plt.ylabel("ROC Scores")
plt.xlabel("Model")
ax = fig.add_subplot()
pyplot.boxplot(results)
ax.set_xticklabels(names)
fig.set_size_inches(14,7)
pyplot.show()
```



As we can see from the plot above, the previous results suggested by tue AUC and accuracy scores were correct. Both the Ada Boosting Model based on a Decision Tree Classifier and a learning rate of 0.5 and the Random Forest Model are the best models in fittimg the data.

6 Model Tuning and Grid Search

Based on the ROC comparison above, we choose the Random Forest Classifier as our main model given that it presented a slightly lower volatility in its performance compared to Ada Boosting and it got the better AUC score. For hyperparameter tuning, we will perform many iterations of the entire K-Fold CV process, each time using different model settings.

The hyperparameters to be randomized are: (i) number of trees in forest (ii) number of features at every split (iii) maximum number of levels in tree

[106]:
```
import pprint
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import GridSearchCV
```

[107]:
```
# Creating Random Grid

# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 300, stop = 700, num = 5)]
# Number of features to consider at every split
max_features = ['auto', 'sqrt']
# Maximum number of levels in tree
max_depth = [int(x) for x in np.linspace(50, 200, num = 4)]

random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth}

pp = pprint.PrettyPrinter(indent=4)
pp.pprint(random_grid)
```

```
{   'max_depth': [50, 100, 150, 200],
    'max_features': ['auto', 'sqrt'],
    'n_estimators': [300, 400, 500, 600, 700]}
```

[108]:
```
# Now we use RandomizedSearchCV to compute 30 iterations

RF=RandomForestClassifier()
rf_random = RandomizedSearchCV(estimator = RF, param_distributions =␣
↪random_grid, n_iter = 30, cv = 10, verbose = 2, random_state = 5, n_jobs=-1)
rf_random.fit(X_train, y_train)
```

```
Fitting 10 folds for each of 30 candidates, totalling 300 fits
```

[108]:
```
RandomizedSearchCV(cv=10, estimator=RandomForestClassifier(), n_iter=30,
                   n_jobs=-1,
                   param_distributions={'max_depth': [50, 100, 150, 200],
                                        'max_features': ['auto', 'sqrt'],
                                        'n_estimators': [300, 400, 500, 600,
                                                         700]},
                   random_state=5, verbose=2)
```

[109]:
```
# Best parameters
rf_random.best_params_
```

[109]:
```
{'n_estimators': 400, 'max_features': 'sqrt', 'max_depth': 200}
```

[110]:
```
grid = GridSearchCV(RF, random_grid, verbose=2)
grid.fit(X_train,y_train)
```

```
Fitting 5 folds for each of 40 candidates, totalling 200 fits
[CV] END ..max_depth=50, max_features=auto, n_estimators=300; total time=   5.0s
[CV] END ..max_depth=50, max_features=auto, n_estimators=300; total time=   5.2s
[CV] END ..max_depth=50, max_features=auto, n_estimators=300; total time=   4.7s
[CV] END ..max_depth=50, max_features=auto, n_estimators=300; total time=   4.8s
[CV] END ..max_depth=50, max_features=auto, n_estimators=300; total time=   4.8s
[CV] END ..max_depth=50, max_features=auto, n_estimators=400; total time=   6.1s
[CV] END ..max_depth=50, max_features=auto, n_estimators=400; total time=   6.1s
[CV] END ..max_depth=50, max_features=auto, n_estimators=400; total time=   6.1s
[CV] END ..max_depth=50, max_features=auto, n_estimators=400; total time=   6.4s
[CV] END ..max_depth=50, max_features=auto, n_estimators=400; total time=   5.9s
[CV] END ..max_depth=50, max_features=auto, n_estimators=500; total time=   7.3s
[CV] END ..max_depth=50, max_features=auto, n_estimators=500; total time=   6.5s
[CV] END ..max_depth=50, max_features=auto, n_estimators=500; total time=   6.2s
[CV] END ..max_depth=50, max_features=auto, n_estimators=500; total time=   6.3s
[CV] END ..max_depth=50, max_features=auto, n_estimators=500; total time=   6.3s
[CV] END ..max_depth=50, max_features=auto, n_estimators=600; total time=   7.4s
[CV] END ..max_depth=50, max_features=auto, n_estimators=600; total time=   7.6s
[CV] END ..max_depth=50, max_features=auto, n_estimators=600; total time=   7.5s
[CV] END ..max_depth=50, max_features=auto, n_estimators=600; total time=   7.6s
[CV] END ..max_depth=50, max_features=auto, n_estimators=600; total time=   7.5s
[CV] END ..max_depth=50, max_features=auto, n_estimators=700; total time=   8.8s
[CV] END ..max_depth=50, max_features=auto, n_estimators=700; total time=  10.1s
[CV] END ..max_depth=50, max_features=auto, n_estimators=700; total time=   9.1s
[CV] END ..max_depth=50, max_features=auto, n_estimators=700; total time=   8.9s
[CV] END ..max_depth=50, max_features=auto, n_estimators=700; total time=   8.8s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=300; total time=   3.7s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=300; total time=   3.7s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=300; total time=   3.7s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=300; total time=   3.7s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=300; total time=   3.7s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=400; total time=   5.0s
```

```
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=400; total time=   5.0s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=400; total time=   4.9s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=400; total time=   5.0s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=400; total time=   5.0s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=500; total time=   6.2s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=500; total time=   6.3s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=500; total time=   6.2s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=500; total time=   6.3s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=500; total time=   6.3s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=600; total time=   7.6s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=600; total time=   8.3s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=600; total time=   7.6s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=600; total time=   7.9s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=600; total time=   7.8s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=700; total time=   8.8s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=700; total time=   9.2s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=700; total time=   9.9s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=700; total time=  10.7s
[CV] END ..max_depth=50, max_features=sqrt, n_estimators=700; total time=   9.1s
[CV] END .max_depth=100, max_features=auto, n_estimators=300; total time=   3.8s
[CV] END .max_depth=100, max_features=auto, n_estimators=300; total time=   3.9s
[CV] END .max_depth=100, max_features=auto, n_estimators=300; total time=   3.8s
[CV] END .max_depth=100, max_features=auto, n_estimators=300; total time=   3.8s
[CV] END .max_depth=100, max_features=auto, n_estimators=300; total time=   3.8s
[CV] END .max_depth=100, max_features=auto, n_estimators=400; total time=   5.0s
[CV] END .max_depth=100, max_features=auto, n_estimators=400; total time=   5.0s
[CV] END .max_depth=100, max_features=auto, n_estimators=400; total time=   5.0s
[CV] END .max_depth=100, max_features=auto, n_estimators=400; total time=   5.2s
[CV] END .max_depth=100, max_features=auto, n_estimators=400; total time=   5.8s
[CV] END .max_depth=100, max_features=auto, n_estimators=500; total time=   6.9s
[CV] END .max_depth=100, max_features=auto, n_estimators=500; total time=   6.4s
[CV] END .max_depth=100, max_features=auto, n_estimators=500; total time=   6.2s
[CV] END .max_depth=100, max_features=auto, n_estimators=500; total time=   6.4s
[CV] END .max_depth=100, max_features=auto, n_estimators=500; total time=   6.4s
[CV] END .max_depth=100, max_features=auto, n_estimators=600; total time=   7.7s
[CV] END .max_depth=100, max_features=auto, n_estimators=600; total time=   7.7s
[CV] END .max_depth=100, max_features=auto, n_estimators=600; total time=   7.7s
[CV] END .max_depth=100, max_features=auto, n_estimators=600; total time=   7.8s
[CV] END .max_depth=100, max_features=auto, n_estimators=600; total time=   7.8s
[CV] END .max_depth=100, max_features=auto, n_estimators=700; total time=   8.9s
[CV] END .max_depth=100, max_features=auto, n_estimators=700; total time=   9.0s
[CV] END .max_depth=100, max_features=auto, n_estimators=700; total time=   8.9s
[CV] END .max_depth=100, max_features=auto, n_estimators=700; total time=   9.2s
[CV] END .max_depth=100, max_features=auto, n_estimators=700; total time=   9.4s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=300; total time=   4.5s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=300; total time=   3.8s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=300; total time=   3.7s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=300; total time=   3.8s
```

```
[CV] END .max_depth=100, max_features=sqrt, n_estimators=300; total time=   3.7s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=400; total time=   4.9s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=400; total time=   4.9s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=400; total time=   4.9s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=400; total time=   5.0s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=400; total time=   5.0s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=500; total time=   6.3s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=500; total time=   6.4s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=500; total time=   6.2s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=500; total time=   6.4s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=500; total time=   6.3s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=600; total time=   7.7s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=600; total time=   7.5s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=600; total time=   7.8s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=600; total time=   7.7s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=600; total time=   7.9s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=700; total time=  10.1s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=700; total time=   8.9s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=700; total time=   8.8s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=700; total time=   9.2s
[CV] END .max_depth=100, max_features=sqrt, n_estimators=700; total time=   9.0s
[CV] END .max_depth=150, max_features=auto, n_estimators=300; total time=   3.7s
[CV] END .max_depth=150, max_features=auto, n_estimators=300; total time=   3.9s
[CV] END .max_depth=150, max_features=auto, n_estimators=300; total time=   3.9s
[CV] END .max_depth=150, max_features=auto, n_estimators=300; total time=   3.9s
[CV] END .max_depth=150, max_features=auto, n_estimators=300; total time=   3.9s
[CV] END .max_depth=150, max_features=auto, n_estimators=400; total time=   5.1s
[CV] END .max_depth=150, max_features=auto, n_estimators=400; total time=   5.3s
[CV] END .max_depth=150, max_features=auto, n_estimators=400; total time=   5.0s
[CV] END .max_depth=150, max_features=auto, n_estimators=400; total time=   5.1s
[CV] END .max_depth=150, max_features=auto, n_estimators=400; total time=   5.1s
[CV] END .max_depth=150, max_features=auto, n_estimators=500; total time=   6.5s
[CV] END .max_depth=150, max_features=auto, n_estimators=500; total time=   6.4s
[CV] END .max_depth=150, max_features=auto, n_estimators=500; total time=   6.5s
[CV] END .max_depth=150, max_features=auto, n_estimators=500; total time=   6.4s
[CV] END .max_depth=150, max_features=auto, n_estimators=500; total time=   6.9s
[CV] END .max_depth=150, max_features=auto, n_estimators=600; total time=   8.4s
[CV] END .max_depth=150, max_features=auto, n_estimators=600; total time=   7.7s
[CV] END .max_depth=150, max_features=auto, n_estimators=600; total time=   7.7s
[CV] END .max_depth=150, max_features=auto, n_estimators=600; total time=   7.9s
[CV] END .max_depth=150, max_features=auto, n_estimators=600; total time=   7.6s
[CV] END .max_depth=150, max_features=auto, n_estimators=700; total time=   9.0s
[CV] END .max_depth=150, max_features=auto, n_estimators=700; total time=   9.1s
[CV] END .max_depth=150, max_features=auto, n_estimators=700; total time=   9.1s
[CV] END .max_depth=150, max_features=auto, n_estimators=700; total time=   9.0s
[CV] END .max_depth=150, max_features=auto, n_estimators=700; total time=   9.0s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=300; total time=   3.8s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=300; total time=   3.9s
```

```
[CV] END .max_depth=150, max_features=sqrt, n_estimators=300; total time=   3.8s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=300; total time=   3.8s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=300; total time=   3.8s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=400; total time=   5.1s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=400; total time=   5.0s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=400; total time=   5.8s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=400; total time=   6.1s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=400; total time=   5.3s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=500; total time=   6.3s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=500; total time=   6.3s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=500; total time=   6.3s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=500; total time=   7.6s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=500; total time=   7.4s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=600; total time=  11.9s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=600; total time=   9.5s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=600; total time=   7.7s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=600; total time=   9.7s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=600; total time=   8.2s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=700; total time=   9.2s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=700; total time=   9.2s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=700; total time=  11.0s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=700; total time=   9.9s
[CV] END .max_depth=150, max_features=sqrt, n_estimators=700; total time=   8.9s
[CV] END .max_depth=200, max_features=auto, n_estimators=300; total time=   3.9s
[CV] END .max_depth=200, max_features=auto, n_estimators=300; total time=   4.0s
[CV] END .max_depth=200, max_features=auto, n_estimators=300; total time=   3.9s
[CV] END .max_depth=200, max_features=auto, n_estimators=300; total time=   4.1s
[CV] END .max_depth=200, max_features=auto, n_estimators=300; total time=   3.9s
[CV] END .max_depth=200, max_features=auto, n_estimators=400; total time=   5.2s
[CV] END .max_depth=200, max_features=auto, n_estimators=400; total time=   5.0s
[CV] END .max_depth=200, max_features=auto, n_estimators=400; total time=   5.1s
[CV] END .max_depth=200, max_features=auto, n_estimators=400; total time=   5.3s
[CV] END .max_depth=200, max_features=auto, n_estimators=400; total time=   5.1s
[CV] END .max_depth=200, max_features=auto, n_estimators=500; total time=   6.3s
[CV] END .max_depth=200, max_features=auto, n_estimators=500; total time=   6.4s
[CV] END .max_depth=200, max_features=auto, n_estimators=500; total time=   6.4s
[CV] END .max_depth=200, max_features=auto, n_estimators=500; total time=   6.6s
[CV] END .max_depth=200, max_features=auto, n_estimators=500; total time=   6.4s
[CV] END .max_depth=200, max_features=auto, n_estimators=600; total time=   7.5s
[CV] END .max_depth=200, max_features=auto, n_estimators=600; total time=   7.5s
[CV] END .max_depth=200, max_features=auto, n_estimators=600; total time=   9.9s
[CV] END .max_depth=200, max_features=auto, n_estimators=600; total time=   8.4s
[CV] END .max_depth=200, max_features=auto, n_estimators=600; total time=   7.9s
[CV] END .max_depth=200, max_features=auto, n_estimators=700; total time=  10.0s
[CV] END .max_depth=200, max_features=auto, n_estimators=700; total time=   9.5s
[CV] END .max_depth=200, max_features=auto, n_estimators=700; total time=   9.1s
[CV] END .max_depth=200, max_features=auto, n_estimators=700; total time=   8.9s
[CV] END .max_depth=200, max_features=auto, n_estimators=700; total time=   8.8s
```

```
[CV] END .max_depth=200, max_features=sqrt, n_estimators=300; total time=   3.7s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=300; total time=   3.8s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=300; total time=   3.7s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=300; total time=   3.8s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=300; total time=   3.7s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=400; total time=   4.9s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=400; total time=   4.9s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=400; total time=   5.0s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=400; total time=   5.0s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=400; total time=   4.9s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=500; total time=   6.6s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=500; total time=   6.9s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=500; total time=   6.3s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=500; total time=   6.3s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=500; total time=   6.2s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=600; total time=   7.5s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=600; total time=   7.6s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=600; total time=   7.8s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=600; total time=   7.9s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=600; total time=   7.9s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=700; total time=   9.2s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=700; total time=   9.0s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=700; total time=   9.0s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=700; total time=   9.0s
[CV] END .max_depth=200, max_features=sqrt, n_estimators=700; total time=   9.1s
```

[110]:
```
GridSearchCV(estimator=RandomForestClassifier(),
             param_grid={'max_depth': [50, 100, 150, 200],
                         'max_features': ['auto', 'sqrt'],
                         'n_estimators': [300, 400, 500, 600, 700]},
             verbose=2)
```

[111]:
```
grid.best_params_
```

[111]:
```
{'max_depth': 100, 'max_features': 'sqrt', 'n_estimators': 500}
```

We used two different methods to tune the hyperparameters of the model. The first one is RandomizedSearchCV which in contrast to GridSearchCV, do not try out all parameter, but rather a fixed number of parameter settings is sampled from the specified distributions. The number of parameter settings that are tried is given by n_iter, which in our case is 30. The main consequence of this is that this method takes less time. The downside is that it might not get the optimal value for the parameters, although it will yield a good approximation. The second methods is the classical GridSearchCV. We will tryout the output from both methods next to choose the best parameter combination for our Random Forest model,

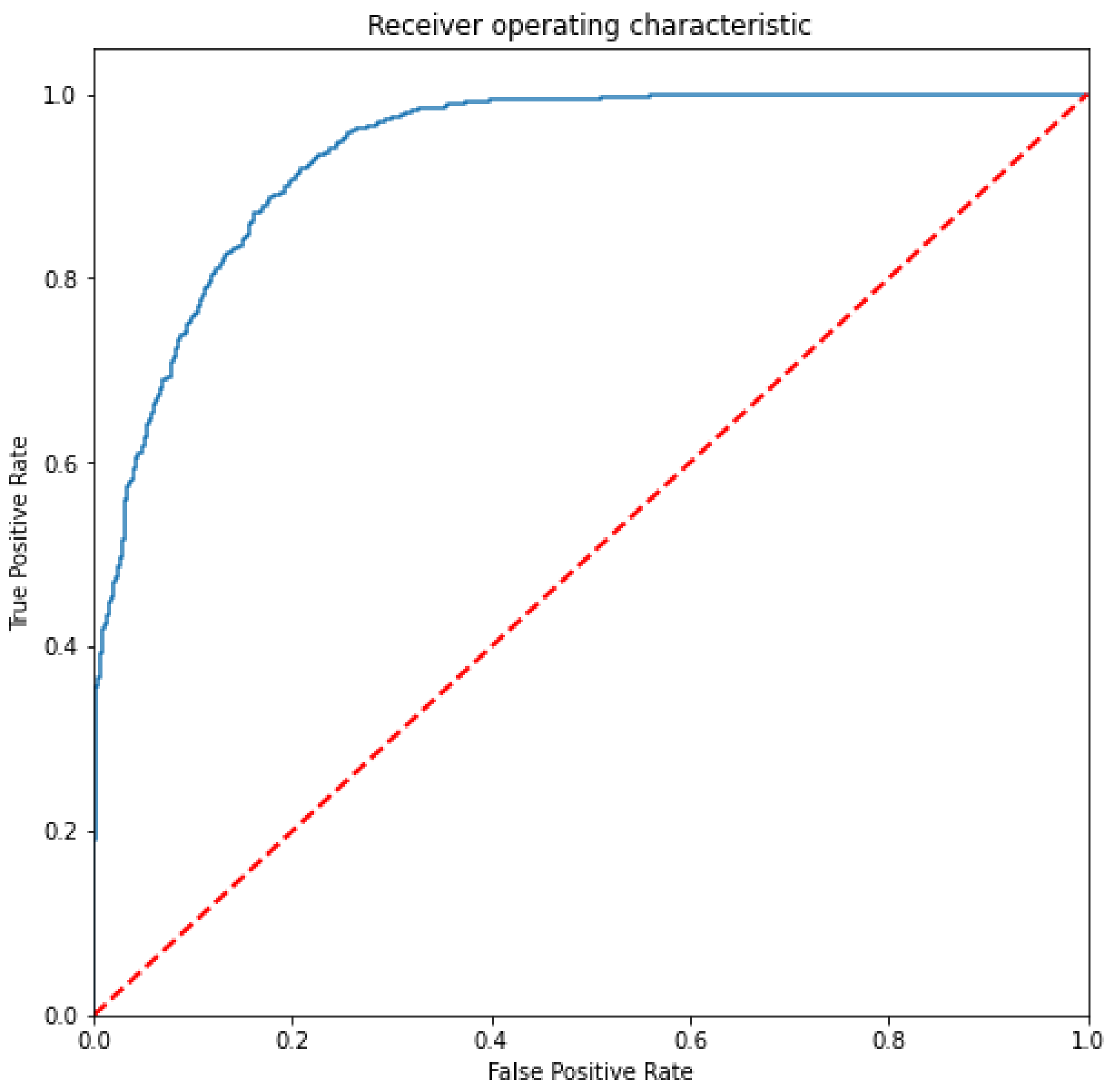
7 Finalize the Model

Finalized Model with best parameters found during tuning step

[112]:
```
clsf = RandomForestClassifier(n_estimators = 600, max_depth=50,␣
 ↪max_features='auto',max_leaf_nodes = 64, n_jobs=-1,random_state=8)
a=aucs2(clsf,X_train,X_test,y_train,y_test)
a
```

```
AUC: 0.9356765209241678
```

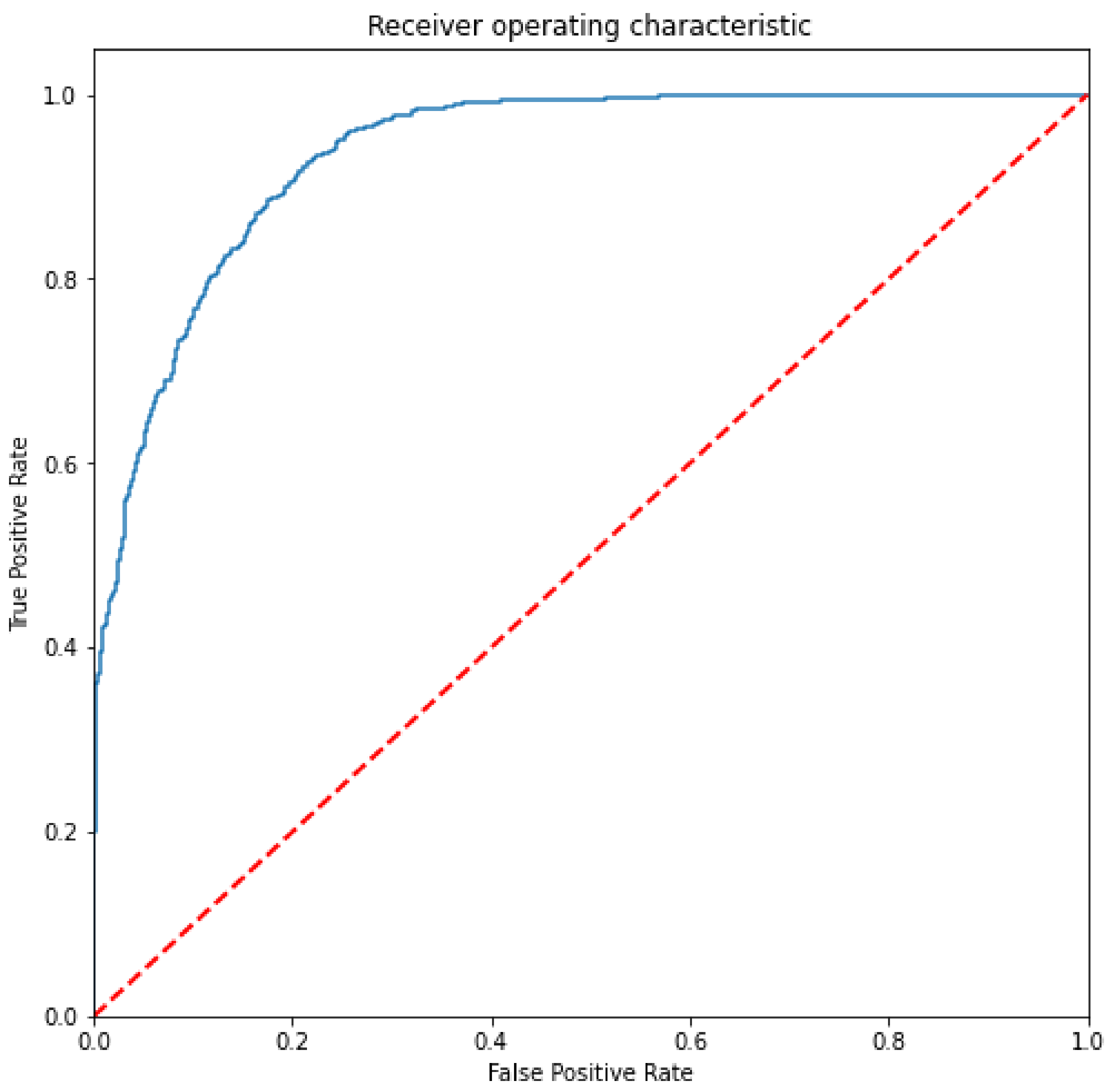
[112]:
```
(0.9356765209241678, 0.8530631045601106)
```



[113]:
```
clsf = RandomForestClassifier(n_estimators = 500, max_depth=100,␣
 ↪max_features='auto',max_leaf_nodes = 64, n_jobs=-1,random_state=8)
a=aucs2(clsf,X_train,X_test,y_train,y_test)
a
```

```
AUC: 0.9357266071743886
```

[113]:
```
(0.9357266071743886, 0.852141870105942)
```



As we can see from the graphs above. The parameters suggested by RandomizedSearchCV yields a slightly lower AUC score and a better Accuracy score. We will then keep the results suggested by GridSearchCV as the AUC is a more robust performance score.

7.1 Results on the Test Dataset

Calculation of the fine-tuned model results on the test dataset. Computation of accuracy, confusion matrix, the classification report

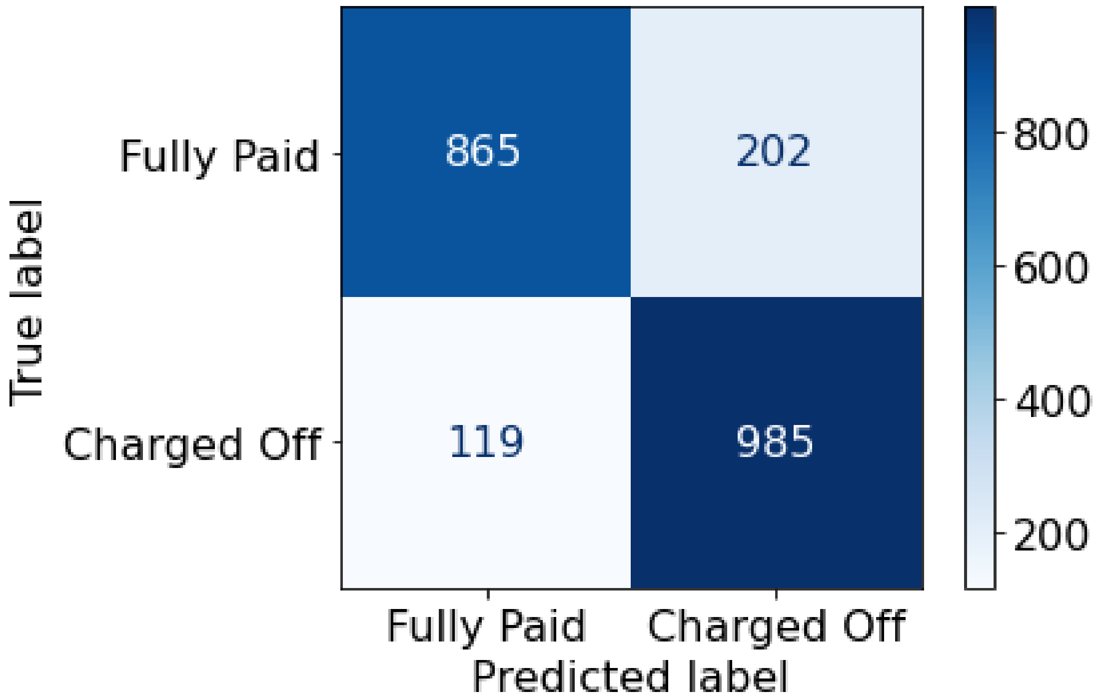
[114]:
```
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
from sklearn.metrics import plot_confusion_matrix
import matplotlib as mpl

def plot_cm(clf, X, y, labs):
    mpl.rcParams.update({'font.size': 16})
    cm = plot_confusion_matrix(clf, X_test, y_test,␣
 ↪display_labels=labs,cmap=mpl.cm.Blues);
```

[115]:
```
def evaluate(model, X_train, X_test, y_train, y_test):
    y_pred = model.predict(X_test)
    model.fit(X_train,y_train)
    y_score = model.predict_proba(X_test)
    fpr, tpr, thresholds = roc_curve(y_test, y_score[:,1])
    area = auc(fpr,tpr)
    acc_score = accuracy_score(y_test, y_pred)
    accuracy_list.append(acc_score)
    auc_list.append(area)
    report = metrics.classification_report(y_test, y_pred, output_dict=True)
    df_report = pd.DataFrame(report).transpose()
    df_report = df_report.sort_values(by=['f1-score'], ascending=False)

    print('Model Performance')
    print('AUC = {:0.2f}%.'.format(area*100))
    print('Accuracy = {:0.2f}%.'.format(acc_score*100))
    print('Classification Reoport:')
    print(df_report)
    plot_cm(model, X=X_test, y=y_test, labs=('Fully Paid', 'Charged Off'))

accuracy_list = []
auc_list = []
```

[116]:
```
# Computing evaluation with original parameters
base_model = RandomForestClassifier(n_estimators = 500, max_leaf_nodes = 64,␣
 ↪n_jobs=-1,random_state=8)
base_model.fit(X_train, y_train)
evaluate(base_model, X_train, X_test, y_train, y_test)
```

```
Model Performance
AUC = 93.57%.
Accuracy = 85.21%.
Classification Reoport:
                precision    recall  f1-score      support
```

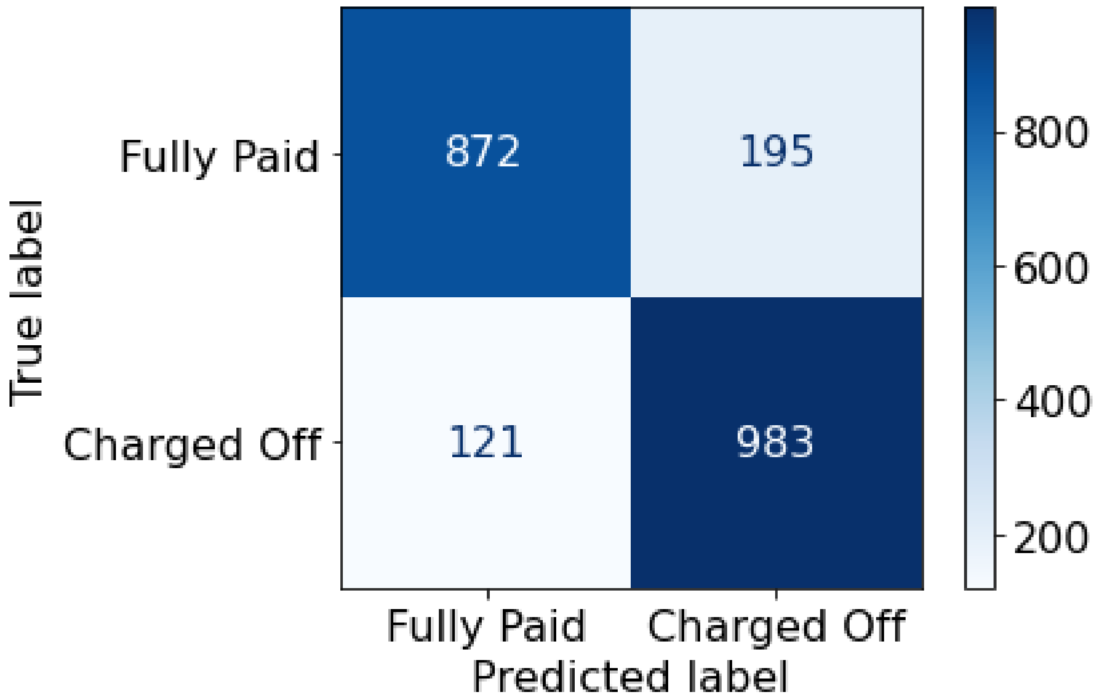
1.0	0.829823	0.892210	0.859887	1104.000000
accuracy	0.852142	0.852142	0.852142	0.852142
weighted avg	0.854024	0.852142	0.851828	2171.000000
macro avg	0.854444	0.851447	0.851689	2171.000000
0.0	0.879065	0.810684	0.843491	1067.000000



[118]:
```
# Computing evaluation with optimized parameters
best_model = grid.best_estimator_
evaluate(best_model, X_train, X_test, y_train, y_test)
```

Model Performance
AUC = 93.63%.
Accuracy = 85.54%.
Classification Reoport:

	precision	recall	f1-score	support
1.0	0.831376	0.897645	0.863240	1104.000000
accuracy	0.855366	0.855366	0.855366	0.855366
weighted avg	0.857523	0.855366	0.855028	2171.000000
macro avg	0.857976	0.854633	0.854885	2171.000000
0.0	0.884576	0.811621	0.846530	1067.000000



[119]:
```
# Computing Improvement using AUC
print('Improvement of AUC by {:0.2f}%.'.format( 100 * (auc_list[1] -␣
↪auc_list[0]) / auc_list[0]))
```

```
Improvement of AUC by 0.06%.
```

[120]:
```
# Computing Improvement using accuracy
print('Improvement of Accuracy by {:0.2f}%.'.format( 100 * (accuracy_list[1] -␣
↪accuracy_list[0]) / accuracy_list[0]))
```

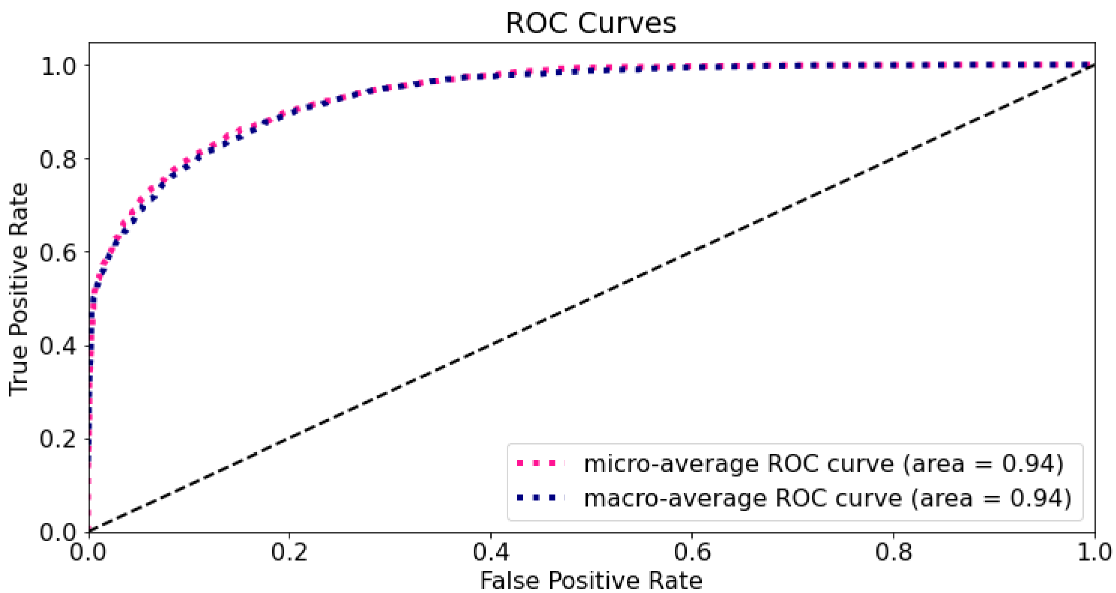
```
Improvement of Accuracy by 0.38%.
```

As we can see, both Accuracy and AUC improved slightly with the new parameters chosen, which suggest that our hyperparameter tuning was sucessful.

Computation of the ROC curve for the model

[121]:
```
import sklearn.metrics as metrics

probs = best_model.predict_proba(X_test)
y_pred = probs[:,1]
fpr, tpr, threshold = metrics.roc_curve(y_test, y_pred)
roc_auc = metrics.auc(fpr, tpr)
```

[122]:
```
import scikitplot as skplt

skplt.metrics.plot_roc_curve(y_test, probs, curves=('micro', 'macro'),␣
 ↪figsize=(12,6))
plt.show()
```



We finally computed the ROC curve for our tunned model. As we can see, the ROC curve reflects a pretty high skill in prediction. This can be seen also in the AUC which is around 94%. It is important to rember that a model with a ROC curve too close to the diagonal line or with an AUC of 50% will not have any predictive power. As the AUC increases, or the ROC curve steps further from the diagonla line, the model starts to become good at avoiding False Positives and then it becomes good at identifying True Positives. Given the results yielded by our model we can state that it is good at identifying True Positives.

7.2 Variable Intuition/Feature Importance

The last step in our analysis will be to look a the feature importance for our model. Before diving directly in our model we will fit an additional model which is the Gradient Boosting model. This is an ensamble method that is based on the same idea that Gradient Descent. This model is capable of capturing complex relationships and offers more flexibility than other ensemble methods such as the Random Forest model. After fitting the model we will examine the 'feature importance' property embedded in the model and, then, we will compare it to what we get with our chosen model.

[123]:
```
from sklearn.ensemble import GradientBoostingClassifier

# Training the GBM model
model_gbm = GradientBoostingClassifier(n_estimators=5000,
```

```
                                           learning_rate=0.05,
                                            max_depth=3,
                                            subsample=0.5,
                                            validation_fraction=0.1,
                                            n_iter_no_change=20,
                                            max_features='log2',
                                            verbose=1)

model_gbm.fit(X_train, y_train)
```

```
      Iter       Train Loss      OOB Improve   Remaining Time
         1           1.3748           0.0103           44.88s
         2           1.3632           0.0106           37.48s
         3           1.3194           0.0447           34.99s
         4           1.2904           0.0267           34.93s
         5           1.2806           0.0080           34.87s
         6           1.2529           0.0252           34.87s
         7           1.2513           0.0054           34.88s
         8           1.2426           0.0074           34.87s
         9           1.2326           0.0109           35.42s
        10           1.2086           0.0237           34.35s
        20           1.0729           0.0118           31.06s
        30           0.9636           0.0136           27.92s
        40           0.8842           0.0068           28.39s
        50           0.8059           0.0078           27.90s
        60           0.7792           0.0075           27.74s
        70           0.7490           0.0006           25.96s
        80           0.7163           0.0011           25.55s
        90           0.7035           0.0018           26.68s
       100           0.6975           0.0001           26.26s
       200           0.5959          -0.0001           24.31s
       300           0.5509          -0.0000           23.07s
```

[123]:
```
GradientBoostingClassifier(learning_rate=0.05, max_features='log2',
                           n_estimators=5000, n_iter_no_change=20,
                           subsample=0.5, verbose=1)
```

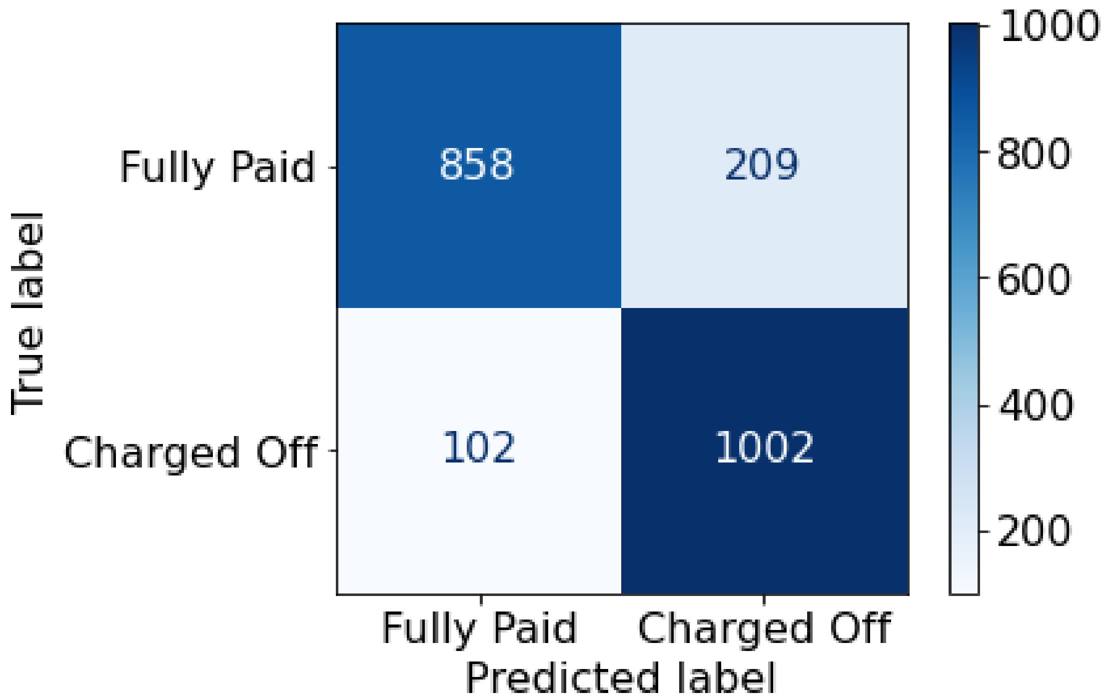
[124]:
```
len(model_gbm.estimators_) # Number of trees created by the GBM model
```

[124]: 391

[125]:
```
evaluate(model_gbm, X_train, X_test, y_train, y_test)
```

```
      Iter       Train Loss      OOB Improve   Remaining Time
         1           1.3772           0.0084           25.03s
         2           1.3575           0.0199           27.43s
         3           1.3349           0.0186           28.18s
         4           1.3279           0.0079           29.85s
```

```
         5           1.2822            0.0471            31.88s
         6           1.2742            0.0078            31.51s
         7           1.2401            0.0311            32.01s
         8           1.2134            0.0293            31.13s
         9           1.1920            0.0174            30.43s
        10           1.1649            0.0296            30.34s
        20           1.0359            0.0016            29.05s
        30           0.9230            0.0092            27.43s
        40           0.8393            0.0015            26.95s
        50           0.7909            0.0010            26.45s
        60           0.7590            0.0025            24.69s
        70           0.7269            0.0064            24.79s
        80           0.6883            0.0004            24.53s
        90           0.6810            0.0008            24.90s
       100           0.6625            0.0000            24.57s
       200           0.5897            0.0007            22.73s
       300           0.5359           -0.0002            22.30s
Model Performance
AUC = 93.69%.
Accuracy = 85.95%.
Classification Reoport:
              precision    recall  f1-score      support
1.0            0.833194  0.904891  0.867564  1104.000000
accuracy       0.859512  0.859512  0.859512     0.859512
weighted avg   0.862084  0.859512  0.859137  2171.000000
macro avg      0.862585  0.858725  0.858990  2171.000000
0.0            0.891975  0.812559  0.850417  1067.000000
```



[126]:

```
# Creating Feature Importance Grid

pd.DataFrame({"Feature_Name":X.columns,
             "Importance":model_gbm.feature_importances_}).
↪sort_values('Importance', ascending=False)
```

[126]:

	Feature_Name	Importance
14	last_pymnt_amnt	0.623287
1	term	0.067556
4	sub_grade	0.053748
2	int_rate	0.045130
0	loan_amnt	0.032501
3	grade	0.024589
10	dti	0.020007
15	acc_open_past_24mths	0.015416
16	avg_cur_bal	0.014430
6	verification_status	0.014093
23	annual_inc_log	0.013587
24	FICO_Score	0.011690
19	mo_sin_old_rev_tl_op	0.009913
17	bc_open_to_buy	0.009527
12	revol_util	0.008548
20	mo_sin_rcnt_rev_tl_op	0.006288
11	open_acc	0.006064

```
18                bc_util    0.005074
21               mort_acc    0.003696
9              addr_state    0.003580
22         num_actv_rev_tl    0.003322
8                   title    0.002701
7                 purpose    0.002109
13    initial_list_status    0.001917
5          home_ownership    0.001225
```

[127]:
```python
pd.DataFrame({"Feature_Name":X.columns,
             "Importance":best_model.feature_importances_}).
 ↪sort_values('Importance', ascending=False)
```

[127]:

	Feature_Name	Importance
14	last_pymnt_amnt	0.410929
2	int_rate	0.051098
4	sub_grade	0.041921
0	loan_amnt	0.038383
1	term	0.035566
10	dti	0.034663
19	mo_sin_old_rev_tl_op	0.031536
16	avg_cur_bal	0.031277
17	bc_open_to_buy	0.030022
23	annual_inc_log	0.029325
12	revol_util	0.028466
18	bc_util	0.028340
3	grade	0.028321
24	FICO_Score	0.023865
15	acc_open_past_24mths	0.023071
20	mo_sin_rcnt_rev_tl_op	0.022499
9	addr_state	0.022114
11	open_acc	0.020786
22	num_actv_rev_tl	0.017193
21	mort_acc	0.012229
6	verification_status	0.010293
8	title	0.008547
7	purpose	0.008285
5	home_ownership	0.006045
13	initial_list_status	0.005224

As we can see, the first top 4 features are the same in both cases, although in a slightly different order. From an intuitive perspective, the rank from the GB model is better as it ranks the feature 'grade' higher, which should be an important feature determining if a loan is charged-off or not. It is interesting to notice that our GB model fit actually yielded both higher AUC and Accuracy score than our Random Forest model with finetuned hyperparameters, although not by much.

We will now, run an additional analysis by fitting our model to only one of our fueatures at a time and then running a recursive feature elimination algorithim which will keep only the most important 5 features to compare our results.

Feature Importance using Individual Features

[128]:
```
model=RandomForestClassifier(n_estimators = 500, max_depth=100,␣
 ↪max_features='auto',max_leaf_nodes = 64, n_jobs=-1,random_state=8)
n_feats=len(X.columns)

print("----------------------------------------------------")
print(model.__class__)
scores_list = []

for i in range(n_feats):
    X_one_feature = X_train.iloc[:, i:i+1]
    scores = cross_val_score(model, X_one_feature, y_train, cv=5)
    scores_mean = scores.mean()
    scores_list.append(scores.mean())

sorted_indices = np.argsort(np.array(scores_list) * -1) # negate to have␣
 ↪descending

for i in range(0,5): # top 5 features
    index = sorted_indices[i]
    print(i, ":", X.columns[index], scores_list[index])

print("----------------------------------------------------")
```

```
----------------------------------------------------
<class 'sklearn.ensemble._forest.RandomForestClassifier'>
0 : last_pymnt_amnt 0.8103686635944701
1 : grade 0.6444700460829493
2 : int_rate 0.6426267281105991
3 : sub_grade 0.6425115207373272
4 : term 0.6237327188940093
----------------------------------------------------
```

Feature Importance using Recursive Feature Elimination

[129]:
```
from sklearn.feature_selection import RFE
```

[130]:
```
print("-------------------------------------------------")

rfe = RFE(estimator=model, n_features_to_select=5)

print(model.__class__)
rfe.fit(X_train, y_train)

for i in range(0,n_feats):
```

```
    if rfe.support_[i] == True:
        print(X.columns[i], end="\n")
```

```
--------------------------------------------------
<class 'sklearn.ensemble._forest.RandomForestClassifier'>
term
int_rate
grade
sub_grade
last_pymnt_amnt
```

As we suggested above, it seems that the feature importance we got from the GB model is more relevant, as we see that 'grade' is in the top 6 whilst the feature 'loan_amnt' is not. In terms of the ranking, the list of features from the recursive feature elimination algorithim is not actually ranked so the order between the last result and the previous one should not be compared.

8 Conclusion

We started with a 100,000 observations of a feature space containing 149 features and 1 label. By running several types of feature space reduction criteria we ended up with a 10,851 observations of a feature space containing 25 features and 1 label. By splitting the resulting data set into train and test sample we fitted different models and we found that the Random Forest model yielded the most robust perfomance scores. This was somehow expcted as the ensemble methods are designed to improve the predictibility power from regular models.

We then, fine tuned our parameters and managed to get an Area Under the Curve and Accuracy score of 93.63% and 85.54%, respectively. The last check we did on our model was to see if the features importance embedded in the model were logic. We implemented several methods and found that the loan term, the interest rate on the loan, the grade, the subgrade and the last payment amount to be the most relevant features. This result speaks very well of our model as those features are among the most relevant regarding the probability of a loan being charged off from a logical point of view. This suggest that our model is based on sound economic relationships between the features and the label.

Lastly, we also fit an additional model, the Gradient Boosting model, which is also and ensembled model and we got even better performance results. This suggest that the GB model is a model worth considering in any classification problem as it seems to have very high predictive power. Overall, we feel pretty comfrotable with the results obtained and with recommeding the use of both our Random Forest model with finetunned hyperparaments and the Gradient Boosting model to predict if a loan will end up being charged off or not, based on our data set.

[ ]:

```

```