In [2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

In [387]:
```
df = pd.read_csv("C:/Users/asus/Downloads/logistic_regression.csv")
df
```

Out[387]:

	loan_amnt	term	int_rate	installment	grade	sub_grade	emp_title	emp_length	home_ownership	annual_inc	...	ope
0	10000.0	36 months	11.44	329.48	B	B4	Marketing	10+ years	RENT	117000.0	...	
1	8000.0	36 months	11.99	265.68	B	B5	Credit analyst	4 years	MORTGAGE	65000.0	...	
2	15600.0	36 months	10.49	506.97	B	B3	Statistician	< 1 year	RENT	43057.0	...	
3	7200.0	36 months	6.49	220.65	A	A2	Client Advocate	6 years	RENT	54000.0	...	
4	24375.0	60 months	17.27	609.33	C	C5	Destiny Management Inc.	9 years	MORTGAGE	55000.0	...	
...	...	...	...	...	...	...	...	...	...	...	...	
396025	10000.0	60 months	10.99	217.38	B	B4	licensed bankere	2 years	RENT	40000.0	...	
396026	21000.0	36 months	12.29	700.42	C	C1	Agent	5 years	MORTGAGE	110000.0	...	
396027	5000.0	36 months	9.99	161.32	B	B1	City Carrier	10+ years	RENT	56500.0	...	
396028	21000.0	60 months	15.31	503.02	C	C2	Gracon Services, Inc	10+ years	MORTGAGE	64000.0	...	
396029	2000.0	36 months	13.61	67.98	C	C2	Internal Revenue Service	10+ years	RENT	42996.0	...	

396030 rows × 27 columns

EDA

In [5]:
```
df.columns
```

Out[5]:
```
Index(['loan_amnt', 'term', 'int_rate', 'installment', 'grade', 'sub_grade',
       'emp_title', 'emp_length', 'home_ownership', 'annual_inc',
       'verification_status', 'issue_d', 'loan_status', 'purpose', 'title',
       'dti', 'earliest_cr_line', 'open_acc', 'pub_rec', 'revol_bal',
       'revol_util', 'total_acc', 'initial_list_status', 'application_type',
       'mort_acc', 'pub_rec_bankruptcies', 'address'],
      dtype='object')
```

In [154]:
```
df.shape
```

Out[154]:
```
(396030, 27)
```

In [13]:
```
loan_status = df['loan_status'].value_counts(normalize=True)
loan_status
```

Out[13]:
```
Fully Paid     0.803871
Charged Off    0.196129
Name: loan_status, dtype: float64
```

- **80% of the customers have fully paid their loan amount**

In [20]:
```
df.describe()
```

Out[20]:

	loan_amnt	int_rate	installment	annual_inc	dti	open_acc	pub_rec	revol_bal
count	396030.000000	396030.000000	396030.000000	3.960300e+05	396030.000000	396030.000000	396030.000000	3.960300e+05
mean	14113.888089	13.639400	431.849698	7.420318e+04	17.379514	11.311153	0.178191	1.584454e+04
std	8357.441341	4.472157	250.727790	6.163762e+04	18.019092	5.137649	0.530671	2.059184e+04
min	500.000000	5.320000	16.080000	0.000000e+00	0.000000	0.000000	0.000000	0.000000e+00
25%	8000.000000	10.490000	250.330000	4.500000e+04	11.280000	8.000000	0.000000	6.025000e+03
50%	12000.000000	13.330000	375.430000	6.400000e+04	16.910000	10.000000	0.000000	1.118100e+04
75%	20000.000000	16.490000	567.300000	9.000000e+04	22.980000	14.000000	0.000000	1.962000e+04
max	40000.000000	30.990000	1533.810000	8.706582e+06	9999.000000	90.000000	86.000000	1.743266e+06

In [22]:
```
df.info()
```

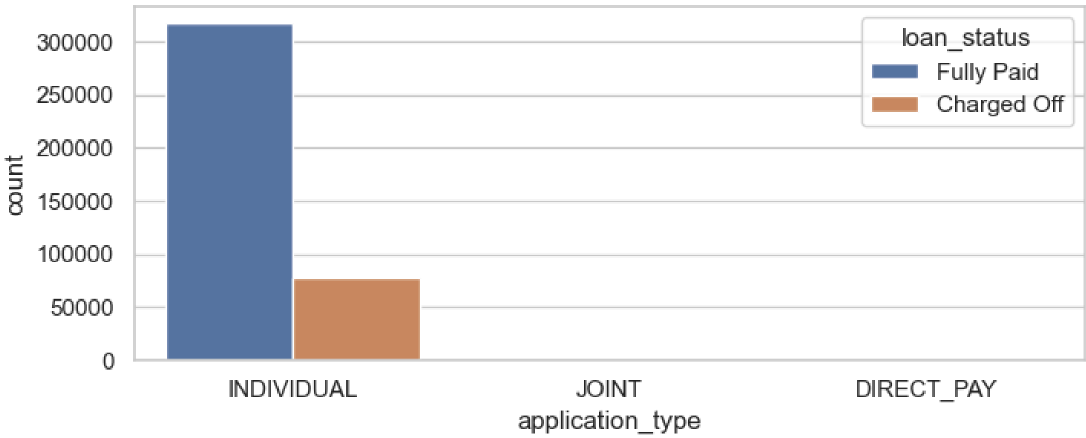
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 396030 entries, 0 to 396029
Data columns (total 27 columns):
 #   Column                Non-Null Count   Dtype  
---  ------                --------------   -----  
 0   loan_amnt             396030 non-null  float64
 1   term                  396030 non-null  object 
 2   int_rate              396030 non-null  float64
 3   installment           396030 non-null  float64
 4   grade                 396030 non-null  object 
 5   sub_grade             396030 non-null  object 
 6   emp_title             373103 non-null  object 
 7   emp_length            377729 non-null  object 
 8   home_ownership        396030 non-null  object 
 9   annual_inc            396030 non-null  float64
 10  verification_status   396030 non-null  object 
 11  issue_d               396030 non-null  object 
 12  loan_status           396030 non-null  object 
 13  purpose               396030 non-null  object 
 14  title                 394275 non-null  object 
 15  dti                   396030 non-null  float64
 16  earliest_cr_line      396030 non-null  object 
 17  open_acc              396030 non-null  float64
 18  pub_rec               396030 non-null  float64
 19  revol_bal             396030 non-null  float64
 20  revol_util            395754 non-null  float64
 21  total_acc             396030 non-null  float64
 22  initial_list_status   396030 non-null  object 
 23  application_type      396030 non-null  object 
 24  mort_acc              358235 non-null  float64
 25  pub_rec_bankruptcies  395495 non-null  float64
 26  address               396030 non-null  object 
dtypes: float64(12), object(15)
memory usage: 81.6+ MB
```

In [24]:
```
df.isnull().sum()
```

Out[24]:
```
loan_amnt                   0
term                        0
int_rate                    0
installment                 0
grade                       0
sub_grade                   0
emp_title               22927
emp_length              18301
home_ownership              0
annual_inc                  0
verification_status         0
issue_d                     0
loan_status                 0
purpose                     0
title                    1755
dti                         0
earliest_cr_line            0
open_acc                    0
pub_rec                     0
revol_bal                   0
revol_util                276
total_acc                   0
initial_list_status         0
application_type            0
mort_acc                37795
pub_rec_bankruptcies      535
address                     0
dtype: int64
```

In [155]:
```
plt.figure(figsize=(8,3))
sns.countplot(data=df, x='application_type', hue='loan_status')
```

Out[155]:
```
<AxesSubplot:xlabel='application_type', ylabel='count'>
```



In [156]:
```
df['application_type'].value_counts()
```

Out[156]:
```
INDIVIDUAL    395319
JOINT            425
DIRECT_PAY       286
Name: application_type, dtype: int64
```

The feature "application_type" has no significance to the target variable "loan_status", only one category dominates almost 100%. hence it can be dropped in model training

In [29]:
```
# Checking the categories in features having <= 20 categories
for i in df.columns:
    if df[i].nunique() <=20:
        print(i, '--->', df[i].nunique())
```

```
term ---> 2
grade ---> 7
emp_length ---> 11
home_ownership ---> 6
verification_status ---> 3
loan_status ---> 2
purpose ---> 14
pub_rec ---> 20
initial_list_status ---> 2
application_type ---> 3
pub_rec_bankruptcies ---> 9
```

In [41]:
```
# Unique values of categorical variables
print('Unique Values for Categorical Variables:')
for col in df.select_dtypes(include=['object']).columns:
    print(f'{col} :')
    print(df[col].value_counts())
    print()
```

```
Unique Values for Categorical Variables:
term :
 36 months    302005
 60 months     94025
Name: term, dtype: int64

grade :
B    116018
C    105987
A     64187
D     63524
E     31488
F     11772
G      3054
Name: grade, dtype: int64

sub_grade :
B3    26655
B4    25601
```

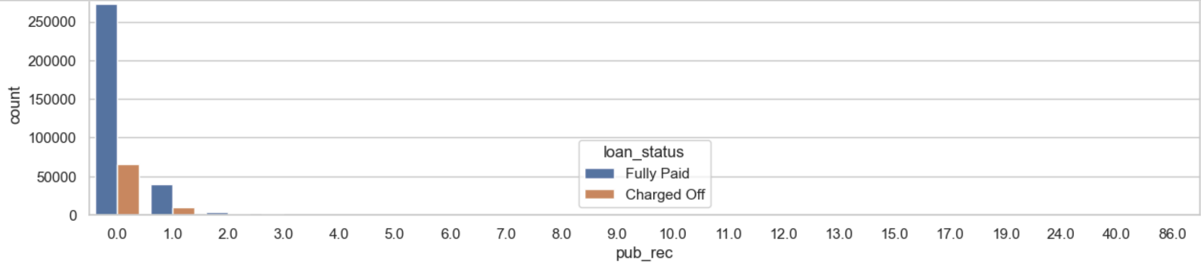
In [115]:
```
df.select_dtypes(include='number')
```

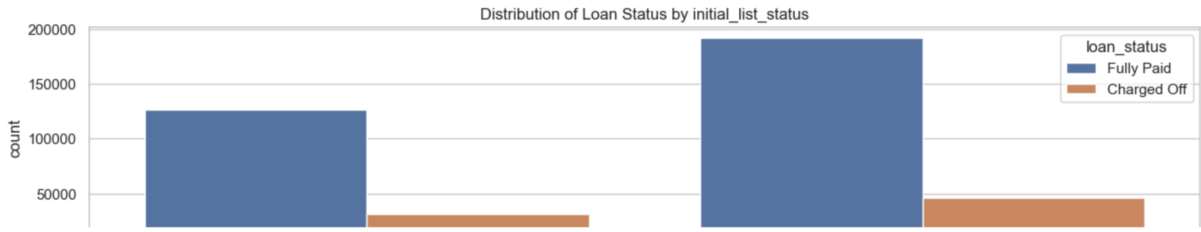
Out[115]:

	loan_amnt	int_rate	installment	annual_inc	dti	open_acc	pub_rec	revol_bal	revol_util	total_acc	mort_acc	pub_rec_
0	10000.0	11.44	329.48	117000.0	26.24	16.0	0.0	36369.0	41.8	25.0	0.0	
1	8000.0	11.99	265.68	65000.0	22.05	17.0	0.0	20131.0	53.3	27.0	3.0	
2	15600.0	10.49	506.97	43057.0	12.79	13.0	0.0	11987.0	92.2	26.0	0.0	
3	7200.0	6.49	220.65	54000.0	2.60	6.0	0.0	5472.0	21.5	13.0	0.0	
4	24375.0	17.27	609.33	55000.0	33.95	13.0	0.0	24584.0	69.8	43.0	1.0	
...	...	...	...	...	...	...	...	...	...	...	...	
396025	10000.0	10.99	217.38	40000.0	15.63	6.0	0.0	1990.0	34.3	23.0	0.0	
396026	21000.0	12.29	700.42	110000.0	21.45	6.0	0.0	43263.0	95.7	8.0	1.0	
396027	5000.0	9.99	161.32	56500.0	17.56	15.0	0.0	32704.0	66.9	23.0	0.0	
396028	21000.0	15.31	503.02	64000.0	15.88	9.0	0.0	15704.0	53.8	20.0	5.0	
396029	2000.0	13.61	67.98	42996.0	8.32	3.0	0.0	4292.0	91.3	19.0	NaN	

396030 rows × 12 columns

In [114]:
```
# Count plot for all the categorical variables
categorical_vars = ['term', 'grade', 'emp_length', 'home_ownership', 'verification_status', 'purpose',
                    'pub_rec', 'initial_list_status', 'application_type', 'pub_rec_bankruptcies', 'mort_ac

for col in categorical_vars:
    plt.figure(figsize=(15, 3))
    sns.countplot(x=col, hue='loan_status', data=df)
    if col == 'purpose':
        plt.xticks(rotation=60)
    plt.title(f'Distribution of Loan Status by {col}')
    plt.show()
```





36 months of loan term has higher percentage of 'Fully Paid' Loan_status.

When features 'pub_rec' and 'pub_rec_bankruptcies' are >1, there is no loan, this can be ustilized for feature engineering, may be to create new features

In [98]:
```
df['home_ownership'].value_counts(normalize=True)*100
```

Out[98]:
```
MORTGAGE    50.084085
RENT        40.347953
OWN          9.531096
OTHER        0.028281
NONE         0.007828
ANY          0.000758
Name: home_ownership, dtype: float64
```

The majority of people have home ownership as :

- **MORTAGE : This is about 50%.**
- Next highest is: **RENT : 40%**

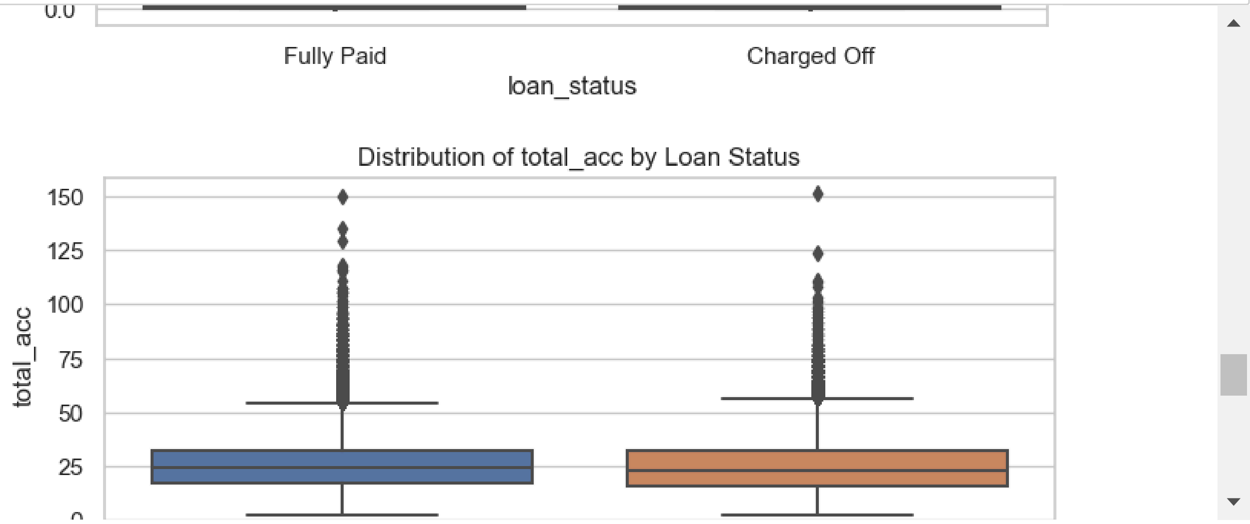
In [108]:
```
# total number of loans for each grade
total_loans_per_grade = df['grade'].value_counts()

# number of 'Fully Paid' loans for each grade
fully_paid_loans_per_grade = df[df['loan_status'] == 'Fully Paid']['grade'].value_counts()

# percentage of 'Fully Paid' loans for each grade
fully_paid_percentage_per_grade = (fully_paid_loans_per_grade / total_loans_per_grade) * 100

print("Percentage of 'Fully Paid' loans for each grade:")
print(fully_paid_percentage_per_grade)
```

```
Percentage of 'Fully Paid' loans for each grade:
B    87.426951
C    78.819100
A    93.712122
D    71.132171
E    62.636560
F    57.212029
G    52.161100
Name: grade, dtype: float64
```

Calculating on each grade, the highest percentage of loans are paid off by grade A, compared to other grades

Outliers detection adn treatment

In [84]:
```
# Box plot for numeric variables by loan_status
numeric_vars = ['loan_amnt', 'int_rate', 'installment', 'annual_inc', 'dti', 'pub_rec', 'open_acc', 'revol_
                'total_acc', 'mort_acc', 'pub_rec_bankruptcies']

for var in numeric_vars:
    plt.figure(figsize=(8, 3))
    sns.boxplot(x='loan_status', y=var, data=df)
    plt.title(f'Distribution of {var} by Loan Status')
    plt.show()
```



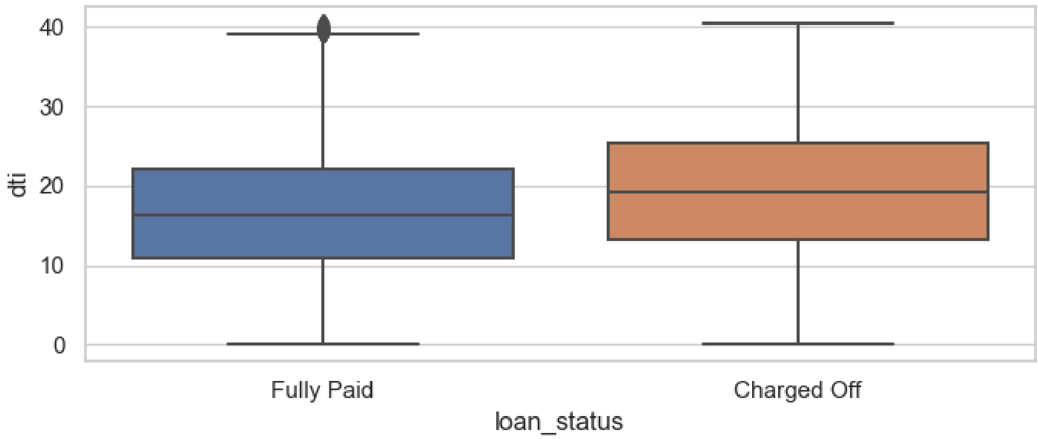
In [179]:
```
# Defing a function to remove outliers
def remove_outliers(df, columns):
    for col in columns:
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1

        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR

        df = df[(df[col] >= lower_bound) & (df[col] <= upper_bound)]

    return df
```

Removing outliers from the columns "dti", "annual_inc", "open_acc" and "rev_balance"

In [388]:
```
outlier_columns = ["dti", "annual_inc", "open_acc", "revol_bal"]
df = remove_outliers(df, outlier_columns)
```

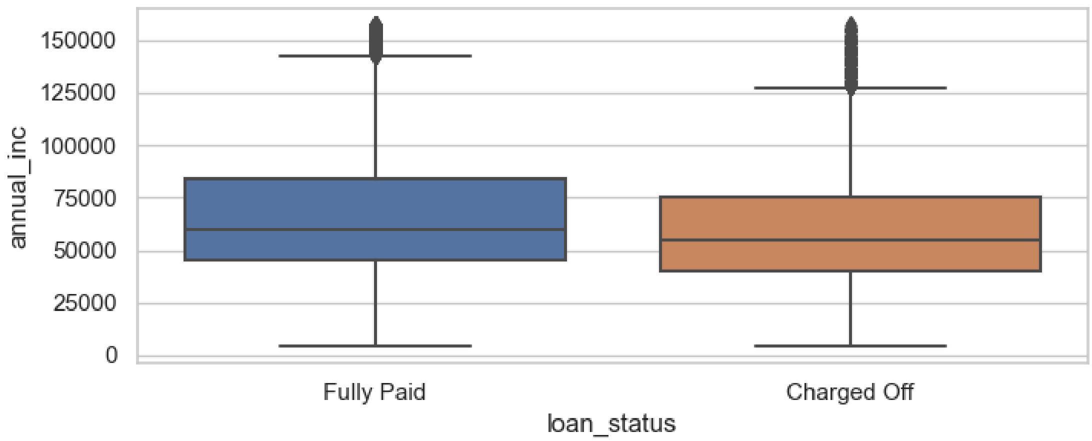
In [350]:
```
plt.figure(figsize=(8, 3))
sns.boxplot(x='loan_status', y='dti', data=df)
```

Out[350]: <AxesSubplot:xlabel='loan_status', ylabel='dti'>



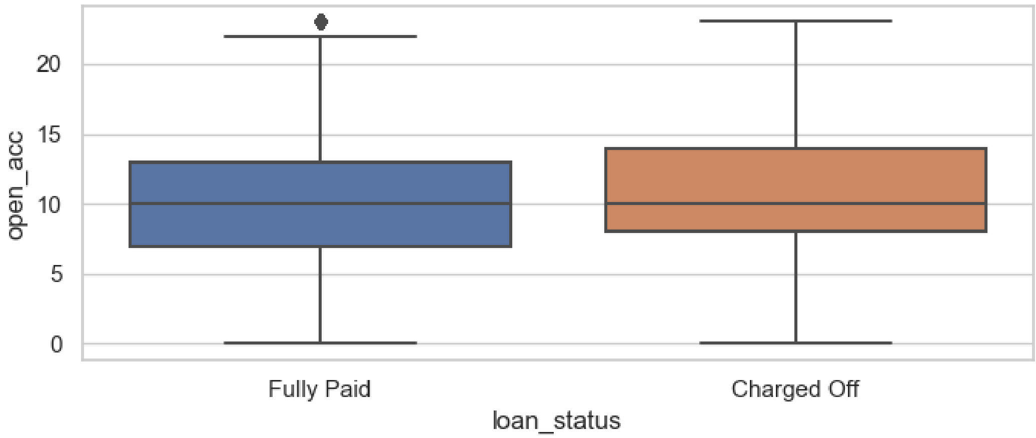
In [160]:
```
plt.figure(figsize=(8, 3))
sns.boxplot(x='loan_status', y='annual_inc', data=df)
```

Out[160]: <AxesSubplot:xlabel='loan_status', ylabel='annual_inc'>



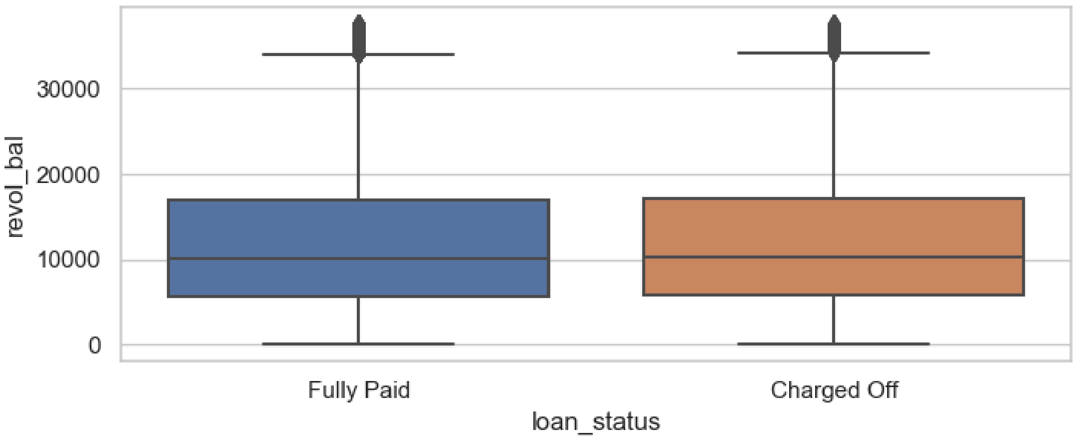
In [145]:
```
plt.figure(figsize=(8, 3))
sns.boxplot(x='loan_status', y='open_acc', data=df)
```

Out[145]: <AxesSubplot:xlabel='loan_status', ylabel='open_acc'>



In [189]:
```
plt.figure(figsize=(8, 3))
sns.boxplot(x='loan_status', y='revol_bal', data=df)
```

Out[189]: <AxesSubplot:xlabel='loan_status', ylabel='revol_bal'>



The feature 'revol_bal' has same median for both the loan_status. This feature can be dropped in model training

Correlation and Treatment

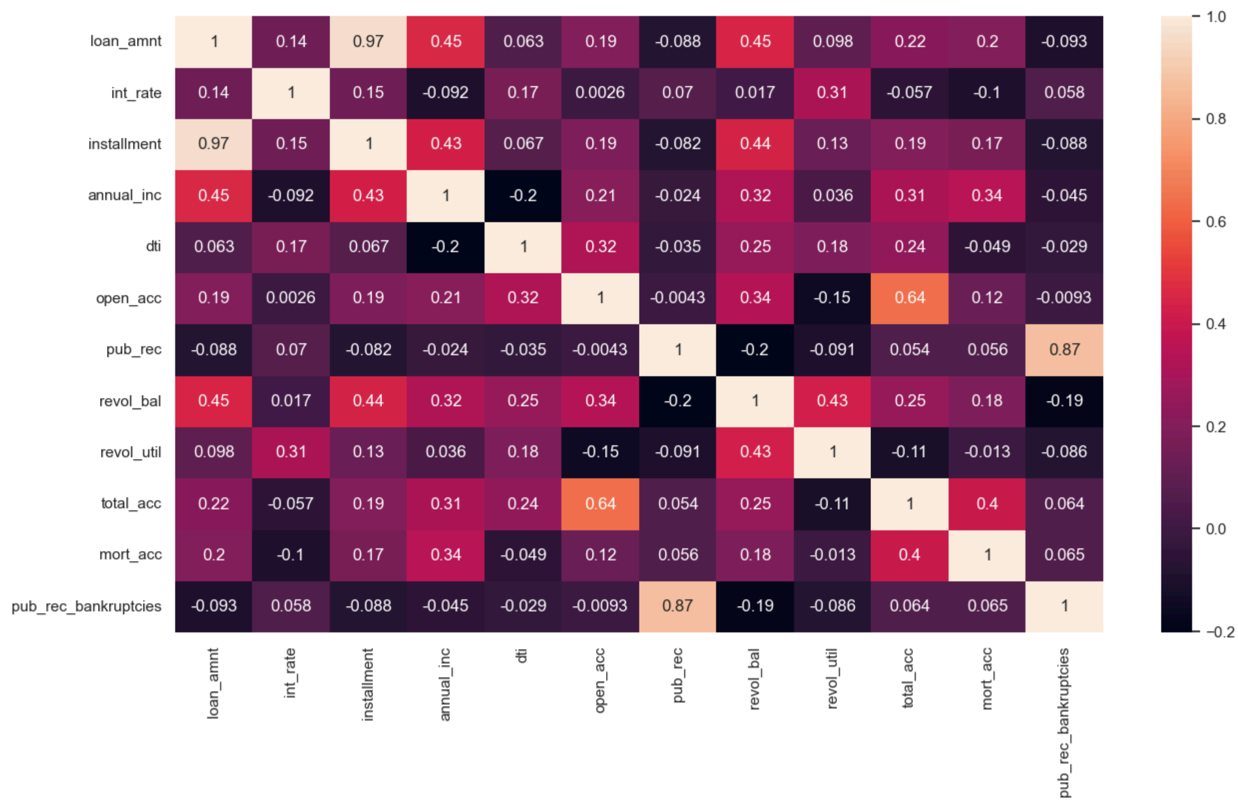
In [162]:
```
df.corr(method='spearman')
```

Out[162]:

	loan_amnt	int_rate	installment	annual_inc	dti	open_acc	pub_rec	revol_bal	revol_util	total_acc
loan_amnt	1.000000	0.136591	0.966980	0.452706	0.062948	0.193731	-0.088490	0.446069	0.097894	0.215537
int_rate	0.136591	1.000000	0.145299	-0.092412	0.170675	0.002550	0.069802	0.017422	0.309684	-0.057087
installment	0.966980	0.145299	1.000000	0.433686	0.066540	0.187150	-0.082129	0.439161	0.127245	0.194354
annual_inc	0.452706	-0.092412	0.433686	1.000000	-0.202571	0.209984	-0.023964	0.318542	0.036352	0.309880
dti	0.062948	0.170675	0.066540	-0.202571	1.000000	0.318960	-0.035354	0.251918	0.179794	0.235292
open_acc	0.193731	0.002550	0.187150	0.209984	0.318960	1.000000	-0.004257	0.338569	-0.150686	0.642498
pub_rec	-0.088490	0.069802	-0.082129	-0.023964	-0.035354	-0.004257	1.000000	-0.196398	-0.090788	0.054197
revol_bal	0.446069	0.017422	0.439161	0.318542	0.251918	0.338569	-0.196398	1.000000	0.428933	0.253502
revol_util	0.097894	0.309684	0.127245	0.036352	0.179794	-0.150686	-0.090788	0.428933	1.000000	-0.113456
total_acc	0.215537	-0.057087	0.194354	0.309880	0.235292	0.642498	0.054197	0.253502	-0.113456	1.000000
mort_acc	0.199463	-0.099902	0.169009	0.343299	-0.048725	0.122485	0.056288	0.180093	-0.012703	0.400154
pub_rec_bankruptcies	-0.093270	0.058440	-0.088016	-0.045229	-0.028791	-0.009309	0.870454	-0.188318	-0.085611	0.064128

In [163]:
```
plt.figure(figsize = (15,8))
sns.heatmap(df.corr(method='spearman'), annot=True)
```

Out[163]: <AxesSubplot:>



- The spearman correlation coefficient between **Loan Amount** and **Installment** is very high (0.97).
- **=> This indicates high multicollinearity between these two features. This will result in high VIF value.**
- **=> One of them can be dropped while training the model**

Dropping the columns "application_type", "revol_bal", 'Loan_amount', 'Issue_d', 'purpose', 'title', 'address'

'tilte' and 'purpose' are just the type of loan that is filled by the customers only, which should not have a significance on the loan repayment

In [389]:
```
df.drop(columns=['application_type', 'revol_bal', 'loan_amnt', 'issue_d', 'purpose', 'title', 'address'],
```

Missing Values Treatment

In [390]:
```
df.isnull().sum()/len(df)*100
```

Out[390]:
```
term                    0.000000
int_rate                0.000000
installment             0.000000
grade                   0.000000
sub_grade               0.000000
emp_title               5.973426
emp_length              4.893534
home_ownership          0.000000
annual_inc              0.000000
verification_status     0.000000
loan_status             0.000000
dti                     0.000000
earliest_cr_line        0.000000
open_acc                0.000000
pub_rec                 0.000000
revol_util              0.066621
total_acc               0.000000
initial_list_status     0.000000
mort_acc                9.823795
pub_rec_bankruptcies    0.134666
dtype: float64
```

Median Imputation for these 3 columns

In [391]:
```
df['mort_acc'].fillna(df['mort_acc'].median(), inplace=True)
df['pub_rec_bankruptcies'].fillna(df['pub_rec_bankruptcies'].median(), inplace=True)
df['revol_util'].fillna(df['revol_util'].median(), inplace=True)
```

In [392]:
```
df.isnull().sum()/len(df)*100
```

Out[392]:
```
term                    0.000000
int_rate                0.000000
installment             0.000000
grade                   0.000000
sub_grade               0.000000
emp_title               5.973426
emp_length              4.893534
home_ownership          0.000000
annual_inc              0.000000
verification_status     0.000000
loan_status             0.000000
dti                     0.000000
earliest_cr_line        0.000000
open_acc                0.000000
pub_rec                 0.000000
revol_util              0.000000
total_acc               0.000000
initial_list_status     0.000000
mort_acc                0.000000
pub_rec_bankruptcies    0.000000
dtype: float64
```

The maximum percentage of missing value is 5.9%

Mean imputation doeas not make sense here as the target variable is a binary class; Mode imputation might lead to imbalance data, KNN can be doen on numerical features.

Dropping the missing values seems ok

In [393]:
```
df.dropna(inplace=True)
```

In [394]:
```
df.shape
```

Out[394]:
```
(330093, 20)
```

Feature Engineering

In [395]:
```
# Creation of Flags
df['pub_rec_flag'] = (df['pub_rec'] > 1.0).astype(int)

df['mort_acc_flag'] = (df['mort_acc'] > 1.0).astype(int)

df['pub_rec_bankruptcies_flag'] = (df['pub_rec_bankruptcies'] > 1.0).astype(int)
```

Using "OHE Encoder" on Categorical feature with 2 categories

In [396]:
```
# Mapping of target variable
df['loan_status']=df.loan_status.map({'Fully Paid':0, 'Charged Off':1})
```

In [397]:
```
columns_to_encode = ['term', 'initial_list_status']

df = pd.get_dummies(df, columns=columns_to_encode)
```

Using "Ordinal Encoder" on features 'grade' and 'sub_grade'. There is inherent order in these 2 categories

In [398]:
```
from sklearn.preprocessing import OrdinalEncoder

encoder = OrdinalEncoder()

df[['grade', 'sub_grade']] = encoder.fit_transform(df[['grade', 'sub_grade']])
```

Creating a new feture 'Credit_line_age' and dropping the feature 'earliest_cr_line'

In [399]:
```
# Convert to datetime
df['earliest_cr_line'] = pd.to_datetime(df['earliest_cr_line'])

# Extract year
df['earliest_cr_year'] = df['earliest_cr_line'].dt.year

# Calculate age of credit line
current_year = pd.to_datetime('now').year
df['credit_line_age'] = current_year - df['earliest_cr_year']

# Drop the original and Year column
df.drop(columns=['earliest_cr_line', 'earliest_cr_year'], inplace=True)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\arrays\datetimes.py:2224: FutureWarning: The parsi
ng of 'now' in pd.to_datetime without `utc=True` is deprecated. In a future version, this will match Time
stamp('now') and Timestamp.now()
  result, tz_parsed = tslib.array_to_datetime(
```

Using "Target Encoder" on rest of the Categorical feature

In [400]:
```
from category_encoders import TargetEncoder

# Define the columns to encode
columns_to_encode = ['emp_title', 'emp_length', 'home_ownership', 'verification_status']

encoder = TargetEncoder()

for col in columns_to_encode:
    df[col] = encoder.fit_transform(df[col], df['loan_status'])
```

In [401]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 330093 entries, 0 to 396029
Data columns (total 25 columns):
 #   Column                    Non-Null Count   Dtype  
---  ------                    --------------   -----  
 0   int_rate                  330093 non-null  float64
 1   installment               330093 non-null  float64
 2   grade                     330093 non-null  float64
 3   sub_grade                 330093 non-null  float64
 4   emp_title                 330093 non-null  float64
 5   emp_length                330093 non-null  float64
 6   home_ownership            330093 non-null  float64
 7   annual_inc                330093 non-null  float64
 8   verification_status       330093 non-null  float64
 9   loan_status               330093 non-null  int64  
 10  dti                       330093 non-null  float64
 11  open_acc                  330093 non-null  float64
 12  pub_rec                   330093 non-null  float64
 13  revol_util                330093 non-null  float64
 14  total_acc                 330093 non-null  float64
 15  mort_acc                  330093 non-null  float64
 16  pub_rec_bankruptcies      330093 non-null  float64
 17  pub_rec_flag              330093 non-null  int32  
 18  mort_acc_flag             330093 non-null  int32  
 19  pub_rec_bankruptcies_flag 330093 non-null  int32  
 20  term_ 36 months           330093 non-null  uint8  
 21  term_ 60 months           330093 non-null  uint8  
 22  initial_list_status_f     330093 non-null  uint8  
 23  initial_list_status_w     330093 non-null  uint8  
 24  credit_line_age           330093 non-null  int64  
dtypes: float64(16), int32(3), int64(2), uint8(4)
memory usage: 52.9 MB
```

In [ ]:

Model Training

In [430]:
```
from sklearn.model_selection import train_test_split

X = df.drop(columns=['loan_status'])
y = df['loan_status']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25,random_state =42)
```

In [431]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

X_train_cols = X_train.columns
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

In [432]:
```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression()
model.fit(X_train, y_train)
```

Out[432]:
```
LogisticRegression()
```

In [421]:
```
model.coef_
```

Out[421]:
```
array([[-0.01687345,  0.09795484, -0.0278359 ,  0.44913378,  1.4156598 ,
         0.06555052,  0.12690726, -0.01140279,  0.05294805,  0.1466086 ,
         0.13691803,  0.03029027,  0.14306217, -0.03885671, -0.05125207,
        -0.06831451, -0.01824214, -0.0350281 ,  0.02082104, -0.11967345,
         0.11967345,  0.04260717, -0.04260717,  0.0337882 ]])
```

In [422]:
```
model.intercept_
```

Out[422]:
```
array([-2.08055966])
```

Accuracy score of train and test data

In [433]:
```
print(f'Training Accuracy:{model.score(X_train,y_train)}')
print(f'Test Accuracy:{model.score(X_test,y_test)}')
```

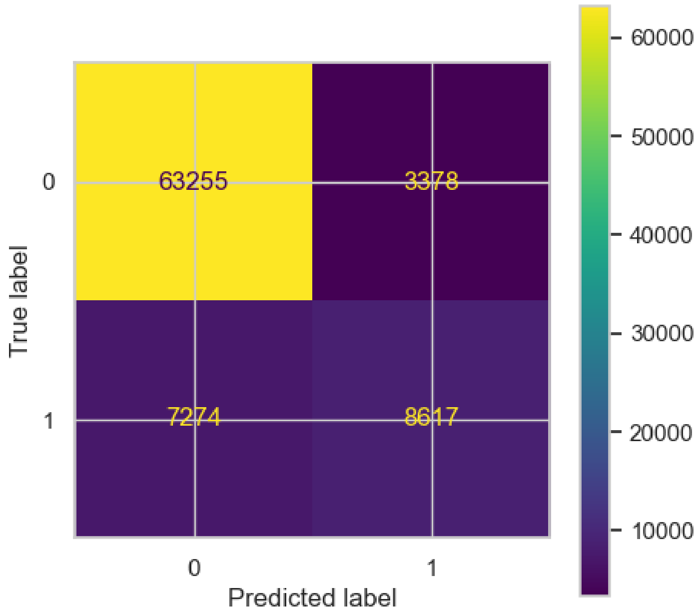
```
Training Accuracy:0.8695232440248981
Test Accuracy:0.8709223983326063
```

In [434]:
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

y_pred = model.predict(X_test)

conf_matrix = confusion_matrix(y_test, y_pred)
conf_matrix
```

Out[434]:
```
array([[63255,  3378],
       [ 7274,  8617]], dtype=int64)
```

In [435]:
```
# ax used here to control the size of confusion matrix
fig, ax = plt.subplots(figsize=(5,5))
ConfusionMatrixDisplay(conf_matrix).plot(ax = ax)
```

Out[435]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x21ac7c05fa0>
```



Finding Accuracy using Confusion Matrix

In [436]:
```
np.diag(conf_matrix).sum() / conf_matrix.sum()
```

Out[436]:
```
0.8709223983326063
```

In [437]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.90      0.95      0.92     66633
           1       0.72      0.54      0.62     15891

    accuracy                           0.87     82524
   macro avg       0.81      0.75      0.77     82524
weighted avg       0.86      0.87      0.86     82524
```

In [439]:
```
total_instances = 63255 + 3378 + 7274 + 8617  # Total instances in the confusion matrix

# Calculate percentages
percentage_FP = (3378 / total_instances) * 100
percentage_FN = (7274 / total_instances) * 100

print("Percentage of False Positives (FP): {:.2f}%".format(percentage_FP))
print("Percentage of False Negatives (FN): {:.2f}%".format(percentage_FN))
```

```
Percentage of False Positives (FP): 4.09%
Percentage of False Negatives (FN): 8.81%
```

Interpretation:¶

- **90% of the instances predicted as class 0 were actually class 0, and 72% of the instances predicted as class 1 were actually class 1.**
- **95% of the actual instances of class 0 were correctly predicted as class 0, but only 54% of the actual instances of class 1 were correctly predicted as class 1.**
- **F1-score balances precision and recall. A higher F1-score indicates a better balance between precision and recall.**
- **Overall accuracy is 0.87, meaning that the model correctly predicted the target variable in approximately 87% of the instances.**
- **Weighted Avg precision is 0.86, recall is 0.87, and F1-score is 0.86.**

=> Recall score: 0.95 and Precision score: 0.90. Which tells us that there are more false positives than the false negatives.

=> From Confusion Matrix it can be seen that FP = 4% of total cases & FN = 9% of Total Cases

=> If Recall value is low (i.e. FN are high), it means Bank's NPA (defaulters) may increase.

=> If Precision value is low (i.e. FP are high), it means Bank might loose out on an opportunity to finance more individuals and earn interest on it.

Hyperparameter Tuning

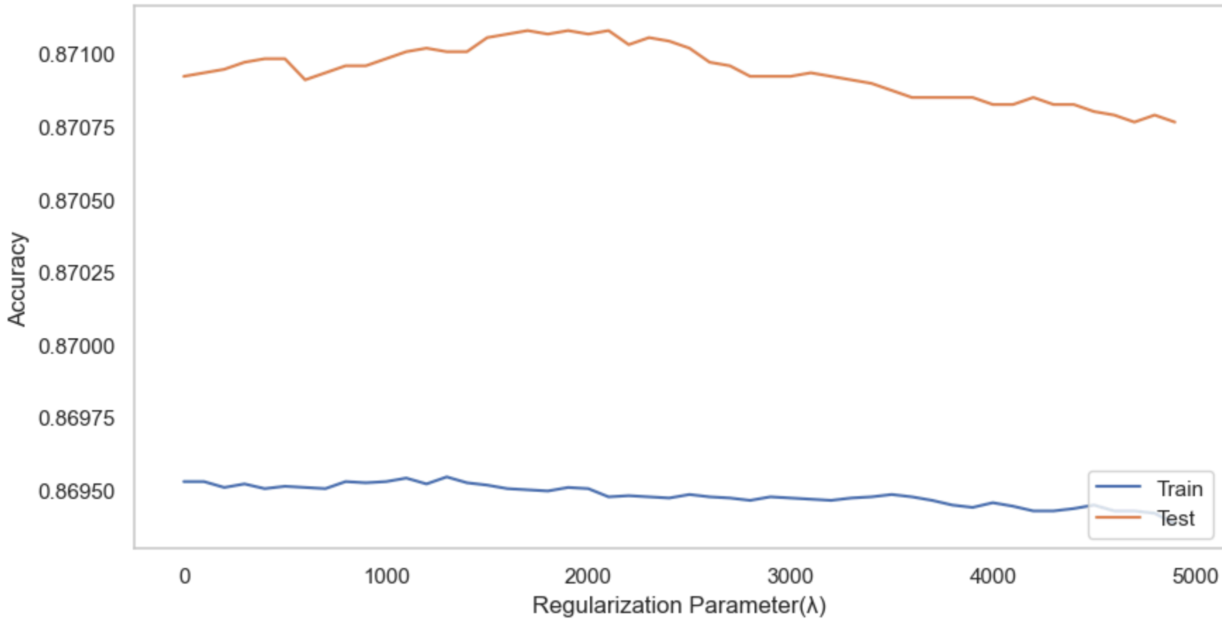
In [484]:
```
def acc(y_test,y_pred):
    return np.sum(y_test==y_pred)/len(y_test)

c = np.logspace(-3, 3, 7)

Accuracy_score = []
for i in c:
    lgr = LogisticRegression(penalty='l2',random_state=42, C=i)
    lgr.fit(X_train,y_train)
    y_pred = lgr.predict(X_test)
    Accuracy_score.append(acc(y_test,y_pred))
    print(i,acc(y_test,y_pred))
```

```
0.001 0.8705831030972808
0.01 0.8709829867674859
0.1 0.8709345160195822
1.0 0.8709223983326063
10.0 0.8709223983326063
100.0 0.8709223983326063
1000.0 0.8709223983326063
```

In [ ]:
```
from sklearn.pipeline import make_pipeline
train_scores = []
val_scores = []
scaler = StandardScaler()
for la in np.arange(0.01, 5000.0, 100): # range of values of Lambda
    scaled_lr = make_pipeline(scaler, LogisticRegression(C=1/la))
    scaled_lr.fit(X_train, y_train)
    train_score = accuracy(y_train, scaled_lr.predict(X_train))
    val_score = accuracy(y_val, scaled_lr.predict(X_val))
    train_scores.append(train_score)
    val_scores.append(val_score)
```

In [446]:
```
range_of_C = np.arange(0.01, 500.0, 10)
train_scores = []
test_scores = []
for i in range_of_C:
    model = LogisticRegression(C=1/i)
    model.fit(X_train, y_train)
    tr_score = model.score(X_train,y_train)
    train_scores.append(tr_score)
    tst_score = model.score(X_test,y_test)
    test_scores.append(tst_score)
```

In [447]:
```
plt.figure(figsize=(10,5))
plt.plot(list(np.arange(0.01, 5000.0, 100)), train_scores, label="Train")
plt.plot(list(np.arange(0.01, 5000.0, 100)), test_scores, label="Test")
plt.legend(loc='lower right')

plt.xlabel("Regularization Parameter(λ)")
plt.ylabel("Accuracy")
plt.grid()
plt.show()
```



- **We see that changing the value of C does not make any difference in performance**

In [459]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score

param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000], 'penalty': ['l2']}
logreg = LogisticRegression()

# Using K-fold method with k=5
grid_search = GridSearchCV(logreg, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

print('Best Hyperparameters:', grid_search.best_params_)

best_model = grid_search.best_estimator_
y_pred_test = best_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred_test)
print('*' * 50)
print('Accuracy:', round(accuracy * 100, 2), '%')
```

```
Best Hyperparameters: {'C': 0.01, 'penalty': 'l2'}
**************************************************
Accuracy: 87.1 %
```

Regularization and hyperparameter tuning(c) is not making any difference

In [ ]:

ROC Curve - An ROC curve (receiver operating characteristic curve) is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters:

- True Positive Rate
- False Positive Rate
- True Positive Rate (TPR) is a synonym for recall and is therefore defined as follows: TPR=(TP)/(TP+FN)
- False Positive Rate (FPR) is defined as follows: FPR=(FP)/(FP+TN)
- An ROC curve plots TPR vs. FPR at different classification thresholds. Lowering the classification threshold classifies more items as positive, thus increasing both False Positives and True Positives. The following figure shows a typical ROC curve.

In [460]:
```
from sklearn.metrics import roc_auc_score, roc_curve

probability = model.predict_proba(X_test)
```

In [463]:
```
probability
```

Out[463]:
```
array([[0.89100821, 0.10899179],
       [0.34676247, 0.65323753],
       [0.26874922, 0.73125078],
       ...,
       [0.87951164, 0.12048836],
       [0.96266857, 0.03733143],
       [0.2970843 , 0.7029157 ]])
```

Observe

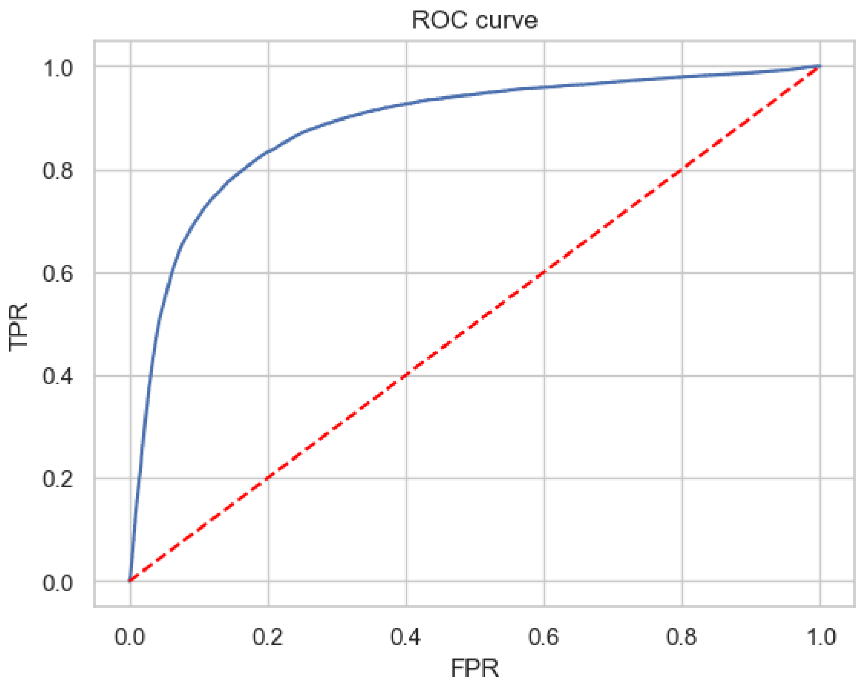
`Probability` variable contains 2 probability $P(Y=0|X)$ and $P(Y=1|X)$

But for thresholding we need only one probability, what can be done ?

Ans: lets consider only $p = P(Y=1|X)$

In [464]:
```
probabilites = probability[:,1]
fpr, tpr, thr = roc_curve(y_test,probabilites)
```

In [466]:
```
plt.plot(fpr,tpr)

#random model
plt.plot(fpr,fpr,'--',color='red' )
plt.title('ROC curve')
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.show()
```



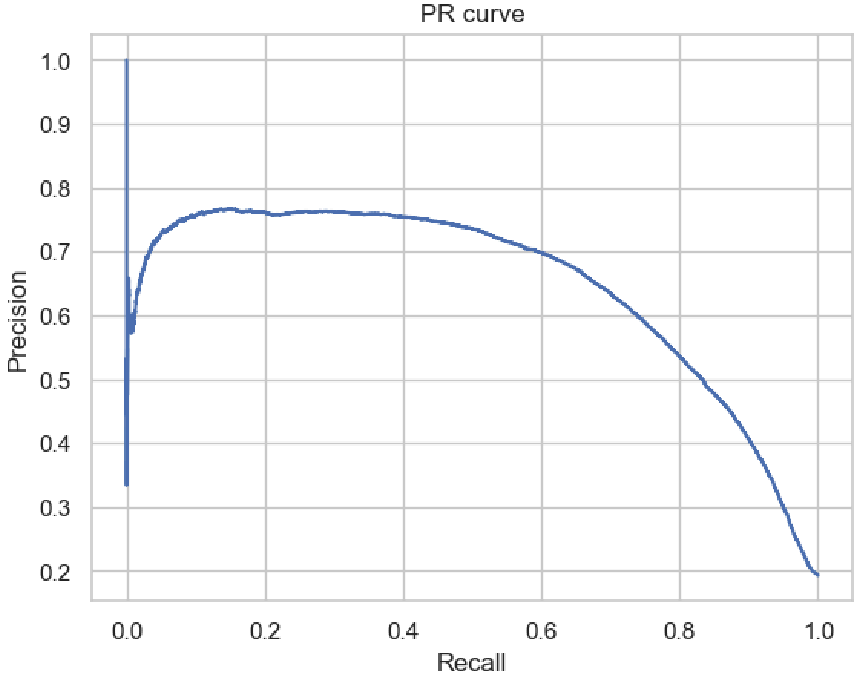
In [467]:
```
# Calculate ROC AUC score
roc_auc_score(y_test,probabilites)
```

Out[467]: 0.8836359699764297

In [ ]:

In [468]:
```
from sklearn.metrics import precision_recall_curve, auc
precision, recall, thr = precision_recall_curve(y_test, probabilites)
```

In [469]:
```
plt.plot(recall, precision)

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('PR curve')
plt.show()
```



In [470]:
```
auc(recall, precision)
```

Out[470]:
```
0.6478590628636132
```

In [ ]:

In [472]:
```
# Model creation, prediction

def training(model,X_train,y_train,X_test,y_test):

    model.fit(X_train, y_train)

    train_y_pred = model.predict(X_train)
    test_y_pred = model.predict(X_test)

    train_score = f1_score(y_train, train_y_pred)
    test_score = f1_score(y_test, test_y_pred)

    return train_score,test_score
```

In [475]:
```
pip install imbalanced-learn
```

```
Defaulting to user installation because normal site-packages is not writeable
Collecting imbalanced-learn
  Downloading imbalanced_learn-0.12.2-py3-none-any.whl (257 kB)
     -------------------------------------- 258.0/258.0 kB 2.6 MB/s eta 0:00:00
Requirement already satisfied: scipy>=1.5.0 in c:\programdata\anaconda3\lib\site-packages (from imbalance
d-learn) (1.9.1)
Requirement already satisfied: threadpoolctl>=2.0.0 in c:\programdata\anaconda3\lib\site-packages (from i
mbalanced-learn) (2.2.0)
Requirement already satisfied: scikit-learn>=1.0.2 in c:\programdata\anaconda3\lib\site-packages (from im
balanced-learn) (1.0.2)
Requirement already satisfied: numpy>=1.17.3 in c:\programdata\anaconda3\lib\site-packages (from imbalanc
ed-learn) (1.21.5)
Collecting joblib>=1.1.1
  Downloading joblib-1.4.2-py3-none-any.whl (301 kB)
     -------------------------------------- 301.8/301.8 kB 6.2 MB/s eta 0:00:00
Installing collected packages: joblib, imbalanced-learn
Successfully installed imbalanced-learn-0.12.2 joblib-1.4.2
Note: you may need to restart the kernel to use updated packages.
```

In [477]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import f1_score
from imblearn.over_sampling import SMOTE

# Create an instance of SMOTE
smt = SMOTE()


# Perform SMOTE on the training data
print('Before SMOTE')
print(y_train.value_counts())

X_sm, y_sm = smt.fit_resample(X_train, y_train)
print('After Oversampling')
print(y_sm.value_counts())

lgr_model = LogisticRegression(C= 5, penalty= 'l1', solver = 'liblinear')

f1_train,f1_test = training(lgr_model,X_sm, y_sm,X_test,y_test)

print(f'Training F1 score:{f1_train}, Testing F1 score:{f1_test}')
```

```
Before SMOTE
0    199441
1     48128
Name: loan_status, dtype: int64
After Oversampling
0    199441
1    199441
Name: loan_status, dtype: int64
Training F1 score:0.8171816767006594, Testing F1 score:0.6482421120412105
```

In [ ]:

In [479]:
```
from statsmodels.stats.outliers_influence import variance_inflation_factor

def calc_vif(X):
    # Calculating the VIF
    vif=pd.DataFrame()
    vif['Feature']=X.columns
    vif['VIF']=[variance_inflation_factor(X.values,i) for i in range(X.shape[1])]
    vif['VIF']=round(vif['VIF'],2)
    vif=vif.sort_values(by='VIF',ascending=False)
    return vif

calc_vif(X)[:5]
```

```
C:\ProgramData\Anaconda3\lib\site-packages\statsmodels\stats\outliers_influence.py:195: RuntimeWarning: d
ivide by zero encountered in double_scalars
  vif = 1. / (1. - r_squared_i)
```

Out[479]:

	Feature	VIF
22	initial_list_status_w	inf
21	initial_list_status_f	inf
20	term_ 60 months	inf
19	term_ 36 months	inf
3	sub_grade	42.64

In [480]:
```
calc_vif(X)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\statsmodels\stats\outliers_influence.py:195: RuntimeWarning: d
ivide by zero encountered in double_scalars
  vif = 1. / (1. - r_squared_i)
```

Out[480]:

	Feature	VIF
22	initial_list_status_w	inf
21	initial_list_status_f	inf
20	term_ 60 months	inf
19	term_ 36 months	inf
3	sub_grade	42.64
2	grade	22.26
0	int_rate	20.81
11	pub_rec	3.66
14	mort_acc	3.09
17	mort_acc_flag	2.94
15	pub_rec_bankruptcies	2.85
16	pub_rec_flag	2.64
13	total_acc	2.12
18	pub_rec_bankruptcies_flag	1.93
10	open_acc	1.88
7	annual_inc	1.61
1	installment	1.41
9	dti	1.38
6	home_ownership	1.36
12	revol_util	1.24
23	credit_line_age	1.19
8	verification_status	1.16
4	emp_title	1.07
5	emp_length	1.06

There are a few features with vif=inf and a few more with vif>5

we can also consider removing those features and retraining the model

In [ ]:

Insights and Recommendations:¶

1. Model Performance Metrics:

- Accuracy: Achieved an overall accuracy of 87% indicating the model's ability to correctly classify instances.

2. Classification Report Analysis:

- Class 0 (Non-Defaulters): High precision (90%) and recall (95%). Demonstrates reliable identification of non-defaulters, minimizing false negatives.
- Class 1 (Defaulters): Precision at 72%, indicating areas for improvement in avoiding false positives. Recall at 54%, suggesting the need to capture more instances of actual defaulters.