Exploratory Data Analysis

In [ ]:
```
import os
import re
# Data manipulation
import numpy as np
import pandas as pd
# Visualization
import matplotlib.pyplot as plt
import seaborn as sns
# Preprocessing
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Statistical inference
import pingouin as pg
from scipy.stats import kstest, shapiro
from scipy import stats

# Set pandas dataframe display options
pd.options.display.max_columns = None
pd.options.display.max_rows = 500
```

Load Dataset

In [ ]:
```
df = pd.read_csv('../data/raw/loan_data.csv', sep=',')
```

In [ ]:
```
# Adjust columns names
df.columns = [col.replace('.', '_') for col in df.columns.values]
# Rename target column
df = df.rename(columns={'not_fully_paid': 'default'})
```

In [ ]:
```
# Checking shape of the data
df.shape
```

Out[ ]: (9578, 14)

This dataframe consists of 9578 observatios and 14 variables(13 predictors and 1 target variable)

In [ ]:
```
# Check columns general information
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9578 entries, 0 to 9577
Data columns (total 14 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   credit_policy      9578 non-null   int64  
 1   purpose            9578 non-null   object 
 2   int_rate           9578 non-null   float64
 3   installment        9578 non-null   float64
 4   log_annual_inc     9578 non-null   float64
 5   dti                9578 non-null   float64
 6   fico               9578 non-null   int64  
 7   days_with_cr_line  9578 non-null   float64
 8   revol_bal          9578 non-null   int64  
 9   revol_util         9578 non-null   float64
 10  inq_last_6mths     9578 non-null   int64  
 11  delinq_2yrs        9578 non-null   int64  
 12  pub_rec            9578 non-null   int64  
 13  default            9578 non-null   int64  
dtypes: float64(6), int64(7), object(1)
memory usage: 1.0+ MB
```

Missing values

In [ ]:
```
# Check for missing values
df.isna().sum()
```

Out[ ]:
```
credit_policy        0
purpose              0
int_rate             0
installment          0
log_annual_inc       0
dti                  0
fico                 0
days_with_cr_line    0
revol_bal            0
revol_util           0
inq_last_6mths       0
delinq_2yrs          0
pub_rec              0
default              0
dtype: int64
```

The is not missing values in the dataset

Duplicated records

In [ ]:
```
df.duplicated().sum()
```

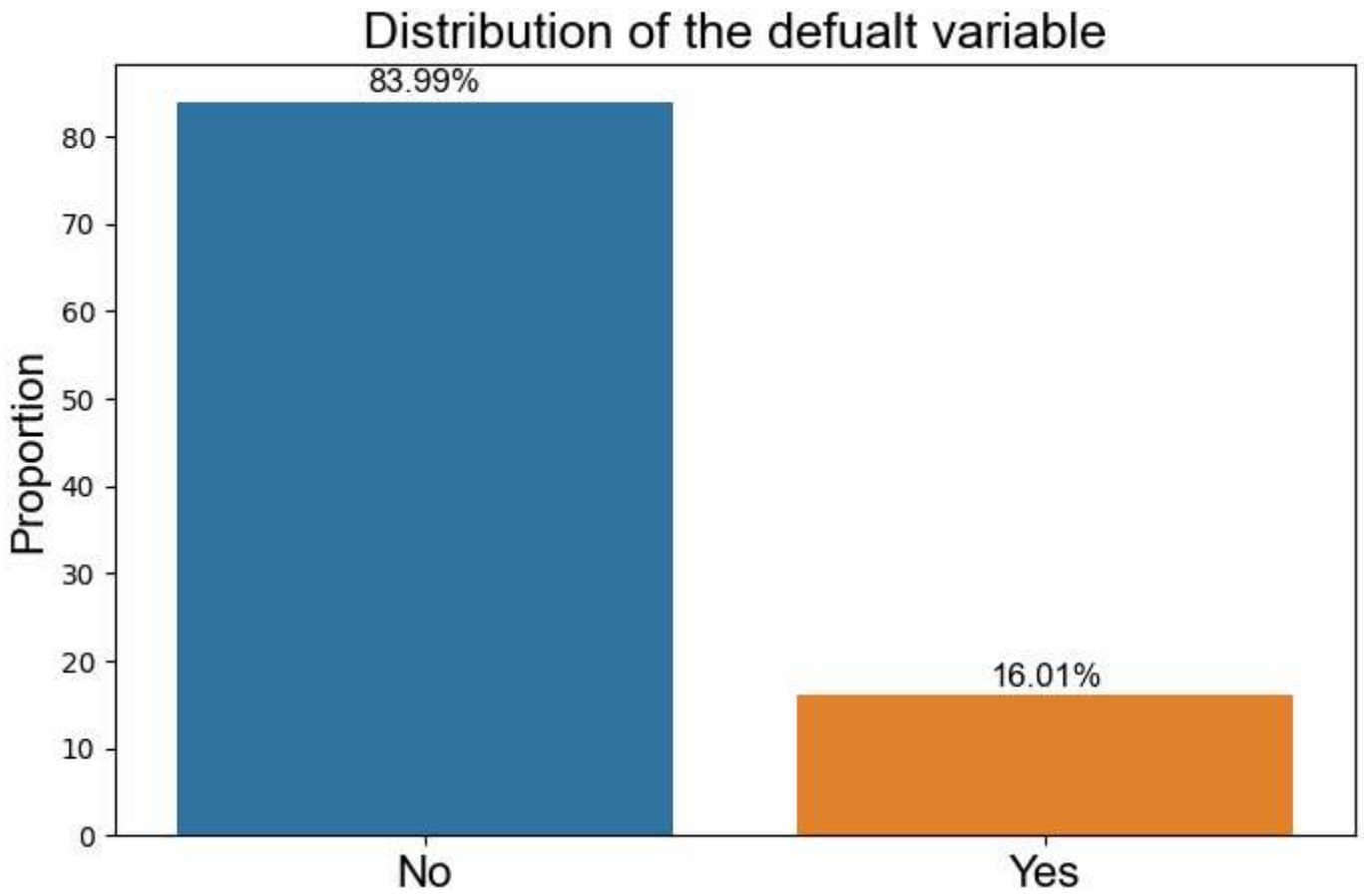
Out[ ]:
```
0
```

There is not duplicated records in the dataset

Target Variable Distribution

In [ ]:
```
target = df.default.value_counts(normalize=True).to_frame().reset_index()
target['proportion'] = round(target.proportion * 100, 2)

fig, ax = plt.subplots(figsize=(8, 5))
ax = sns.barplot(data=target, x='default', y='proportion', hue='default', legend=False)
ax.set_xlabel(None)
ax.set_title("Distribution of the defualt variable",
             fontdict={'fontname': 'Arial', 'fontsize': 18})
ax.set_xticks(range(2))
ax.set_xticklabels(['No', 'Yes'], fontdict={'fontname': 'Arial', 'fontsize': 16})
```

```
ax.set_ylabel("Proportion", fontdict={'fontname': 'Arial', 'fontsize': 16})

for p in ax.patches:
    height = p.get_height()
    ax.text(
        p.get_x() + p.get_width() / 2,
        height + 1,
        f'{height:.2f}%',
        ha='center',
        fontname='Arial',
        fontsize=12
    )

plt.show()
```



We can see that the target variable is unbalanced, 84% of the clients did not default and 16% are default.

Univariate Analysis

In this seccion numerical and categorical variables are analized independently in order to have an idea of their distributions

In [ ]:
```python
def univariated_num_plot(df, var_name, box_xlabel, figheight=5, figwidth=8,
                         fuente='Arial', fig_fontsize=18, title_fontsize=16,
                         ylabel_fontsize=14):
    "Create univariated plots for numerical variables"

    # mean (mu) and standar deviation (sigma) from int_rate data
    mu, sigma = stats.norm.fit(df[var_name])

    # Theorical values for the normal distribution
    x_hat = np.linspace(min(df[var_name]), max(df[var_name]), num=100)
    y_hat = stats.norm.pdf(x_hat, mu, sigma)

    fig, axes = plt.subplots(1, 2, figsize=(figwidth, figheight))
    fig.suptitle(f"Distribution of the variable - {var_name.capitalize()}", fontsize=fig_fontsize)
    # Boxplot
    sns.boxplot(data=df, y=var_name, ax=axes[0])
    # Density
    axes[1].plot(x_hat, y_hat, linewidth=2, color='r', label='Normal')
    axes[1].hist(x=df[var_name], density=True, bins=30, color="#3182bd", alpha=0.5)
    sns.rugplot(data=df, x=var_name, color='#343a40', alpha=0.5, ax=axes[1])
    # Text format
    axes[0].set_title('Box Plot', fontdict={'fontname': fuente,
                                            'size': title_fontsize})
    axes[0].set_ylabel(box_xlabel, fontdict={'fontname': fuente,
                                             'size': ylabel_fontsize})
    axes[0].set_xlabel(None)
    axes[1].set_title('Density Plot', fontdict={'fontname': fuente,
                                                'size': title_fontsize})
    axes[1].set_ylabel("Frequency", fontdict={'fontname': fuente, 'size': ylabel_fontsize})
    axes[1].set_xlabel(box_xlabel)
    axes[1].legend()
    plt.tight_layout()
    plt.show()
```

In [ ]:
```python
# Split the original dataframen into 2 dataframes, one for numerical variables and another for categorical variables
num_df = df.select_dtypes(exclude=['object', 'category'])
num_df = num_df.drop('default', axis=1) # drop target variable

cat_df = df.select_dtypes(include=['object', 'category'])
```

Numeric Variables

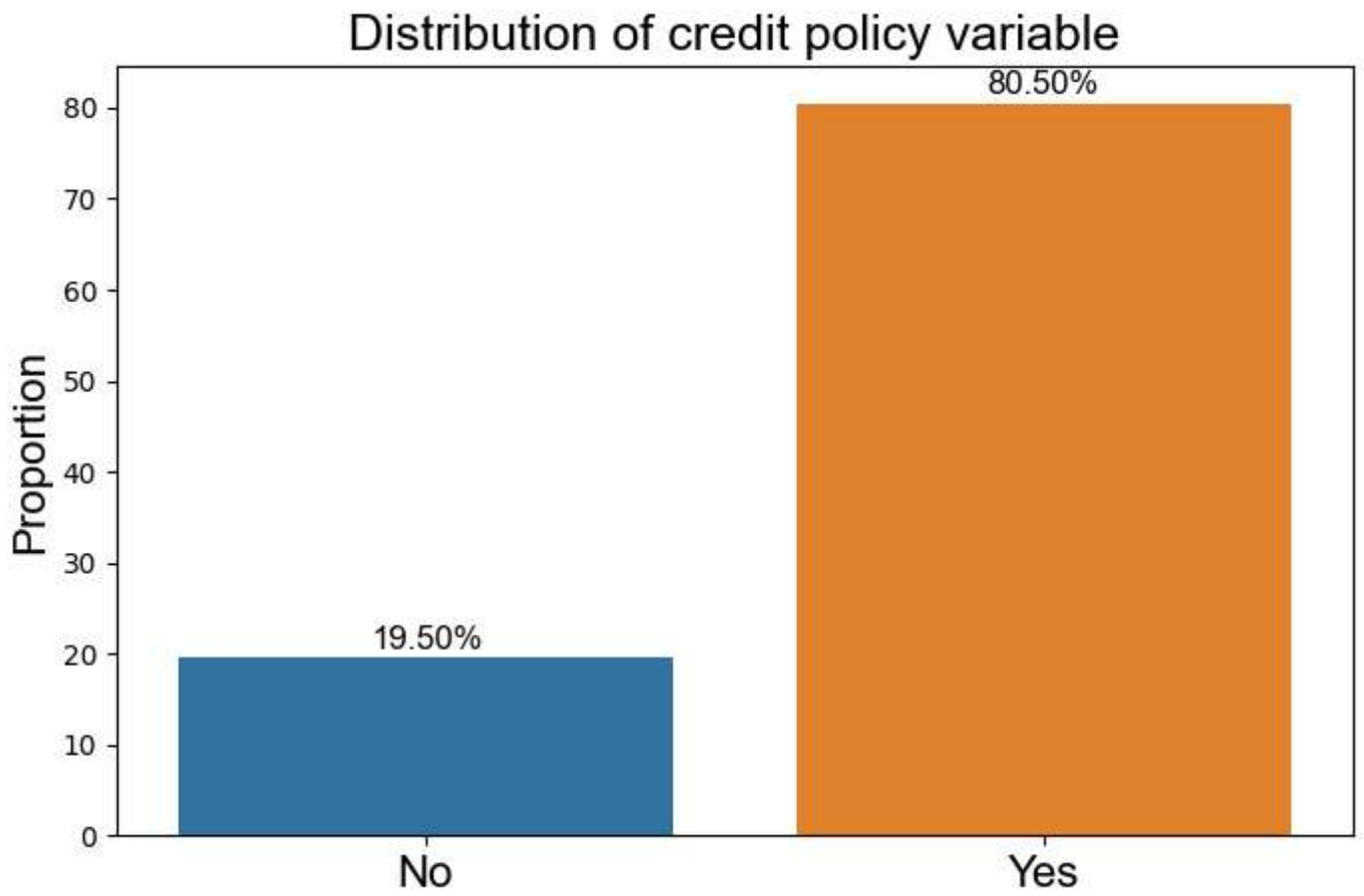
In [ ]:
```
(num_df
 .describe()
 .T
 .style
 .format({
     'count': '{0:.0f}',
     'mean': '{0:.4f}',
     'std': '{0:.4f}',
     '25%': '{0:.4f}',
     '50%': '{0:.4f}',
     '75%': '{0:.4f}',
     'max': '{0:.4f}'
     })
)
```

Out[ ]:

	count	mean	std	min	25%	50%	75%	max
credit_policy	9578	0.8050	0.3962	0.000000	1.0000	1.0000	1.0000	1.0000
int_rate	9578	0.1226	0.0268	0.060000	0.1039	0.1221	0.1407	0.2164
installment	9578	319.0894	207.0713	15.670000	163.7700	268.9500	432.7625	940.1400
log_annual_inc	9578	10.9321	0.6148	7.547502	10.5584	10.9289	11.2913	14.5284
dti	9578	12.6067	6.8840	0.000000	7.2125	12.6650	17.9500	29.9600
fico	9578	710.8463	37.9705	612.000000	682.0000	707.0000	737.0000	827.0000
days_with_cr_line	9578	4560.7672	2496.9304	178.958333	2820.0000	4139.9583	5730.0000	17639.9583
revol_bal	9578	16913.9639	33756.1896	0.000000	3187.0000	8596.0000	18249.5000	1207359.0000
revol_util	9578	46.7992	29.0144	0.000000	22.6000	46.3000	70.9000	119.0000
inq_last_6mths	9578	1.5775	2.2002	0.000000	0.0000	1.0000	2.0000	33.0000
delinq_2yrs	9578	0.1637	0.5462	0.000000	0.0000	0.0000	0.0000	13.0000
pub_rec	9578	0.0621	0.2621	0.000000	0.0000	0.0000	0.0000	5.0000

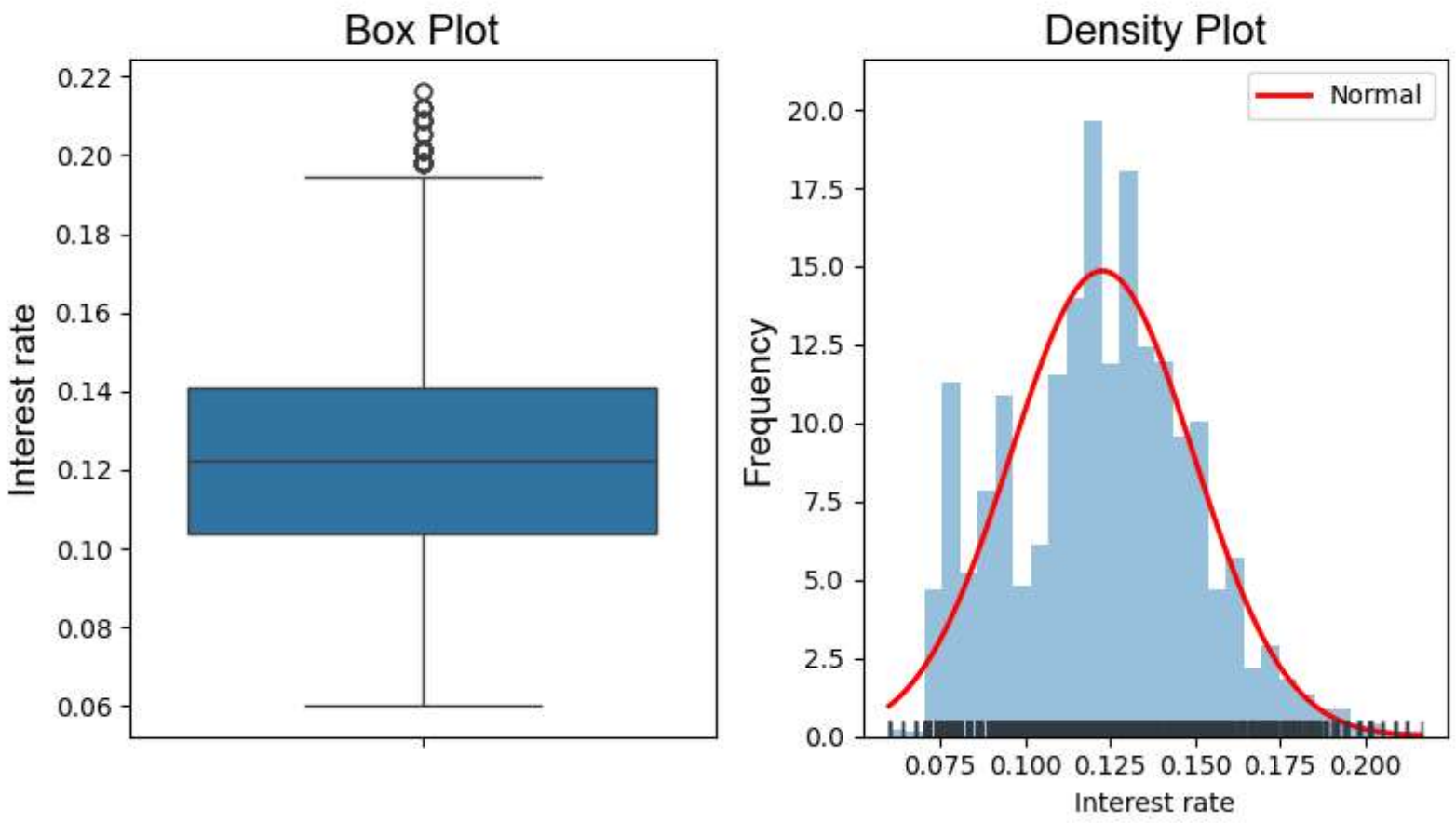
In [ ]:

```
# univariated_plot(num_df, var_name='credit_policy', box_xlabel='Credit Policy')
policy = df.credit_policy.value_counts(normalize=True).to_frame().reset_index()
policy['proportion'] = round(policy.proportion * 100, 2)

fig, ax = plt.subplots(figsize=(8, 5))
ax = sns.barplot(data=policy, x='credit_policy', y='proportion', hue='credit_policy', legend=False)
ax.set_xlabel(None)
ax.set_title("Distribution of credit policy variable",
             fontdict={'fontname': 'Arial', 'fontsize': 18})
ax.set_xticks(range(2))
ax.set_xticklabels(['No', 'Yes'], fontdict={'fontname': 'Arial', 'fontsize': 16})
ax.set_ylabel("Proportion", fontdict={'fontname': 'Arial', 'fontsize': 16})

for p in ax.patches:
    height = p.get_height()
    ax.text(
        p.get_x() + p.get_width() / 2,
        height + 1,
        f'{height:.2f}%',
        ha='center',
        fontname='Arial',
        fontsize=12
    )

plt.show()
```



Most of the customers meet the Lending Club credit policy

In [ ]:
```
univariated_num_plot(num_df, var_name='int_rate', box_xlabel='Interest rate')
```



50% of the customers has an interes rate lower than 12.21%. The interest rate appearst to follow a normal distribution, so I will perfmon the ks test for testing normality in the interes rate distribution.

The null and alternative hypothesis are:

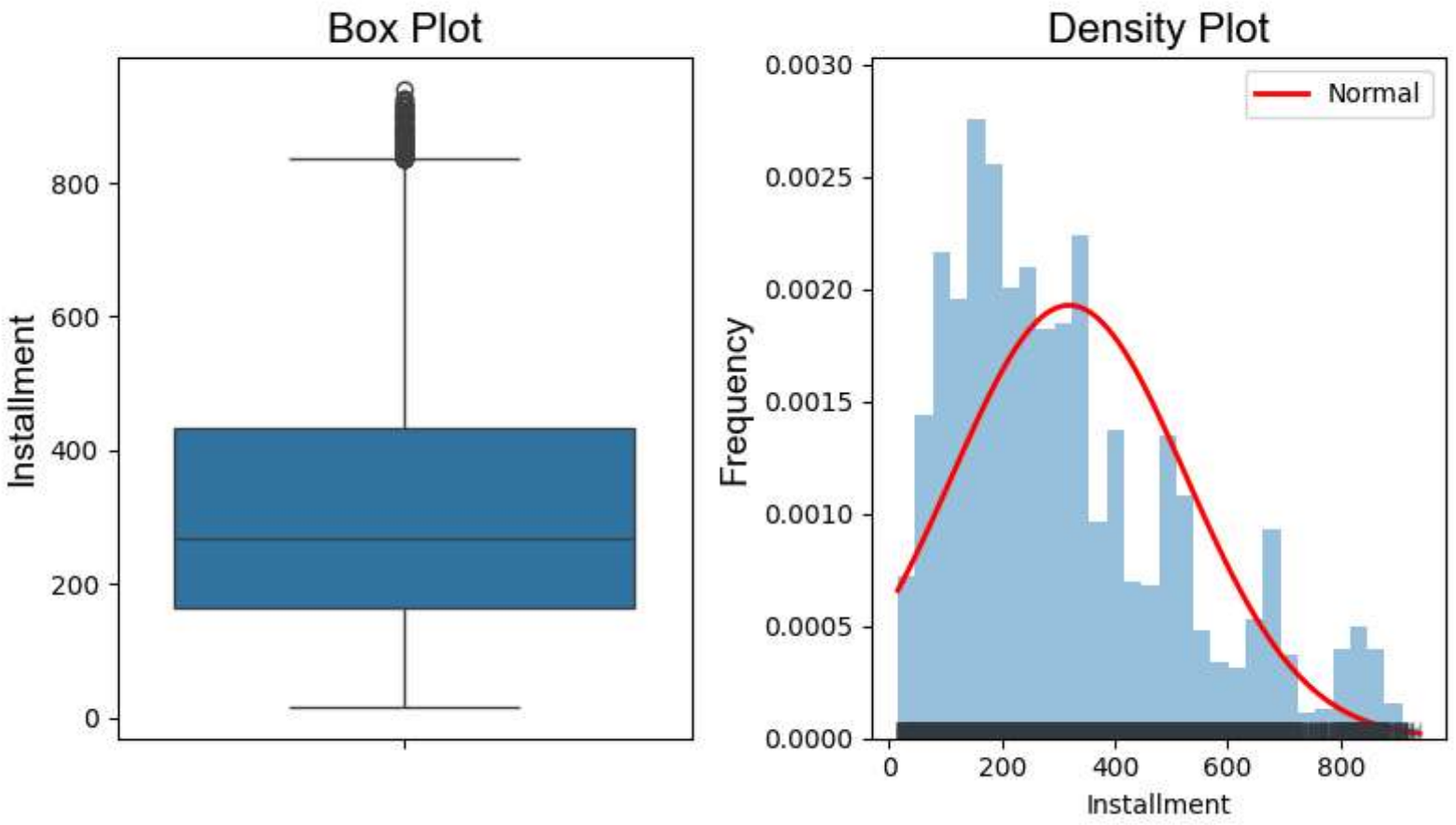
Ho: The int_rate variable follows a normal distribution
Ha: The int_rate variable doesn't follow a normal distribution

The test is perfomed with a significance level of 5%

In [ ]:
```
alpha = 0.05

# Normality test
stat, p_value = kstest(num_df.int_rate, stats.norm.cdf)

if p_value > alpha:
    print("Fail to reject Ho. The int_rate variable does follow a normal distribution")
else:
    print("Reject Ho. The int_rate variable does not follow a normal distribution")
```

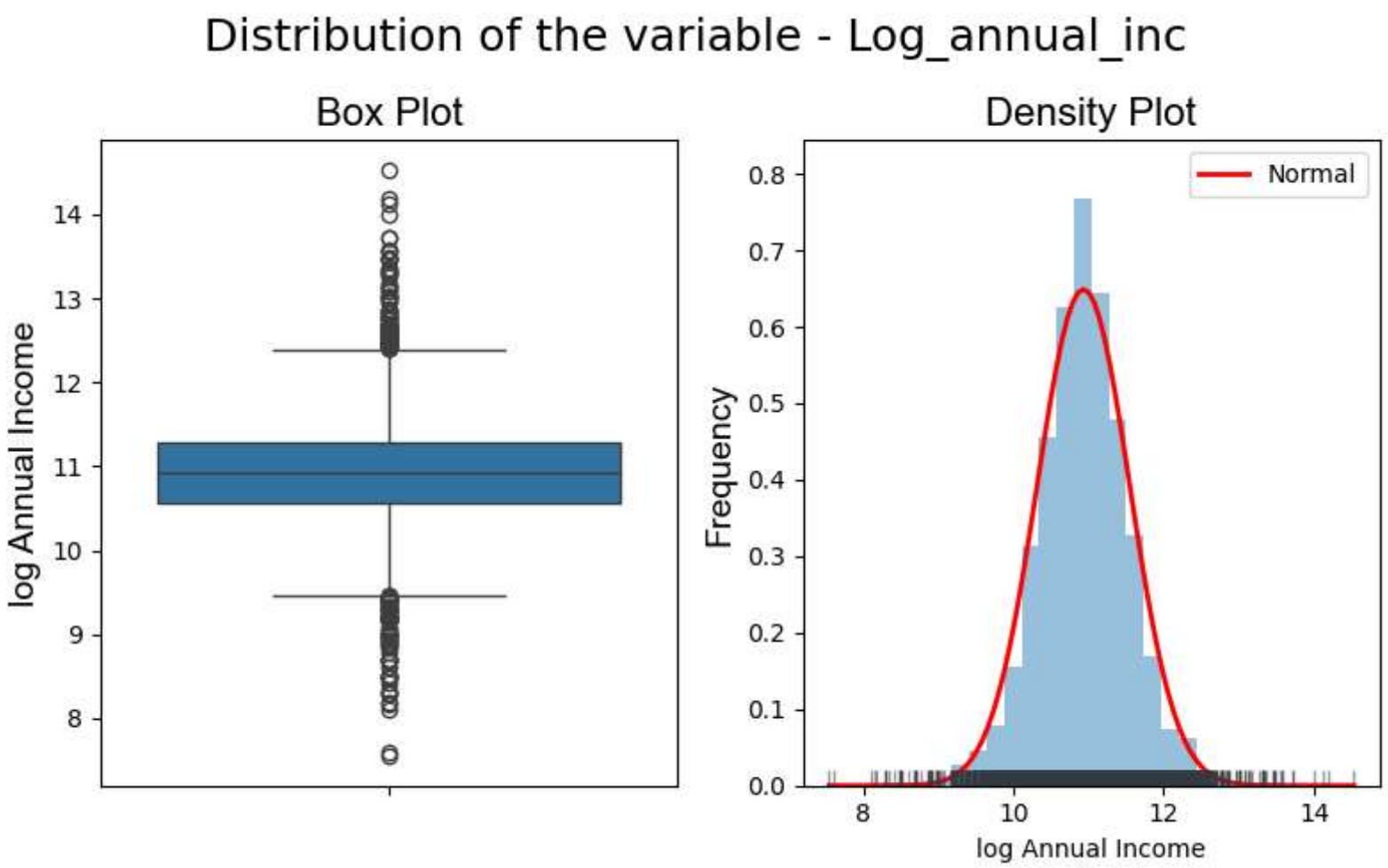
```
Reject Ho. The int_rate variable does not follow a normal distribution
```

In [ ]:
```
univariated_num_plot(num_df, var_name='installment', box_xlabel='Installment')
```



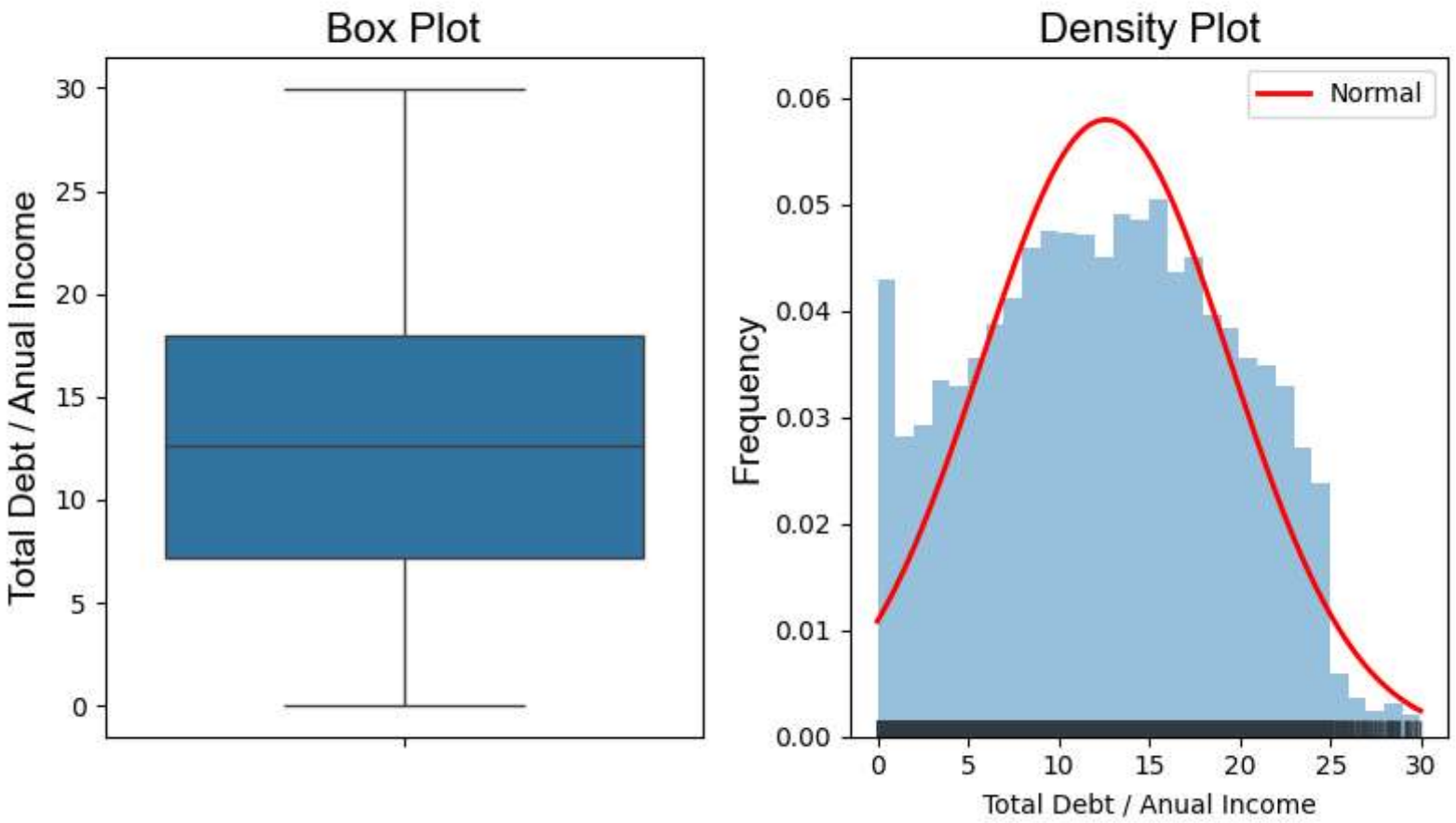
Most of the customers have a monthly installment under 400. The variable doesn't follow a normal distribution, its distribution is right skedwd

In [ ]:
```
univariated_num_plot(num_df, var_name='log_annual_inc', box_xlabel='log Annual Income')
```



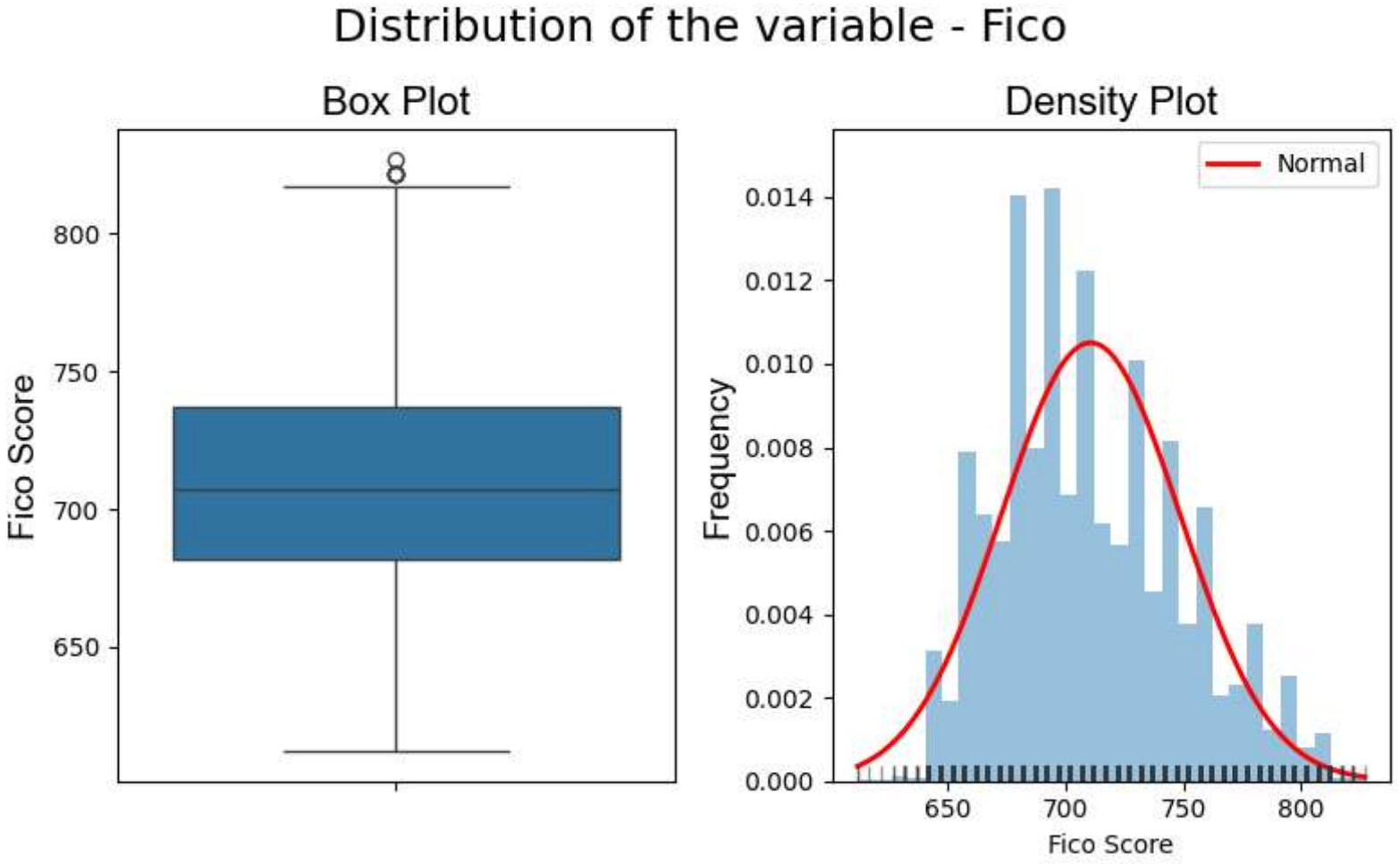
The annual income was transformed in the dataset using the natural log. According to the plot, the variable follows a normal distribution and there are some posible outlier in both tails

In [ ]:
```
univariated_num_plot(num_df, var_name='dti', box_xlabel='Total Debt / Anual Income')
```



The dti variable tells us at hos indebted the clients are. The higher the value the riskier a customer is. In the case of the LendingClub's customers, the greatest debt to income ratio is about 30% and most of customers have a dti around 15%.

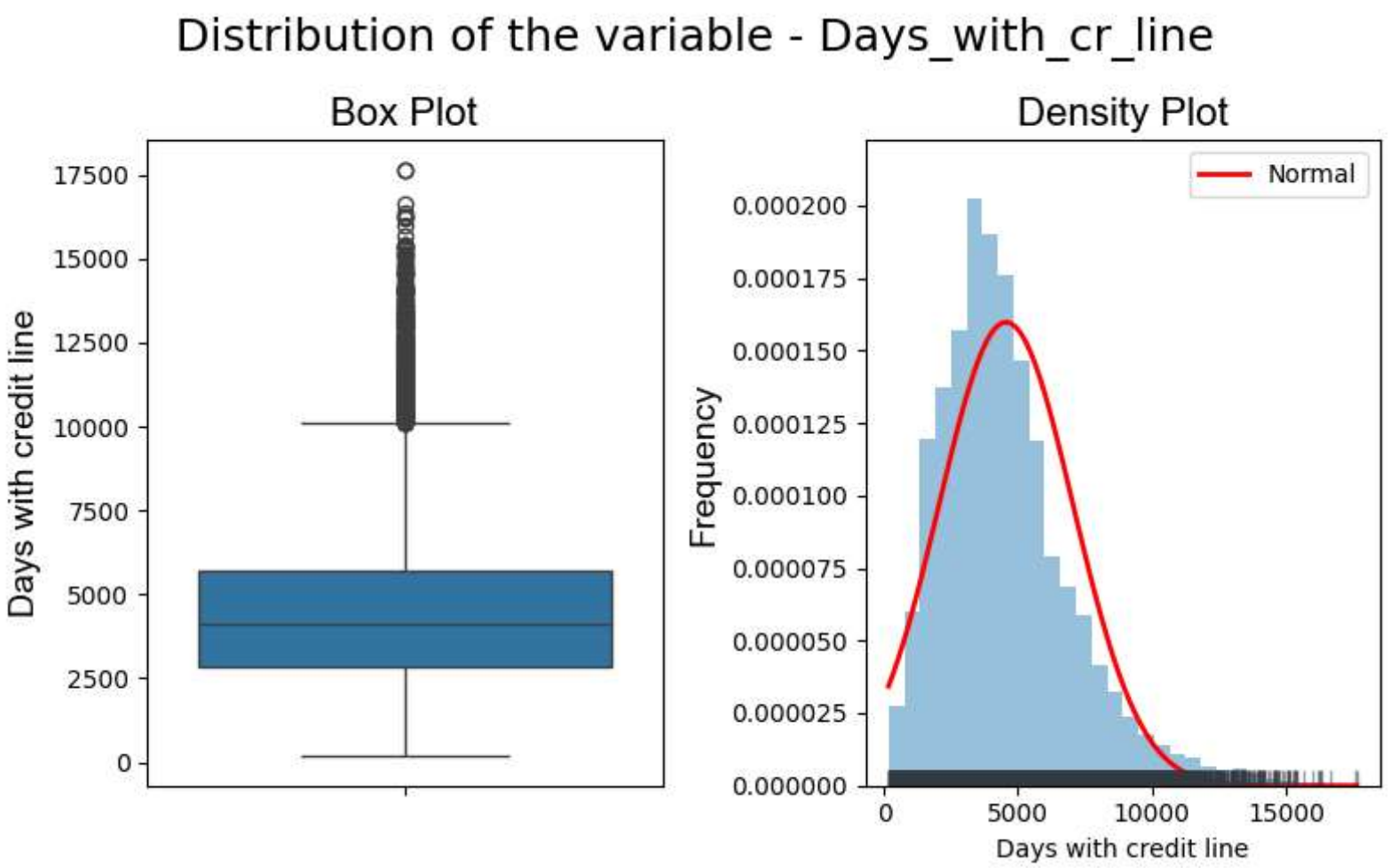
In [ ]:
```
univariated_num_plot(num_df, var_name='fico', box_xlabel='Fico Score')
```



FICO score is a credit score which uses socio-demographic, financial and behavioral information, among others. This score typically has a value from 300 to 850. The higher the value the better. According to the grahps we can see tha 50% of the customers has a FICO score between 682 (Q1) and 737 (Q3).

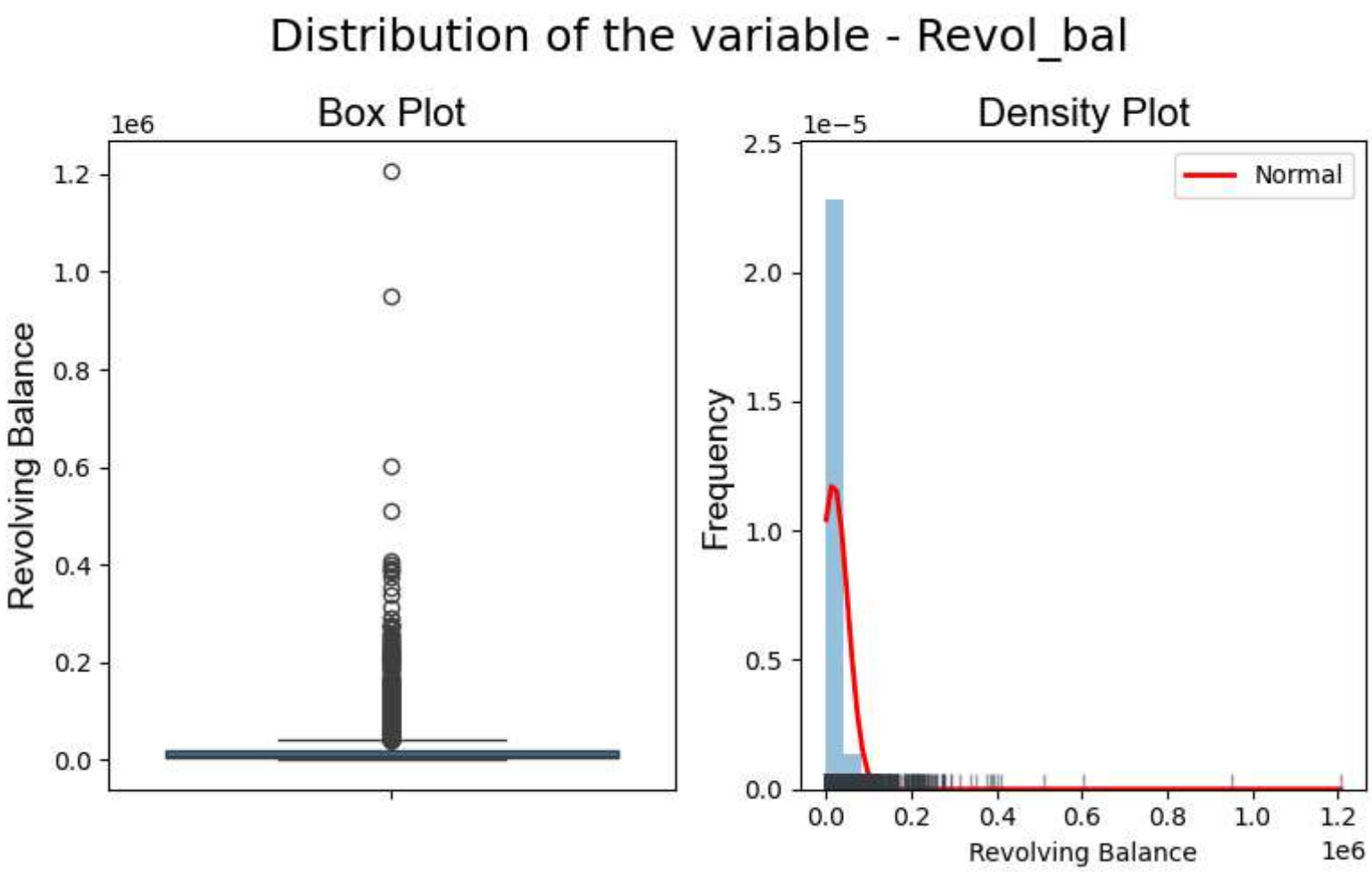
The distribution seems to be simetric, with a range between 612 and 827

In [ ]:
```
univariated_num_plot(num_df, var_name='days_with_cr_line', box_xlabel='Days with credit line')
```



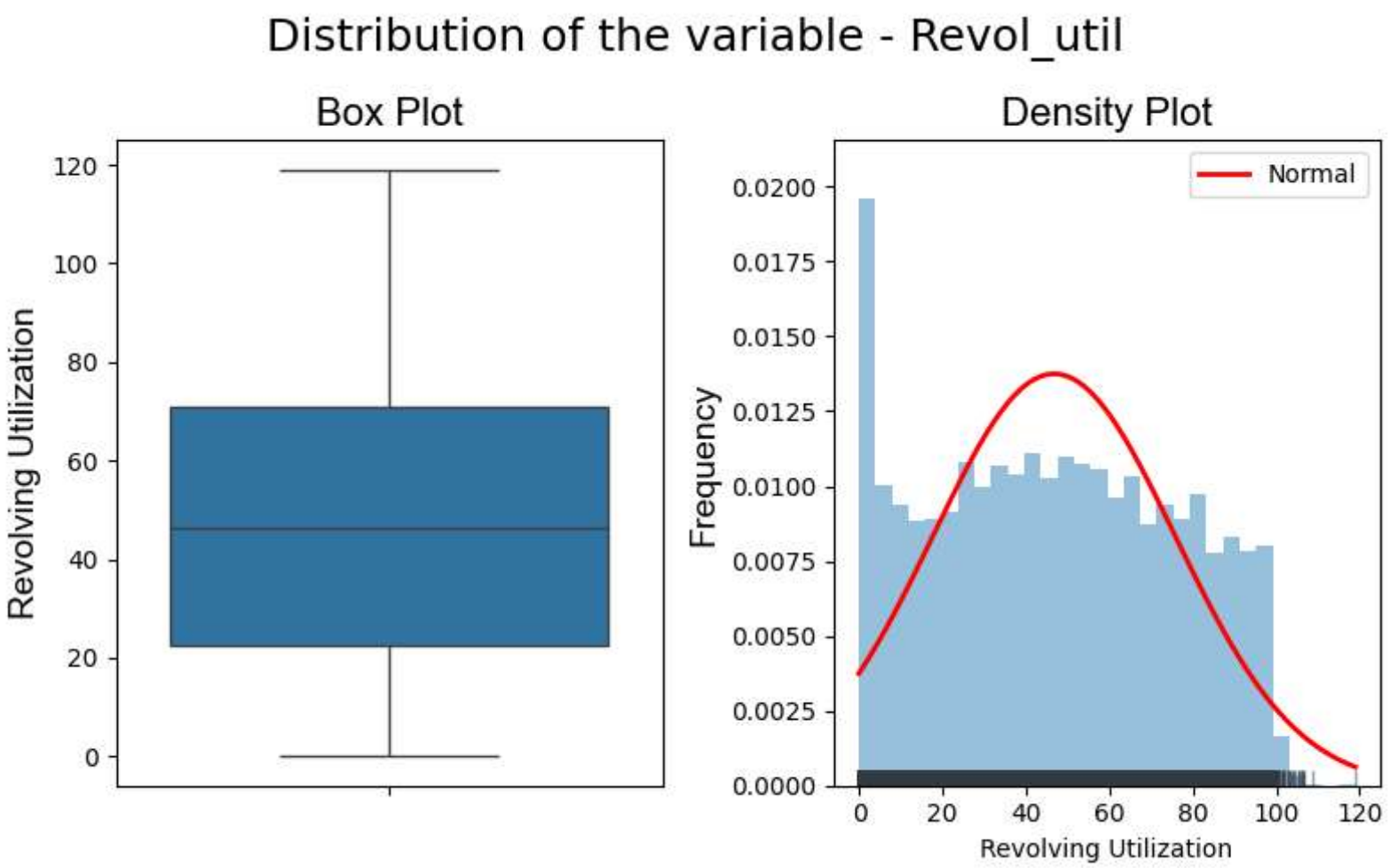
The days_with_cr_line variable has a right skewed distribution, so there are posible outlier to the upper tail. Most of customer have had credit line for 4139 days (10 year aprox) or les

In [ ]:
```
univariated_num_plot(num_df, var_name='revol_bal', box_xlabel='Revolving Balance')
```



The revol_val variable tells us the amount unpaid for the customer at the end of the credit card billing cycle. We can see the distribution is heavy-tailed to the right.

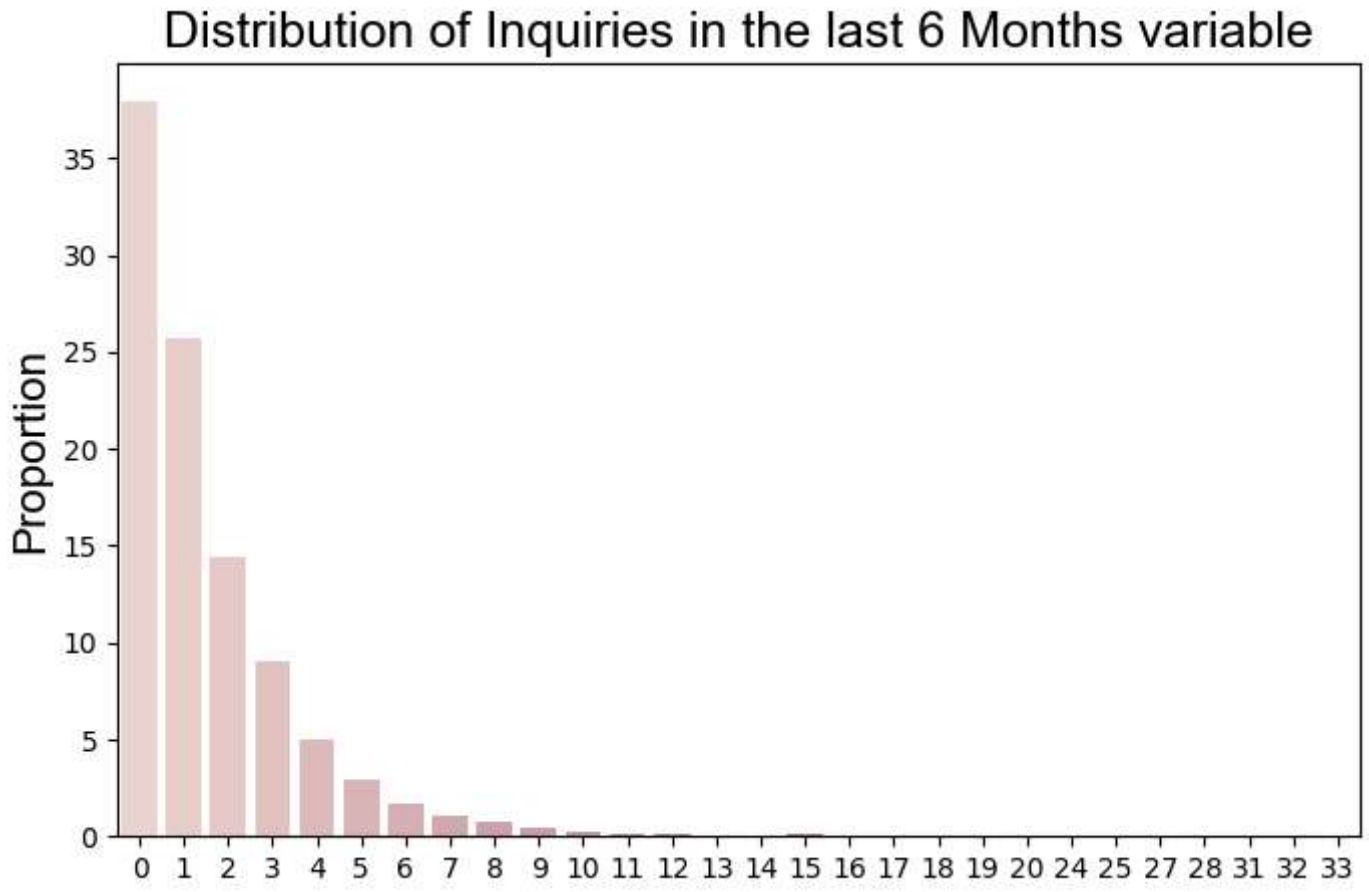
In [ ]:
```
univariated_num_plot(num_df, var_name='revol_util', box_xlabel='Revolving Utilization')
```



The revolving utilization rate varible seems to follow a uniform distribution. Apparently there are not outliers. However the distribution shows there are customers who have used more than the total credit available.

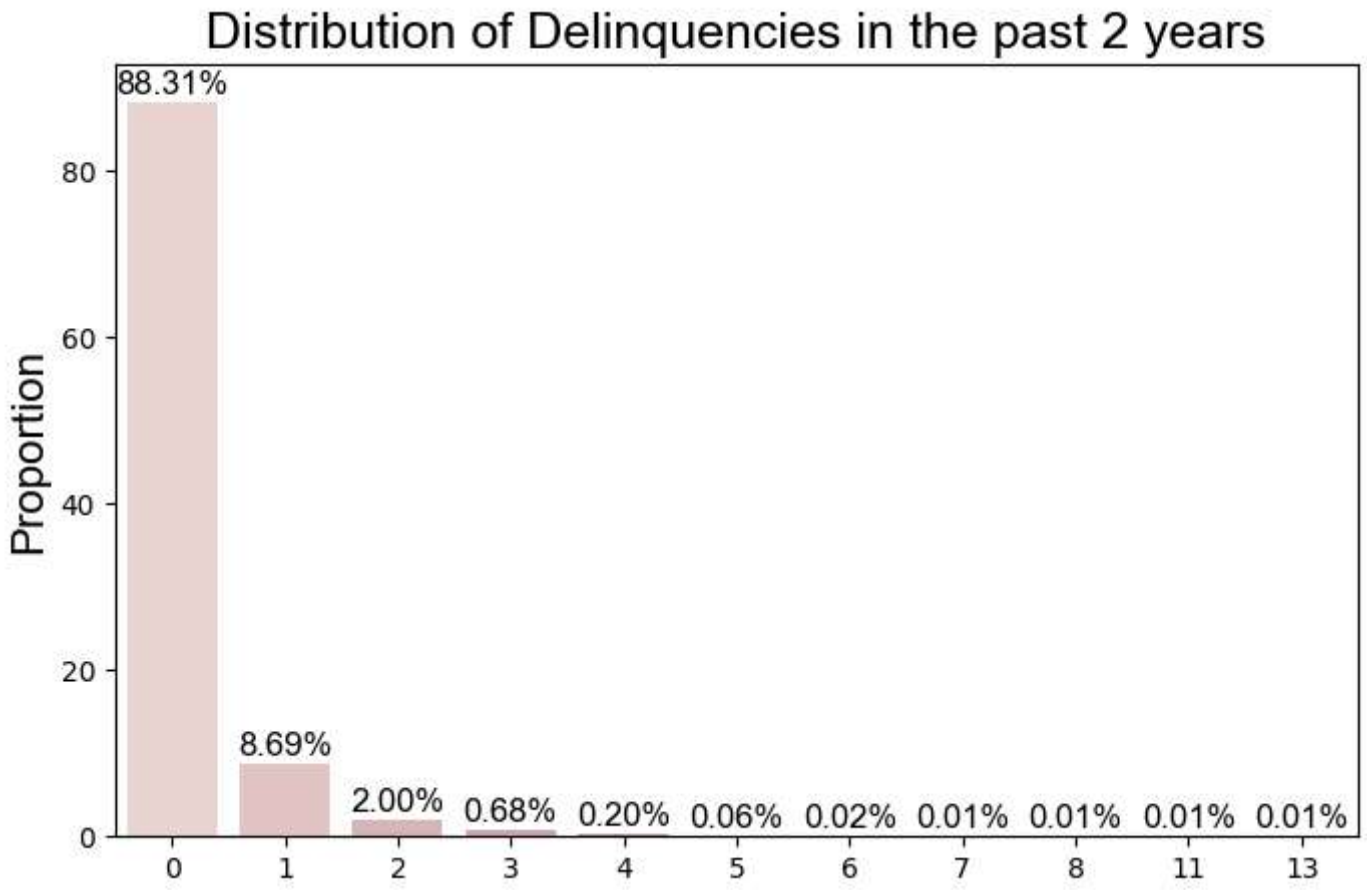
In [ ]:
```
# univariated_plot(num_df, var_name='inq_last_6mths', box_xlabel='Inquiries in last 6 months')
inquiries_6m = df.inq_last_6mths.value_counts(normalize=True).to_frame().reset_index()
inquiries_6m['proportion'] = round(inquiries_6m.proportion * 100, 2)

fig, ax = plt.subplots(figsize=(8, 5))
ax = sns.barplot(data=inquiries_6m, x='inq_last_6mths', y='proportion', hue='inq_last_6mths', legend=False)
ax.set_xlabel(None)
ax.set_title("Distribution of Inquiries in the last 6 Months variable",
```

```
            fontdict={'fontname': 'Arial', 'fontsize': 18})
# ax.set_xticklabels(['No', 'Yes'], fontdict={'fontname': 'Arial', 'fontsize': 16})
ax.set_ylabel("Proportion", fontdict={'fontname': 'Arial', 'fontsize': 16})

# for p in ax.patches:
#     height = p.get_height()
#     ax.text(
#         p.get_x() + p.get_width() / 2,
#         height + 1,
#         f'{height:.2f}%',
#         ha='center',
#         fontname='Arial',
#         fontsize=12
#     )

plt.show()
```



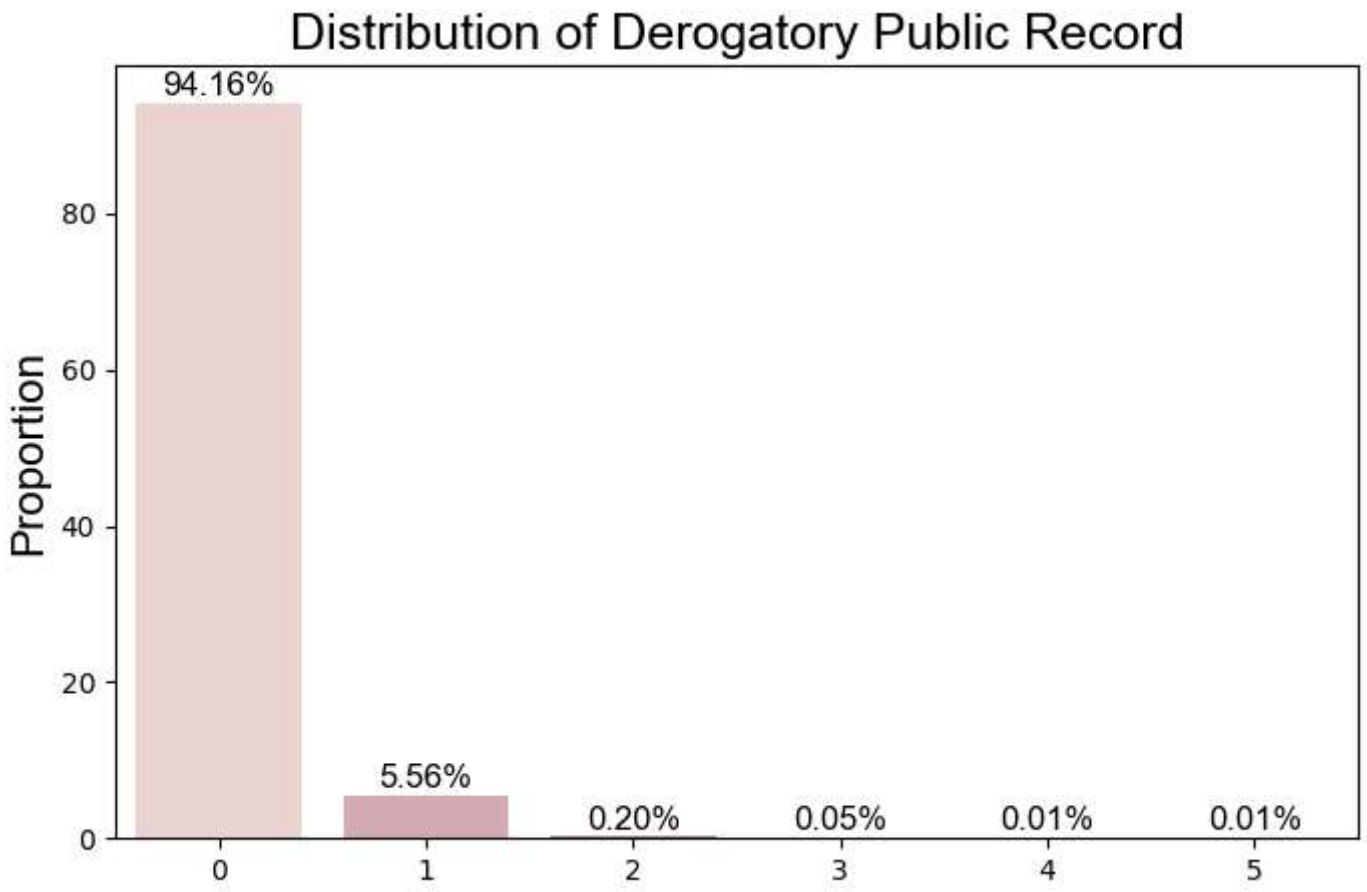
In [ ]:
```
# univariated_plot(num_df, var_name='delinq_2yrs', box_xlabel='# times 30+ past due in the past 2 years')
delinq_2y = df.delinq_2yrs.value_counts(normalize=True).to_frame().reset_index()
delinq_2y['proportion'] = round(delinq_2y.proportion * 100, 2)

fig, ax = plt.subplots(figsize=(8, 5))
ax = sns.barplot(data=delinq_2y, x='delinq_2yrs', y='proportion', hue='delinq_2yrs', legend=False)
ax.set_xlabel(None)
ax.set_title("Distribution of Delinquencies in the past 2 years",
             fontdict={'fontname': 'Arial', 'fontsize': 18})
# ax.set_xticklabels(['No', 'Yes'], fontdict={'fontname': 'Arial', 'fontsize': 16})
ax.set_ylabel("Proportion", fontdict={'fontname': 'Arial', 'fontsize': 16})

for p in ax.patches:
    height = p.get_height()
```

```
    ax.text(
        p.get_x() + p.get_width() / 2,
        height + 1,
        f'{height:.2f}%',
        ha='center',
        fontname='Arial',
        fontsize=12
    )

plt.show()
```



In [ ]:
```
# univariated_plot(num_df, var_name='pub_rec', box_xlabel='Derogatory Public Record')
pub_rec = df.pub_rec.value_counts(normalize=True).to_frame().reset_index()
pub_rec['proportion'] = round(pub_rec.proportion * 100, 2)
```

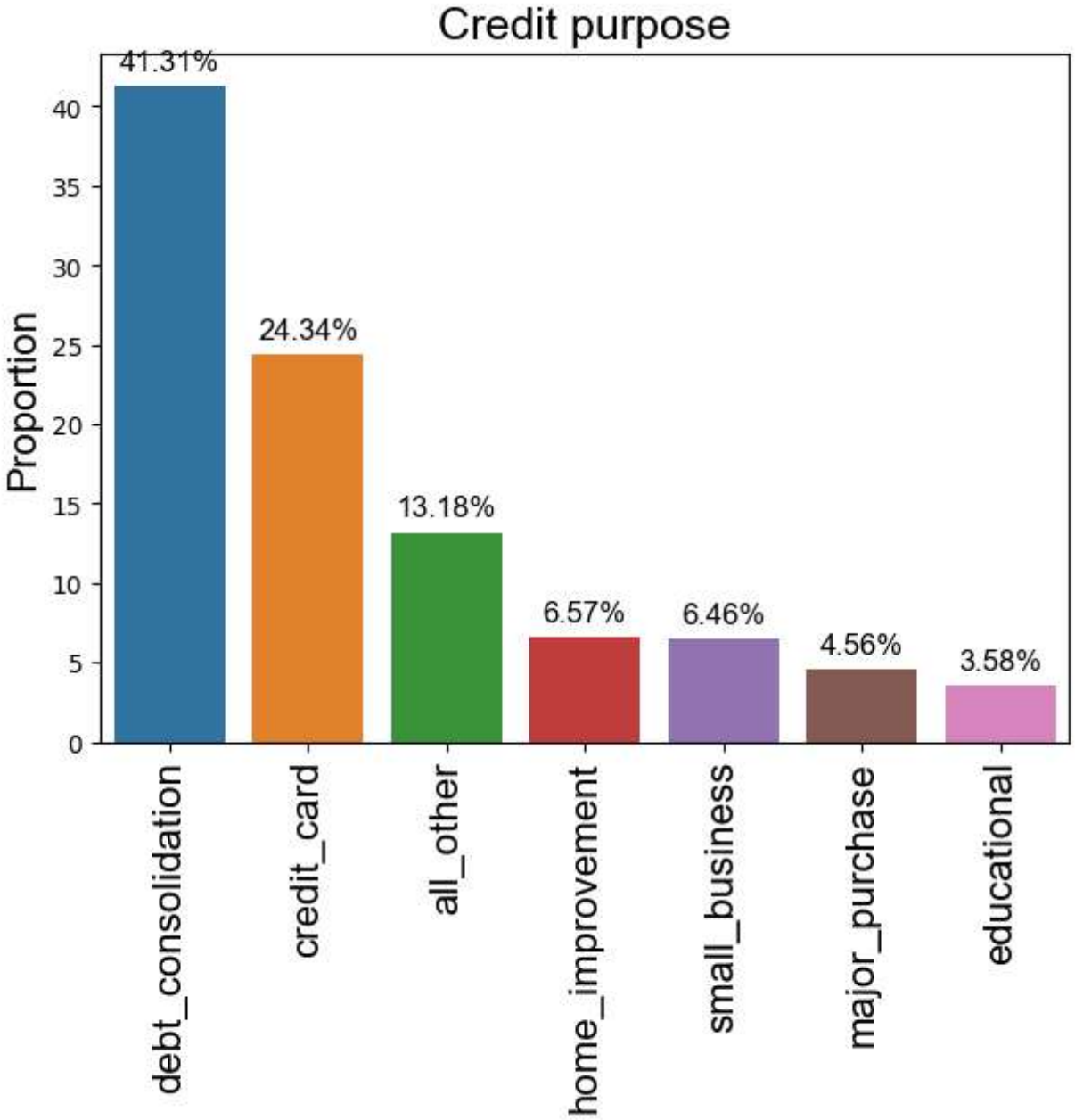
```
fig, ax = plt.subplots(figsize=(8, 5))
sns.barplot(data=pub_rec, x='pub_rec', y='proportion', hue='pub_rec', legend=False, ax=ax)
ax.set_xlabel(None)
ax.set_title("Distribution of Derogatory Public Record",
             fontdict={'fontname': 'Arial', 'fontsize': 18})
# ax.set_xticklabels(['No', 'Yes'], fontdict={'fontname': 'Arial', 'fontsize': 16})
ax.set_ylabel("Proportion", fontdict={'fontname': 'Arial', 'fontsize': 16})

for p in ax.patches:
    height = p.get_height()
    ax.text(
        p.get_x() + p.get_width() / 2,

        height + 1,
        f'{height:.2f}%',
        ha='center',
        fontname='Arial',
        fontsize=12
    )

plt.show()
```



Categorical Variables

In [ ]:
```
cred_purpose = df.purpose.value_counts(normalize=True).to_frame().reset_index()
cred_purpose['proportion'] = round(cred_purpose.proportion * 100, 2)

fig, ax = plt.subplots(figsize=(7, 5))
ax = sns.barplot(data=cred_purpose, x='purpose', y='proportion', hue='purpose', legend=False)
ax.set_xlabel(None)
ax.set_title("Credit purpose",
             fontdict={'fontname': 'Arial', 'fontsize': 18})
ax.set_xticks(range(len(cat_df.purpose.unique())))
ax.set_xticklabels(cat_df.purpose.unique(),
                   fontdict={'fontname': 'Arial', 'fontsize': 16},
```

```
                    rotation=90)
ax.set_ylabel("Proportion", fontdict={'fontname': 'Arial', 'fontsize': 16})

for p in ax.patches:
    height = p.get_height()
    ax.text(
        p.get_x() + p.get_width() / 2,

        height + 1,
        f'{height:.2f}%',
        ha='center',
        fontname='Arial',
        fontsize=12
    )

plt.show()
```



From the plot we can tell that the paretto for the purpose of the credit is for debt consolidation, credit card and some other purposes.

Bivariate Analysis

From the univariate analysis we saw that the base defualt rate is 16%.

In this section a bivariate analysis will be perfomed in order to identify which of the predictor variables can help us detecting default.

For plotting the bivariate plots, the function "bivariated_num_default_plot" is created

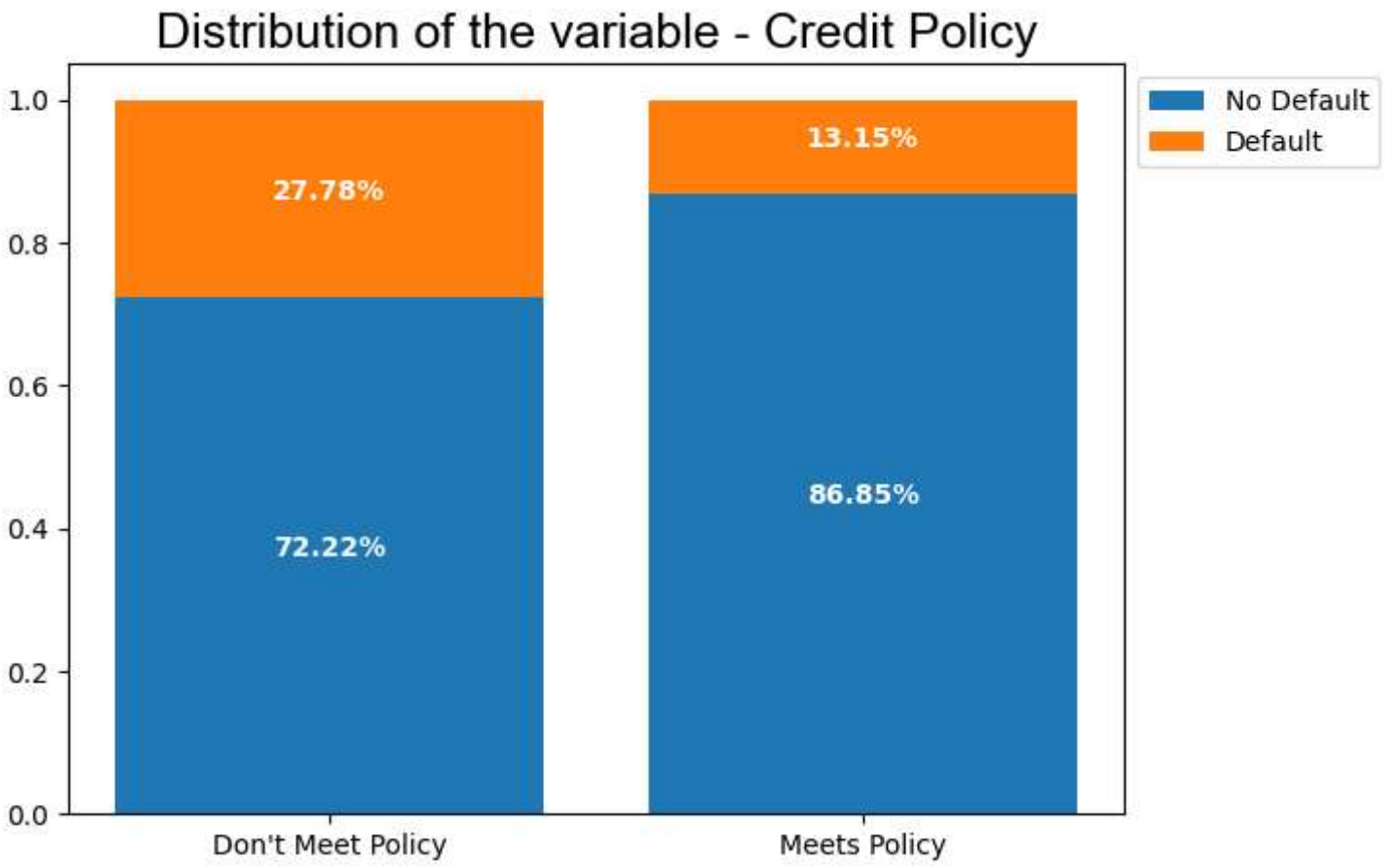
In [ ]:
```
def bivariated_num_default_plot(df,
                                var_name,
                                box_xlabel,
                                figheight=5,
                                figwidth=10,
                                fontname='Arial',
                                fig_fontsize=18,
                                title_fontsize=16,
                                ylabel_fontsize=14,
                                xtick_fontsize=12,
                                ytick_fontsize=14):
    """Function to create a bivariate plot for numerical variables against
    the target variable (Default)
    """
    fig, axes = plt.subplots(1, 2, figsize=(figwidth, figheight))
    fig.suptitle(f"Distribution of the variable - {var_name.capitalize()}", fontsize=fig_fontsize)
    sns.boxplot(data=df, y=var_name, x='default', hue='default', ax=axes[0])
    sns.kdeplot(data=df, x=var_name, hue='default', fill=True, ax=axes[1])
    axes[0].set_title('Box Plot', fontdict={'fontname': fontname, 'size': title_fontsize})
    axes[0].set_ylabel(box_xlabel, fontdict={'fontname': fontname,'size': ylabel_fontsize})
    axes[0].set_xlabel(None)
    axes[0].set_xticks(range(2))
    axes[0].set_xticklabels(['No Default', 'Default'],
                            fontdict={'fontname': fontname,
                                      'fontsize': xtick_fontsize})
    axes[0].get_legend().set_visible(False)
    axes[1].set_title('Density Plot', fontdict={'fontname': fontname,
                                                'size': title_fontsize})
    axes[1].set_ylabel("Density", fontdict={'fontname': fontname,
                                            'size': ylabel_fontsize})
```

```
    axes[1].set_xlabel(None)
    axes[1].legend(['Default', 'No Default'], bbox_to_anchor=(1, 1))
    plt.tight_layout()
    plt.show()
```

Credit Policy vs Default

In [ ]:

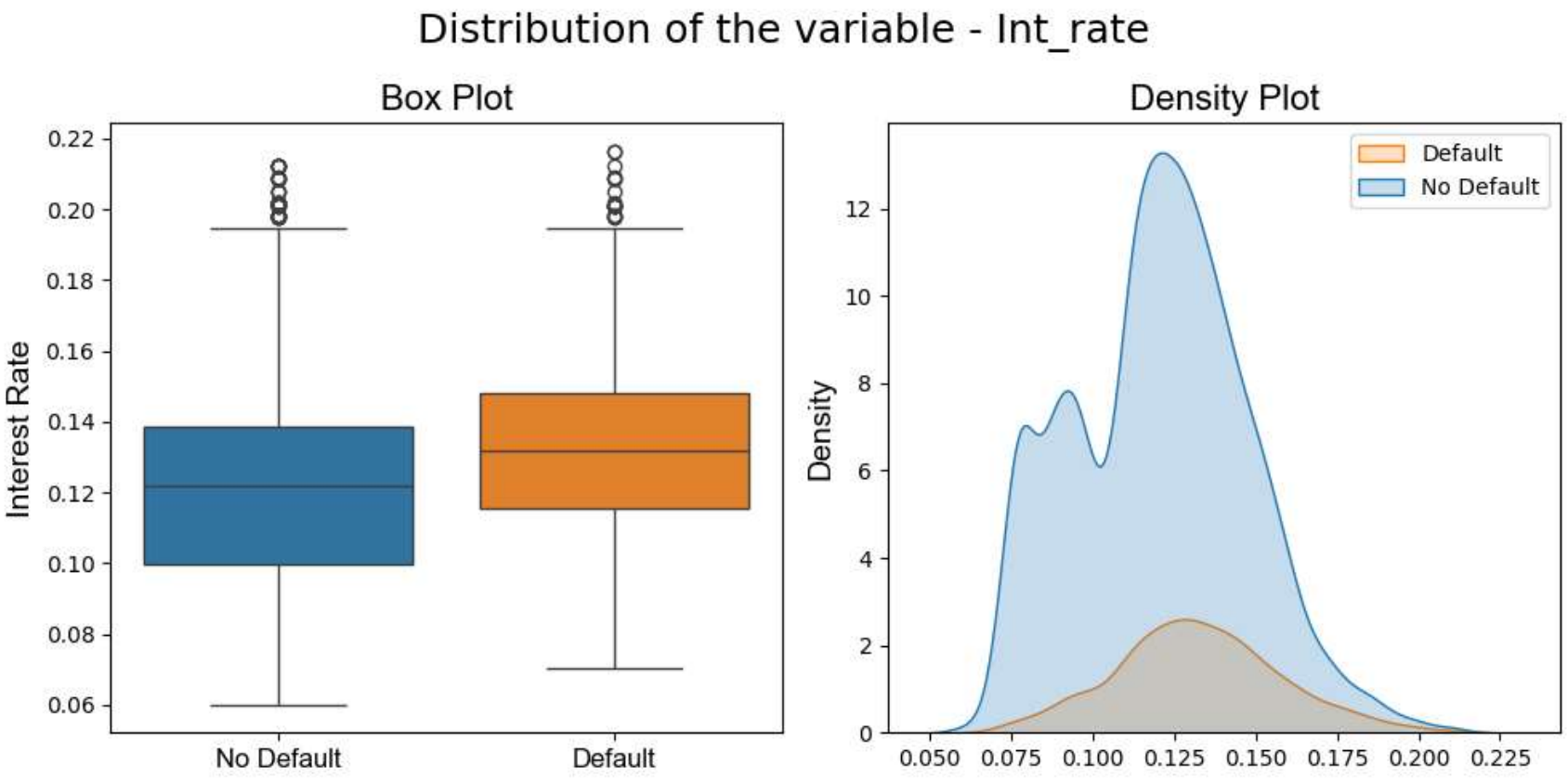
```
credit_policy = df.groupby('credit_policy')['default'].value_counts(normalize=True).unstack()
credit_policy.columns = ['no_default', 'default']
credit_policy = credit_policy.reset_index()

groups = ["Don't Meet Policy", 'Meets Policy']
no_default = credit_policy.no_default.tolist()
default = credit_policy.default.tolist()

fig, ax = plt.subplots(figsize=(7, 5))
ax.bar(groups, no_default, label = "No Default")
ax.bar(groups, default, bottom=no_default, label="Default")
ax.set_title("Distribution of the variable - Credit Policy",
             fontdict={'fontname': 'Arial', 'fontsize':18})
ax.legend(bbox_to_anchor=(1, 1))
# Labels
for bar in ax.patches:
    ax.text(
        bar.get_x() + bar.get_width() / 2,
        bar.get_y() + bar.get_height() / 2,
        f'{round(bar.get_height() * 100, 2)}%',
        ha='center',
        color='w',
        weight='bold'
    )

plt.show()
```



From this plot we can see that customer who meets the Lendin Club credit policy tends to default lesser than the ones who doesn't meet the policy

Interest Rate vs Default

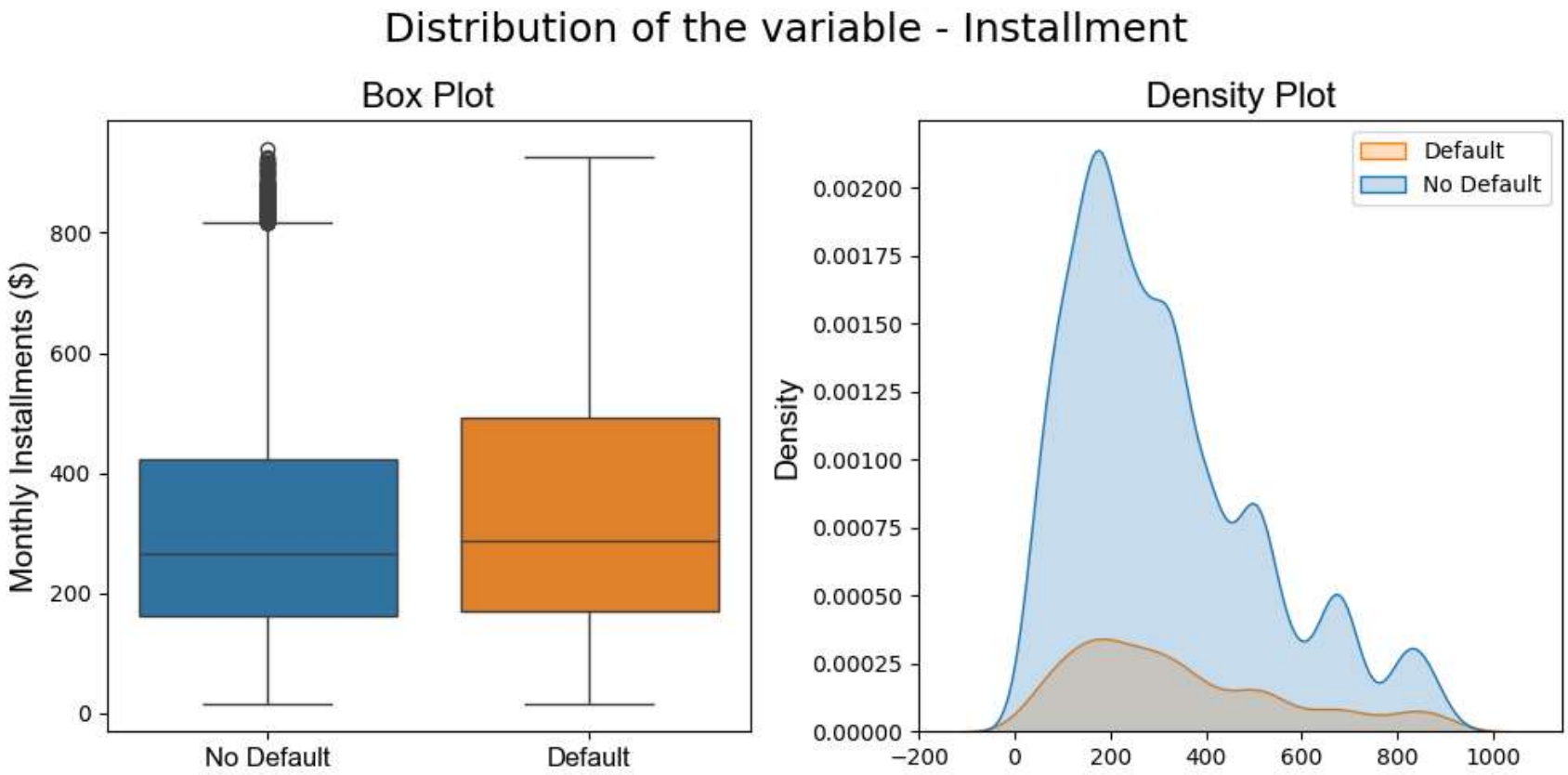
In [ ]:
```
bivariated_num_default_plot(df, var_name='int_rate', box_xlabel='Interest Rate')
```



Customers with higher interest rate tends to default the more

Installment vs Default

In [ ]:
```
bivariated_num_default_plot(df, var_name='installment', box_xlabel='Monthly Installments ($)')
```



At glance it seems that there is not difference in the installments for default and non default customers. So we perform a hypotesis testig to evaluate if the means difference are statiscally significant.

In [ ]:
```
# Hypotesis testing for installment
pg.ttest(x=df.query("default == 1").installment,
         y=df.query("default == 0").installment,
         correction=True,
         alternative='greater'
         )
```

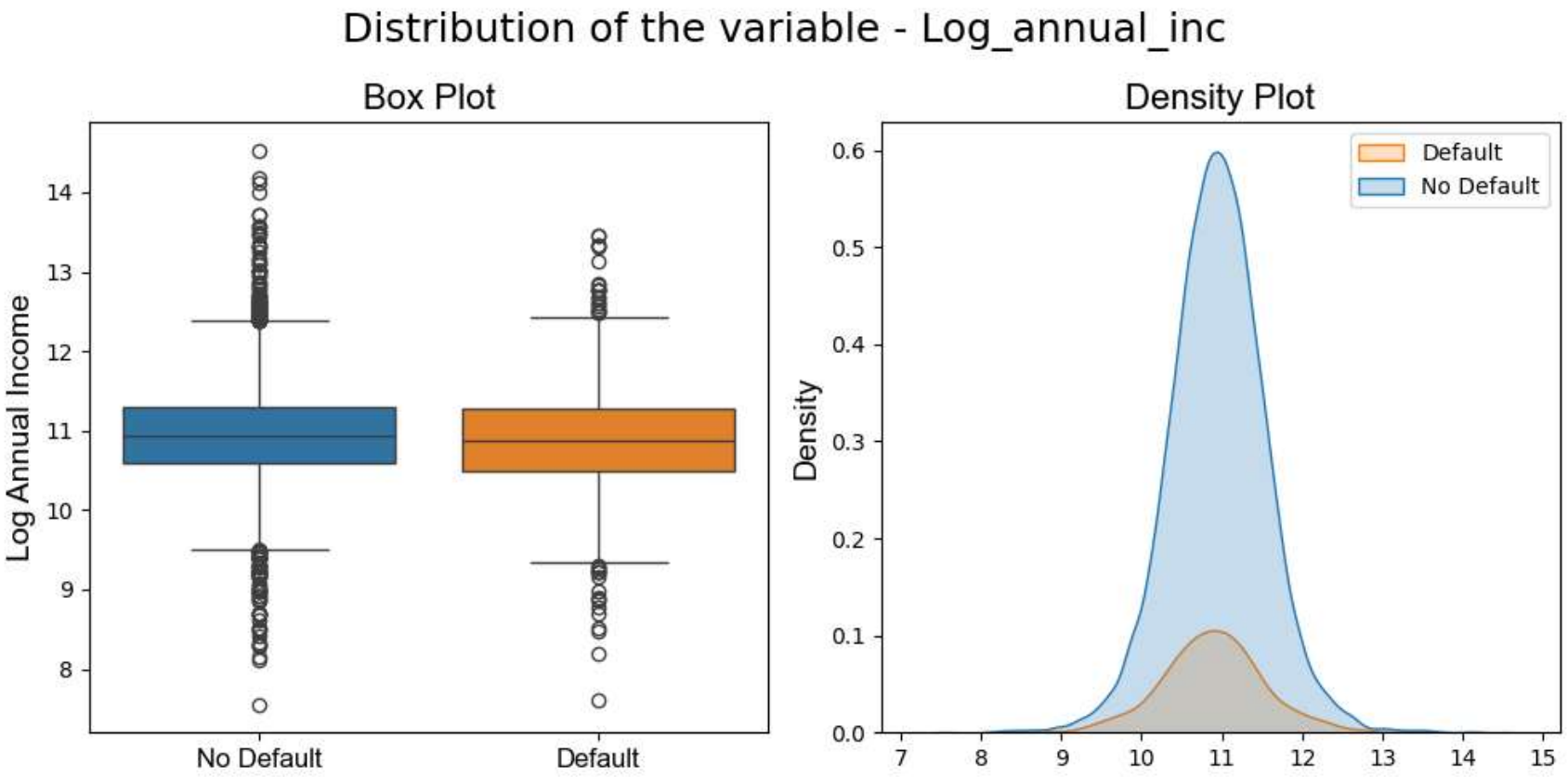
Out[ ]:

	T	dof	alternative	p-val	CI95%	cohen-d	BF10	power
T-test	4.585001	2041.864143	greater	0.000002	[18.09, inf]	0.136401	2210.645	0.999422

According the the test, there is a significant difference between the two means. So we can think, that defualt customers have a higher average installment

Log Annual Income vs Default

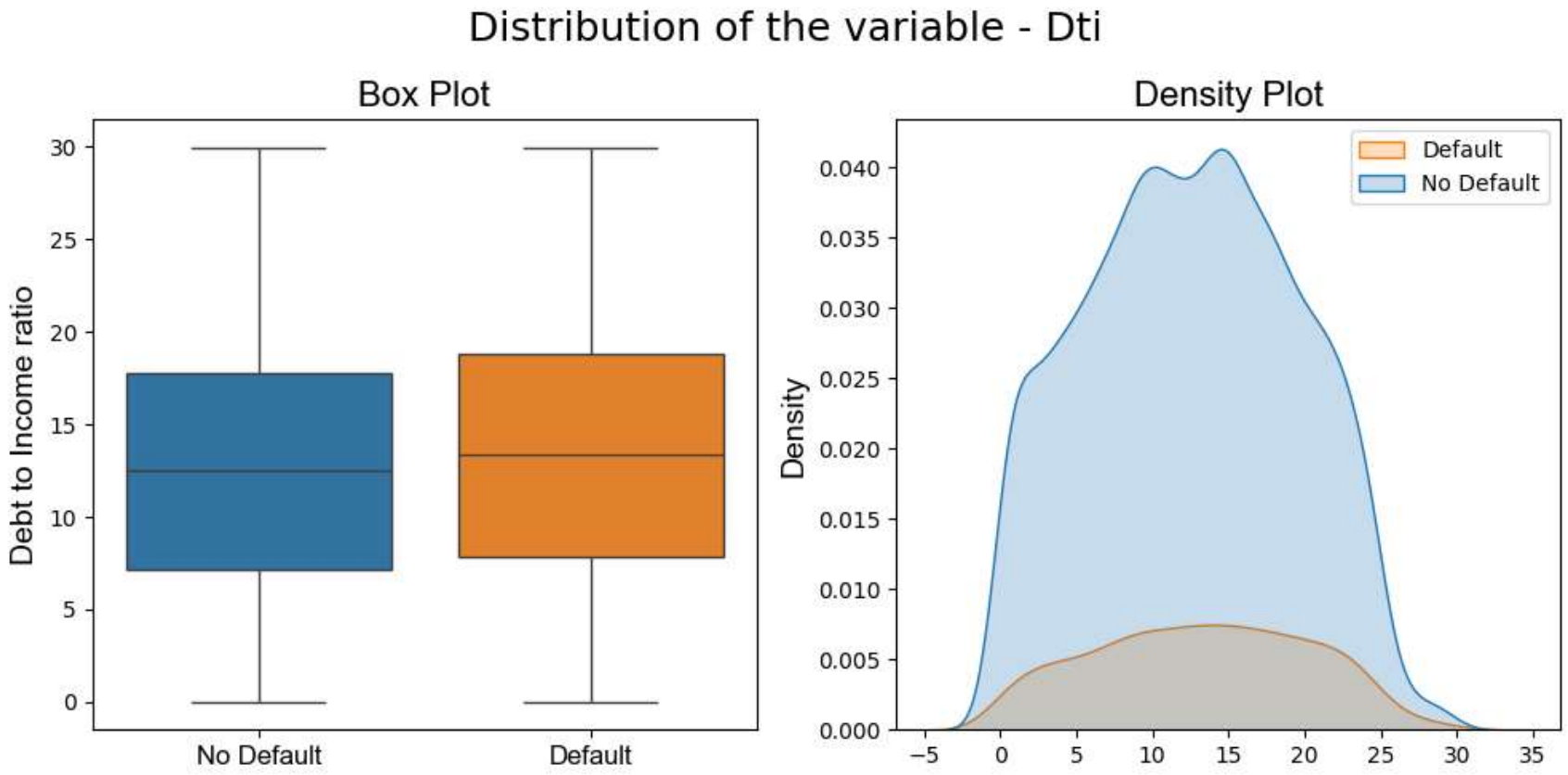
In [ ]:
```
bivariated_num_default_plot(df, var_name='log_annual_inc', box_xlabel='Log Annual Income')
```



In terms of the income, it seems there is no differences in the distributions

Debt to Income vs Default

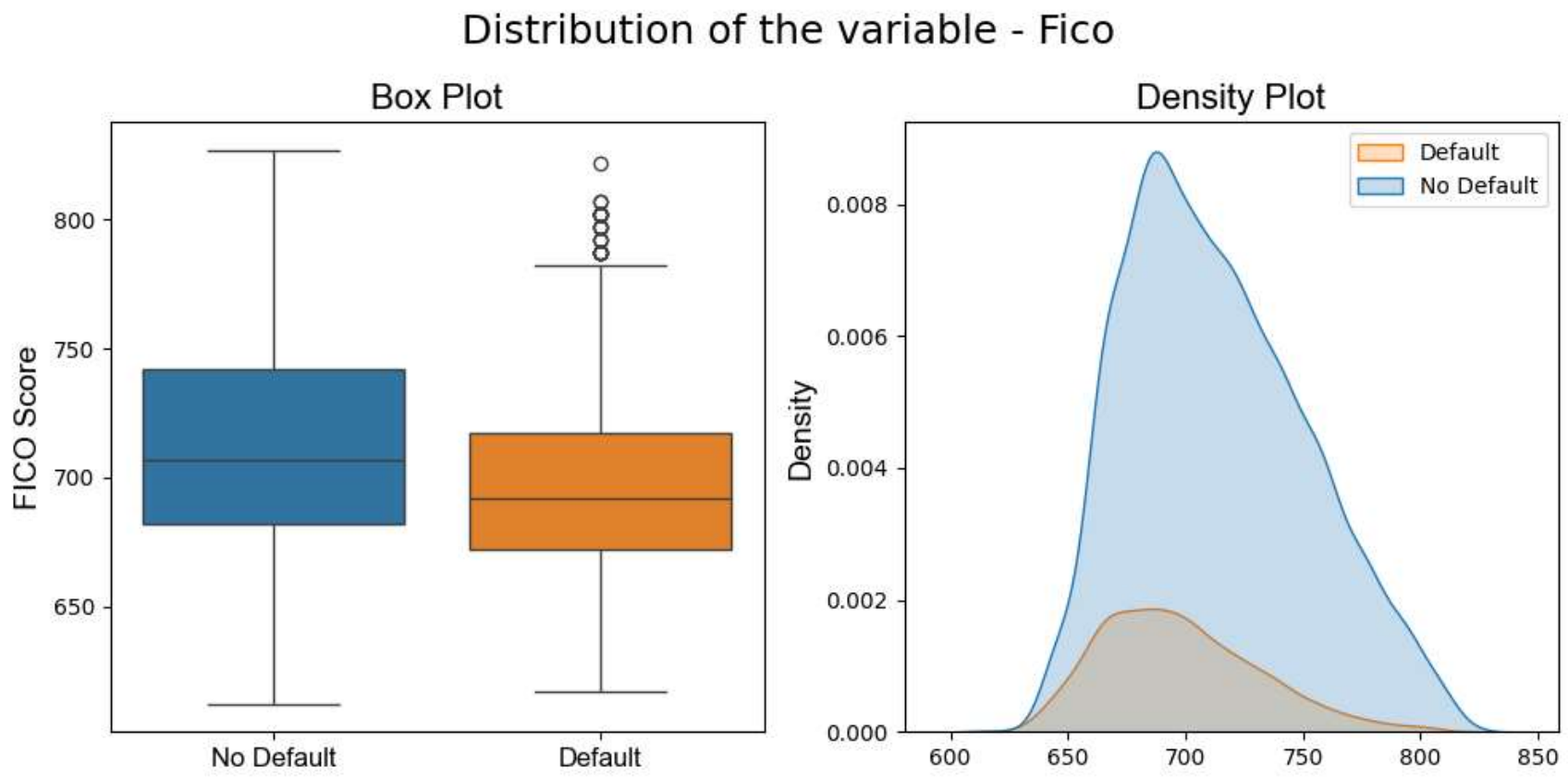
In [ ]:
```
bivariated_num_default_plot(df, var_name='dti', box_xlabel='Debt to Income ratio')
```



From the plot we can tell that customer who defualt tends to have a greater debt to income ratio. It means, that customer who defualts has higher amount of debt compare to their incomes

FICO Score vs Default

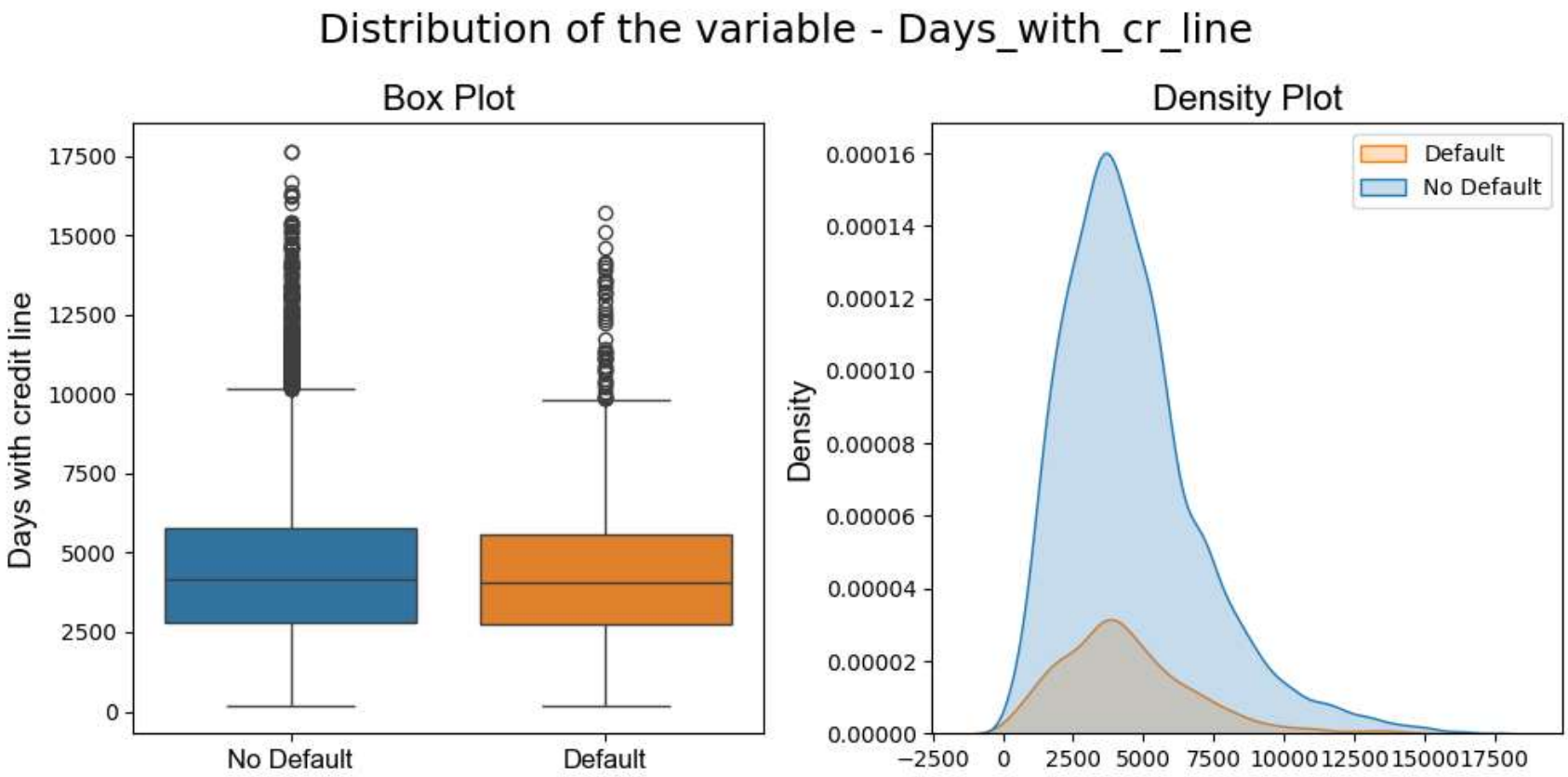
In [ ]:
```
bivariated_num_default_plot(df, var_name='fico', box_xlabel='FICO Score')
```



Customers with lower FICO scores defaults the more

Days with Credit Line vs Default

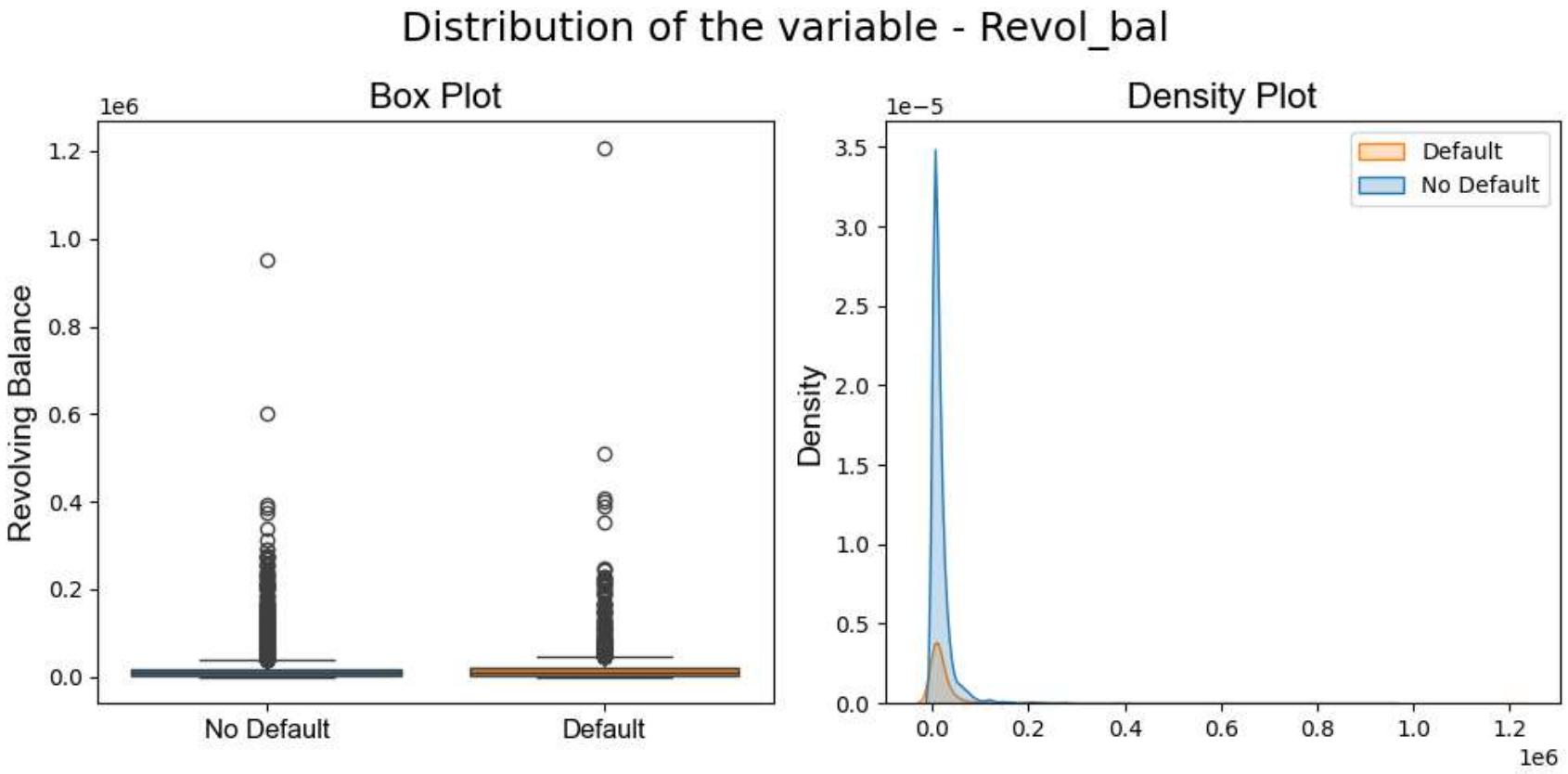
In [ ]:
```
bivariated_num_default_plot(df, var_name='days_with_cr_line', box_xlabel='Days with credit line')
```



There is not differences in the distributions for the variable Days with credit line

Revolving Balance vs Default

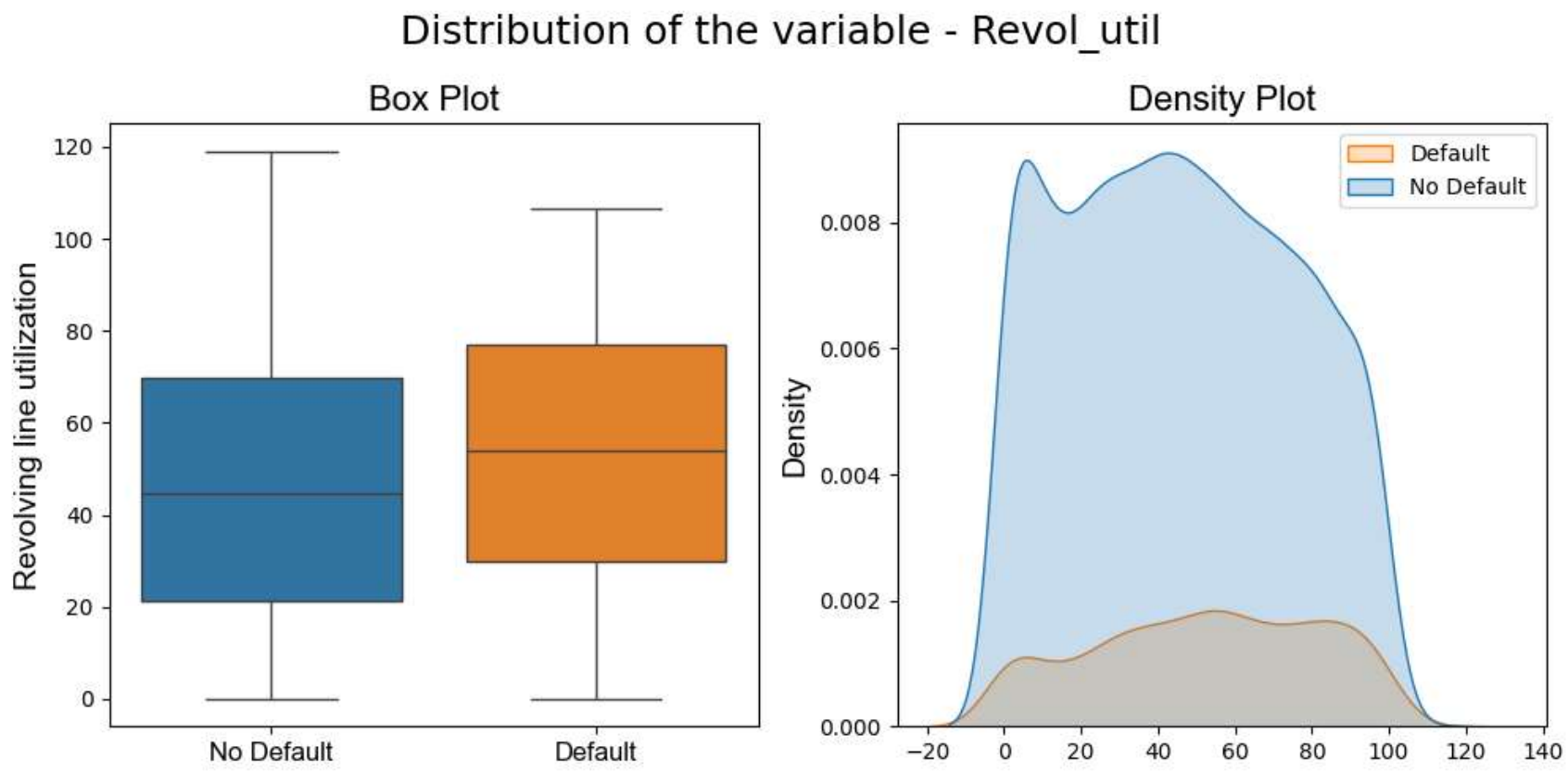
In [ ]:
```
bivariated_num_default_plot(df, var_name='revol_bal', box_xlabel='Revolving Balance')
```



The revolving balance dont seem to have a significant difference

Revolving Line Utilization vs Default

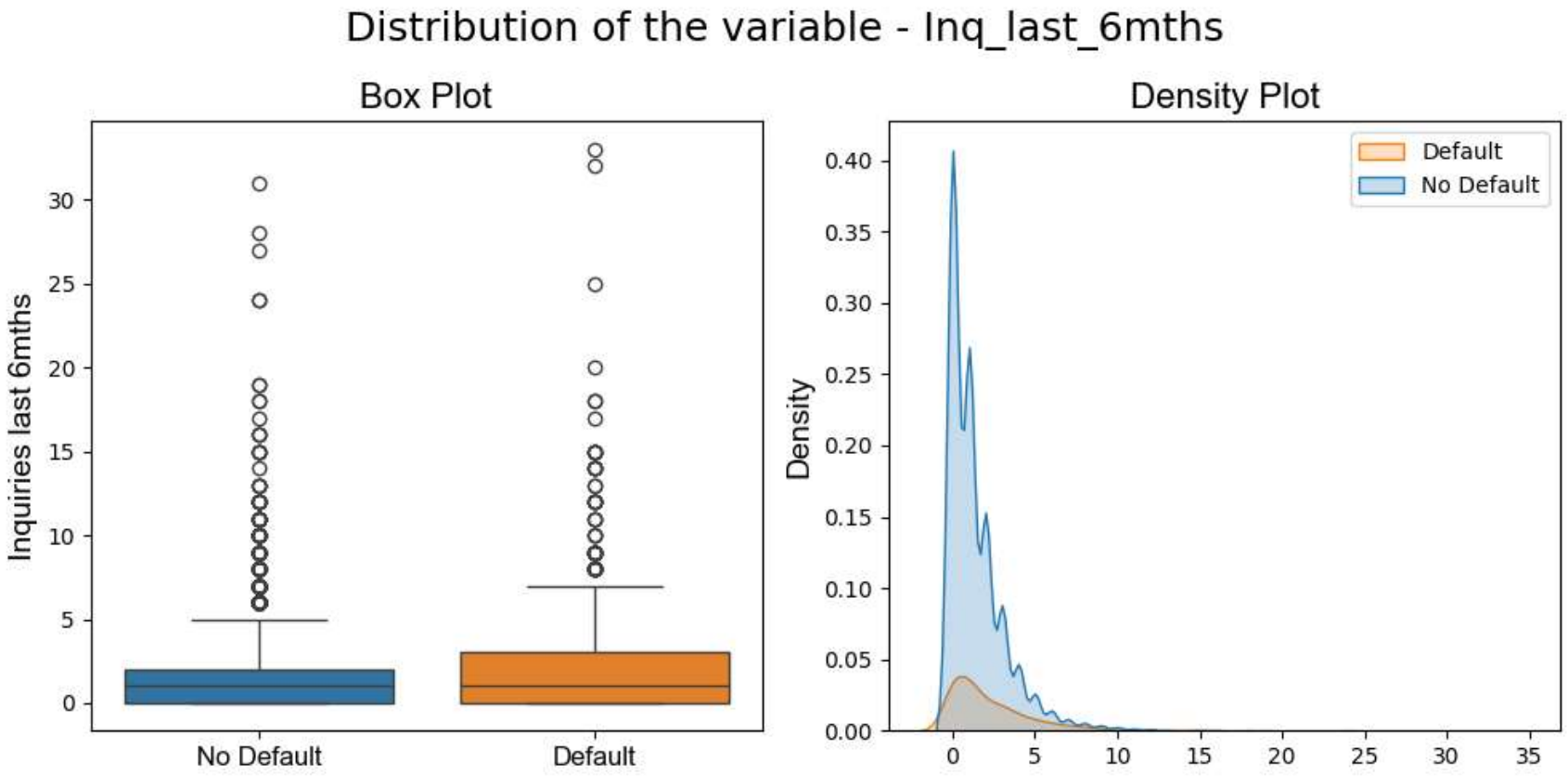
In [ ]:
```
bivariated_num_default_plot(df, var_name='revol_util', box_xlabel='Revolving line utilization')
```



Customers with higher levels of line utilization default the more

Inquiries Last 6mths vs Default

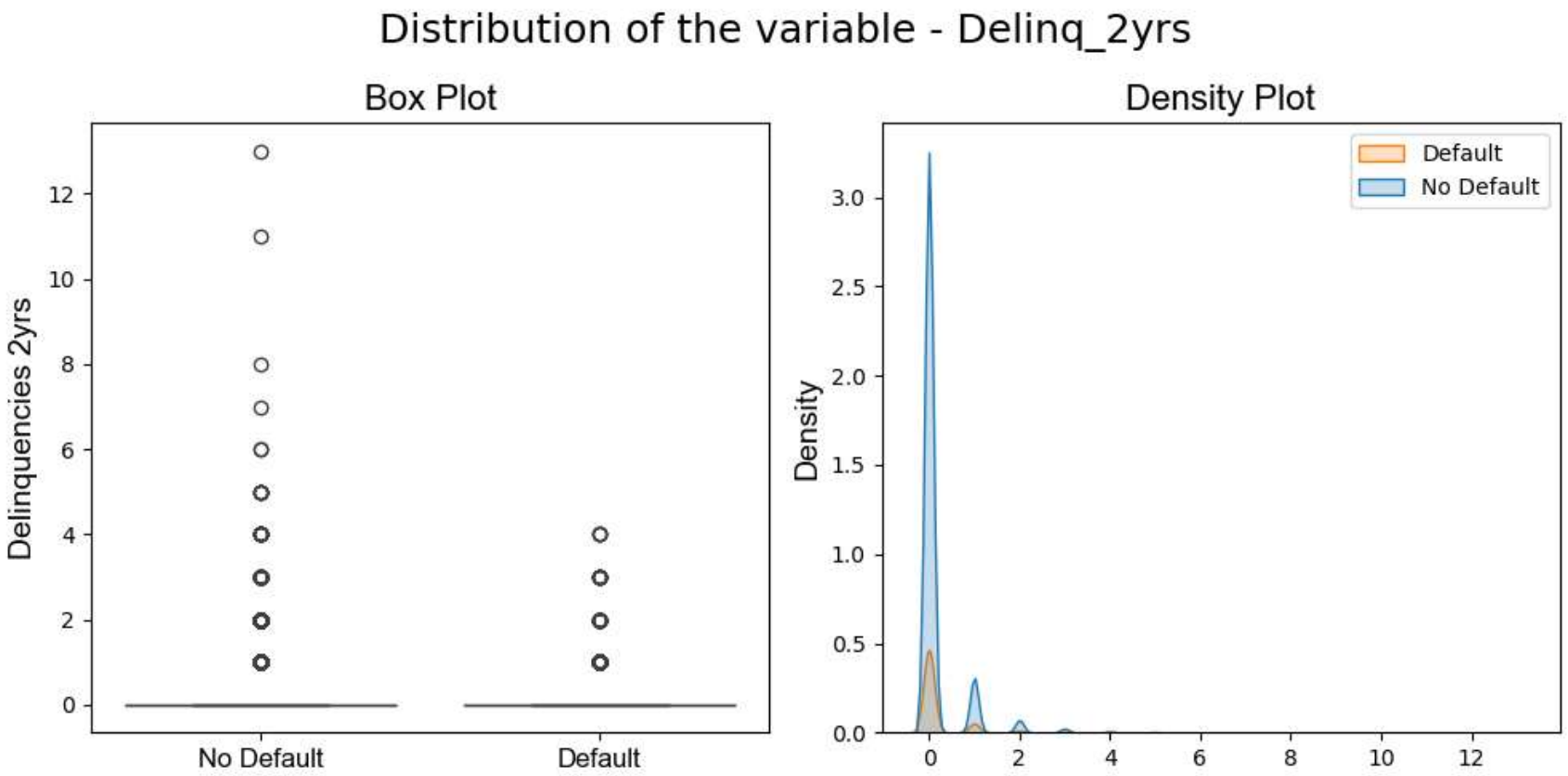
In [ ]:
```
bivariated_num_default_plot(df, var_name='inq_last_6mths', box_xlabel='Inquiries last 6mths')
```



Customers who defaulted have more inquiries in the last 6 months

Delinquencies in the past 2 years vs Default

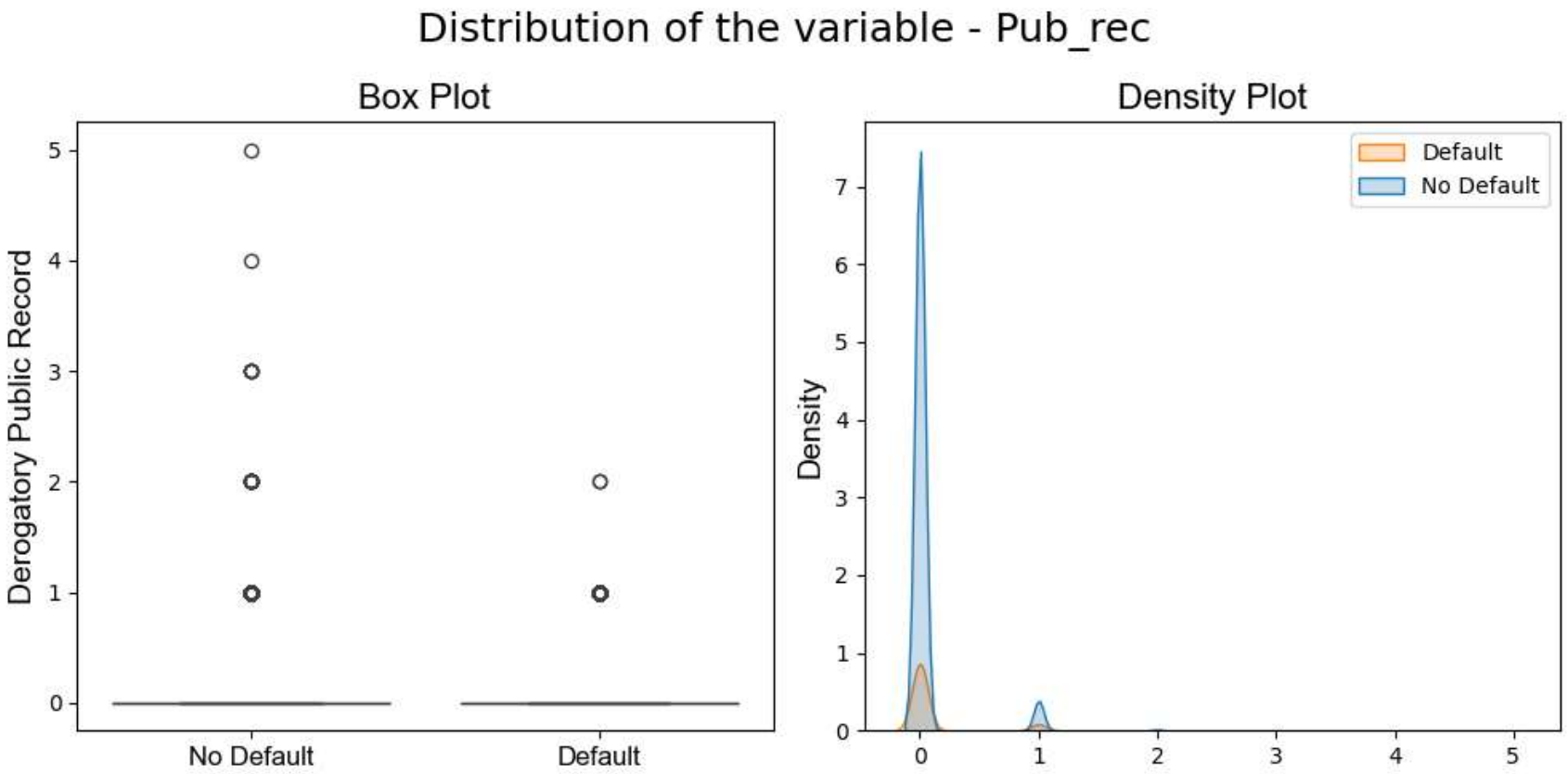
In [ ]:
```
bivariated_num_default_plot(df, var_name='delinq_2yrs', box_xlabel='Delinquencies 2yrs')
```



It does not seem to be a difference in the distributios of the variable delinquencies in the past 2 years since most of the customers have 0 delinquencies

Derogatory Public Records vs Default

In [ ]:
```
bivariated_num_default_plot(df, var_name='pub_rec', box_xlabel='Derogatory Public Record')
```

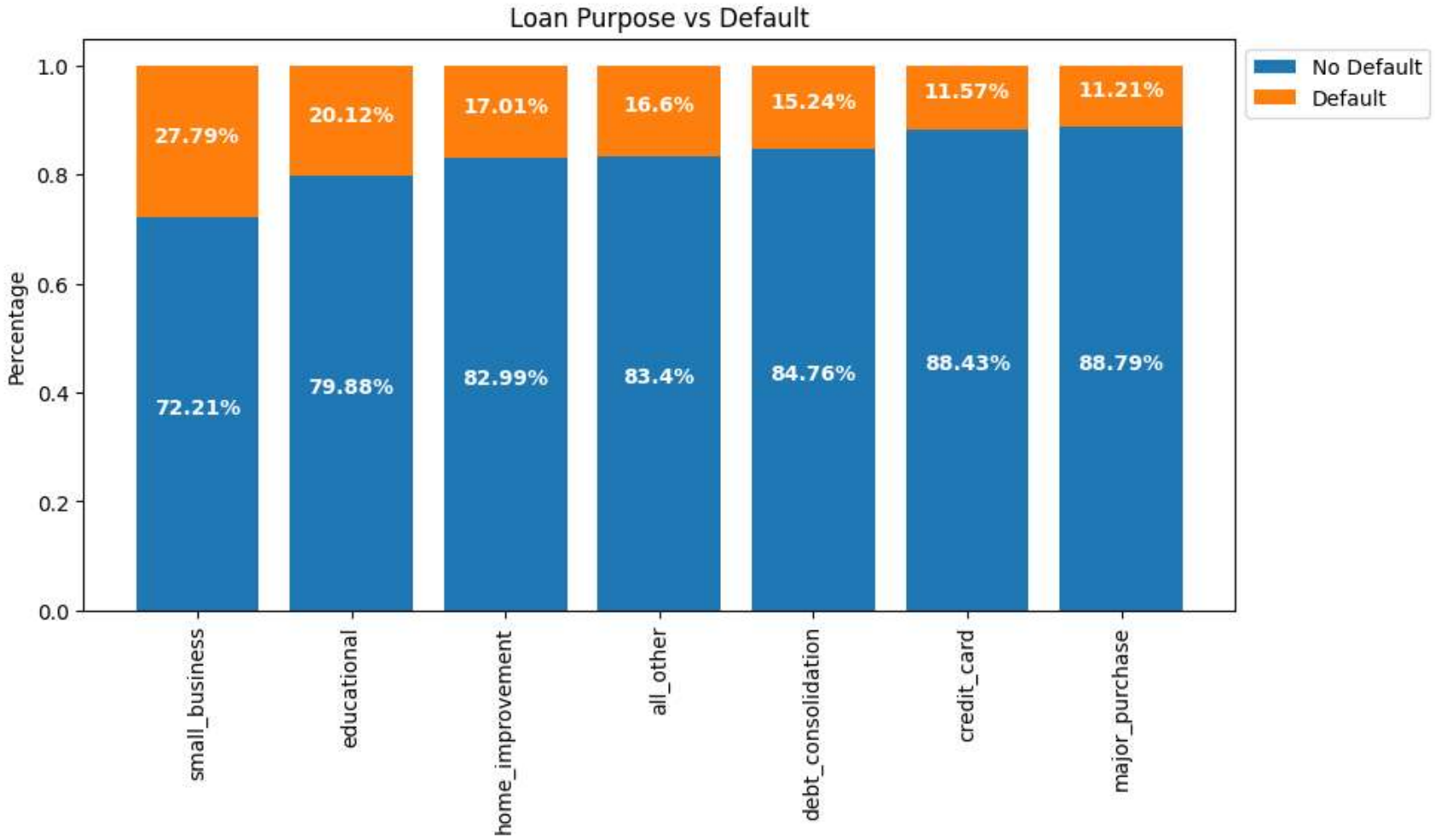


It does not seem to be a difference in the distributios of the variable derogatory public records, since most of the customers have a value of 0

Loan Purpose vs Default

In [ ]:
```
purpose = df.groupby(by='purpose')['default'].value_counts(normalize=True)
purpose = purpose.unstack()
purpose = purpose.reset_index()
purpose = purpose.rename(columns={0: 'no_default', 1: 'default'})
purpose = purpose.sort_values(by='default', ascending=False)

groups = purpose.purpose.tolist()
no_default = purpose.no_default.tolist()
default = purpose.default.tolist()
```

```
fig, ax = plt.subplots(figsize=(10, 5))
ax.bar(groups, no_default, label='No Default')
ax.bar(groups, default, bottom=no_default, label='Default')
ax.set(
    title="Loan Purpose vs Default",
    ylabel='Percentage'
)
ax.set_xticks(range(len(groups)))
ax.set_xticklabels(rotation=90, labels=groups)
# Labels
for bar in ax.patches:
    ax.text(
        bar.get_x() + bar.get_width() / 2,
        bar.get_y() + bar.get_height() / 2,
        f'{round(bar.get_height() * 100, 2)}%',
        ha = 'center',
        color = 'w',
        weight = 'bold',
        size = 10
    )
ax.legend(bbox_to_anchor=(1, 1))
plt.show()
```



small business, educational and home improvement are the purposes with the higest levels of default

Correlation Analysis

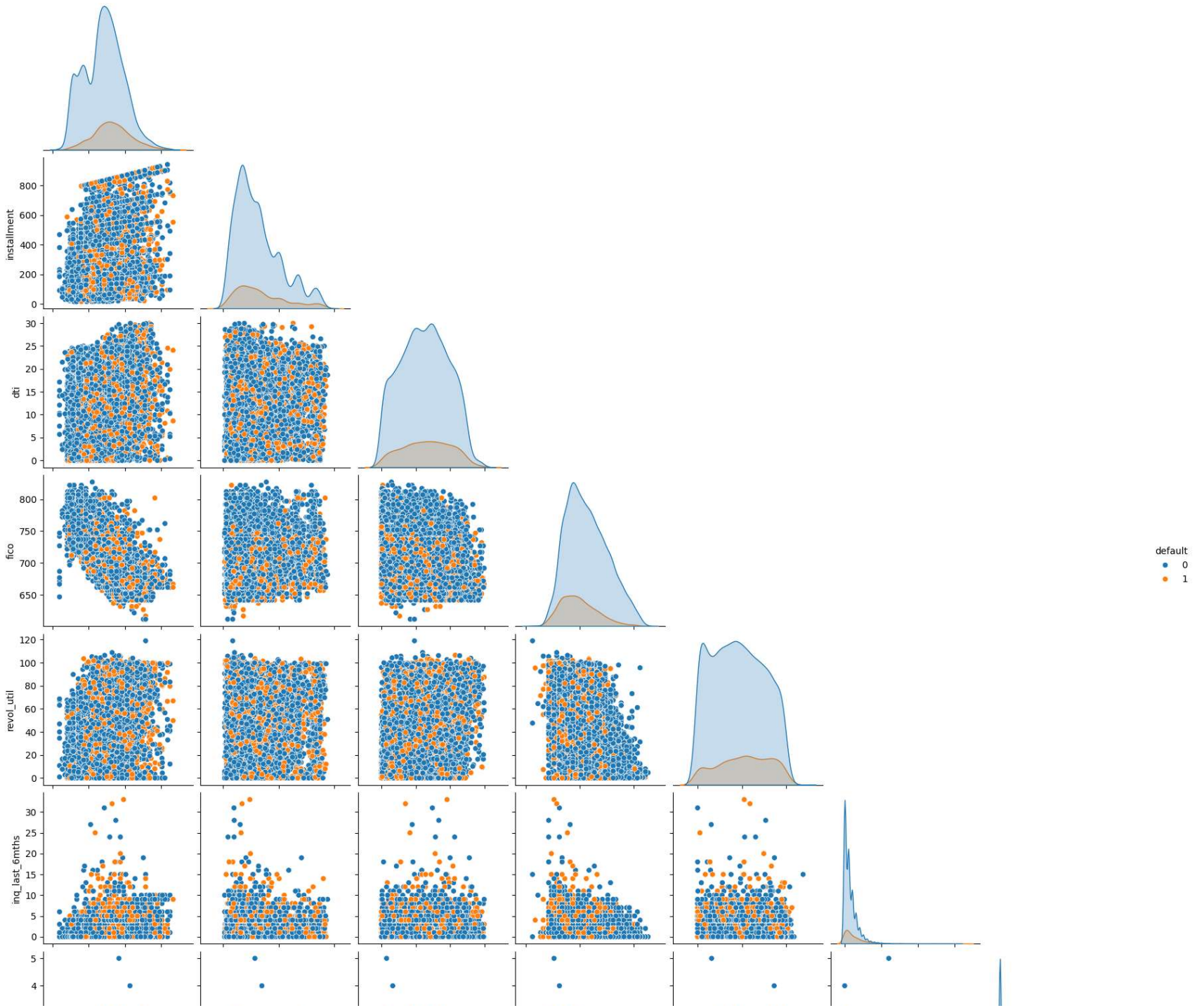
From the exploratory data anaylisis we can see that some predictors which could help us to identify default customers are:

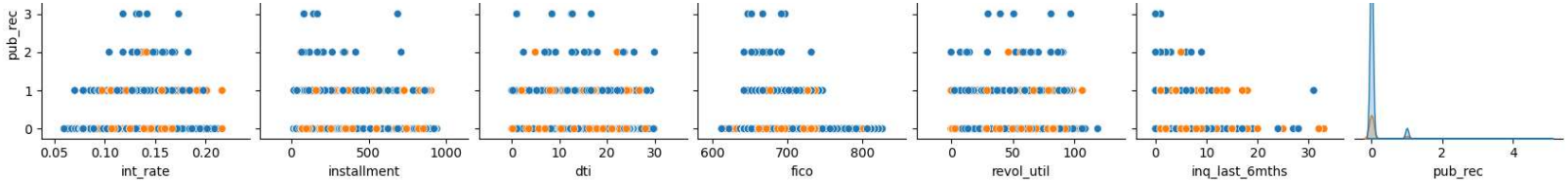
- credit_policy
- int_rate
- installment

- dti
- fico
- revol_util
- inq_last_6mths
- purpose

Next, we evaluate if some of this variable are correlated each other

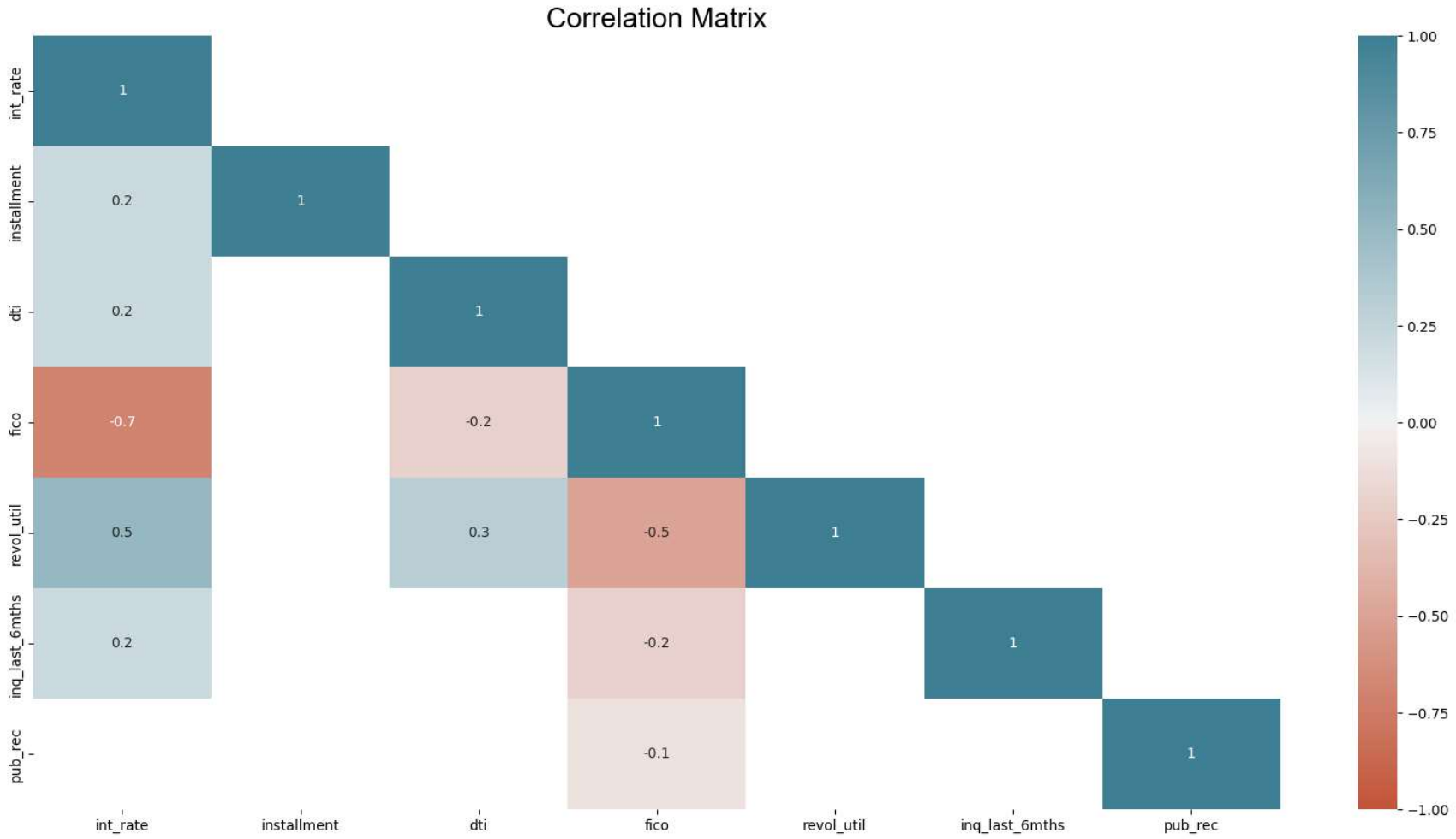
In [ ]:
```
num_vars = df.drop(['purpose', 'log_annual_inc', 'days_with_cr_line', 'revol_bal', 'delinq_2yrs', 'credit_policy'], a
sns.pairplot(data=num_vars, hue='default', corner=True)
plt.show()
```


installment
dti
fico
revol_util
inq_last_6mths
default
0
1



In [ ]:

```
# Correlation Matrix
num_vars = df.drop(['default', 'purpose', 'log_annual_inc', 'days_with_cr_line', 'revol_bal', 'delinq_2yrs', 'credit_
corr_mat = num_vars.corr(method='spearman')
# Select variable with a relationship over 10%
corr_mat = corr_mat[abs(corr_mat) >=  0.1]
corr_mat = corr_mat.round(1)

lower_mat = np.triu(corr_mat, k=1)

plt.figure(figsize=(20, 10))
sns.heatmap(corr_mat,
            annot=True,
            vmin=-1,
            vmax=1,
            cmap=sns.diverging_palette(20, 220, n=200),
            mask=lower_mat
           )
plt.title("Correlation Matrix", fontdict={'fontname': 'Arial', 'fontsize': 20})
plt.show()
```



- Fico score has a strong inverse relationship with the int_rate and revol_util variables
- The int_rate has a strong positive relationship with the revol_util variable

Principal Component Analysis

In [ ]:
```
def biplot(data, loadings, index1, index2, labels=None):
    """
    Function for plot principal components
    """
```

```
    plt.figure(figsize=(15, 7))
    xs = data[:,index1]
    ys = data[:,index2]
    n=loadings.shape[0]
    scalex = 1.0/(xs.max()- xs.min())
    scaley = 1.0/(ys.max()- ys.min())
    plt.scatter(xs*scalex,ys*scaley)
    for i in range(n):
        plt.arrow(0, 0, loadings[i,index1], loadings[i,index2],color='r',alpha=0.5)
        if labels is None:
            plt.text(loadings[i,index1]* 1.15, loadings[i,index2] * 1.15, "Var"+str(i+1), color='g', ha='center', va=
        else:
            plt.text(loadings[i,index1]* 1.15, loadings[i,index2] * 1.15, labels[i], color='r', ha='center', va='cent
    plt.xlim(-1,1)
    plt.ylim(-1,1)
    plt.xlabel("PC{}".format(index1))
    plt.ylabel("PC{}".format(index2))
    plt.axvline(x=0, color='black', linestyle='--')
    plt.axhline(y=0, color='black', linestyle='--')
    plt.grid()
```

In [ ]:
```
num_vars = df.select_dtypes(exclude=['object', 'category'])
num_vars = num_vars.drop('default', axis=1)
```

In [ ]:
```
# Scale data
scaler = StandardScaler()
scaled_data = scaler.fit_transform(num_vars.values)
scaled_data = pd.DataFrame(scaled_data,
                           columns=num_vars.columns,
                           index=num_vars.index
                           )
```

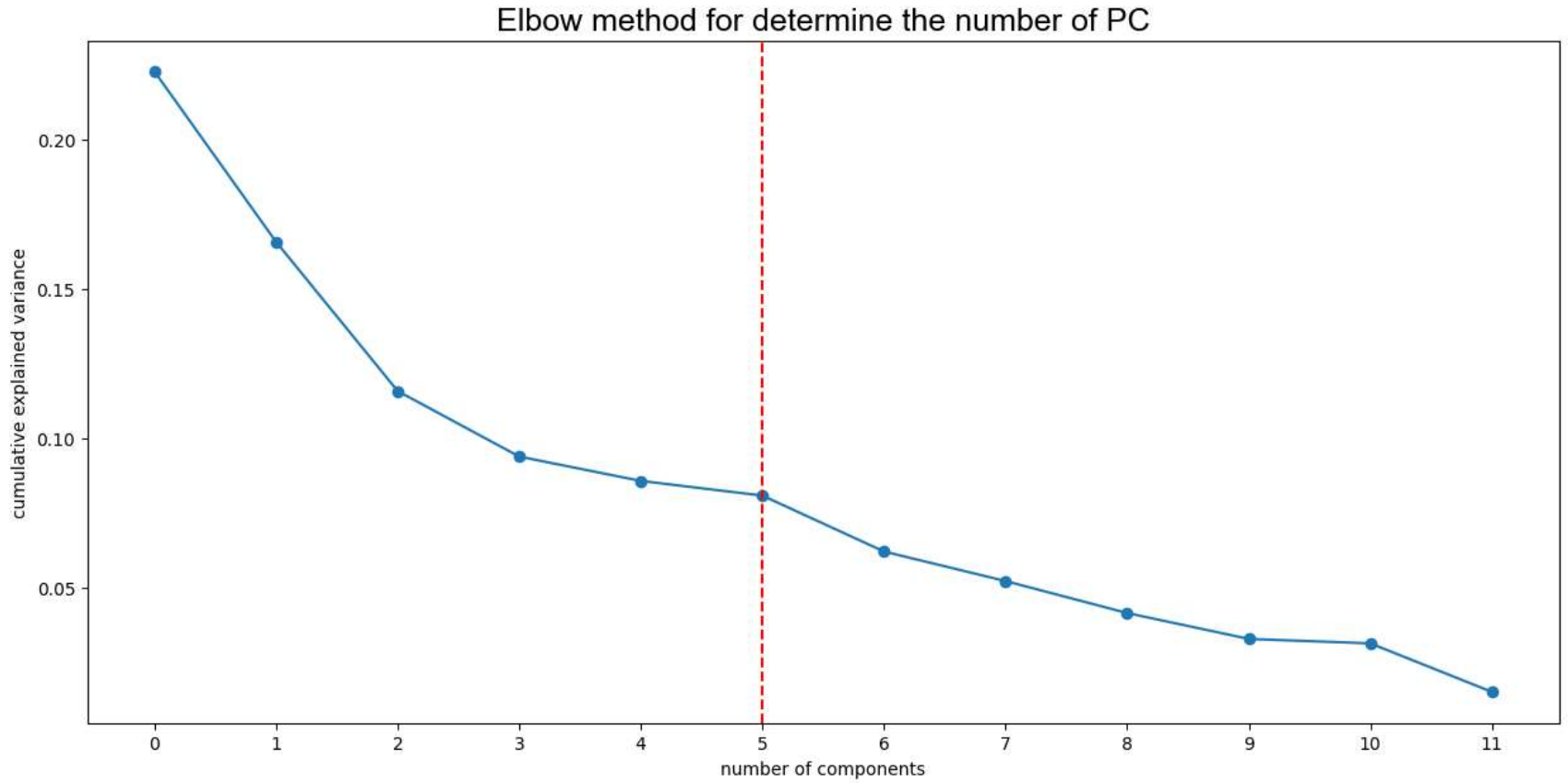
In [ ]:
```
# Principal Components
for comp in range(2, scaled_data.shape[1]):
    pca = PCA(n_components=comp, random_state=42)
    pca.fit(scaled_data)
    variance_acum = pca.explained_variance_ratio_
    final_comp = comp

    # Threshold for determine the number of principal components needed for a specified variance level
    threshold = 0.85
```

```
    if variance_acum.sum() >= threshold:
        print(f"Principal components that collect {threshold * 100}% of the variance: {final_comp}")
        break
```

```
Principal components that collect 85.0% of the variance: 8
```

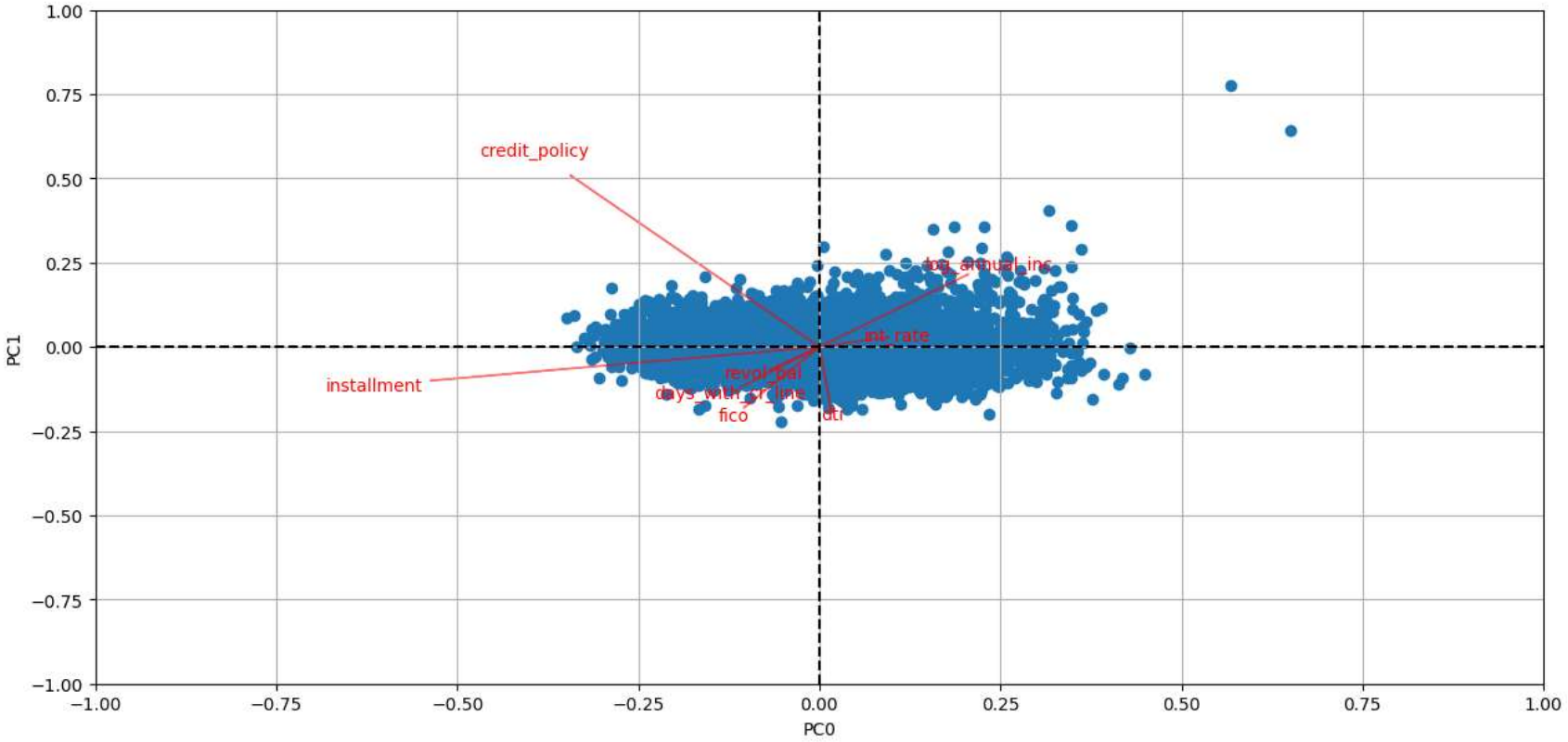
In [ ]:

```
pca = PCA(random_state=42)
pca.fit(scaled_data)
var_exp = pca.explained_variance_ratio_
cum_var_exp = np.cumsum(var_exp)

fig, ax = plt.subplots(figsize=(15, 7))
plt.plot(var_exp, marker='o')
plt.title("Elbow method for determine the number of PC", fontdict={'fontname': 'Arial', 'fontsize':18})
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance')
plt.xticks(range(0, len(pca.components_)))
plt.axvline(x=5, linestyle='--', color='red')
plt.show()
```

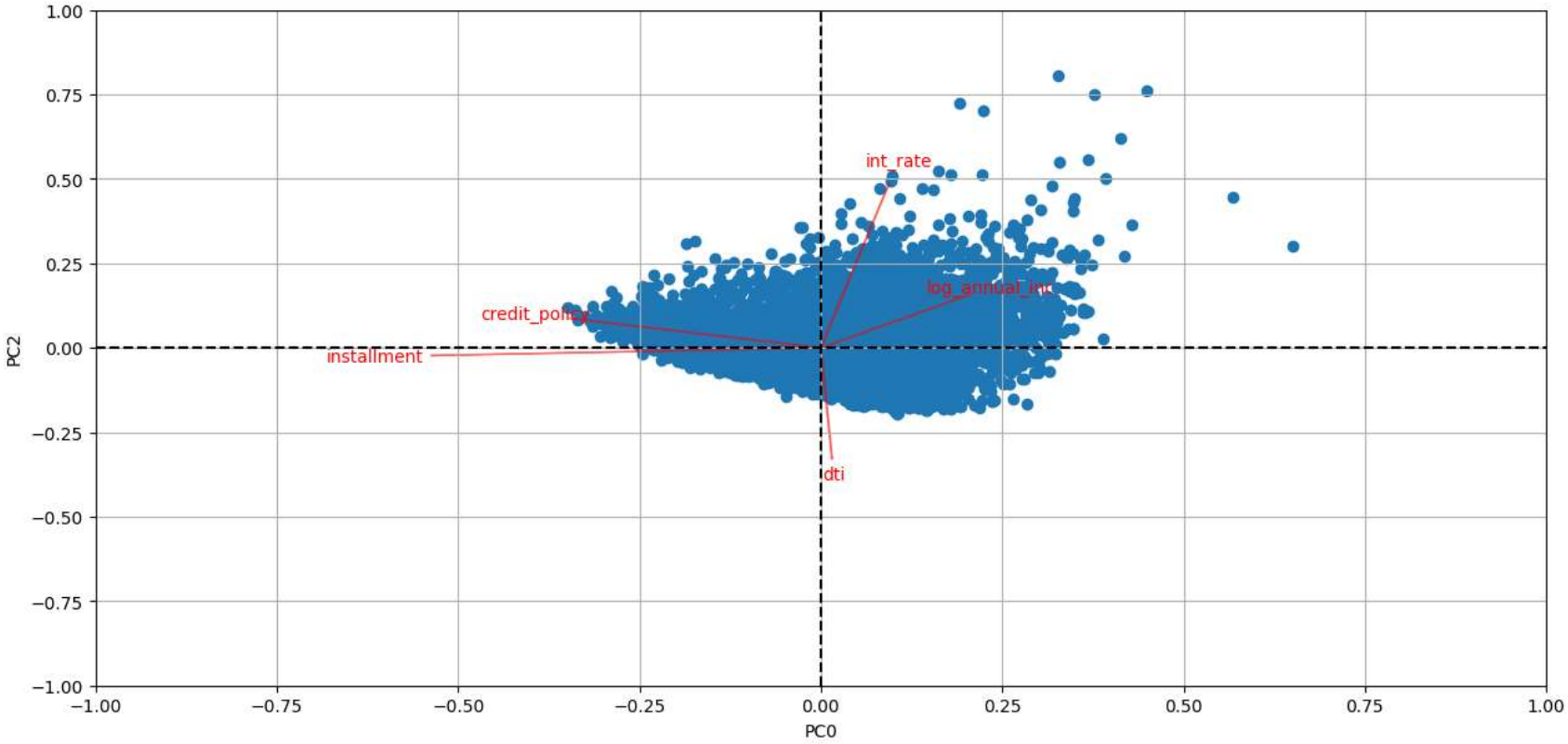


In [ ]:
```
scaler = StandardScaler()
scaled_data = scaler.fit_transform(num_vars.values)
scaled_data = pd.DataFrame(scaled_data,
                           columns=num_vars.columns,
                           index=num_vars.index)
pca = PCA(n_components=5, random_state=42)
pca.fit(scaled_data)
pca_data = pca.transform(scaled_data)
```

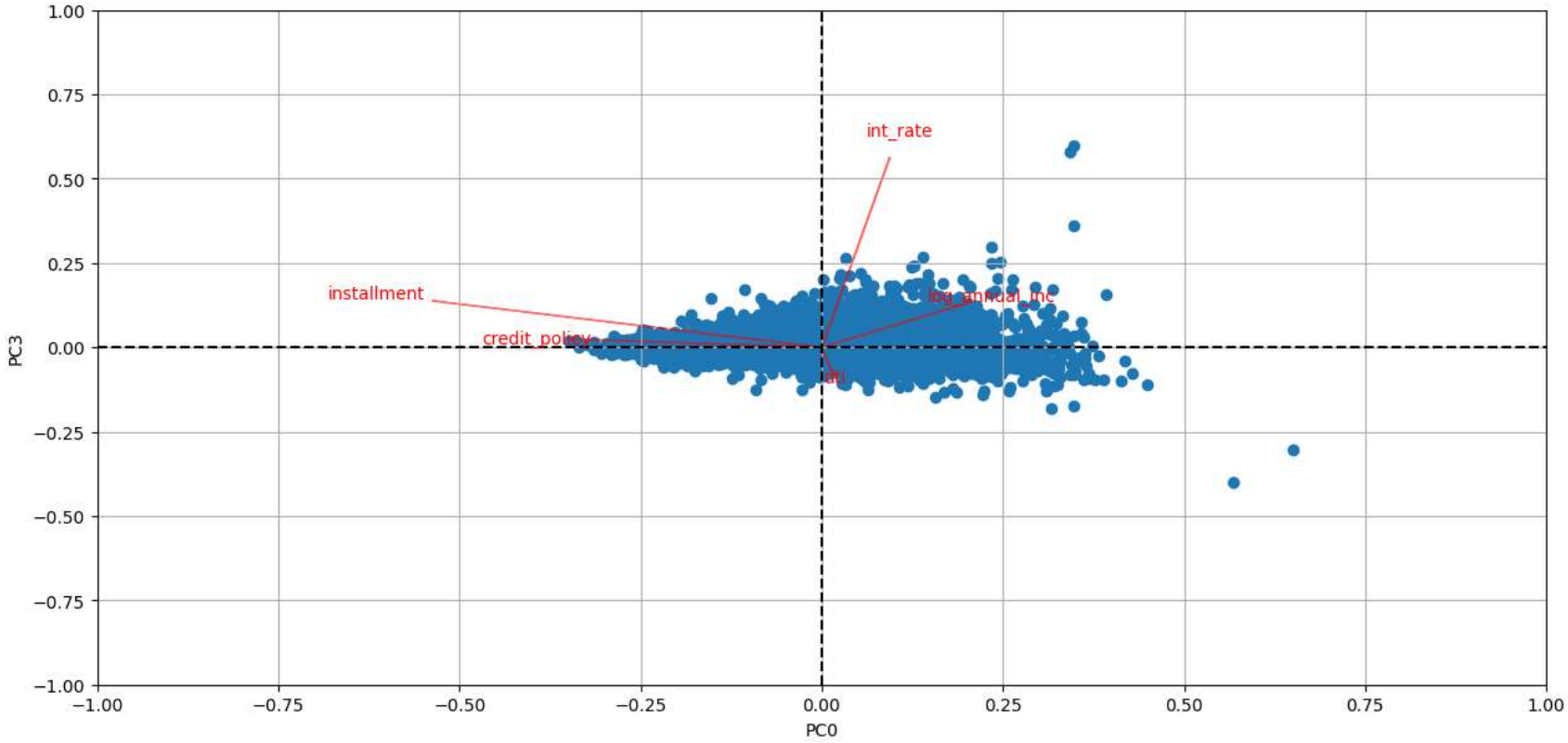
In [ ]:
```
biplot(pca_data, pca.components_, 0, 1, scaled_data.columns.tolist())
```



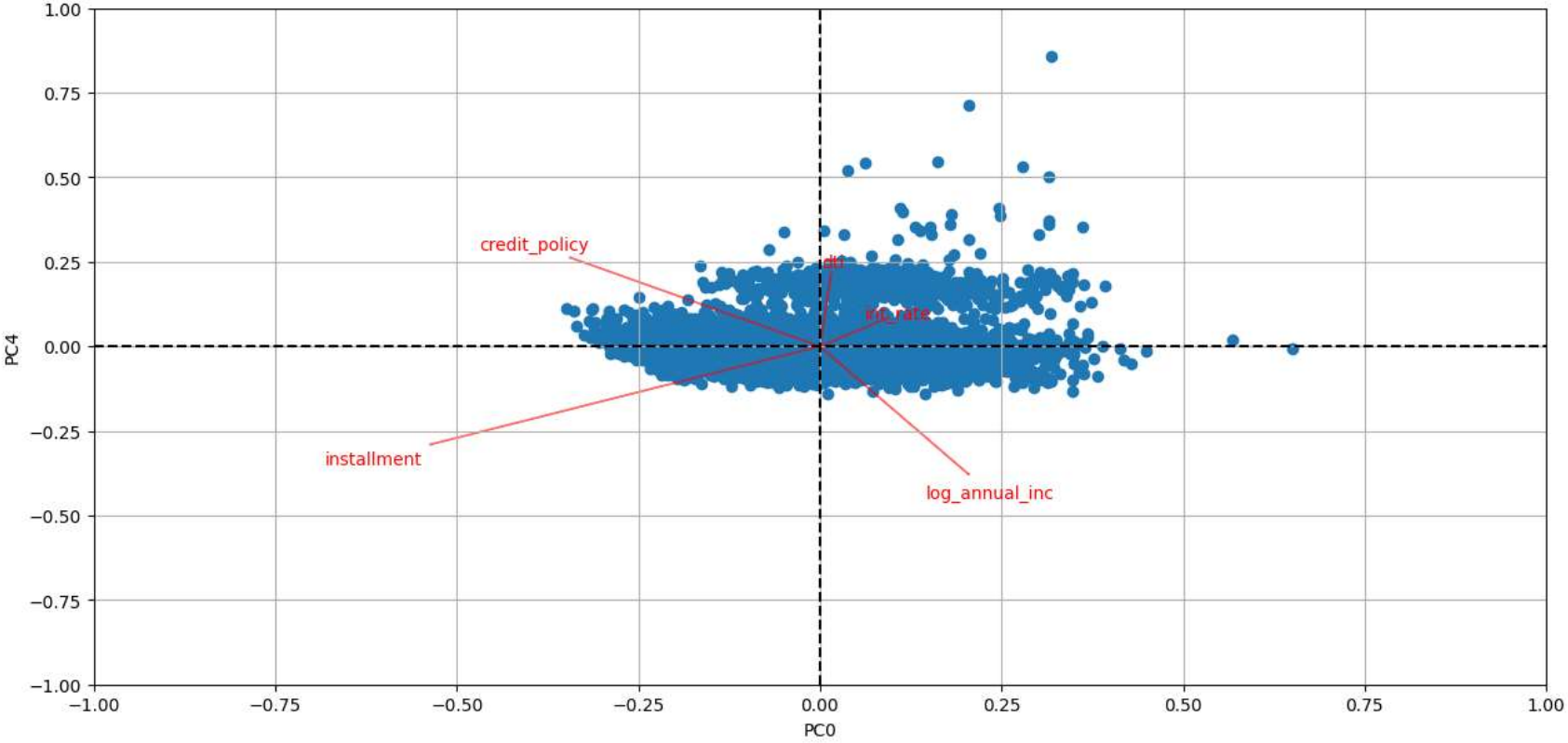
In [ ]:
```
biplot(pca_data, pca.components_, 0, 2, scaled_data.columns.tolist())
```



In [ ]:
```
biplot(pca_data, pca.components_, 0, 3, scaled_data.columns.tolist())
```



In [ ]:
```
biplot(pca_data, pca.components_, 0, 4, scaled_data.columns.tolist())
```


credit_policy
dti
int_rate
installment
log_annual_inc
PC0
PC4
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

Data Preprocessing

In [ ]:
```
import os
import re
import warnings
# Data manipulation
import numpy as np
import pandas as pd
# Visualizatoin
import matplotlib.pyplot as plt
import seaborn as sns
# Preprocessing
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.covariance import MinCovDet
# Inference
from scipy.stats import chi2
from sklearn.feature_selection import f_classif
from scipy.stats import chi2_contingency

# Pandas configuration
pd.options.display.max_columns = None
pd.options.display.max_rows = 200

warnings.filterwarnings('ignore')
```

Load Data

In [ ]:
```
df = pd.read_csv('../data/raw/loan_data.csv', sep=',')
# Adjust columns names
df.columns = [col.replace('.', '_') for col in df.columns.values]
# Rename target column
df = df.rename(columns={'not_fully_paid': 'default'})
```

Train and Test Sets

Before starting data preprocessin, we split the data set into 3 sets:

- **Training:** this set contains 70% of the data and will be use to train the model
- **Validation:** this set contains 15% of the data and will be use to tune hyperparams
- **Test:** this set contains 15% of the data and will be use to select the final model

In [ ]:
```
X = df.drop('default', axis=1)
y = df.default

x_train_temp, x_test, y_train_temp, y_test = train_test_split(X, y,
                                                              test_size=0.15,
                                                              stratify=y,
                                                              random_state=42)

x_train, x_val, y_train, y_val = train_test_split(x_train_temp, y_train_temp,
                                                  test_size=0.15,
                                                  stratify=y_train_temp,
                                                  random_state=42
                                                  )
```

Check all datasets has the same distribution for the target variable

In [ ]:
```
print("Original distribution")
print(df.default.value_counts(normalize=True))
print("\nTrain set distribution")
print(y_train.value_counts(normalize=True))
print("\nValidation set distribution")
print(y_val.value_counts(normalize=True))
print("\nTest set distribution")
print(y_test.value_counts(normalize=True))
```

```
Original distribution
default
0    0.839946
1    0.160054
Name: proportion, dtype: float64

Train set distribution
default
0    0.840006
1    0.159994
Name: proportion, dtype: float64

Validation set distribution
default
0    0.839607
1    0.160393
Name: proportion, dtype: float64

Test set distribution
default
0    0.839944
1    0.160056
Name: proportion, dtype: float64
```

Auxiliary functions for preprocessing

In [ ]:
```python
def bivariated_plot(df, x, y, hue=None, ylabel=None, title=None,
                    legend_title=None, xlabel=None, kind='box', figheight=5,
                    figwidth=10):
    fig, ax = plt.subplots(figsize=(figwidth, figheight))
    if kind == 'box':
        g = sns.boxplot(data=df, y=y, x=x, hue=hue, ax=ax)
    elif kind == 'boxen':
        g = sns.boxenplot(data=df, y=y, x=x, hue=hue, ax=ax)
    elif kind == 'scatter':
        g = sns.scatterplot(data=df, y=y, x=x, hue=hue, s=100, ax=ax)
    else:
        g = sns.kdeplot(data=df, x=y, fill=True, ax=ax)
        sns.rugplot(data=df, x=y, color='red', alpha=0.1, ax=ax)
```

```
    plt.title(title, size=20)
    plt.ylabel(ylabel, size=18)
    plt.yticks(fontsize=16)
    plt.xlabel(xlabel, size=18)
    plt.xticks(fontsize=16)
    plt.legend(title=legend_title, fontsize='50')
    sns.move_legend(g, "upper left", bbox_to_anchor=(1, 1))
    plt.show()

def woe_cat_discrete(df, cat_variabe_name, y_df):
    df = pd.concat([df[cat_variabe_name], y_df], axis = 1)
    df = pd.concat([df.groupby(df.columns.values[0], as_index = False)[df.columns.values[1]].count(),
                    df.groupby(df.columns.values[0], as_index = False)[df.columns.values[1]].mean()], axis = 1)
    df = df.iloc[:, [0, 1, 3]]
    df.columns = [df.columns.values[0], 'n_obs', 'prop_good']
    df['prop_n_obs'] = df['n_obs'] / df['n_obs'].sum()
    df['n_good'] = df['prop_good'] * df['n_obs']
    df['n_bad'] = (1 - df['prop_good']) * df['n_obs']
    df['prop_n_good'] = df['n_good'] / df['n_good'].sum()
    df['prop_n_bad'] = df['n_bad'] / df['n_bad'].sum()
    df['WoE'] = np.log(df['prop_n_good'] / df['prop_n_bad'])
    df = df.sort_values(['WoE'])
    df = df.reset_index(drop = True)
    df['diff_prop_good'] = df['prop_good'].diff().abs()
    df['diff_WoE'] = df['WoE'].diff().abs()
    df['IV'] = (df['prop_n_good'] - df['prop_n_bad']) * df['WoE']
    df['IV'] = df['IV'].sum()
    return df

def woe_num_discrete(df, continuous_variabe_name, y_df):
    df = pd.concat([df[continuous_variabe_name], y_df], axis = 1)
    df = pd.concat([df.groupby(df.columns.values[0], as_index = False)[df.columns.values[1]].count(),
                    df.groupby(df.columns.values[0], as_index = False)[df.columns.values[1]].mean()], axis = 1)
    df = df.iloc[:, [0, 1, 3]]
    df.columns = [df.columns.values[0], 'n_obs', 'prop_good']
    df['prop_n_obs'] = df['n_obs'] / df['n_obs'].sum()
    df['n_good'] = df['prop_good'] * df['n_obs']
    df['n_bad'] = (1 - df['prop_good']) * df['n_obs']
    df['prop_n_good'] = df['n_good'] / df['n_good'].sum()
    df['prop_n_bad'] = df['n_bad'] / df['n_bad'].sum()
    df['WoE'] = np.log(df['prop_n_good'] / df['prop_n_bad'])
    #df = df.sort_values(['WoE'])
```

```
    #df = df.reset_index(drop = True)
    df['diff_prop_good'] = df['prop_good'].diff().abs()
    df['diff_WoE'] = df['WoE'].diff().abs()
    df['IV'] = (df['prop_n_good'] - df['prop_n_bad']) * df['WoE']
    df['IV'] = df['IV'].sum()
    return df

def plot_by_woe(df_WoE, rotation_of_x_axis_labels = 0):
    x = np.array(df_WoE.iloc[:, 0].apply(str))
    y = df_WoE['WoE']
    plt.figure(figsize=(18, 6))
    plt.plot(x, y, marker = 'o', linestyle = '--', color = 'k')
    plt.xlabel(df_WoE.columns[0])
    plt.ylabel('Weight of Evidence')
    plt.title(str('Weight of Evidence by ' + df_WoE.columns[0]))
    plt.xticks(rotation = rotation_of_x_axis_labels)
    plt.axhline(0, linestyle='--', color='r', lw=2.5)

def text_normalized(text: str) -> str:
    """Funcion para quitar tildes de las vocales
    Args:
        text (str): cadena de texto con tildes
    Returns:
        str: cadena de texto sin tildes
    """
    replacements = (
        ("á", "a"),
        ("é", "e"),
        ("í", "i"),
        ("ó", "o"),
        ("ú", "u"),
    )
    for a, b in replacements:
        text = text.replace(a, b).replace(a.upper(), b.upper())
    return text

def iv_woe(data, target, bins=10, show_woe=False):

    #Empty Dataframe
    newDF,woeDF = pd.DataFrame(), pd.DataFrame()

    #Extract Column Names
```

```
    cols = data.columns

    #Run WOE and IV on all the independent variables
    for ivars in cols[~cols.isin([target])]:
        if (data[ivars].dtype.kind in 'bifc') and (len(np.unique(data[ivars]))>10):
            binned_x = pd.qcut(data[ivars], bins,  duplicates='drop')
            d0 = pd.DataFrame({'x': binned_x, 'y': data[target]})
        else:
            d0 = pd.DataFrame({'x': data[ivars], 'y': data[target]})
        d = d0.groupby("x", as_index=False).agg({"y": ["count", "sum"]})
        d.columns = ['Cutoff', 'N', 'Events']
        d['% of Events'] = np.maximum(d['Events'], 0.5) / d['Events'].sum()
        d['Non-Events'] = d['N'] - d['Events']
        d['% of Non-Events'] = np.maximum(d['Non-Events'], 0.5) / d['Non-Events'].sum()
        d['WoE'] = np.log(d['% of Events']/d['% of Non-Events'])
        d['IV'] = d['WoE'] * (d['% of Events'] - d['% of Non-Events'])
        d.insert(loc=0, column='Variable', value=ivars)
        # print("Information value of " + ivars + " is " + str(round(d['IV'].sum(),6)))
        temp =pd.DataFrame({"Variable" : [ivars], "IV" : [d['IV'].sum()]}, columns = ["Variable", "IV"])
        newDF=pd.concat([newDF,temp], axis=0)
        woeDF=pd.concat([woeDF,d], axis=0)

        #Show WOE Table
        if show_woe == True:
            print(d)
    return newDF, woeDF

def cov(data, scaled=False, scaler='standar', robust=False):
    df = data.copy()
    df1 = pd.DataFrame(columns=['variable', 'cov'])

    if robust:
        i = 0
        for col in df.columns:
            q1, q3 = np.percentile(df[col], [25, 75])
            iqr = q3 - q1
            median = df[col].median()
            if median == 0:
                cov = 0
            else:
                cov = iqr / median
            df1.loc[i] = [col, cov]
```

```
            i+= 1
    else:
        if scaled:
            if scaler == 'standar':
                df = df.apply(lambda row: (row - row.mean())/row.std(), axis=1)
            else:
                df = df.apply(lambda row: (row - row.min())/(row.max() - row.min()), axis=1)

        i = 0
        for col in df.columns:
            mean = np.mean(df[col].dropna())
            std_dev = np.std(df[col].dropna())
            if mean == 0:
                cov = 0
            else:
                cov = std_dev / mean
            df1.loc[i] = [col, cov]
            i+= 1

    df1 = df1.sort_values(by='cov', ascending=False)
    df1 = df1.reset_index(drop=True)

    return df1

def dummy_creation(df, columns_list):
    """Funcion para crear variables dummies
    """
    df_dummies = []
    for col in columns_list:
        df_dummies.append(pd.get_dummies(df[col], prefix = col, prefix_sep = ':'))
    df_dummies = pd.concat(df_dummies, axis = 1)
    df = pd.concat([df, df_dummies], axis = 1)
    return df
```

Preprocessing

Numerical variables

In [ ]:
```
num_df = x_train.drop('purpose', axis=1)
```

In [ ]:
```
(num_df
 .describe()
 .T
 .style
 .format({'count': '{:.0f}',
          'mean': '{:.2f}',
          'std': '{:.2f}',
          'min': '{:.2f}',
          '25%': '{:.2f}',
          '50%': '{:.2f}',
          '75%': '{:.2f}',
          'max': '{:.2f}'
          })
 )
```

Out[ ]:

	count	mean	std	min	25%	50%	75%	max
credit_policy	6919	0.80	0.40	0.00	1.00	1.00	1.00	1.00
int_rate	6919	0.12	0.03	0.06	0.10	0.12	0.14	0.22
installment	6919	319.22	206.24	15.67	163.70	269.07	437.72	940.14
log_annual_inc	6919	10.94	0.61	7.55	10.57	10.92	11.29	14.18
dti	6919	12.61	6.90	0.00	7.16	12.58	18.03	29.96
fico	6919	711.02	38.07	612.00	682.00	707.00	737.00	827.00
days_with_cr_line	6919	4544.48	2492.79	180.04	2790.00	4110.04	5700.04	17639.96
revol_bal	6919	17242.54	34909.10	0.00	3204.50	8613.00	18341.00	1207359.00
revol_util	6919	46.94	29.13	0.00	22.75	46.40	71.20	119.00
inq_last_6mths	6919	1.56	2.14	0.00	0.00	1.00	2.00	31.00
delinq_2yrs	6919	0.16	0.53	0.00	0.00	0.00	0.00	13.00
pub_rec	6919	0.06	0.26	0.00	0.00	0.00	0.00	5.00

In [ ]:
```
x_train.purpose.value_counts(normalize=True) * 100
```

Out[ ]:
```
purpose
debt_consolidation    41.263188
all_other             24.093077
credit_card           13.065472
small_business         6.749530
home_improvement       6.576095
major_purchase         4.567134
educational            3.685504
Name: proportion, dtype: float64
```

Outlier Detection

For outlier detection the Mahalanobis distance is used

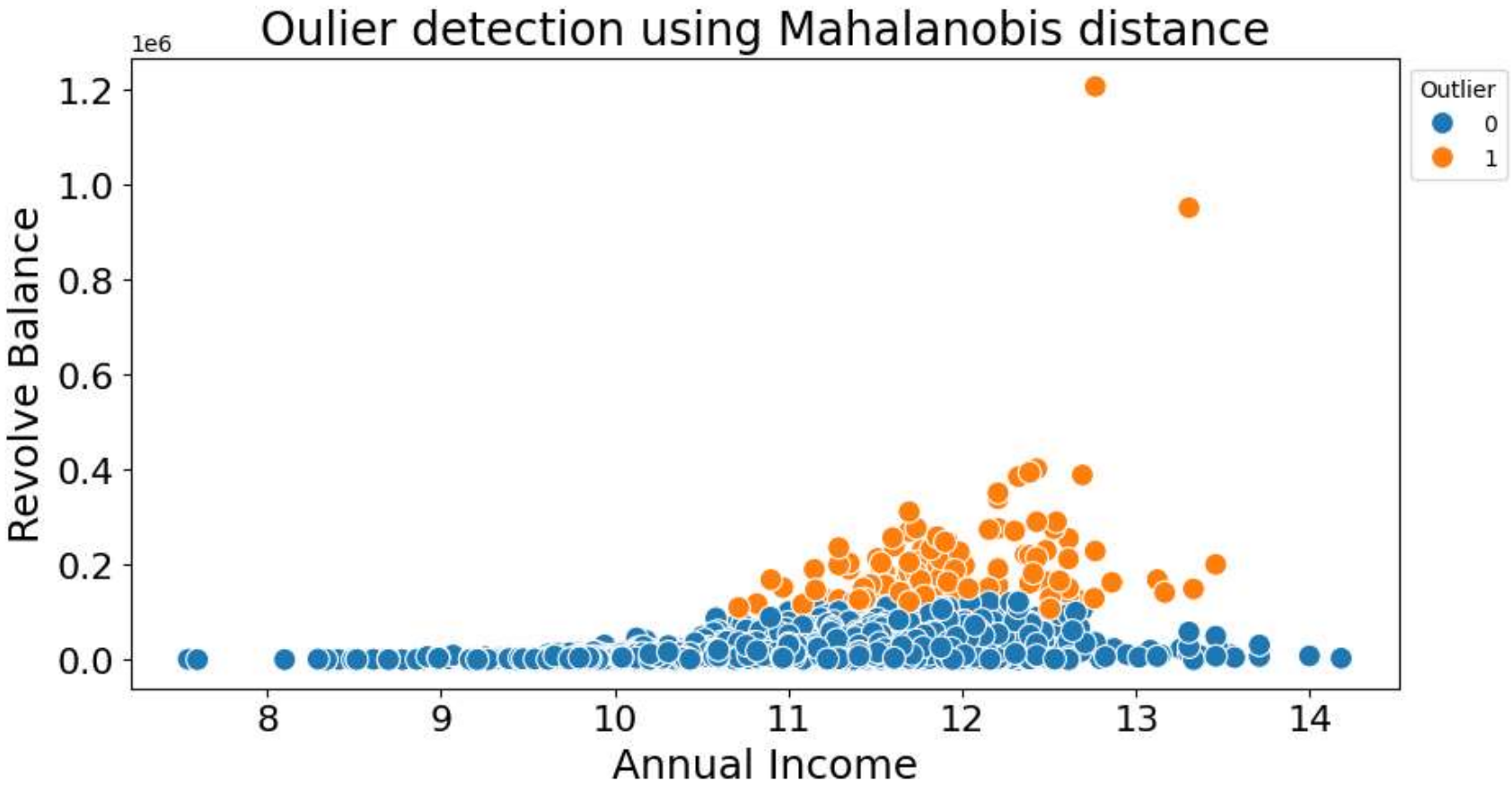
In [ ]:
```
outlier_df = x_train.select_dtypes(exclude=['object', 'category'])
outlier_df.head()
```

Out[ ]:

	credit_policy	int_rate	installment	log_annual_inc	dti	fico	days_with_cr_line	revol_bal	revol_util	inq_last_6mths	del
3950	1	0.1284	268.95	10.714418	8.32	712	2100.000000	9572	76.6	0	
7112	1	0.1645	583.73	11.472103	15.10	667	4620.041667	19257	92.6	1	
610	1	0.0976	482.32	10.165852	5.77	777	4560.041667	3050	2.9	1	
966	1	0.1229	213.46	10.463103	20.78	687	4620.000000	9408	64.0	1	
4794	1	0.0894	254.18	11.652687	2.70	762	6900.000000	5914	15.9	4	

In [ ]:
```
target = y_train

mcd = MinCovDet().fit(num_df)
mahadist = mcd.mahalanobis(num_df)
cutoff = np.percentile(mahadist, 98)
# cutoff = chi2.ppf(0.95, num_df.shape[1])
outlier_df['maha_out'] = mahadist
outlier_df.maha_out = [1 if mcd > cutoff else 0 for mcd in outlier_df.maha_out]

bivariated_plot(outlier_df,#.query("maha_out == 0"),
```

```
            y='revol_bal',
            x='log_annual_inc',
            hue='maha_out',
            kind='scatter',
            xlabel='Annual Income',
            ylabel='Revolve Balance',
            title='Oulier detection using Mahalanobis distance',
            legend_title='Outlier',
            figwidth=10,
            figheight=5)
```



Feature Selection

For selecting variables, many criteria such us informatio value, chi-square test, anova test and the results from the exploratoy analysis are taken into account

Categorical Variables

Information Value

In [ ]:
```
cat_df = x_train.select_dtypes(include=['object', 'category']).copy()

# Invertir valores de clases para calcular y graficar WoE
target = y_train.copy()
target = target + 1
target.loc[target == 2] = 0

cat_df = x_train.select_dtypes(include=['object', 'category'])
cat_df['bad'] = target
iv, woe = iv_woe(data=cat_df, target='bad', bins=10, show_woe=False)

iv.columns = [col.lower() for col in iv.columns]
iv.loc[iv.iv <= 0.02,'predictividad'] = 'None'
iv.loc[(iv.iv > 0.02) & (iv.iv <= 0.1),'predictividad'] = 'Weak'
iv.loc[(iv.iv > 0.1) & (iv.iv <= 0.3),'predictividad'] = 'Medium'
iv.loc[(iv.iv > 0.3) & (iv.iv <= 0.5),'predictividad'] = 'Strong'
iv.loc[iv.iv > 0.5,'predictividad'] = 'Suspiciously strong'
iv.sort_values(by='iv', ascending=False)
```

Out[ ]:

	variable	iv	predictividad
0	purpose	0.07901	Weak

Chi squared test

In [ ]:
```
cat_df = x_train.select_dtypes(include=['object', 'category'])
cat_df['bad'] = y_train

chi2_check = {}
for column in cat_df:
    if column == 'bad':
```

```
        pass
    else:
        chi, p, dof, ex = chi2_contingency(pd.crosstab(cat_df.bad, cat_df[column]))
        chi2_check.setdefault('Feature',[]).append(column)
        chi2_check.setdefault('p-value',[]).append(round(p, 10))

# convert the dictionary to a DF
chi2_result = pd.DataFrame(data=chi2_check)
chi2_result.sort_values(by = ['p-value'], ascending = True, ignore_index = True, inplace = True)
chi2_result['significant'] = np.where(chi2_result['p-value'] < 0.05, 'yes', 'no')
chi2_result
```

Out[ ]:

	Feature	p-value	significant
0	purpose	0.0	yes

Numerical Variables

Anova Test

In [ ]:

```
# Calcular estadistico F y p-value
F_statistic, p_values = f_classif(num_df, y_train)

# DF
ANOVA_F_table = pd.DataFrame(data = {'variable': num_df.columns.values,
                                     'F-Score': F_statistic,
                                     'p-value': p_values.round(decimals=10)})
ANOVA_F_table = ANOVA_F_table.reset_index(drop=True)
ANOVA_F_table.sort_values(by = ['F-Score'],
                          ascending = False,
                          ignore_index = True,
                          inplace = True)
ANOVA_F_table = ANOVA_F_table.loc[ANOVA_F_table.variable != 'bad', :]
ANOVA_F_table['significant'] = np.where(ANOVA_F_table['p-value'] < 0.05, 'yes', 'no')

ANOVA_F_table
```

Out[ ]:

	variable	F-Score	p-value	significant
1	int_rate	174.070624	0.000000	yes
2	credit_policy	144.943501	0.000000	yes
3	fico	133.501869	0.000000	yes
4	inq_last_6mths	129.023205	0.000000	yes
5	revol_util	45.757721	0.000000	yes
6	revol_bal	20.619445	0.000006	yes
7	installment	17.548604	0.000028	yes
8	pub_rec	16.634476	0.000046	yes
9	dti	8.446555	0.003669	yes
10	log_annual_inc	4.070289	0.043682	yes
11	days_with_cr_line	1.154512	0.282644	no
12	delinq_2yrs	0.295565	0.586694	no

Information Value

In [ ]:

```
# Invertir valores de clases para calcular y graficar WoE
target = y_train.copy()
target = target + 1
target.loc[target == 2] = 0
num_df = x_train.select_dtypes(exclude=['object', 'category']).copy()
num_df['bad'] = target
iv, woe = iv_woe(data=num_df, target='bad', bins=10, show_woe=False)

iv.columns = [col.lower() for col in iv.columns]
iv.loc[iv.iv <= 0.02,'predictividad'] = 'None'
iv.loc[(iv.iv > 0.02) & (iv.iv <= 0.1),'predictividad'] = 'Weak'
iv.loc[(iv.iv > 0.1) & (iv.iv <= 0.3),'predictividad'] = 'Medium'
iv.loc[(iv.iv > 0.3) & (iv.iv <= 0.5),'predictividad'] = 'Strong'
```

```
iv.loc[iv.iv > 0.5,'predictividad'] = 'Suspiciously strong'
iv.sort_values(by='iv', ascending=False)
```

Out[ ]:

	variable	iv	predictividad
0	int_rate	0.239938	Medium
0	fico	0.161819	Medium
0	credit_policy	0.132144	Medium
0	inq_last_6mths	0.131994	Medium
0	revol_util	0.066965	Weak
0	installment	0.030106	Weak
0	revol_bal	0.024866	Weak
0	pub_rec	0.022926	Weak
0	log_annual_inc	0.016012	None
0	dti	0.011323	None
0	days_with_cr_line	0.010788	None
0	delinq_2yrs	0.003220	None

Given the EDA analysis and the tests applied in this secction. The following variables are selected:

- credit_policy
- int_rate
- installment
- dti
- fico
- revol_util
- inq_last_6mths
- log_annual_inc
- purpose

Feature Engineering Using WOE

For creating new variables the Weight of Evidence is used.

The variables created using the WoE will be categorical, based on the woe value for each of the variables classes.

- A WoE near to 0 means that the class is not relevant for predicting the target variable.
- A positive value of WoE means the variable tends to predict better the negative class of the target variable
- A negative value of Woe means that the variable tends to predict better the positive class of the target variable.

In this case the positive class means that the customer will default

We will apply WoE anaylsis for numercial and catgorical variables for defining variable classes for each one of the select variables. Next we will define the base category for each variable, as the base category will be selected the one which has the closest value to zero

Finally we will create pipeline in the modeling, using custom sklearn classes, to apply this transformation to the datasets.

Categorical Variables

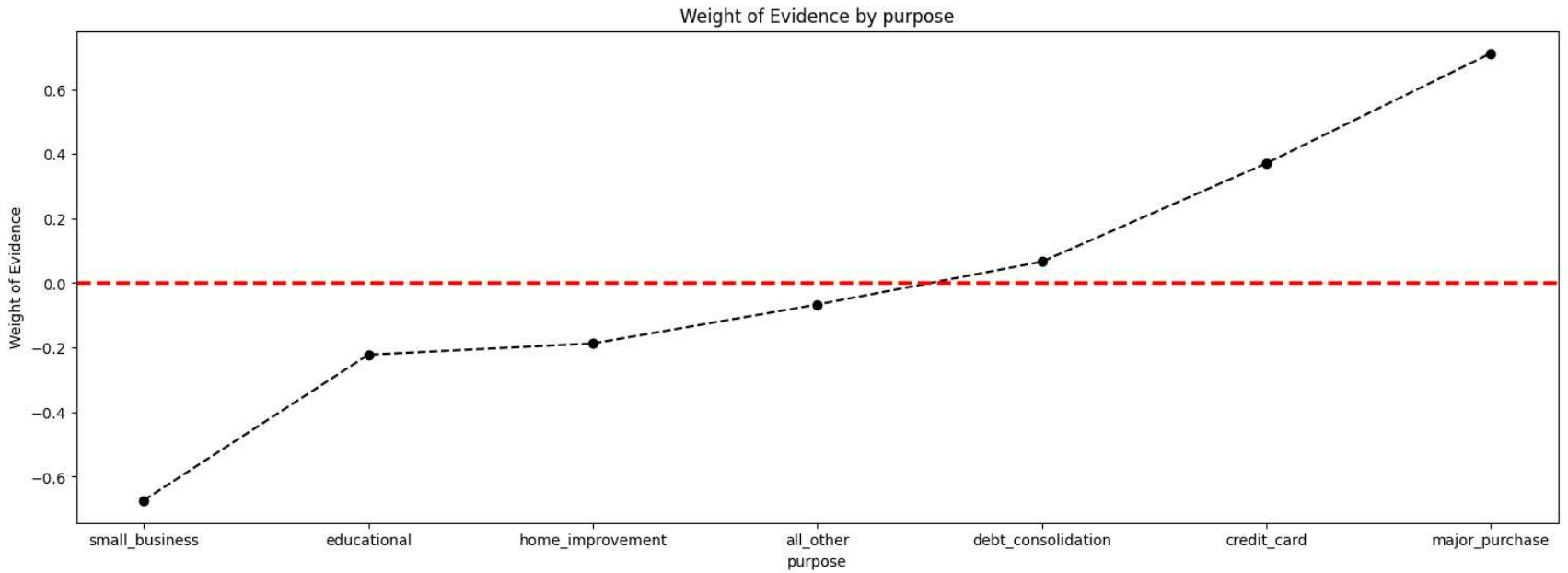
In [ ]:
```
"""
Purposes groups
----------------
small_business
educational | home_improvement (base category)
all_other | debt_consolidation
credit_card
major_purchase

"""

cat_df = x_train.select_dtypes(include=['object', 'category']).copy()

# Invertir valores de clases para calcular y graficar WoE
target = y_train.copy()
target = target + 1
target.loc[target == 2] = 0
```

```
temp_df = woe_cat_discrete(cat_df, 'purpose', target)
temp_df
```

Out[ ]:

	purpose	n_obs	prop_good	prop_n_obs	n_good	n_bad	prop_n_good	prop_n_bad	WoE	diff_prop_good	di
0	small_business	467	0.728051	0.067495	340.0	127.0	0.058500	0.114724	-0.673513	NaN	
1	educational	255	0.807843	0.036855	206.0	49.0	0.035444	0.044264	-0.222215	0.079792	0.
2	home_improvement	455	0.813187	0.065761	370.0	85.0	0.063661	0.076784	-0.187419	0.005344	0.
3	all_other	1667	0.830834	0.240931	1385.0	282.0	0.238300	0.254743	-0.066723	0.017647	0.
4	debt_consolidation	2855	0.848687	0.412632	2423.0	432.0	0.416896	0.390244	0.066065	0.017853	0.
5	credit_card	904	0.883850	0.130655	799.0	105.0	0.137474	0.094851	0.371130	0.035163	0.
6	major_purchase	316	0.914557	0.045671	289.0	27.0	0.049725	0.024390	0.712319	0.030707	0.

In [ ]:
```
plot_by_woe(temp_df)
```



Numerical Variables

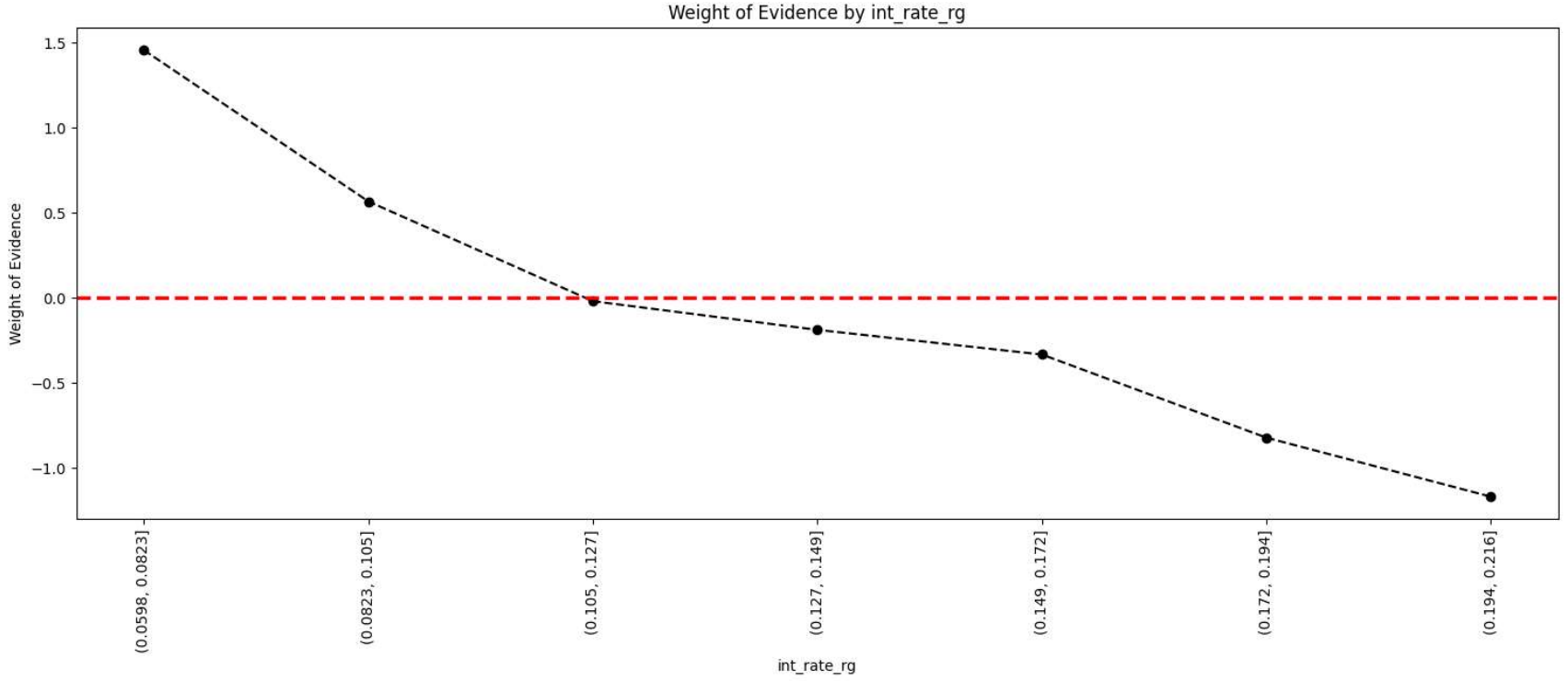
Int_rate

In [ ]:
```
"""
Int rate ranges
----------------
(0.0, 0.0823]
(0.0823, 0.105]
(0.105, 0.127] (base)
(0.127, 0.149]
(0.149, 0.172]
(0.172, 0.194]
(0.194, 0.216]

"""

num_df['int_rate_rg'] = pd.cut(num_df.int_rate, 7) # 7
temp_df = woe_num_discrete(num_df, 'int_rate_rg', target)
temp_df['%_n_obs'] = (temp_df.n_obs / temp_df.n_obs.sum()) * 100
temp_df['%_n_obs_cum'] = temp_df['%_n_obs'].cumsum()
temp_df['val_5%'] = np.where(temp_df.n_obs >= num_df.shape[0] * 0.05, 1, 0)
temp_df['val_10%'] = np.where(temp_df.n_obs >= num_df.shape[0] * 0.1, 1, 0)
temp_df['val_good'] = np.where(temp_df.n_good > 0, 1, 0)
temp_df['val_bad'] = np.where(temp_df.n_bad > 0, 1, 0)

(temp_df
 .iloc[:, [0,1,4,5,8,12,13,14,15,16,17]]
 .style
 .format({'n_good': '{0:.0f}',
          'n_bad': '{0:.0f}',
          '%_n_obs_cum': '{0:.2f}',
          '%_n_obs': '{0:.2f}'
          })
 .bar(subset=['%_n_obs'], color='purple')
 )
```

Out[ ]:

	int_rate_rg	n_obs	n_good	n_bad	WoE	%_n_obs	%_n_obs_cum	val_5%	val_10%	val_good	val_bad
0	(0.0598, 0.0823]	587	562	25	1.454355	8.48	8.48	1	0	1	1
1	(0.0823, 0.105]	1178	1063	115	0.565647	17.03	25.51	1	1	1	1
2	(0.105, 0.127]	2120	1775	345	-0.020260	30.64	56.15	1	1	1	1
3	(0.127, 0.149]	1893	1539	354	-0.188680	27.36	83.51	1	1	1	1
4	(0.149, 0.172]	876	692	184	-0.333621	12.66	96.17	1	1	1	1
5	(0.172, 0.194]	215	150	65	-0.822023	3.11	99.28	0	0	1	1
6	(0.194, 0.216]	50	31	19	-1.168723	0.72	100.00	0	0	1	1

In [ ]:
```
plot_by_woe(temp_df, rotation_of_x_axis_labels = 90)
```



installment

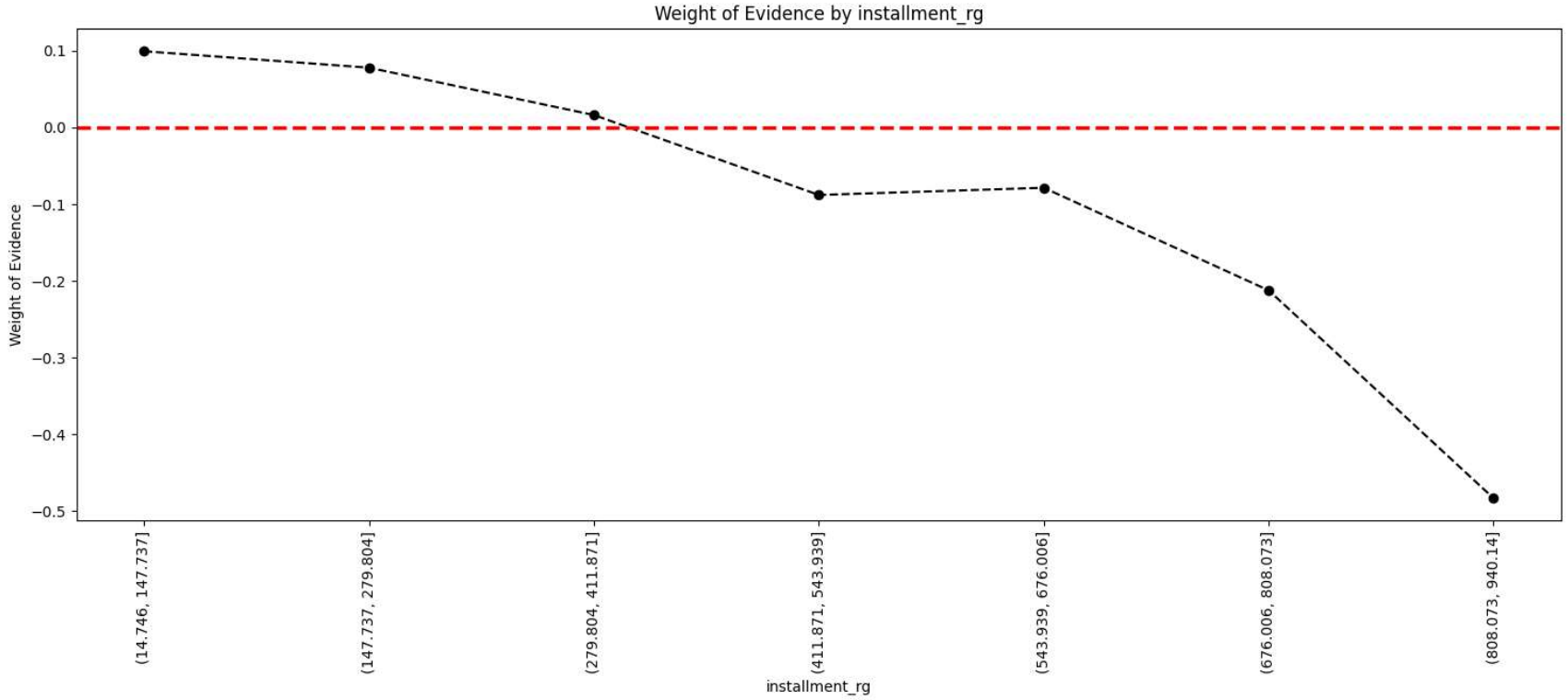
In [ ]:

```
"""
Installment ranges
----------------
(0, 147.737]
(147.737, 279.804]
(279.804, 411.871] (Base)
(411.871, 676.006]
(676.006, 808.073]
(808.073, inf]

"""
num_df = x_train.select_dtypes(exclude=['object', 'category']).copy()
target = y_train.copy()
target = target + 1
target.loc[target == 2] = 0

num_df['installment_rg'] = pd.cut(num_df.installment, 7) # 7
temp_df = woe_num_discrete(num_df, 'installment_rg', target)
temp_df['%_n_obs'] = (temp_df.n_obs / temp_df.n_obs.sum()) * 100
temp_df['%_n_obs_cum'] = temp_df['%_n_obs'].cumsum()
temp_df['val_5%'] = np.where(temp_df.n_obs >= num_df.shape[0] * 0.05, 1, 0)
temp_df['val_10%'] = np.where(temp_df.n_obs >= num_df.shape[0] * 0.1, 1, 0)
temp_df['val_good'] = np.where(temp_df.n_good > 0, 1, 0)
temp_df['val_bad'] = np.where(temp_df.n_bad > 0, 1, 0)

(temp_df
 .iloc[:, [0,1,4,5,8,9,12,13,14,15,16,17]]
 .style
 .format({'n_good': '{0:.0f}',
          'n_bad': '{0:.0f}',
          '%_n_obs_cum': '{0:.2f}',
          '%_n_obs': '{0:.2f}'
          })
 .bar(subset=['%_n_obs'], color='purple')
 )
```

Out[ ]:

	installment_rg	n_obs	n_good	n_bad	WoE	diff_prop_good	%_n_obs	%_n_obs_cum	val_5%	val_10%	val_good	val_ba
0	(14.746, 147.737]	1421	1212	209	0.099422	nan	20.54	20.54	1	1	1	
1	(147.737, 279.804]	2163	1839	324	0.077963	0.002712	31.26	51.80	1	1	1	
2	(279.804, 411.871]	1470	1238	232	0.016244	0.008031	21.25	73.05	1	1	1	
3	(411.871, 543.939]	877	726	151	-0.088001	0.014355	12.68	85.72	1	1	1	
4	(543.939, 676.006]	439	364	75	-0.078605	0.001335	6.34	92.07	1	0	1	
5	(676.006, 808.073]	299	242	57	-0.212385	0.019793	4.32	96.39	0	0	1	
6	(808.073, 940.14]	250	191	59	-0.483535	0.045365	3.61	100.00	0	0	1	

In [ ]:
```
plot_by_woe(temp_df, rotation_of_x_axis_labels = 90)
```



log_annual_inc

In [ ]:
```
"""
Log annual income range
------------------------
(0, 10.949] (Base)
(10.949, 12.026]
(12.026, 13.103]
(13.103, inf]
"""

num_df = x_train.select_dtypes(exclude=['object', 'category']).copy()
target = y_train.copy()
target = target + 1
target.loc[target == 2] = 0

num_df2_temp = num_df.loc[(num_df.log_annual_inc > 9.869), :].copy()
```

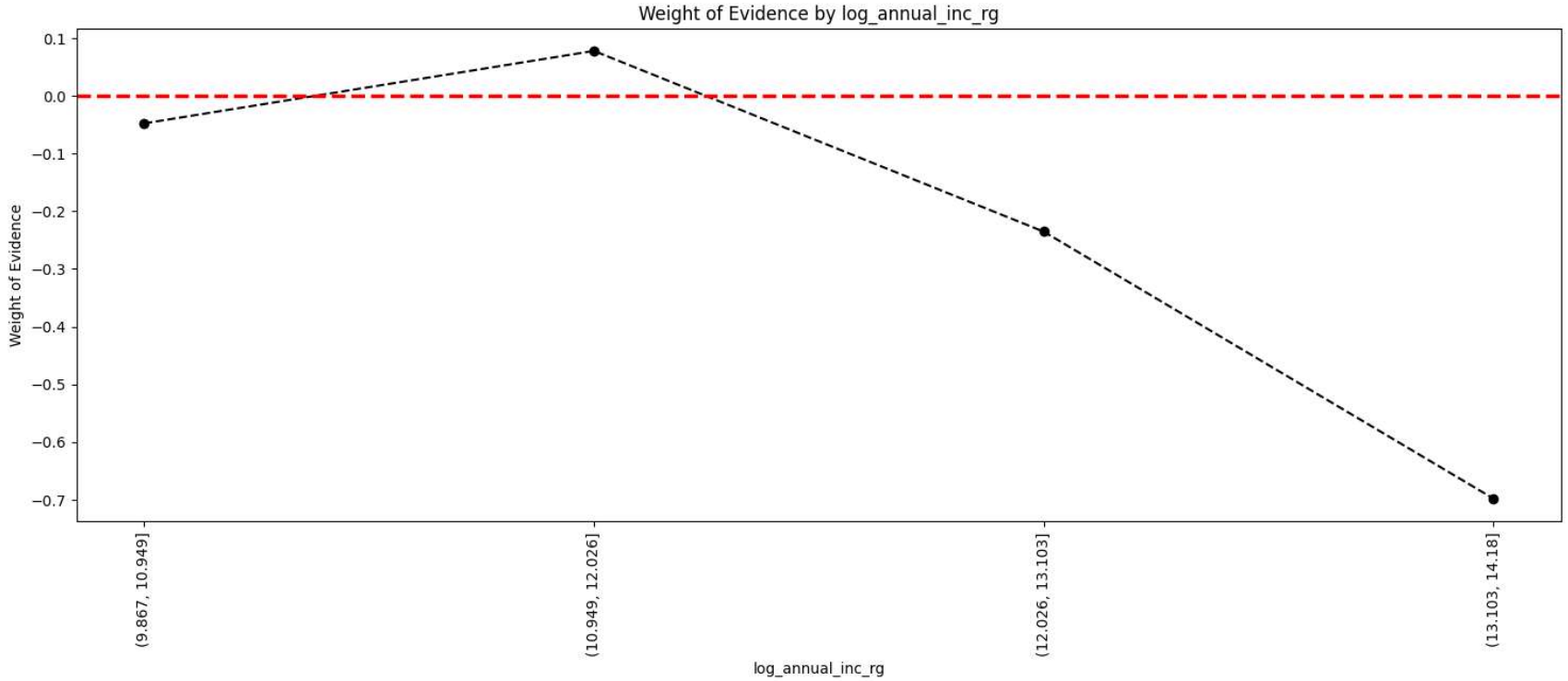
```
num_df2_temp['log_annual_inc_rg'] = pd.cut(num_df2_temp.log_annual_inc, 4)
temp_df = woe_num_discrete(num_df2_temp, 'log_annual_inc_rg', target)

# num_df['log_annual_inc_rg'] = pd.cut(num_df.log_annual_inc, 20)
# temp_df = woe_num_discrete(num_df, 'log_annual_inc_rg', target)
temp_df['%_n_obs'] = (temp_df.n_obs / temp_df.n_obs.sum()) * 100
temp_df['%_n_obs_cum'] = temp_df['%_n_obs'].cumsum()
temp_df['val_5%'] = np.where(temp_df.n_obs >= num_df.shape[0] * 0.05, 1, 0)
temp_df['val_10%'] = np.where(temp_df.n_obs >= num_df.shape[0] * 0.1, 1, 0)
temp_df['val_good'] = np.where(temp_df.n_good > 0, 1, 0)
temp_df['val_bad'] = np.where(temp_df.n_bad > 0, 1, 0)

(temp_df
 .iloc[:, [0,1,4,5,8,9,12,13,14,15,16,17]]
 .style
 .format({'n_good': '{0:.0f}',
          'n_bad': '{0:.0f}',
          '%_n_obs_cum': '{0:.2f}',
          '%_n_obs': '{0:.2f}'
          })
 .bar(subset=['%_n_obs'], color='purple')
 )
```

Out[ ]:

	log_annual_inc_rg	n_obs	n_good	n_bad	WoE	diff_prop_good	%_n_obs	%_n_obs_cum	val_5%	val_10%	val_good	va
0	(9.867, 10.949]	3278	2742	536	-0.047269	nan	49.25	49.25	1	1	1	
1	(10.949, 12.026]	3115	2657	458	0.078506	0.016484	46.80	96.05	1	1	1	
2	(12.026, 13.103]	241	195	46	-0.235219	0.043841	3.62	99.67	0	0	1	
3	(13.103, 14.18]	22	16	6	-0.698748	0.081856	0.33	100.00	0	0	1	

In [ ]:

```
plot_by_woe(temp_df, rotation_of_x_axis_labels = 90)
```



dti

In [ ]:

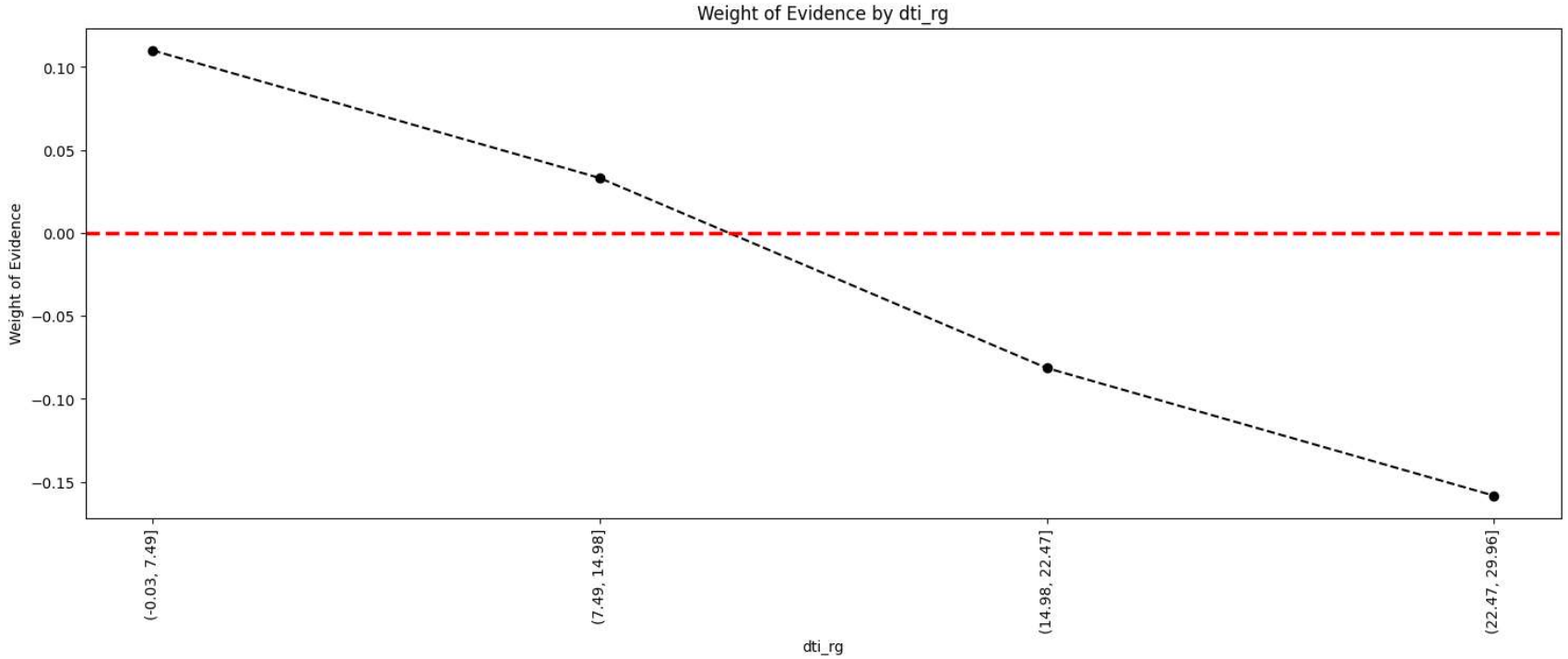
```
"""
dti range
------------------------
(0, 8]
(8, 15] (Base)
(15, 23]
(23, 100]
"""

num_df = x_train.select_dtypes(exclude=['object', 'category']).copy()
target = y_train.copy()
target = target + 1
target.loc[target == 2] = 0

num_df['dti_rg'] = pd.cut(num_df.dti, 4)
temp_df = woe_num_discrete(num_df, 'dti_rg', target)
```

```
temp_df['%_n_obs'] = (temp_df.n_obs / temp_df.n_obs.sum()) * 100
temp_df['%_n_obs_cum'] = temp_df['%_n_obs'].cumsum()
temp_df['val_5%'] = np.where(temp_df.n_obs >= num_df.shape[0] * 0.05, 1, 0)
temp_df['val_10%'] = np.where(temp_df.n_obs >= num_df.shape[0] * 0.1, 1, 0)
temp_df['val_good'] = np.where(temp_df.n_good > 0, 1, 0)
temp_df['val_bad'] = np.where(temp_df.n_bad > 0, 1, 0)

(temp_df
 .iloc[:, [0,1,4,5,8,9,12,13,14,15,16,17]]
 .style
 .format({'n_good': '{0:.0f}',
          'n_bad': '{0:.0f}',
          '%_n_obs_cum': '{0:.2f}',
          '%_n_obs': '{0:.2f}'
          })
 .bar(subset=['%_n_obs'], color='purple')
 )
```

Out[ ]:

	dti_rg	n_obs	n_good	n_bad	WoE	diff_prop_good	%_n_obs	%_n_obs_cum	val_5%	val_10%	val_good	val_bad
0	(-0.03, 7.49]	1825	1559	266	0.110032	nan	26.38	26.38	1	1	1	1
1	(7.49, 14.98]	2423	2046	377	0.033126	0.009839	35.02	61.40	1	1	1	1
2	(14.98, 22.47]	2079	1723	356	-0.081380	0.015644	30.05	91.44	1	1	1	1
3	(22.47, 29.96]	592	484	108	-0.158317	0.011196	8.56	100.00	1	0	1	1

In [ ]:
```
plot_by_woe(temp_df, rotation_of_x_axis_labels = 90)
```



fico

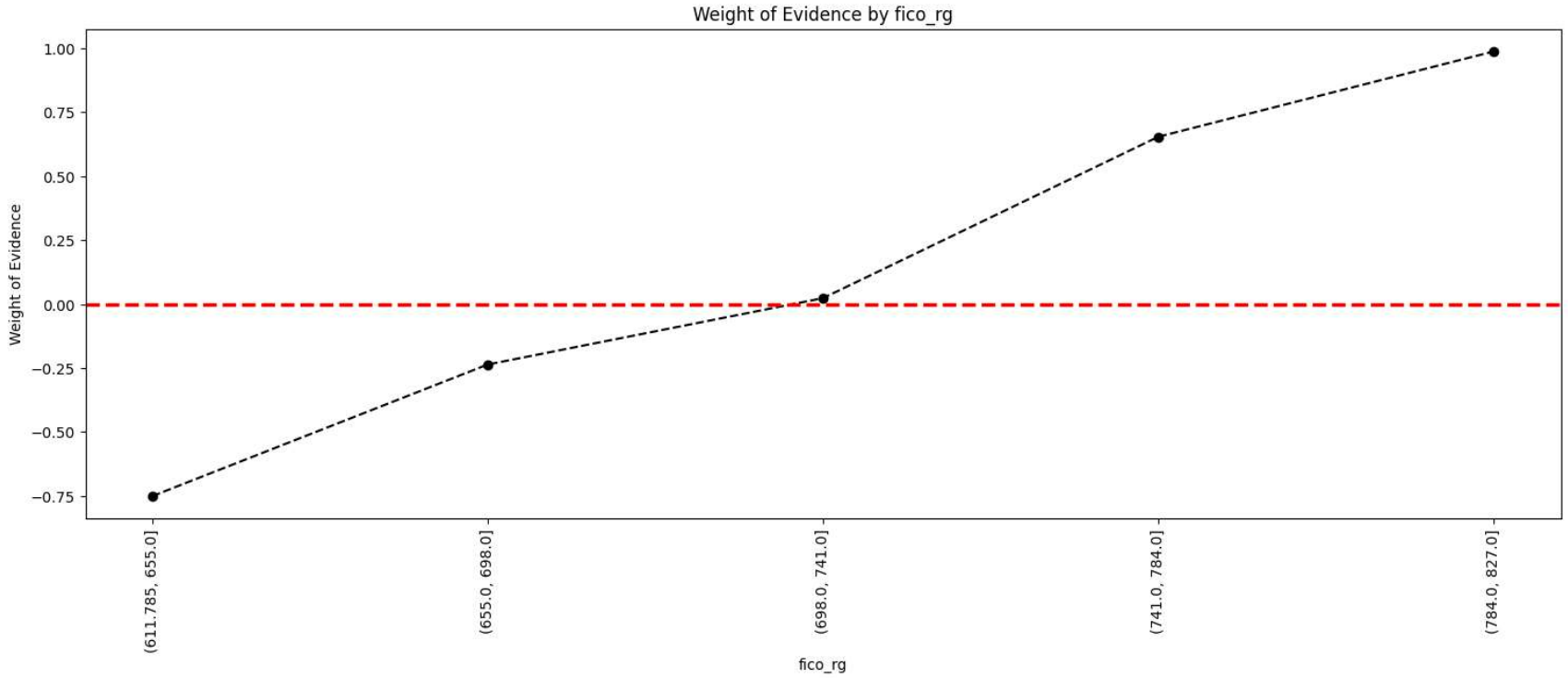
In [ ]:
```
"""
Fico range
------------------------
(0, 655]
(655, 698]
(698, 741] (Base)
(741, 784]
(784, 850]
"""

num_df = x_train.select_dtypes(exclude=['object', 'category']).copy()
target = y_train.copy()
target = target + 1
target.loc[target == 2] = 0

num_df['fico_rg'] = pd.cut(num_df.fico, 5)
temp_df = woe_num_discrete(num_df, 'fico_rg', target)
```

```
temp_df['%_n_obs'] = (temp_df.n_obs / temp_df.n_obs.sum()) * 100
temp_df['%_n_obs_cum'] = temp_df['%_n_obs'].cumsum()
temp_df['val_5%'] = np.where(temp_df.n_obs >= num_df.shape[0] * 0.05, 1, 0)
temp_df['val_10%'] = np.where(temp_df.n_obs >= num_df.shape[0] * 0.1, 1, 0)
temp_df['val_good'] = np.where(temp_df.n_good > 0, 1, 0)
temp_df['val_bad'] = np.where(temp_df.n_bad > 0, 1, 0)

(temp_df
 .iloc[:, [0,1,4,5,8,9,12,13,14,15,16,17]]
 .style
 .format({'n_good': '{0:.0f}',
          'n_bad': '{0:.0f}',
          '%_n_obs_cum': '{0:.2f}',
          '%_n_obs': '{0:.2f}'
          })
 .bar(subset=['%_n_obs'], color='purple')
 )
```

Out[ ]:

	fico_rg	n_obs	n_good	n_bad	WoE	diff_prop_good	%_n_obs	%_n_obs_cum	val_5%	val_10%	val_good	val_bad
0	(611.785, 655.0]	254	181	73	-0.750234	nan	3.67	3.67	0	0	1	1
1	(655.0, 698.0]	2764	2227	537	-0.235859	0.093118	39.95	43.62	1	1	1	1
2	(698.0, 741.0]	2289	1930	359	0.023682	0.037447	33.08	76.70	1	1	1	1
3	(741.0, 784.0]	1310	1192	118	0.654432	0.066761	18.93	95.64	1	1	1	1
4	(784.0, 827.0]	302	282	20	0.987904	0.023851	4.36	100.00	0	0	1	1

In [ ]:
```
plot_by_woe(temp_df, rotation_of_x_axis_labels = 90)
```



revol_util

In [ ]:

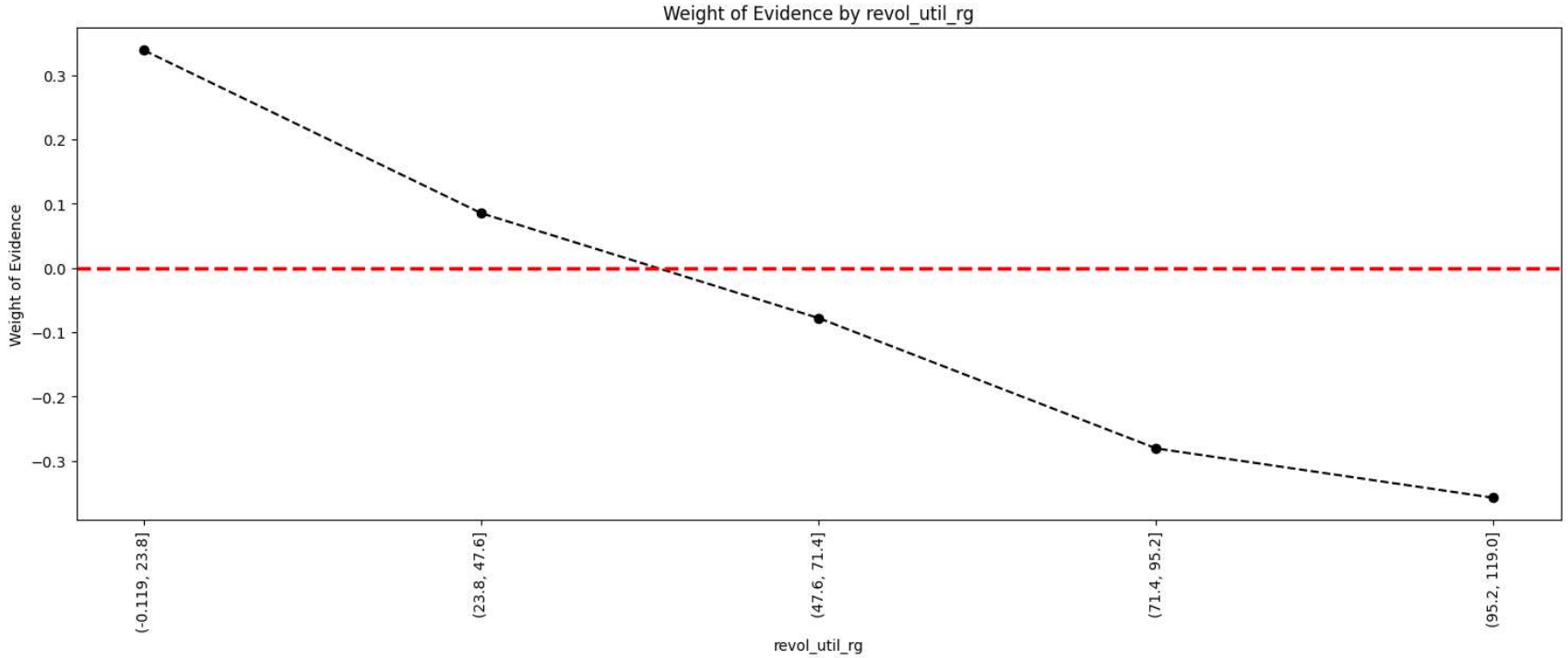
```
"""
Revol util ranges
-------------------------
(0, 25]
(25, 50]
(50, 70] (Base)
(70, 95]
(95, inf]
"""

num_df = x_train.select_dtypes(exclude=['object', 'category']).copy()
target = y_train.copy()
target = target + 1
target.loc[target == 2] = 0

num_df['revol_util_rg'] = pd.cut(num_df.revol_util, 5)
```

```
temp_df = woe_num_discrete(num_df, 'revol_util_rg', target)
temp_df['%_n_obs'] = (temp_df.n_obs / temp_df.n_obs.sum()) * 100
temp_df['%_n_obs_cum'] = temp_df['%_n_obs'].cumsum()
temp_df['val_5%'] = np.where(temp_df.n_obs >= num_df.shape[0] * 0.05, 1, 0)
temp_df['val_10%'] = np.where(temp_df.n_obs >= num_df.shape[0] * 0.1, 1, 0)
temp_df['val_good'] = np.where(temp_df.n_good > 0, 1, 0)
temp_df['val_bad'] = np.where(temp_df.n_bad > 0, 1, 0)

(temp_df
 .iloc[:, [0,1,4,5,8,9,12,13,14,15,16,17]]
 .style
 .format({'n_good': '{0:.0f}',
          'n_bad': '{0:.0f}',
          '%_n_obs_cum': '{0:.2f}',
          '%_n_obs': '{0:.2f}'
          })
 .bar(subset=['%_n_obs'], color='purple')
 )
```

Out[ ]:

	revol_util_rg	n_obs	n_good	n_bad	WoE	diff_prop_good	%_n_obs	%_n_obs_cum	val_5%	val_10%	val_good	val_bad
0	(-0.119, 23.8]	1815	1598	217	0.338340	nan	26.23	26.23	1	1	1	1
1	(23.8, 47.6]	1726	1469	257	0.084990	0.029340	24.95	51.18	1	1	1	1
2	(47.6, 71.4]	1663	1379	284	-0.078131	0.021877	24.04	75.21	1	1	1	1
3	(71.4, 95.2]	1430	1142	288	-0.280695	0.030623	20.67	95.88	1	1	1	1
4	(95.2, 119.0]	285	224	61	-0.357499	0.012636	4.12	100.00	0	0	1	1

In [ ]:
```
plot_by_woe(temp_df, rotation_of_x_axis_labels = 90)
```



inq_last_6mths

In [ ]:

```
"""
Revol util ranges
-------------------------
0
1-2
2+
"""

num_df = x_train.select_dtypes(exclude=['object', 'category']).copy()
target = y_train.copy()
target = target + 1
target.loc[target == 2] = 0

num_df2_temp = num_df.loc[(num_df.inq_last_6mths < 2), :].copy()
num_df2_temp['inq_last_6mths_rg'] = pd.cut(num_df2_temp.inq_last_6mths, 2)
temp_df = woe_num_discrete(num_df2_temp, 'inq_last_6mths_rg', target)
```

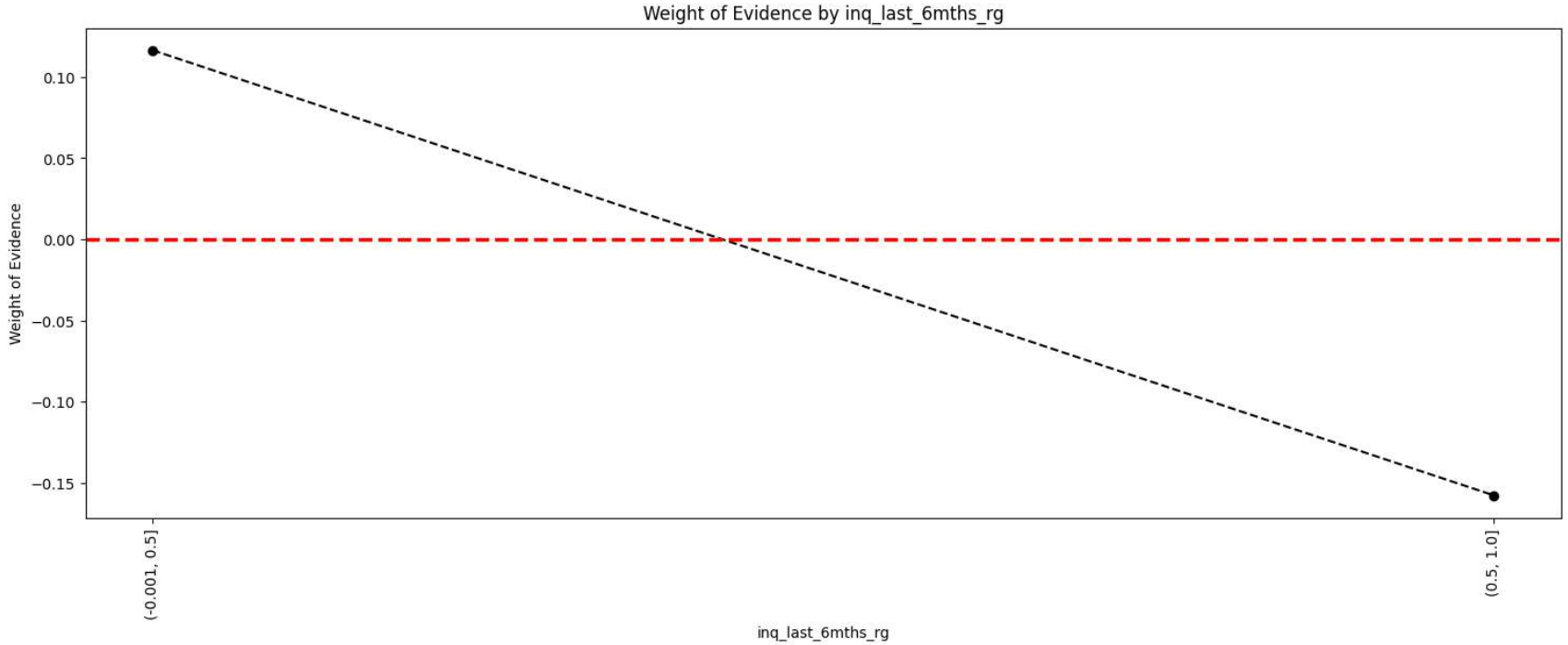
```
# num_df['inq_last_6mths_rg'] = pd.cut(num_df.inq_last_6mths, 2)
# temp_df = woe_num_discrete(num_df, 'inq_last_6mths_rg', target)
temp_df['%_n_obs'] = (temp_df.n_obs / temp_df.n_obs.sum()) * 100
temp_df['%_n_obs_cum'] = temp_df['%_n_obs'].cumsum()
temp_df['val_5%'] = np.where(temp_df.n_obs >= num_df.shape[0] * 0.05, 1, 0)
temp_df['val_10%'] = np.where(temp_df.n_obs >= num_df.shape[0] * 0.1, 1, 0)
temp_df['val_good'] = np.where(temp_df.n_good > 0, 1, 0)
temp_df['val_bad'] = np.where(temp_df.n_bad > 0, 1, 0)

(temp_df
 .iloc[:, [0,1,4,5,8,9,12,13,14,15,16,17]]
 .style
 .format({'n_good': '{0:.0f}',
          'n_bad': '{0:.0f}',
          '%_n_obs_cum': '{0:.2f}',
          '%_n_obs': '{0:.2f}'
          })
 .bar(subset=['%_n_obs'], color='purple')
 )
```

Out[ ]:

	inq_last_6mths_rg	n_obs	n_good	n_bad	WoE	diff_prop_good	%_n_obs	%_n_obs_cum	val_5%	val_10%	val_good	va
0	(-0.001, 0.5]	2653	2333	320	0.116481	nan	59.94	59.94	1	1	1	
1	(0.5, 1.0]	1773	1502	271	-0.157675	0.032230	40.06	100.00	1	1	1	

In [ ]:

```
plot_by_woe(temp_df, rotation_of_x_axis_labels = 90)
```



Save Train and Test sets

In [ ]:

```
#======================
# Final datasets
#======================

# Training
cols = ['credit_policy', 'int_rate', 'installment', 'dti',
        'fico', 'revol_util', 'inq_last_6mths', 'log_annual_inc',
        'purpose']
x_train = x_train.loc[:, cols]
x_train['maha_out'] = outlier_df.maha_out

# Drop outliers
x_train = x_train.loc[x_train.maha_out == 0, cols]
# Add target variable
x_train['default'] = y_train
```

```
# Validation
x_val = x_val.loc[:, cols]
x_val['default'] = y_val

# Test
x_test = x_test.loc[:, cols]
x_test['default'] = y_test
```

In [ ]:

```
# Training set
x_train.to_parquet('../data/processed/training.gzip', compression='gzip')
# Validation set
x_val.to_parquet('../data/processed/validation.gzip', compression='gzip')
# Test set
x_test.to_parquet('../data/processed/test.gzip', compression='gzip')
```

Modeling

Load Libraries

In [ ]:

```
# Librerias base
import os
import re
import json

# Manipulacion de datos
import pandas as pd
import numpy as np

# Visualizacion
import matplotlib.pyplot as plt
import seaborn as sns
from IPython.display import clear_output
import collections
from pprint import pprint

# Pipelines
from sklearn.pipeline import Pipeline
from imblearn.pipeline import Pipeline as imbPipeline

# Preprocesamiento
from sklearn.preprocessing import MinMaxScaler, StandardScaler, OneHotEncoder
from sklearn.decomposition import PCA
from sklearn.linear_model import Lasso, LassoCV
from imblearn.over_sampling import SMOTE, RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.feature_selection import SelectFromModel
from sklearn.utils.validation import check_is_fitted
from sklearn.compose import ColumnTransformer, make_column_selector
from sklearn.model_selection import train_test_split

# Datos faltantes
```

```
from sklearn.impute import KNNImputer, SimpleImputer

# Metrics
from sklearn.metrics import confusion_matrix, roc_auc_score
from sklearn.metrics import precision_recall_curve, roc_curve
from sklearn.metrics import fbeta_score, recall_score, precision_score
from sklearn.metrics import mean_squared_error, r2_score

# Modelamiento
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
import xgboost as xgb

# Serializar modelo
import pickle

# Optimización Hiperparametros
from sklearn.model_selection import GridSearchCV
from bayes_opt import BayesianOptimization
from bayes_opt.logger import JSONLogger
from bayes_opt.event import Events
from sklearn.model_selection import KFold

# Configurar pandas
pd.options.display.max_columns = None
pd.options.display.max_rows = 200
#pd.options.display.float_format = '{:,.2f}'.format
```

Load Data

In [ ]:
```
# Training set
train = pd.read_parquet('../data/processed/training.gzip')
# Validation set
val = pd.read_parquet('../data/processed/validation.gzip')
# Test set
test = pd.read_parquet('../data/processed/test.gzip')
```

Datasets preparation

In [ ]:
```
x_train = train.drop('default', axis=1)
y_train = train.default

x_val = val.drop('default', axis=1)
y_val = val.default

x_test = test.drop('default', axis=1)
y_test = test.default
```

In [ ]:
```
print(x_train.shape)
print(x_val.shape)
print(x_test.shape)
```

```
(6780, 9)
(1222, 9)
(1437, 9)
```

Custom class for discretizing numercial variables

In [ ]:
```
class NumBinning(BaseEstimator, TransformerMixin):
    def __init__(self, imputer_strategy, knn_imputer):
        # self.X = X
        self.imputer_strategy = imputer_strategy
        self.knn_imputer = knn_imputer

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        # Selet numerical variables
        X = X.select_dtypes(exclude=['object', 'category'])
        num_cols = X.columns.values
        num_index = X.index.values

        # Impute missing data
        imputer_strategy = "mean" if self.imputer_strategy > 0.5 else "median"
        if self.knn_imputer > 0.5:
            imputer = KNNImputer(n_neighbors=int(5))
        else:
            imputer = SimpleImputer(strategy=imputer_strategy, missing_values=np.nan)
```

```
pipe = Pipeline(steps=[('imputer', imputer)])
X = pipe.fit_transform(X)
X = pd.DataFrame(X, columns=num_cols, index=num_index)

X_new = pd.DataFrame(index=X.index.values)
# Int rate
X_new['int_rate:(0.0, 0.0823]'] = np.where(X.int_rate <= 0.0823, 1, 0)
X_new['int_rate:(0.0823, 0.105]'] = np.where((X.int_rate > 0.0823) & (X.int_rate <= 0.105), 1, 0)
X_new['int_rate:(0.105, 0.127]'] = np.where((X.int_rate > 0.105) & (X.int_rate <= 0.127), 1, 0)
X_new['int_rate:(0.127, 0.149]'] = np.where((X.int_rate > 0.127) & (X.int_rate <= 0.149), 1, 0)
X_new['int_rate:(0.149, 0.172]'] = np.where((X.int_rate > 0.149) & (X.int_rate <= 0.172), 1, 0)
X_new['int_rate:(0.172, 0.194]'] = np.where((X.int_rate > 0.172) & (X.int_rate <= 0.194), 1, 0)
X_new['int_rate:0.194+'] = np.where(X.int_rate > 0.194, 1, 0)
# Installments
X_new['installment:(0, 147.737]'] = np.where(X.installment <= 147.737, 1, 0)
X_new['installment:(147.737, 279.804]'] = np.where((X.installment > 147.737) & (X.installment <= 279.804), 1,
X_new['installment:(279.804, 411.871]'] = np.where((X.installment > 279.804) & (X.installment <= 411.871), 1,
X_new['installment:(411.871, 676.006]'] = np.where((X.installment > 411.871) & (X.installment <= 676.006), 1,
X_new['installment:(676.006, 808.073]'] = np.where((X.installment > 676.006) & (X.installment <= 808.073), 1,
X_new['installment:808.073+'] = np.where(X.installment > 808.073, 1, 0)
# log_annual_inc
X_new['income:(0, 10.949]'] = np.where(X.log_annual_inc > 10.949, 1, 0)
X_new['income:(10.949, 12.026]'] = np.where((X.log_annual_inc > 10.949) & (X.log_annual_inc <= 12.026), 1, 0)
X_new['income:(12.026, 13.103]'] = np.where((X.log_annual_inc > 12.026) & (X.log_annual_inc <= 13.103), 1, 0)
X_new['income:13.103+'] = np.where(X.log_annual_inc > 13.103, 1, 0)
# dti
X_new['dti:(0, 8]'] = np.where(X.dti <= 8, 1, 0)
X_new['dti:(8, 15]'] = np.where((X.dti > 8) & (X.dti <= 15), 1, 0)
X_new['dti:(15, 23]'] = np.where((X.dti > 15) & (X.dti <= 23), 1, 0)
X_new['dti:23+'] = np.where(X.dti > 23, 1, 0)
# Fico
X_new['fico:(0, 655]'] = np.where(X.fico <= 655, 1, 0)
X_new['fico:(655, 698]'] = np.where((X.fico > 655) & (X.fico <= 698), 1, 0)
X_new['fico:(698, 741]'] = np.where((X.fico > 698) & (X.fico <= 741), 1, 0)
X_new['fico:(741, 784]'] = np.where((X.fico > 741) & (X.fico <= 784), 1, 0)
X_new['fico:784+'] = np.where(X.fico > 784, 1, 0)
# revol_util
X_new['revol_util:(0, 25]'] = np.where(X.revol_util > 25, 1, 0)
X_new['revol_util:(25, 50]'] = np.where((X.revol_util > 25) & (X.revol_util <= 50), 1, 0)
X_new['revol_util:(50, 70]'] = np.where((X.revol_util > 50) & (X.revol_util <= 70), 1, 0)
X_new['revol_util:(70, 95]'] = np.where((X.revol_util > 70) & (X.revol_util <= 95), 1, 0)
```

```
        X_new['revol_util:95+'] = np.where(X.revol_util > 95, 1, 0)
        # inq_last_6mths
        X_new['inquiries:0'] = np.where(X.inq_last_6mths == 0, 1, 0)
        X_new['inquiries:1-2'] = np.where((X.inq_last_6mths > 0) & (X.inq_last_6mths <= 2), 1, 0)
        X_new['inquiries:2+'] = np.where(X.inq_last_6mths > 2, 1, 0)
        # credit_policy
        X_new['cred_pol:0'] = np.where(X.credit_policy == 0, 1, 0)
        X_new['cred_pol:1'] = np.where(X.credit_policy == 1, 1, 0)

        # Columnas de referencia
        ref_cols = ['int_rate:(0.105, 0.127]', 'installment:(279.804, 411.871]',
                    'income:(0, 10.949]', 'dti:(8, 15]', 'fico:(698, 741]',
                    'revol_util:(50, 70]', 'cred_pol:0', 'inquiries:0'
                    ]

        X_new = X_new.drop(ref_cols, axis=1)
        return X_new
```

Custom class for preprocessing categorical variables

In [ ]:
```
def dummy_creation(df, columns_list):
    """Funcion para crear variables dummies
    """
    df_dummies = []
    for col in columns_list:
        df_dummies.append(pd.get_dummies(df[col], prefix = col, prefix_sep = ':'))
    df_dummies = pd.concat(df_dummies, axis = 1)
    df = pd.concat([df, df_dummies], axis = 1)
    return df
```

In [ ]:
```
class CatDummies(BaseEstimator, TransformerMixin):
    def __init__(self, cat_ref=None) -> None:
        # self.X = X
        self.columns = None
        self.cat_ref = cat_ref
        return None

    def fit(self, X, y=None):
        self.columns = X.columns
        return self
```

```
    def transform(self, X):
        # Seleccionar variables categoricas
        X = X.select_dtypes(include=['object', 'category'])
        cat_cols = X.columns.values
        cat_index = X.index.values

        # Impute missing values
        pipe = Pipeline(steps=[('imputer', SimpleImputer(strategy="most_frequent", missing_values=np.nan))])
        X = pipe.fit_transform(X)
        X = pd.DataFrame(X, columns=cat_cols, index=cat_index)

        # Preparar dataframe
        X = dummy_creation(X, cat_cols)
        if self.cat_ref is not None:
            X = X.reindex(labels=self.cat_ref, axis=1, fill_value=0)

        X_new = X.loc[:, ['purpose:small_business', 'purpose:credit_card', 'purpose:major_purchase']].copy()
        X_new.loc[:, 'purpose:education_home_improv'] = sum([X.loc[:, 'purpose:educational'],
                                                             X.loc[:, 'purpose:home_improvement']])
        X_new.loc[:,'purpose:other_debt_consolidation'] = sum([X.loc[:, 'purpose:all_other'],
                                                               X.loc[:, 'purpose:debt_consolidation']
                                                               ])
        # X_new.loc[:,'purpose:credit_card'] = X.loc[:, 'purpose:credit_card']
        # X_new.loc[:,'purpose:major_purchase'] = X.loc[:, 'purpose:major_purchase']

        # Base category column
        ref_cols = ['purpose:education_home_improv']

        X_new = X_new.drop(ref_cols, axis=1)
        X_new = X_new * 1

        return X_new

    def get_feature_names(self):
        return list(self.columns)
```

Modeling

Aux Functions

In [ ]:

```python
def live_plot(data_dict, figsize=(7,5), title='', win_size: int = 100):
    """
    Function for plotting in real time bayesian optimization process
    """
    clear_output(wait=True)
    plt.figure(figsize=figsize)
    for label,data in data_dict.items():
        if len(data) > win_size:
            data = data[-win_size:]
            iterations = np.arange(len(data))[-win_size:]
        else:
            iterations = np.arange(len(data))
        plt.plot(iterations, data, label=label)
    plt.title(title)
    plt.grid(True)
    plt.xlabel('Iteration')
    plt.legend(loc='center left') # the plot evolves to the right
    plt.show()

def adj_live_plot(data_dict, subplot, figsize=(7,5), title='', win_size: int = 100):
    """
    Function for plotting in real time bayesian optimization process
    """
    clear_output(wait=True)
    plt.subplot(*subplot)
    for label,data in data_dict.items():
        if len(data) > win_size:
            data = data[-win_size:]
            iterations = np.arange(len(data))[-win_size:]
        else:
            iterations = np.arange(len(data))
        plt.plot(iterations, data, label=label)
    plt.title(title)
    plt.grid(True)
    plt.xlabel('Iteration')
    plt.legend(loc='center left') # the plot evolves to the right

def add_model(data_pipeline, model, smote=False) -> Pipeline:
    """Function for createing whole pipeline for model training

        Args
```

```
        ----
        data_pipeline: sklearn.pipeline
        model: sklearn model
        smote: boolean, default False
    """
    if smote:
        whole_pipeline = imbPipeline([
            ("data_pipeline", data_pipeline),
            ("model", model)
        ])
    else:
        whole_pipeline = Pipeline([
            ("data_pipeline", data_pipeline),
            ("model", model)
            ])
    return whole_pipeline
```

Función para entrenar Regresión Logística

In [ ]:

```
# Data para visualizacion de la optimizacion
roc_auc_data = collections.defaultdict(list)
ks_data = collections.defaultdict(list)
gini_data = collections.defaultdict(list)

cat_col_ref = ['purpose:all_other', 'purpose:credit_card',
               'purpose:debt_consolidation', 'purpose:educational',
               'purpose:home_improvement', 'purpose:major_purchase',
               'purpose:small_business']

# Funcion para entrenar y evaluar el modelo con KFold
def train_and_evaluate_rl(
                        # Data
                        scaler_choice,
                        imputer_strategy,
                        knn_imputer,
                        knn_imputer_k,
                        pca_components,
                        model_penalty,
                        model_C,
                        model_pos_class_weight,
                        sampling_strategy,
```

```
                    lasso_alpha
                    ) -> float:
"""  Funcion para entrenamiento y validacion del modelo

Args:
    scaler_choice (_type_): _description_
    imputer_strategy (_type_): _description_
    knn_imputer (_type_): _description_
    knn_imputer_k (_type_): _description_
    pca_components (_type_): _description_
    model_penalty (_type_): _description_
    model_C (_type_): _description_
    model_pos_class_weight (_type_): _description_
    lasso_alpha

Returns:
    float: ROC AUC score
    float: KS score
    float: GINI score
"""

#================================
# Configuracion inicial
#================================

#======================
# Configuraciones
#======================
custom_binning = custom_binning_conf
smote = smote_conf
custom_categorical = custom_categorical_conf

# Modelo a utilizar
model_selection = 1
cv = cv_conf

# Steps pipeline preprocesor
feature_eng_step = feature_eng_step_conf

# Mostrar grafico optimizacion
show_live_plot=show_live_plot_cong
```

```
#========
# Datos
#========
X_train = x_data.copy()
Y_train = y_data.copy()

# Definicion de columnas numericas y categoricas
cat_vars = [*X_train.select_dtypes(include='object').columns]
num_vars = [*X_train.select_dtypes(exclude='object').columns]

# Estrategia de escalamiento de datos
scaler_cls = StandardScaler if scaler_choice > 0.5 else MinMaxScaler

# Estrategia de imputacion de datos numericos
impute_strategy = "mean" if imputer_strategy > 0.5 else "median"
if knn_imputer > 0.5:
    imputer = KNNImputer(n_neighbors=int(knn_imputer_k))
else:
    imputer = SimpleImputer(strategy=impute_strategy)


#=======================================
# Seleccion del modelo
#=======================================
print("Modelo seleccionado: Regresión Logistica")
model =  LogisticRegression(
    penalty="l1" if model_penalty > 0.5 else "l2",
    solver='liblinear',
    C=model_C,
    class_weight={0: 1, 1: model_pos_class_weight}
)

#=======================================
# Preprocesamientos
#=======================================
if custom_binning:

    # Numerical variables transformations
    numeric_transformer = Pipeline(
        steps=[
            ("num_binning", NumBinning(imputer_strategy, knn_imputer))
            ]
```

```
        )

        # Categorical variables transformations
        categorical_transformer = Pipeline(
            steps=[
                ('cat_dummy', CatDummies(cat_ref=cat_col_ref))
                ]
        )

        # Transformer consolidado
        preprocessor = ColumnTransformer(
            transformers=[
                ("num", numeric_transformer, num_vars),
                ("cat", categorical_transformer, cat_vars)
            ],
            remainder='passthrough'
        )

        if smote:
            data_pipeline = imbPipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                        ('model', model)
                                    ])
        else:
            data_pipeline = Pipeline(steps=[
                                    ("data_processor", preprocessor),
                                    ('model', model)
                                ])

    else:

        # Categorical variables transformations
        if custom_categorical:
            categorical_transformer = Pipeline(
                steps=[('cat_dummy', CatDummies(cat_ref=cat_col_ref))]
            )
        else:
            categorical_transformer = Pipeline(
                steps=[
                    ("imputer", SimpleImputer(strategy="most_frequent")),
                    ('ohe', OneHotEncoder(drop='first', handle_unknown="ignore"))
```

```
            ]
        )

    # Base transformations
    transformers = []

    if feature_eng_step is None:
        numeric_transformer = Pipeline(
                              steps=[
                                  ("imputer", imputer)
                                  #("scaler", scaler_cls())
                              ])

        # Categorical variables transformations
        transformers.append(("cat", categorical_transformer, cat_vars))

        # Numerical variables transformations
        transformers.append(("num", numeric_transformer, num_vars))

        preprocessor = ColumnTransformer(
            transformers=transformers,
            remainder='passthrough'
        )

        if smote:
            data_pipeline = imbPipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                        ('model', model)
                                    ])
        else:
            data_pipeline = Pipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('model', model)
                                    ])

    elif feature_eng_step == 'pca':

        numeric_transformer = Pipeline(
                    steps=[
                        ("imputer", imputer),
                        ("scaler", scaler_cls()),
```

```
                        ("pca", PCA(n_components=int(pca_components)))
                    ])

        # Categorical variables transformations
        transformers.append(("cat", categorical_transformer, cat_vars))

        # Numerical variables transformations
        transformers.append(("num", numeric_transformer, num_vars))

        preprocessor = ColumnTransformer(
            transformers=transformers,
            remainder='passthrough'
        )

        if smote:
            data_pipeline = imbPipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                        ('model', model)
                                    ])
        else:
            data_pipeline = Pipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('model', model)
                                    ])

    elif feature_eng_step == 'lda':
        numeric_transformer = Pipeline(
                    steps=[
                        ("imputer", imputer),
                        ("scaler", scaler_cls()),
                        ("lda", LDA(n_components=1))
                    ])

        # Numerical variables transformations
        transformers.append(("num", numeric_transformer, num_vars))

        # Categorical variables transformations
        transformers.append(("cat", categorical_transformer, cat_vars))

        preprocessor = ColumnTransformer(
            transformers=transformers,
```

```
            remainder='passthrough'
        )

        if smote:
            data_pipeline = imbPipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                        ('model', model)
                                    ])
        else:
            data_pipeline = Pipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('model', model)
                                    ])

    elif feature_eng_step == 'lasso':

        numeric_transformer = Pipeline(
                                    steps=[
                                        ("imputer", imputer),
                                        ("scaler", scaler_cls())
                                    ])

        # Categorical variables transformations
        transformers.append(("cat", categorical_transformer, cat_vars))

        # Numerical variables transformations
        transformers.append(("num", numeric_transformer, num_vars))

        preprocessor = ColumnTransformer(
            transformers=transformers,
            remainder='passthrough'
        )

        # Apply lasso if the model is not a logistic regression
        if model_selection == 1:
            if smote:
                data_pipeline = imbPipeline(steps=[
                                            ("data_processor", preprocessor),
                                            ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                            ('model', model)
                                        ])
```

```
                else:
                    data_pipeline = Pipeline(steps=[
                                            ("data_processor", preprocessor),
                                            ('model', model)
                                        ])
            else:
                if smote:
                    data_pipeline = imbPipeline(steps=[
                                            ("data_processor", preprocessor),
                                            ("lasso", SelectFromModel(Lasso(alpha=lasso_alpha))),
                                            ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                            ('model', model)
                                        ])
                else:
                    data_pipeline = Pipeline(steps=[
                                            ("data_processor", preprocessor),
                                            ("lasso", SelectFromModel(Lasso(alpha=lasso_alpha))),
                                            ('model', model)
                                        ])

        elif feature_eng_step == 'pca_lda':

            pca_transformer = Pipeline(
                        steps=[
                            ("imputer", imputer),
                            ("scaler", scaler_cls()),
                            ("pca", PCA(n_components=int(pca_components)))
                        ])

            lda_transformer = Pipeline(
                        steps=[
                            ("imputer", imputer),
                            ("scaler", scaler_cls()),
                            ("lda", LDA(n_components=1))
                        ])

            # Numerical variables transformations
            transformers.append(("pca", pca_transformer, num_vars))
            transformers.append(("lda", lda_transformer, num_vars))

            # Categorical variables transformations
            transformers.append(("cat", categorical_transformer, cat_vars))
```

```
        preprocessor = ColumnTransformer(
            transformers=transformers,
            remainder='passthrough'
        )

        if smote:
            data_pipeline = imbPipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                        ('model', model)
                                    ])
        else:
            data_pipeline = Pipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('model', model)
                                    ])

    elif feature_eng_step == 'all':
        pca_transformer = Pipeline(
                    steps=[
                        ("imputer", imputer),
                        ("scaler", scaler_cls()),
                        ("pca", PCA(n_components=int(pca_components)))
                    ])

        lda_transformer = Pipeline(
                    steps=[
                        ("imputer", imputer),
                        ("scaler", scaler_cls()),
                        ("lda", LDA(n_components=1))
                    ])

        # Numerical variables transformations
        transformers.append(("pca", pca_transformer, num_vars))
        transformers.append(("lda", lda_transformer, num_vars))

        # Categorical variables transformations
        transformers.append(("cat", categorical_transformer, cat_vars))

        preprocessor = ColumnTransformer(
            transformers=transformers,
```

```
            remainder='passthrough'
        )

        if smote:
            data_pipeline = imbPipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ("lasso", SelectFromModel(Lasso(alpha=lasso_alpha))),
                                        ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                        ('model', model)
                                    ])
        else:
            data_pipeline = Pipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('lasso', SelectFromModel(Lasso(alpha=lasso_alpha))),
                                        ('model', model)
                                    ])

#==============================
# Model Pipeline
#==============================
#pipeline = add_model(data_pipeline, model, smote)
kf = KFold(n_splits=cv, random_state=42, shuffle=True)

#=========================
# Metrics
#=========================
# Train
roc_auc_train = []
ks_train = []
gini_train = []
# Validation
roc_auc_val = []
ks_val = []
gini_val = []

# K-Fold cross val
for i, (train_index, test_index) in enumerate(kf.split(X_train)):
    # print(f"Fold number: {i+1}")
    kX_train, kX_val = X_train.iloc[train_index], X_train.iloc[test_index]
    ky_train, ky_val = Y_train.iloc[train_index], Y_train.iloc[test_index]

    data_pipeline.fit(kX_train, ky_train.astype(np.float32))
```

```
        val_preds = data_pipeline.predict_proba(kX_val)[:, 1]
        val_auc = roc_auc_score(ky_val.astype(np.float32), val_preds)
        val_fpr, val_tpr, _ = roc_curve(ky_val.astype(np.float32), val_preds)
        val_ks = np.max(abs(val_fpr - val_tpr))
        val_gini = 2 * val_auc - 1

        train_preds = data_pipeline.predict_proba(kX_train)[:, 1]
        train_auc = roc_auc_score(ky_train.astype(np.float32), train_preds)
        train_fpr, train_tpr, _ = roc_curve(ky_train.astype(np.float32), train_preds)
        train_ks = np.max(abs(train_fpr - train_tpr))
        train_gini = 2 * train_auc - 1

        # Add train values
        roc_auc_train.append(train_auc)
        ks_train.append(train_ks)
        gini_train.append(train_gini)

        # Add validation values
        roc_auc_val.append(val_auc)
        ks_val.append(val_ks)
        gini_val.append(val_gini)

# Convert to an array
# Validation
roc_auc_val = np.array(roc_auc_val)
ks_val = np.array(ks_val)
gini_val = np.array(gini_val)
# Training
roc_auc_train = np.array(roc_auc_train)
ks_train = np.array(ks_train)
gini_train = np.array(gini_train)

# ROC AUC ajusted
adj_val_roc_auc = (roc_auc_val.mean() -  roc_auc_val.std())
adj_train_roc_auc = (roc_auc_train.mean() -  roc_auc_train.std())
obj_roc_auc = adj_val_roc_auc - abs(adj_val_roc_auc - adj_train_roc_auc)

# KS ajusted
adj_val_ks = (ks_val.mean() - ks_val.std())
adj_train_ks = (ks_train.mean() - ks_train.std())
obj_ks = adj_val_ks - abs(adj_val_ks - adj_train_ks)
```

```
    # Gini ajusted
    adj_val_gini = (gini_val.mean() - gini_val.std())
    adj_train_gini = (gini_train.mean() - gini_train.std())
    obj_gini = adj_val_gini - abs(adj_val_gini - adj_train_gini)

    print(f"Validation ROC AUC adjusted score: {adj_val_roc_auc}")
    print(f"Train ROC AUC adjusted score: {adj_train_roc_auc}")
    print("\n")
    print(f"Validation KS adjusted score: {adj_val_ks}")
    print(f"Train KS adjusted score: {adj_train_ks}")
    print("\n")
    print(f"Validation GINI adjusted score: {adj_val_gini}")
    print(f"Train GINI adjusted score: {adj_train_gini}")
    print("\n")

    if show_live_plot:
        # ROC-AUC
        roc_auc_data['train_roc_auc'].append(adj_train_roc_auc)
        roc_auc_data['val_roc_auc'].append(adj_val_roc_auc)
        roc_auc_data['objective'].append(obj_roc_auc)
        # KS
        ks_data['train_ks'].append(adj_train_ks)
        ks_data['val_ks'].append(adj_val_ks)
        ks_data['objective'].append(obj_ks)
        # Gini
        gini_data['train_gini'].append(adj_train_gini)
        gini_data['val_gini'].append(adj_val_gini)
        gini_data['objective'].append(obj_gini)
        # live_plot(roc_auc_data)
        fig, axes = plt.subplots(3, 1, figsize=(12, 8))
        adj_live_plot(roc_auc_data, subplot=(3, 1, 1), win_size=160)
        adj_live_plot(ks_data, subplot=(3, 1, 2), win_size=160)
        adj_live_plot(gini_data, subplot=(3, 1, 3), win_size=160)
        plt.tight_layout()
        plt.show()

    return data_pipeline, obj_roc_auc
```

Función para entrenar Random Forest

In [ ]:

```python
# Data para visualizacion de la optimizacion
roc_auc_data = collections.defaultdict(list)
ks_data = collections.defaultdict(list)
gini_data = collections.defaultdict(list)

cat_col_ref = ['purpose:all_other', 'purpose:credit_card',
               'purpose:debt_consolidation', 'purpose:educational',
               'purpose:home_improvement', 'purpose:major_purchase',
               'purpose:small_business']

# Funcion para entrenar y evaluar el modelo con KFold
def train_and_evaluate_rf(
                        # Data
                        scaler_choice,
                        imputer_strategy,
                        knn_imputer,
                        knn_imputer_k,
                        pca_components,
                        model_penalty,
                        sampling_strategy,
                        lasso_alpha,
                        # Random Forest
                        rf_n_estimators,
                        rf_max_depth,
                        rf_max_features,
                        rf_min_samples_split,
                        rf_criterion,
                        model_pos_class_weight
                        ) -> float:
    """  Funcion para entrenamiento y validacion del modelo

    Args:
        scaler_choice (_type_): _description_
        imputer_strategy (_type_): _description_
        knn_imputer (_type_): _description_
        knn_imputer_k (_type_): _description_
        pca_components (_type_): _description_
        model_penalty (_type_): _description_
        model_C (_type_): _description_
        model_pos_class_weight (_type_): _description_
        model_hidden_layer_size_exp (_type_): _description_
```

```
    model_lr_init (_type_): _description_
    model_alpha (_type_): _description_
    model_batch_size (_type_): _description_
    model_max_iter (_type_): _description_
    model_solver (_type_): _description_
    verbose (int, optional): _description_. Defaults to 1.


    model_selection: Indica el modelo a entrenar
                    1 - Regresion logística
                    2 - SVM
                    3 - RED NUERONAL
                    4 - Random Forest
                    5 - XGboost

Returns:
    float: ROC AUC score
    float: KS score
    float: GINI score
"""

#================================
# Configuracion inicial
#================================

#======================
# Configuraciones
#======================
custom_binning = custom_binning_conf
smote = smote_conf
custom_categorical = custom_categorical_conf

# Modelo a utilizar
model_selection = 2
cv = cv_conf

# Steps pipeline preprocesor
feature_eng_step = feature_eng_step_conf

# Mostrar grafico optimizacion
show_live_plot=show_live_plot_cong
```

```python
#========
# Datos
#========
X_train = x_data.copy()
Y_train = y_data.copy()

# Numerical and categorical variables
cat_vars = [*X_train.select_dtypes(include='object').columns]
num_vars = [*X_train.select_dtypes(exclude='object').columns]

# Scaling strategy
scaler_cls = StandardScaler if scaler_choice > 0.5 else MinMaxScaler

# Imputation strategy
impute_strategy = "mean" if imputer_strategy > 0.5 else "median"
if knn_imputer > 0.5:
    imputer = KNNImputer(n_neighbors=int(knn_imputer_k))
else:
    imputer = SimpleImputer(strategy=impute_strategy)


#========================================
# Model
#========================================
print("Modelo seleccionado: Random Forest")

if rf_criterion < 0.33:
    criterion = 'gini'
elif model_penalty < 0.67:
    criterion = 'entropy'
else:
    criterion = 'log_loss'

rf_n_estimators = int(round(rf_n_estimators, 0))
model = RandomForestClassifier(n_estimators=rf_n_estimators,
                               max_depth=int(rf_max_depth),
                               max_features=int(rf_max_features),
                               min_samples_split=int(rf_min_samples_split),
                               criterion=criterion,
                               class_weight={0: 1, 1: model_pos_class_weight}
                               )
```

```
#=========================================
# Preprocessing
#=========================================
if custom_binning:

    # Numercial variables transformation
    numeric_transformer = Pipeline(
        steps=[
            ("num_binning", NumBinning(imputer_strategy, knn_imputer))
            ]
    )

    # Categorical variables transformation
    categorical_transformer = Pipeline(
        steps=[
            ('cat_dummy', CatDummies(cat_ref=cat_col_ref))
            ]
    )

    # Consolidated Transformer
    preprocessor = ColumnTransformer(
        transformers=[
            ("num", numeric_transformer, num_vars),
            ("cat", categorical_transformer, cat_vars)
        ],
        remainder='passthrough'
    )

    if smote:
        data_pipeline = imbPipeline(steps=[
                                    ("data_processor", preprocessor),
                                    ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                    # ('smote', SMOTE(random_state=42)),
                                    ('model', model)
                                ])
    else:
        data_pipeline = Pipeline(steps=[
                                ("data_processor", preprocessor),
                                ('model', model)
                            ])

else:
```

```
    # Categorical variables transformation
    if custom_categorical:
        categorical_transformer = Pipeline(
            steps=[('cat_dummy', CatDummies(cat_ref=cat_col_ref))]
        )
    else:
        categorical_transformer = Pipeline(
            steps=[
                ("imputer", SimpleImputer(strategy="most_frequent")),
                ('ohe', OneHotEncoder(drop='first', handle_unknown="ignore"))
            ]
        )

    # Transformaciones base
    transformers = []

    if feature_eng_step is None:
        numeric_transformer = Pipeline(
                                steps=[
                                    ("imputer", imputer)
                                    #("scaler", scaler_cls())
                                ])

        # Categorical variables transformation
        transformers.append(("cat", categorical_transformer, cat_vars))

        # Numercial variables transformation
        transformers.append(("num", numeric_transformer, num_vars))

        preprocessor = ColumnTransformer(
            transformers=transformers,
            remainder='passthrough'
        )

        if smote:
            data_pipeline = imbPipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                        # ('smote', SMOTE(random_state=42)),
                                        ('model', model)
                                    ])
```

```
            else:
                data_pipeline = Pipeline(steps=[
                                            ("data_processor", preprocessor),
                                            ('model', model)
                                        ])

        elif feature_eng_step == 'pca':

            numeric_transformer = Pipeline(
                        steps=[
                            ("imputer", imputer),
                            ("scaler", scaler_cls()),
                            ("pca", PCA(n_components=int(pca_components)))
                        ])

            # Categorical variables transformation
            transformers.append(("cat", categorical_transformer, cat_vars))

            # Numercial variables transformation
            transformers.append(("num", numeric_transformer, num_vars))

            preprocessor = ColumnTransformer(
                transformers=transformers,
                remainder='passthrough'
            )

            if smote:
                data_pipeline = imbPipeline(steps=[
                                            ("data_processor", preprocessor),
                                            ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                            # ('smote', SMOTE(random_state=42)),
                                            ('model', model)
                                        ])
            else:
                data_pipeline = Pipeline(steps=[
                                            ("data_processor", preprocessor),
                                            ('model', model)
                                        ])

        elif feature_eng_step == 'lda':
            numeric_transformer = Pipeline(
                        steps=[
```

```
                        ("imputer", imputer),
                        ("scaler", scaler_cls()),
                        ("lda", LDA(n_components=1))
                    ])

        # Numercial variables transformation
        transformers.append(("num", numeric_transformer, num_vars))

        # Categorical variables transformation
        transformers.append(("cat", categorical_transformer, cat_vars))

        preprocessor = ColumnTransformer(
            transformers=transformers,
            remainder='passthrough'
        )

        if smote:
            data_pipeline = imbPipeline(steps=[
                                    ("data_processor", preprocessor),
                                    ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                    # ('smote', SMOTE(random_state=42)),
                                    ('model', model)
                                ])
        else:
            data_pipeline = Pipeline(steps=[
                                    ("data_processor", preprocessor),
                                    ('model', model)
                                ])

    elif feature_eng_step == 'lasso':

        numeric_transformer = Pipeline(
                                steps=[
                                    ("imputer", imputer),
                                    ("scaler", scaler_cls())
                                ])

        # Categorical variables transformation
        transformers.append(("cat", categorical_transformer, cat_vars))

        # Numercial variables transformation
        transformers.append(("num", numeric_transformer, num_vars))
```

```
        preprocessor = ColumnTransformer(
            transformers=transformers,
            remainder='passthrough'
        )

        # Apply lasso if the model is not a logistic regression
        if model_selection == 1:
            if smote:
                data_pipeline = imbPipeline(steps=[
                                            ("data_processor", preprocessor),
                                            ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)
                                            # ('smote', SMOTE(random_state=42)),
                                            ('model', model)
                                        ])
            else:
                data_pipeline = Pipeline(steps=[
                                            ("data_processor", preprocessor),
                                            ('model', model)
                                        ])
        else:
            if smote:
                data_pipeline = imbPipeline(steps=[
                                            ("data_processor", preprocessor),
                                            ("lasso", SelectFromModel(Lasso(alpha=lasso_alpha))),
                                            ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)
                                            # ('smote', SMOTE(random_state=42)),
                                            ('model', model)
                                        ])
            else:
                data_pipeline = Pipeline(steps=[
                                            ("data_processor", preprocessor),
                                            ("lasso", SelectFromModel(Lasso(alpha=lasso_alpha))),
                                            ('model', model)
                                        ])

    elif feature_eng_step == 'pca_lda':

        pca_transformer = Pipeline(
                    steps=[
                        ("imputer", imputer),
                        ("scaler", scaler_cls()),
```

```
                        ("pca", PCA(n_components=int(pca_components)))
                    ])

        lda_transformer = Pipeline(
                    steps=[
                        ("imputer", imputer),
                        ("scaler", scaler_cls()),
                        ("lda", LDA(n_components=1))
                    ])

        # Numercial variables transformation
        transformers.append(("pca", pca_transformer, num_vars))
        transformers.append(("lda", lda_transformer, num_vars))

        # Categorical variables transformation
        transformers.append(("cat", categorical_transformer, cat_vars))

        preprocessor = ColumnTransformer(
            transformers=transformers,
            remainder='passthrough'
        )

        if smote:
            data_pipeline = imbPipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                        # ('smote', SMOTE(random_state=42)),
                                        ('model', model)
                                    ])
        else:
            data_pipeline = Pipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('model', model)
                                    ])

    elif feature_eng_step == 'all':
        pca_transformer = Pipeline(
                    steps=[
                        ("imputer", imputer),
                        ("scaler", scaler_cls()),
                        ("pca", PCA(n_components=int(pca_components)))
                    ])
```

```
        lda_transformer = Pipeline(
                    steps=[
                        ("imputer", imputer),
                        ("scaler", scaler_cls()),
                        ("lda", LDA(n_components=1))
                    ])

        # Numercial variables transformation
        transformers.append(("pca", pca_transformer, num_vars))
        transformers.append(("lda", lda_transformer, num_vars))

        # Categorical variables transformation
        transformers.append(("cat", categorical_transformer, cat_vars))

        preprocessor = ColumnTransformer(
            transformers=transformers,
            remainder='passthrough'
        )

        if smote:
            data_pipeline = imbPipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ("lasso", SelectFromModel(Lasso(alpha=lasso_alpha))),
                                        ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                        # ('smote', SMOTE(random_state=42)),
                                        ('model', model)
                                    ])
        else:
            data_pipeline = Pipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('lasso', SelectFromModel(Lasso(alpha=lasso_alpha))),
                                        ('model', model)
                                    ])

    #==============================
    # Model Pipeline
    #==============================
    #pipeline = add_model(data_pipeline, model, smote)
    kf = KFold(n_splits=cv, random_state=42, shuffle=True)

    #========================
```

```
    # Metrics
    #=========================
    # Train
    roc_auc_train = []
    ks_train = []
    gini_train = []
    # Validation
    roc_auc_val = []
    ks_val = []
    gini_val = []

    # K-Fold cross val
    for i, (train_index, test_index) in enumerate(kf.split(X_train)):
        # print(f"Fold number: {i+1}")
        kX_train, kX_val = X_train.iloc[train_index], X_train.iloc[test_index]
        ky_train, ky_val = Y_train.iloc[train_index], Y_train.iloc[test_index]

        data_pipeline.fit(kX_train, ky_train.astype(np.float32))

        val_preds = data_pipeline.predict_proba(kX_val)[:, 1]
        val_auc = roc_auc_score(ky_val.astype(np.float32), val_preds)
        val_fpr, val_tpr, _ = roc_curve(ky_val.astype(np.float32), val_preds)
        val_ks = np.max(abs(val_fpr - val_tpr))
        val_gini = 2 * val_auc - 1

        train_preds = data_pipeline.predict_proba(kX_train)[:, 1]
        train_auc = roc_auc_score(ky_train.astype(np.float32), train_preds)
        train_fpr, train_tpr, _ = roc_curve(ky_train.astype(np.float32), train_preds)
        train_ks = np.max(abs(train_fpr - train_tpr))
        train_gini = 2 * train_auc - 1

        # Add train values
        roc_auc_train.append(train_auc)
        ks_train.append(train_ks)
        gini_train.append(train_gini)

        # Add validation values
        roc_auc_val.append(val_auc)
        ks_val.append(val_ks)
        gini_val.append(val_gini)

    # Convert to an array
```

```
    # Validation
    roc_auc_val = np.array(roc_auc_val)
    ks_val = np.array(ks_val)
    gini_val = np.array(gini_val)
    # Training
    roc_auc_train = np.array(roc_auc_train)
    ks_train = np.array(ks_train)
    gini_train = np.array(gini_train)

    # ROC AUC ajusted
    adj_val_roc_auc = (roc_auc_val.mean() -  roc_auc_val.std())
    adj_train_roc_auc = (roc_auc_train.mean() -  roc_auc_train.std())
    obj_roc_auc = adj_val_roc_auc - abs(adj_val_roc_auc - adj_train_roc_auc)

    # KS ajusted
    adj_val_ks = (ks_val.mean() - ks_val.std())
    adj_train_ks = (ks_train.mean() - ks_train.std())
    obj_ks = adj_val_ks - abs(adj_val_ks - adj_train_ks)

    # Gini ajusted
    adj_val_gini = (gini_val.mean() - gini_val.std())
    adj_train_gini = (gini_train.mean() - gini_train.std())
    obj_gini = adj_val_gini - abs(adj_val_gini - adj_train_gini)

    print(f"Validation ROC AUC adjusted score: {adj_val_roc_auc}")
    print(f"Train ROC AUC adjusted score: {adj_train_roc_auc}")
    print("\n")
    print(f"Validation KS adjusted score: {adj_val_ks}")
    print(f"Train KS adjusted score: {adj_train_ks}")
    print("\n")
    print(f"Validation GINI adjusted score: {adj_val_gini}")
    print(f"Train GINI adjusted score: {adj_train_gini}")
    print("\n")

    if show_live_plot:
        # ROC-AUC
        roc_auc_data['train_roc_auc'].append(adj_train_roc_auc)
        roc_auc_data['val_roc_auc'].append(adj_val_roc_auc)
        roc_auc_data['objective'].append(obj_roc_auc)
        # KS
        ks_data['train_ks'].append(adj_train_ks)
        ks_data['val_ks'].append(adj_val_ks)
```

```
        ks_data['objective'].append(obj_ks)
        # Gini
        gini_data['train_gini'].append(adj_train_gini)
        gini_data['val_gini'].append(adj_val_gini)
        gini_data['objective'].append(obj_gini)
        # live_plot(roc_auc_data)
        fig, axes = plt.subplots(3, 1, figsize=(12, 8))
        adj_live_plot(roc_auc_data, subplot=(3, 1, 1), win_size=160)
        adj_live_plot(ks_data, subplot=(3, 1, 2), win_size=160)
        adj_live_plot(gini_data, subplot=(3, 1, 3), win_size=160)
        plt.tight_layout()
        plt.show()

    return data_pipeline, obj_roc_auc
```

Función para entrenar XGBoost

In [ ]:

```
# Data para visualizacion de la optimizacion
roc_auc_data = collections.defaultdict(list)
ks_data = collections.defaultdict(list)
gini_data = collections.defaultdict(list)

cat_col_ref = ['purpose:all_other', 'purpose:credit_card',
               'purpose:debt_consolidation', 'purpose:educational',
               'purpose:home_improvement', 'purpose:major_purchase',
               'purpose:small_business']

# Funcion para entrenar y evaluar el modelo con KFold
def train_and_evaluate_xgb(
                        # Data
                        scaler_choice,
                        imputer_strategy,
                        knn_imputer,
                        knn_imputer_k,
                        pca_components,
                        sampling_strategy,
                        lasso_alpha,
                        # XGBoost
                        xgb_colsample_bytree,
                        xgb_learning_rate,
                        xgb_max_depth,
```

```
                    xgb_n_estimators,
                    xgb_gamma,
                    xgb_reg_lambda,
                    xgb_reg_alpha,
                    xgb_subsample,
                    xgb_scale_pos_weight
                    ) -> float:
"""  Funcion para entrenamiento y validacion del modelo

Args:
    scaler_choice (_type_): _description_
    imputer_strategy (_type_): _description_
    knn_imputer (_type_): _description_
    knn_imputer_k (_type_): _description_
    pca_components (_type_): _description_
    sampling_strategy (_type_): _description_
    lasso_alpha (_type_): _description_

Returns:
    float: ROC AUC score
    float: KS score
    float: GINI score
"""

#================================
# Configuracion inicial
#================================

#======================
# Configuraciones
#======================
custom_binning = custom_binning_conf
smote = smote_conf
custom_categorical = custom_categorical_conf

# Modelo a utilizar
model_selection = 3
cv = cv_conf

# Steps pipeline preprocesor
feature_eng_step = feature_eng_step_conf
```

```python
# Mostrar grafico optimizacion
show_live_plot=show_live_plot_cong

#========
# Datos
#========
X_train = x_data.copy()
Y_train = y_data.copy()

# Numerical and categorical variables
cat_vars = [*X_train.select_dtypes(include='object').columns]
num_vars = [*X_train.select_dtypes(exclude='object').columns]

# Scaling strategy
scaler_cls = StandardScaler if scaler_choice > 0.5 else MinMaxScaler

# Imputation strategy
impute_strategy = "mean" if imputer_strategy > 0.5 else "median"
if knn_imputer > 0.5:
    imputer = KNNImputer(n_neighbors=int(knn_imputer_k))
else:
    imputer = SimpleImputer(strategy=impute_strategy)


#========================================
# Model
#========================================
print("Modelo seleccionado: XGBoost")
model = xgb.XGBClassifier(objective ='reg:linear',
                          colsample_bytree=xgb_colsample_bytree,
                          learning_rate=xgb_learning_rate,
                          max_depth=int(xgb_max_depth),
                          n_estimators=int(xgb_n_estimators),
                          gamma=xgb_gamma,
                          reg_lambda=xgb_reg_lambda,
                          reg_alpha=xgb_reg_alpha,
                          subsample=xgb_subsample,
                          scale_pos_weight=xgb_scale_pos_weight
                          )

#========================================
# Preprocessing
```

```
#=======================================
if custom_binning:

    # Numerical variable transformation
    numeric_transformer = Pipeline(
        steps=[
            # ("imputer", imputer),
            ("num_binning", NumBinning(imputer_strategy, knn_imputer))
            ]
    )

    # Categorical variables transformations
    categorical_transformer = Pipeline(
        steps=[
            # ("imputer", SimpleImputer(strategy="most_frequent")),
            ('cat_dummy', CatDummies(cat_ref=cat_col_ref))
            ]
    )

    # Consolidated Transformer
    preprocessor = ColumnTransformer(
        transformers=[
            ("num", numeric_transformer, num_vars),
            ("cat", categorical_transformer, cat_vars)
        ],
        remainder='passthrough'
    )

    if smote:
        data_pipeline = imbPipeline(steps=[
                                    ("data_processor", preprocessor),
                                    ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                    ('model', model)
                                ])
    else:
        data_pipeline = Pipeline(steps=[
                                ("data_processor", preprocessor),
                                ('model', model)
                            ])

else:
```

```
        # Categorical variables transformations
        if custom_categorical:
            categorical_transformer = Pipeline(
                steps=[('cat_dummy', CatDummies(cat_ref=cat_col_ref))]
            )
        else:
            categorical_transformer = Pipeline(
                steps=[
                    ("imputer", SimpleImputer(strategy="most_frequent")),
                    ('ohe', OneHotEncoder(drop='first', handle_unknown="ignore"))
                ]
            )

        # Transformaciones base
        transformers = []

        if feature_eng_step is None:
            numeric_transformer = Pipeline(
                                        steps=[
                                            ("imputer", imputer)
                                        ])

            # Categorical variables transformations
            transformers.append(("cat", categorical_transformer, cat_vars))

            # Numerical variable transformation
            transformers.append(("num", numeric_transformer, num_vars))

            preprocessor = ColumnTransformer(
                transformers=transformers,
                remainder='passthrough'
            )

            if smote:
                data_pipeline = imbPipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                        ('model', model)
                                    ])
            else:
                data_pipeline = Pipeline(steps=[
                                        ("data_processor", preprocessor),
```

```
                                        ('model', model)
                                    ])

        elif feature_eng_step == 'pca':

            numeric_transformer = Pipeline(
                        steps=[
                            ("imputer", imputer),
                            ("scaler", scaler_cls()),
                            ("pca", PCA(n_components=int(pca_components)))
                        ])

            # Categorical variables transformations
            transformers.append(("cat", categorical_transformer, cat_vars))

            # Numerical variable transformation
            transformers.append(("num", numeric_transformer, num_vars))

            preprocessor = ColumnTransformer(
                transformers=transformers,
                remainder='passthrough'
            )

            if smote:
                data_pipeline = imbPipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                        ('model', model)
                                    ])
            else:
                data_pipeline = Pipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('model', model)
                                    ])

        elif feature_eng_step == 'lda':
            numeric_transformer = Pipeline(
                        steps=[
                            ("imputer", imputer),
                            ("scaler", scaler_cls()),
                            ("lda", LDA(n_components=1))
                        ])
```

```
        # Numerical variable transformation
        transformers.append(("num", numeric_transformer, num_vars))

        # Categorical variables transformations
        transformers.append(("cat", categorical_transformer, cat_vars))

        preprocessor = ColumnTransformer(
            transformers=transformers,
            remainder='passthrough'
        )

        if smote:
            data_pipeline = imbPipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                        ('model', model)
                                    ])
        else:
            data_pipeline = Pipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('model', model)
                                    ])

    elif feature_eng_step == 'lasso':

        numeric_transformer = Pipeline(
                                    steps=[
                                        ("imputer", imputer),
                                        ("scaler", scaler_cls())
                                    ])

        # Categorical variables transformations
        transformers.append(("cat", categorical_transformer, cat_vars))

        # Numerical variable transformation
        transformers.append(("num", numeric_transformer, num_vars))

        preprocessor = ColumnTransformer(
            transformers=transformers,
            remainder='passthrough'
        )
```

```
        # Apply lasso if the model is not a logistic regression
        if model_selection == 1:
            if smote:
                data_pipeline = imbPipeline(steps=[
                                            ("data_processor", preprocessor),
                                            ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)
                                            ('model', model)
                                        ])
            else:
                data_pipeline = Pipeline(steps=[
                                            ("data_processor", preprocessor),
                                            ('model', model)
                                        ])
        else:
            if smote:
                data_pipeline = imbPipeline(steps=[
                                            ("data_processor", preprocessor),
                                            ("lasso", SelectFromModel(Lasso(alpha=lasso_alpha))),
                                            ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)
                                            ('model', model)
                                        ])
            else:
                data_pipeline = Pipeline(steps=[
                                            ("data_processor", preprocessor),
                                            ("lasso", SelectFromModel(Lasso(alpha=lasso_alpha))),
                                            ('model', model)
                                        ])

    elif feature_eng_step == 'pca_lda':

        pca_transformer = Pipeline(
                    steps=[
                        ("imputer", imputer),
                        ("scaler", scaler_cls()),
                        ("pca", PCA(n_components=int(pca_components)))
                    ])

        lda_transformer = Pipeline(
                    steps=[
                        ("imputer", imputer),
                        ("scaler", scaler_cls()),
```

```
                        ("lda", LDA(n_components=1))
                    ])

        # Numerical variable transformation
        transformers.append(("pca", pca_transformer, num_vars))
        transformers.append(("lda", lda_transformer, num_vars))

        # Categorical variables transformations
        transformers.append(("cat", categorical_transformer, cat_vars))

        preprocessor = ColumnTransformer(
            transformers=transformers,
            remainder='passthrough'
        )

        if smote:
            data_pipeline = imbPipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                        ('model', model)
                                    ])
        else:
            data_pipeline = Pipeline(steps=[
                                        ("data_processor", preprocessor),
                                        ('model', model)
                                    ])

    elif feature_eng_step == 'all':
        pca_transformer = Pipeline(
                    steps=[
                        ("imputer", imputer),
                        ("scaler", scaler_cls()),
                        ("pca", PCA(n_components=int(pca_components)))
                    ])

        lda_transformer = Pipeline(
                    steps=[
                        ("imputer", imputer),
                        ("scaler", scaler_cls()),
                        ("lda", LDA(n_components=1))
                    ])
```

```
            # Numerical variable transformation
            transformers.append(("pca", pca_transformer, num_vars))
            transformers.append(("lda", lda_transformer, num_vars))

            # Categorical variables transformations
            transformers.append(("cat", categorical_transformer, cat_vars))

            preprocessor = ColumnTransformer(
                transformers=transformers,
                remainder='passthrough'
            )

            if smote:
                data_pipeline = imbPipeline(steps=[
                                            ("data_processor", preprocessor),
                                            ("lasso", SelectFromModel(Lasso(alpha=lasso_alpha))),
                                            ('smote', SMOTE(random_state=42, sampling_strategy=sampling_strategy)),
                                            ('model', model)
                                        ])
            else:
                data_pipeline = Pipeline(steps=[
                                            ("data_processor", preprocessor),
                                            ('lasso', SelectFromModel(Lasso(alpha=lasso_alpha))),
                                            ('model', model)
                                        ])

        #==============================
        # Model Pipeline
        #==============================
        #pipeline = add_model(data_pipeline, model, smote)
        kf = KFold(n_splits=cv, random_state=42, shuffle=True)

        #=========================
        # Metrics
        #=========================
        # Train
        roc_auc_train = []
        ks_train = []
        gini_train = []
        # Validation
        roc_auc_val = []
        ks_val = []
```

```
    gini_val = []

    # K-Fold cross val
    for i, (train_index, test_index) in enumerate(kf.split(X_train)):
        # print(f"Fold number: {i+1}")
        kX_train, kX_val = X_train.iloc[train_index], X_train.iloc[test_index]
        ky_train, ky_val = Y_train.iloc[train_index], Y_train.iloc[test_index]

        data_pipeline.fit(kX_train, ky_train.astype(np.float32))

        val_preds = data_pipeline.predict_proba(kX_val)[:, 1]
        val_auc = roc_auc_score(ky_val.astype(np.float32), val_preds)
        val_fpr, val_tpr, _ = roc_curve(ky_val.astype(np.float32), val_preds)
        val_ks = np.max(abs(val_fpr - val_tpr))
        val_gini = 2 * val_auc - 1

        train_preds = data_pipeline.predict_proba(kX_train)[:, 1]
        train_auc = roc_auc_score(ky_train.astype(np.float32), train_preds)
        train_fpr, train_tpr, _ = roc_curve(ky_train.astype(np.float32), train_preds)
        train_ks = np.max(abs(train_fpr - train_tpr))
        train_gini = 2 * train_auc - 1

        # Train values
        roc_auc_train.append(train_auc)
        ks_train.append(train_ks)
        gini_train.append(train_gini)

        # Validation values
        roc_auc_val.append(val_auc)
        ks_val.append(val_ks)
        gini_val.append(val_gini)

    # Convert to an array
    # Validation
    roc_auc_val = np.array(roc_auc_val)
    ks_val = np.array(ks_val)
    gini_val = np.array(gini_val)
    # Training
    roc_auc_train = np.array(roc_auc_train)
    ks_train = np.array(ks_train)
    gini_train = np.array(gini_train)
```

```python
    # ROC AUC ajusted
    adj_val_roc_auc = (roc_auc_val.mean() -  roc_auc_val.std())
    adj_train_roc_auc = (roc_auc_train.mean() -  roc_auc_train.std())
    obj_roc_auc = adj_val_roc_auc - abs(adj_val_roc_auc - adj_train_roc_auc)

    # KS ajusted
    adj_val_ks = (ks_val.mean() - ks_val.std())
    adj_train_ks = (ks_train.mean() - ks_train.std())
    obj_ks = adj_val_ks - abs(adj_val_ks - adj_train_ks)

    # Gini ajusted
    adj_val_gini = (gini_val.mean() - gini_val.std())
    adj_train_gini = (gini_train.mean() - gini_train.std())
    obj_gini = adj_val_gini - abs(adj_val_gini - adj_train_gini)

    print(f"Validation ROC AUC adjusted score: {adj_val_roc_auc}")
    print(f"Train ROC AUC adjusted score: {adj_train_roc_auc}")
    print("\n")
    print(f"Validation KS adjusted score: {adj_val_ks}")
    print(f"Train KS adjusted score: {adj_train_ks}")
    print("\n")
    print(f"Validation GINI adjusted score: {adj_val_gini}")
    print(f"Train GINI adjusted score: {adj_train_gini}")
    print("\n")

    if show_live_plot:
        # ROC-AUC
        roc_auc_data['train_roc_auc'].append(adj_train_roc_auc)
        roc_auc_data['val_roc_auc'].append(adj_val_roc_auc)
        roc_auc_data['objective'].append(obj_roc_auc)
        # KS
        ks_data['train_ks'].append(adj_train_ks)
        ks_data['val_ks'].append(adj_val_ks)
        ks_data['objective'].append(obj_ks)
        # Gini
        gini_data['train_gini'].append(adj_train_gini)
        gini_data['val_gini'].append(adj_val_gini)
        gini_data['objective'].append(obj_gini)
        # live_plot(roc_auc_data)
        fig, axes = plt.subplots(3, 1, figsize=(12, 8))
        adj_live_plot(roc_auc_data, subplot=(3, 1, 1), win_size=160)
        adj_live_plot(ks_data, subplot=(3, 1, 2), win_size=160)
```

```
        adj_live_plot(gini_data, subplot=(3, 1, 3), win_size=160)
        plt.tight_layout()
        plt.show()

    return data_pipeline, obj_roc_auc
```

Configuracion para entrenar modelo

In [ ]:

```
"""
---------------
INSTRUCTIONS
---------------

-------------------------
1. Select model
-------------------------
model_selection_conf :  1 - Regresion logística
                        2 - Random Forest
                        3 - XGboost

# Croos validation
cv_conf : k value for cross validation

-------------------------
2. Preprocesamientos
-------------------------
feature_eng_step_conf:  - None (no preprocessing)
                        - pca (pca to numerical variables)
                        - lda (lda to numerical variables)
                        - lasso (lasso to numerical variables)
                        - pca_lda (pca and lda to numerical variables)
                        - all (all transformations)

custom_binning_conf:    - True (apply woe transformations to numerical and
                              categorical variables)
                        - False

custom_categorical_conf:    - True (apply woe transformation to categorical
                                    variables, it can be used when custom_binning_conf = False)
                            - False
```

```
smote_conf: - True (apply data balancing with smote)
            - False

"""

# Data for visualizing optimization process
roc_auc_data = collections.defaultdict(list)
ks_data = collections.defaultdict(list)
gini_data = collections.defaultdict(list)

# Data
x_data, y_data = x_train, y_train

# Model
model_selection_conf = 1

# k Cross validation
cv_conf = 10

# Woe Transformations
custom_binning_conf = False

# Select preprocessing
feature_eng_step_conf = 'lda'
custom_categorical_conf = False

# Smote
smote_conf = False

# Show optimization plot
show_live_plot_cong = True

pca_components = (x_train
                  .select_dtypes(exclude=['object', 'category'])
                  .shape[1])

#================================
# Optimization params
#================================
if model_selection_conf == 1:
    print("Hyperparamers for RL")
```

```
    pbounds = dict(
        # Data
        scaler_choice=(0, 1),
        imputer_strategy=(0, 1),
        knn_imputer=(0,1),
        knn_imputer_k=(3, 7),
        pca_components=(2, pca_components),
        sampling_strategy=(0.3, 0.5), # SMOTE
        lasso_alpha=(0.00001, 0.001),
        # Model
        model_penalty=(0, 1), # Ridge - Lasso
        model_C=(0.00001, 1), # alpha
        model_pos_class_weight=(1, 100)
    )
elif model_selection_conf == 2:
    print("Hyperparamers for RF")
    pbounds = dict(
        # Data
        scaler_choice=(0, 1),
        imputer_strategy=(0, 1),
        knn_imputer=(0,1),
        knn_imputer_k=(3, 7),
        pca_components=(2, pca_components),
        sampling_strategy=(0.3, 0.5), # SMOTE
        lasso_alpha=(0.00001, 0.001),
        # Hiperparametros random forest
        model_penalty=(0, 1),
        rf_n_estimators=(100, 500),
        rf_max_depth=(2, 20),
        rf_max_features=(8, 60),
        rf_min_samples_split=(2, 10),
        rf_criterion=(0, 1),
        model_pos_class_weight=(1, 100)
    )
else:
    print("Hyperparamers for XGB")
    pbounds = dict(
        # Data
        scaler_choice=(0, 1),
        imputer_strategy=(0, 1),
        knn_imputer=(0,1),
        knn_imputer_k=(3, 7),
```

```
        pca_components=(2, pca_components),
        sampling_strategy=(0.3, 0.5), # SMOTE
        lasso_alpha=(0.00001, 0.001),
        # Hiperparametros xgboost
        xgb_colsample_bytree=(0.1, 1),
        xgb_learning_rate=(0.0001, 1),
        xgb_n_estimators=(100, 500),
        xgb_gamma=(0, 10),
        xgb_reg_lambda=(0, 10),
        xgb_reg_alpha=(0, 10),
        xgb_max_depth=(2, 20),
        xgb_subsample=(0.5, 1),
        xgb_scale_pos_weight=(20, 40)
    )

# Black box funtion
if model_selection_conf == 1:
    print("Target function: RL")
    def target_func(**kwargs):
        """ Black box function for bayes optimization
        """
        model, result = train_and_evaluate_rl(**kwargs)
        return result

elif model_selection_conf == 2:
    print("Target function: RF")
    def target_func(**kwargs):
        """ Black box function for bayes optimization
        """
        model, result = train_and_evaluate_rf(**kwargs)
        return result

else:
    print("Target function: XGB")
    def target_func(**kwargs):
        """ Black box function for bayes optimization
        """
        model, result = train_and_evaluate_xgb(**kwargs)
        return result

# Bayesian Optimization
optimizer = BayesianOptimization(
```

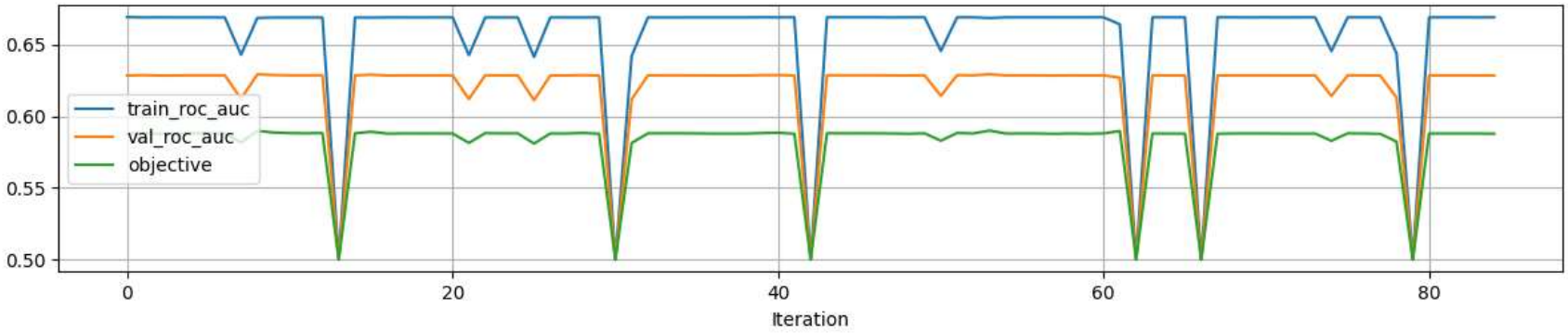
```
    f=target_func,
    pbounds=pbounds,
    verbose=2,
    random_state=42
)
```

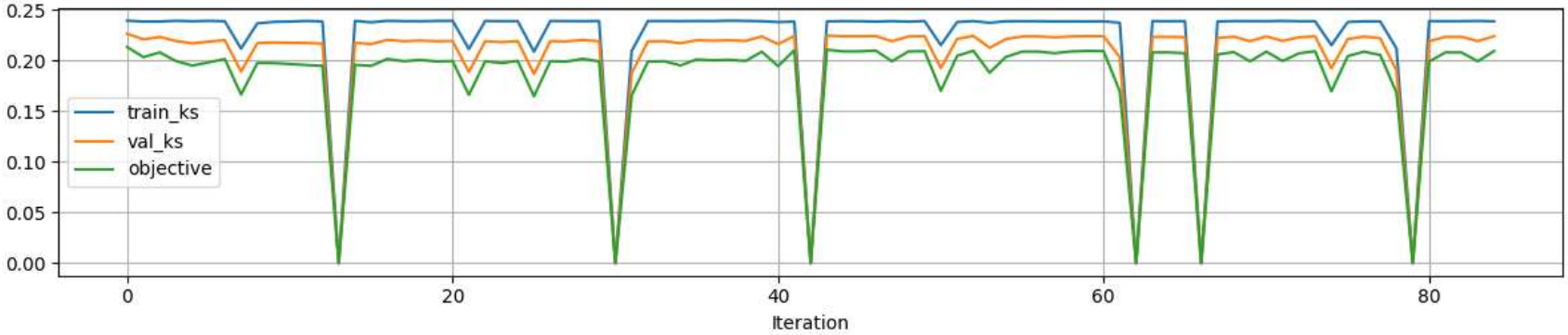
```
Hyperparamers for RL
Target function: RL
```

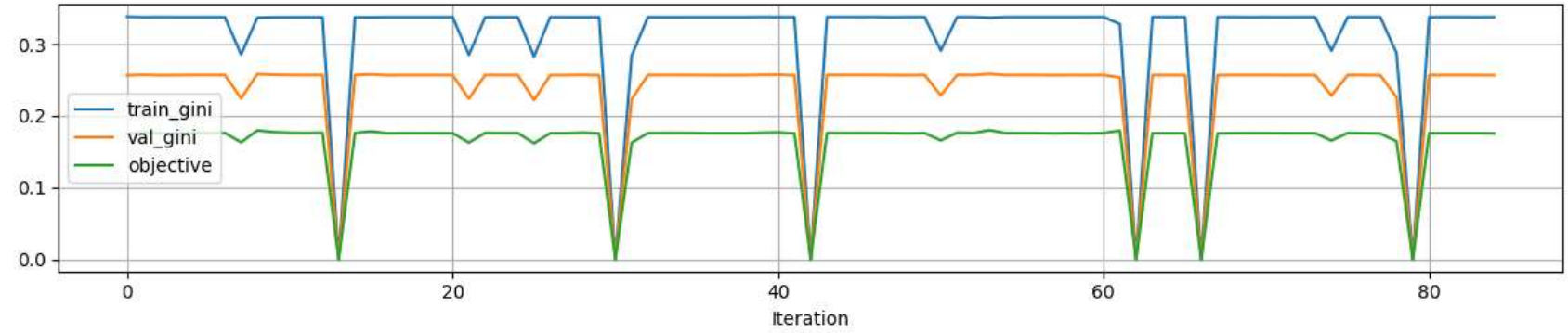
Hyperparameter optimization

In [ ]:

```
models = {1: 'RL', 2:'RF', 3:'XGB'}

# Optimization process logs
model_id = 2
model = models[model_id]
model_seq = 12 #  RL -> 4 , RL -> 5, XGB -> 7, XGB ->  8, XGB ->10, RF -> 11
path = '../data/models/train/hyperparams'
logger = JSONLogger(path=f'{path}/MOD{model_seq}_{model}.json')
optimizer.subscribe(Events.OPTIMIZATION_STEP, logger)

# Optimization
optimizer.maximize(
    n_iter=80
)
```







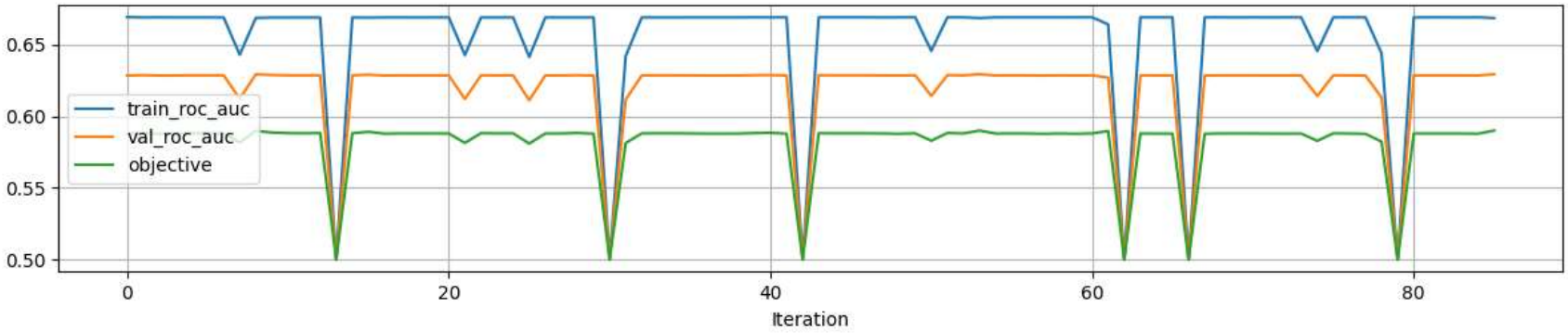
Best Hyperparamers

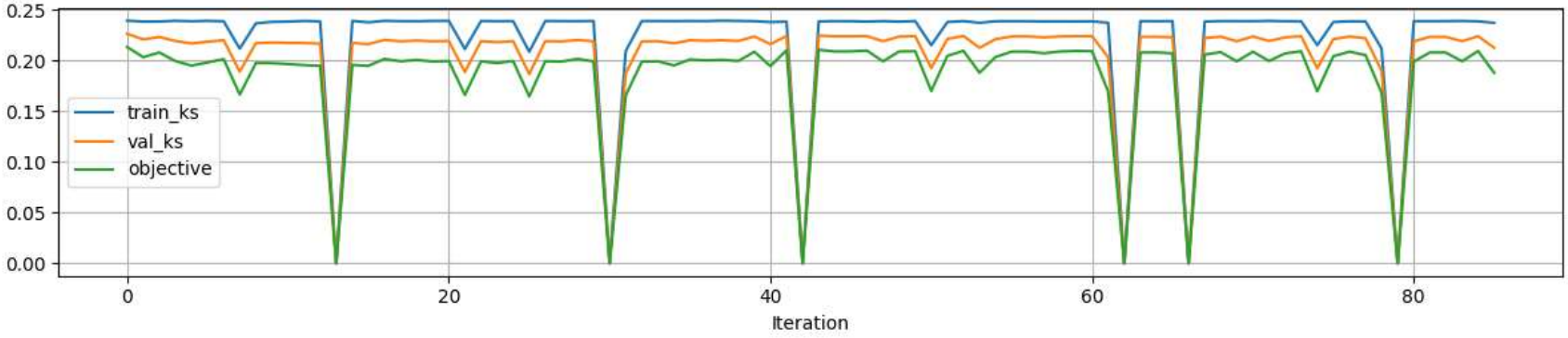
In [ ]:
```
optimizer.max["params"]
```

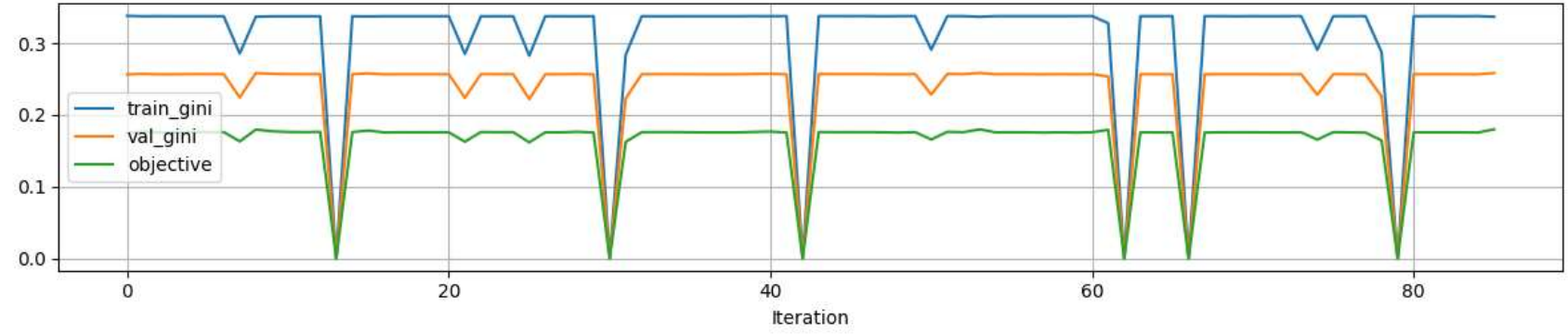
Out[ ]:
```
{'imputer_strategy': 0.012822663913136134,
 'knn_imputer': 0.6228659780543814,
 'knn_imputer_k': 4.218265533763853,
 'lasso_alpha': 0.00031970902581341813,
 'model_C': 0.044483652096020745,
 'model_penalty': 0.8167872374915031,
 'model_pos_class_weight': 22.09815829974078,
 'pca_components': 6.664601635957697,
 'sampling_strategy': 0.3876542667030801,
 'scaler_choice': 0.44634280090867573}
```

Create model using best Hyperparamers

In [ ]:
```python
# Best model
if model_selection_conf == 1:
    print("Target function: RL")
    best_model, best_result = train_and_evaluate_rl(**optimizer.max["params"])

elif model_selection_conf == 2:
    print("Target function: RF")
    best_model, best_result = train_and_evaluate_rf(**optimizer.max["params"])

else:
    print("Target function: XGB")
    best_model, best_result = train_and_evaluate_xgb(**optimizer.max["params"])
```

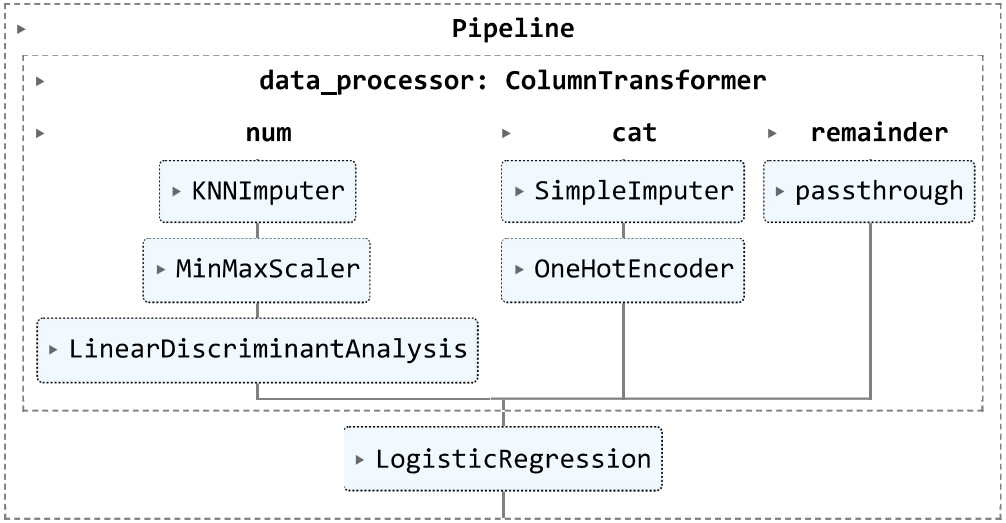






In [ ]:
```
best_model
```

Out[ ]:



In [ ]:
```
best_result
```

Out[ ]:
```
0.5899848225191138
```

Test best model using all datasets

In [ ]:
```
# Function for testing the best model
def test_best_model(x_data=None, y_data=None, dataset_name='test',
                    filename=None, print_pred_file=False,
                    print_metrics_file=False, roc_data=False,
                    file_path=None):

    """ Funcion para probar el modelo con el set de datos completos,
        el set de validación y el set de prueba

    Args:
        dataset_name (str, optional):
                                1. train
                                2. test
                                3. validation
                                4. total
    """
```

```
    if dataset_name == 'train':
        df_x, df_y = x_data, y_data
        val_preds = best_model.predict_proba(df_x)[:, 1]

        # Metrics
        roc_auc = roc_auc_score(df_y.astype(np.int32), val_preds)
        fpr, tpr, threshold = roc_curve(df_y.astype(np.float32), val_preds,
                                        drop_intermediate=False)
        precision, recall, thresholds = precision_recall_curve(df_y.astype(np.float32),
                                                               val_preds)
        ks = np.max(abs(fpr - tpr))
        gini = 2 * roc_auc - 1
        metricas = pd.DataFrame({'ROC AUC': [roc_auc],
                                 'KS': [ks],
                                 'GINI': [gini]
                                 })

        roc_df = pd.DataFrame({'fpr': fpr,
                               'tpr': tpr,
                               'threshold': threshold
                               })

        prec_recall_df = pd.DataFrame({'precision': precision,
                                       'recall': recall
                                       })
        thresholds = pd.DataFrame({'threshold': thresholds})
        prec_recall_df = prec_recall_df.merge(thresholds, left_index=True, right_index=True)

        if roc_data:
            if filename is not None:
                if file_path is not None:
                    roc_df.to_csv(rf"{file_path}/roc/ROC_MOD_{filename}.csv", index=False)
                    prec_recall_df.to_csv(rf"{file_path}/roc/PREC_REC_MOD_{filename}.csv", index=False)
                    print(f"File {filename} generated")
                else:
                    roc_df.to_csv(rf"../data/models/train/roc/ROC_MOD_{filename}.csv", index=False)
                    prec_recall_df.to_csv(rf"../data/models/train/roc/PREC_REC_MOD_{filename}.csv", index=False)
                    print(f"File {filename} generated")

        if print_metrics_file:
            if filename is not None:
                if file_path is not None:
```

```
                metricas.to_csv(rf"{file_path}/metrics/MTR_MOD_{filename}.csv", index=False)
                print(f"File {filename} generated")
            else:
                metricas.to_csv(rf"../data/models/train/metrics/MTR_MOD_{filename}.csv", index=False)
                print(f"File {filename} generated")

    # Predicitons
    if print_pred_file:
        if filename is not None:
            filename = f"{filename}"
            file = pd.DataFrame(data=dict(id=df_x.index,
                                          predicted_bad=val_preds,
                                          actual_bad=df_y)
                                )
            if file_path is not None:
                file.to_csv(rf"{file_path}/predictions/MOD_{filename}.csv", index=False)
                print(f"File {filename} generated")
            else:
                file.to_csv(rf"../data/models/train/predictions/MOD_{filename}.csv", index=False)
                print(f"File {filename} generated")

    return f"Train set ROC_AUC score: {roc_auc} - KS score: {ks} - GINI score: {gini}"

if dataset_name == 'test':
    df_x, df_y = x_data, y_data
    val_preds = best_model.predict_proba(df_x)[:, 1]

    # Metrics
    roc_auc = roc_auc_score(df_y.astype(np.int32), val_preds)
    fpr, tpr, threshold = roc_curve(df_y.astype(np.float32), val_preds,
                                    drop_intermediate=False)
    precision, recall, thresholds = precision_recall_curve(df_y.astype(np.float32),
                                                            val_preds)
    ks = np.max(abs(fpr - tpr))
    gini = 2 * roc_auc - 1
    metricas = pd.DataFrame({'ROC AUC': [roc_auc],
                             'KS': [ks],
                             'GINI': [gini]
                             })

    roc_df = pd.DataFrame({'fpr': fpr,
                           'tpr': tpr,
```

```
                            'threshold': threshold
                            })

    prec_recall_df = pd.DataFrame({'precision': precision,
                                   'recall': recall
                                   })
    thresholds = pd.DataFrame({'threshold': thresholds})
    prec_recall_df = prec_recall_df.merge(thresholds, left_index=True, right_index=True)

    if roc_data:
        if filename is not None:
            if file_path is not None:
                roc_df.to_csv(rf"{file_path}/roc/ROC_MOD_{filename}.csv", index=False)
                prec_recall_df.to_csv(rf"{file_path}/roc/PREC_REC_MOD_{filename}.csv", index=False)
                print(f"File {filename} generated")
            else:
                roc_df.to_csv(rf"../data/models/test/roc/ROC_MOD_{filename}.csv", index=False)
                prec_recall_df.to_csv(rf"../data/models/test/roc/PREC_REC_MOD_{filename}.csv", index=False)
                print(f"File {filename} generated")

    if print_metrics_file:
        if filename is not None:
            if file_path is not None:
                metricas.to_csv(rf"{file_path}/metrics/MTR_MOD_{filename}.csv", index=False)
                print(f"File {filename} generated")
            else:
                metricas.to_csv(rf"../data/models/test/metrics/MTR_MOD_{filename}.csv", index=False)
                print(f"File {filename} generated")

    # Predicitons
    if print_pred_file:
        if filename is not None:
            filename = f"{filename}"
            file = pd.DataFrame(data=dict(id=df_x.index,
                                          predicted_bad=val_preds,
                                          actual_bad=df_y)
                                )
            if file_path is not None:
                file.to_csv(rf"{file_path}/predictions/MOD_{filename}.csv", index=False)
                print(f"File {filename} generated")
            else:
                file.to_csv(rf"../data/models/test/predictions/MOD_{filename}.csv", index=False)
```

```
                print(f"File {filename} generated")

        return f"Test set ROC_AUC score: {roc_auc} - KS score: {ks} - GINI score: {gini}"

    if dataset_name == 'validation':
        df_x, df_y = x_data, y_data
        val_preds = best_model.predict_proba(df_x)[:, 1]

        # Metrics
        roc_auc = roc_auc_score(df_y.astype(np.int32), val_preds)
        fpr, tpr, threshold = roc_curve(df_y.astype(np.float32), val_preds,
                                        drop_intermediate=False)
        precision, recall, thresholds = precision_recall_curve(df_y.astype(np.float32),
                                                               val_preds)
        ks = np.max(abs(fpr - tpr))
        gini = 2 * roc_auc - 1
        metricas = pd.DataFrame({'ROC AUC': [roc_auc],
                                 'KS': [ks],
                                 'GINI': [gini]
                                 })

        roc_df = pd.DataFrame({'fpr': fpr,
                               'tpr': tpr,
                               'threshold': threshold
                               })

        prec_recall_df = pd.DataFrame({'precision': precision,
                                       'recall': recall
                                       })
        thresholds = pd.DataFrame({'threshold': thresholds})
        prec_recall_df = prec_recall_df.merge(thresholds, left_index=True, right_index=True)

        if roc_data:
            if filename is not None:
                if file_path is not None:
                    roc_df.to_csv(rf"{file_path}/roc/ROC_MOD_{filename}.csv", index=False)
                    prec_recall_df.to_csv(rf"{file_path}/roc/PREC_REC_MOD_{filename}.csv", index=False)
                    print(f"File {filename} generated")
                else:
                    roc_df.to_csv(rf"../data/models/validation/roc/ROC_MOD_{filename}.csv", index=False)
                    prec_recall_df.to_csv(rf"../data/models/validation/roc/PREC_REC_MOD_{filename}.csv", index=False)
                    print(f"File {filename} generated")
```

```
    if print_metrics_file:
        if filename is not None:
            if file_path is not None:
                metricas.to_csv(rf"{file_path}/metrics/MTR_MOD_{filename}.csv", index=False)
                print(f"File {filename} generated")
            else:
                metricas.to_csv(rf"../data/models/validation/metrics/MTR_MOD_{filename}.csv", index=False)
                print(f"File {filename} generated")

    # Predicitons
    if print_pred_file:
        if filename is not None:
            filename = f"{filename}"
            file = pd.DataFrame(data=dict(id=df_x.index,
                                          predicted_bad=val_preds,
                                          actual_bad=df_y)
                                )
            if file_path is not None:
                file.to_csv(rf"{file_path}/predictions/MOD_{filename}.csv", index=False)
                print(f"File {filename} generated")
            else:
                file.to_csv(rf"../data/models/validation/predictions/MOD_{filename}.csv", index=False)
                print(f"File {filename} generated")

    return f"Validation set ROC_AUC score: {roc_auc} - KS score: {ks} - GINI score: {gini}"

elif dataset_name == 'total':
    df_x, df_y = x_data, y_data
    val_preds = best_model.predict_proba(df_x)[:, 1]

    # Metrics
    roc_auc = roc_auc_score(df_y.astype(np.int32), val_preds)
    fpr, tpr, threshold = roc_curve(df_y.astype(np.float32), val_preds,
                                    drop_intermediate=False)
    precision, recall, thresholds = precision_recall_curve(df_y.astype(np.float32),
                                                            val_preds)
    ks = np.max(abs(fpr - tpr))
    gini = 2 * roc_auc - 1
    metricas = pd.DataFrame({'ROC AUC': [roc_auc],
                             'KS': [ks],
                             'GINI': [gini]
```

```
                            })

        roc_df = pd.DataFrame({'fpr': fpr,
                               'tpr': tpr,
                               'threshold': threshold
                               })

        prec_recall_df = pd.DataFrame({'precision': precision,
                                       'recall': recall
                                       })
        thresholds = pd.DataFrame({'threshold': thresholds})
        prec_recall_df = prec_recall_df.merge(thresholds, left_index=True, right_index=True)

        if roc_data:
            if filename is not None:
                if file_path is not None:
                    roc_df.to_csv(rf"{file_path}/roc/ROC_MOD_{filename}.csv", index=False)
                    prec_recall_df.to_csv(rf"{file_path}/roc/PREC_REC_MOD_{filename}.csv", index=False)
                    print(f"File {filename} generated")
                else:
                    roc_df.to_csv(rf"../data/models/full/roc/ROC_MOD_{filename}.csv", index=False)
                    prec_recall_df.to_csv(rf"../data/models/full/roc/PREC_REC_MOD_{filename}.csv", index=False)
                    print(f"File {filename} generated")

        if print_metrics_file:
            if filename is not None:
                if file_path is not None:
                    metricas.to_csv(rf"{file_path}/metrics/MTR_MOD_{filename}.csv", index=False)
                    print(f"File {filename} generated")
                else:
                    metricas.to_csv(rf"../data/models/full/metrics/MTR_MOD_{filename}.csv", index=False)
                    print(f"File {filename} generated")

        # Predicitons
        if print_pred_file:
            if filename is not None:
                filename = f"{filename}"
                file = pd.DataFrame(data=dict(id=df_x.index,
                                              predicted_bad=val_preds,
                                              actual_bad=df_y)
                                    )
                if file_path is not None:
```

```
                    file.to_csv(rf"{file_path}/predictions/MOD_{filename}.csv", index=False)
                    print(f"File {filename} generated")
                else:
                    file.to_csv(rf"../data/models/full/predictions/MOD_{filename}.csv", index=False)
                    print(f"File {filename} generated")

        return f"Full set ROC_AUC score: {roc_auc} - KS score: {ks} - GINI score: {gini}"
```

In [ ]:
```
# Test best model using training data
file_path = '../data/models/train'
filename = f'{model_seq}_{model}'
x_data = x_train.copy()
y_data = y_train.copy()
test_best_model(x_data=x_data, y_data=y_data,
                dataset_name='train', filename=filename,
                print_metrics_file=True, print_pred_file=True,
                roc_data=True, file_path=file_path)
```

```
File 12_RF generated
File 12_RF generated
File 12_RF generated
```

Out[ ]:
```
'Train set ROC_AUC score: 0.6712254630143063 - KS score: 0.24128990918388726 - GINI score: 0.3424509260286126'
```

In [ ]:
```
# Test best model using validation data
file_path = '../data/models/validation'
filename = f'{model_seq}_{model}'
x_data = x_val.copy()
y_data = y_val.copy()
test_best_model(x_data=x_data, y_data=y_data,
                dataset_name='validation', filename=filename,
                print_metrics_file=True, print_pred_file=True,
                roc_data=True, file_path=file_path)
```

```
File 12_RF generated
File 12_RF generated
File 12_RF generated
```

Out[ ]:
```
'Validation set ROC_AUC score: 0.7006603811115089 - KS score: 0.3255261168795004 - GINI score: 0.4013207622230177'
```

In [ ]:
```
# Test best model using test data
file_path = '../data/models/test'
filename = f'{model_seq}_{model}'
```

```
x_data = x_test.copy()
y_data = y_test.copy()
test_best_model(x_data=x_data, y_data=y_data,
                dataset_name='test', filename=filename,
                print_metrics_file=True, print_pred_file=True,
                roc_data=True, file_path=file_path)
```

```
File 12_RF generated
File 12_RF generated
File 12_RF generated
```

Out[ ]: 'Test set ROC_AUC score: 0.687604913367674 - KS score: 0.2625013508158929 - GINI score: 0.3752098267353481'

In [ ]:
```
# Test best model using all data
x_full = pd.concat([x_train, x_val, x_test], axis=0)
y_full = pd.concat([y_train, y_val, y_test], axis=0)

file_path = '../data/models/full'
filename = f'{model_seq}_{model}'
test_best_model(x_data=x_full, y_data=y_full,
                dataset_name='total', filename=filename,
                print_metrics_file=True, print_pred_file=True,
                roc_data=True, file_path=file_path)
```

```
File 12_RF generated
File 12_RF generated
File 12_RF generated
```

Out[ ]: 'Full set ROC_AUC score: 0.6777012519674329 - KS score: 0.2446919395925966 - GINI score: 0.35540250393486583'

Serialize selected model

In the model_evaluation notebook logistic regresion, ramdon forest and xgboost models were evaluated.

For this case the logistic regresion was the selected model.

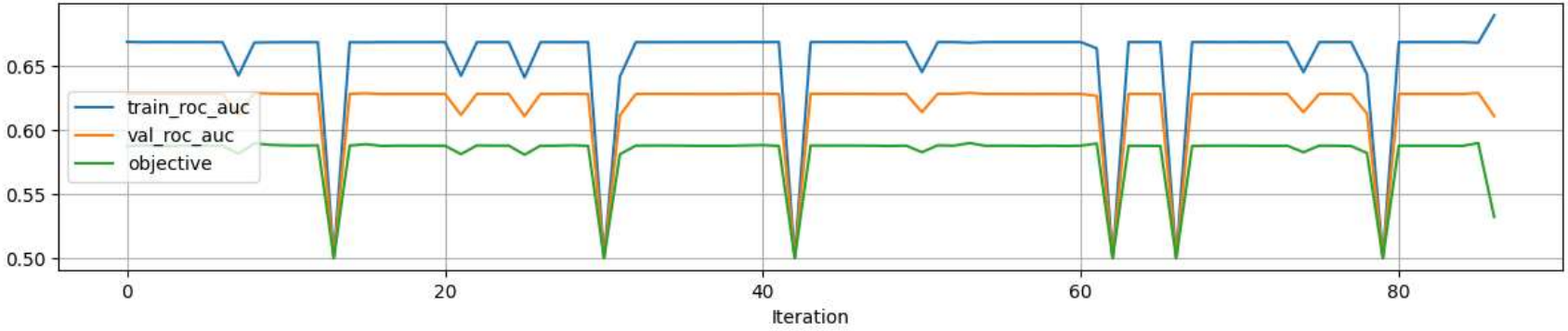
In [ ]:
```
def save_model(filename, object):
    with open(f"{filename}", 'wb') as file:
        pickle.dump(object, file)

def load_model(filename):
```

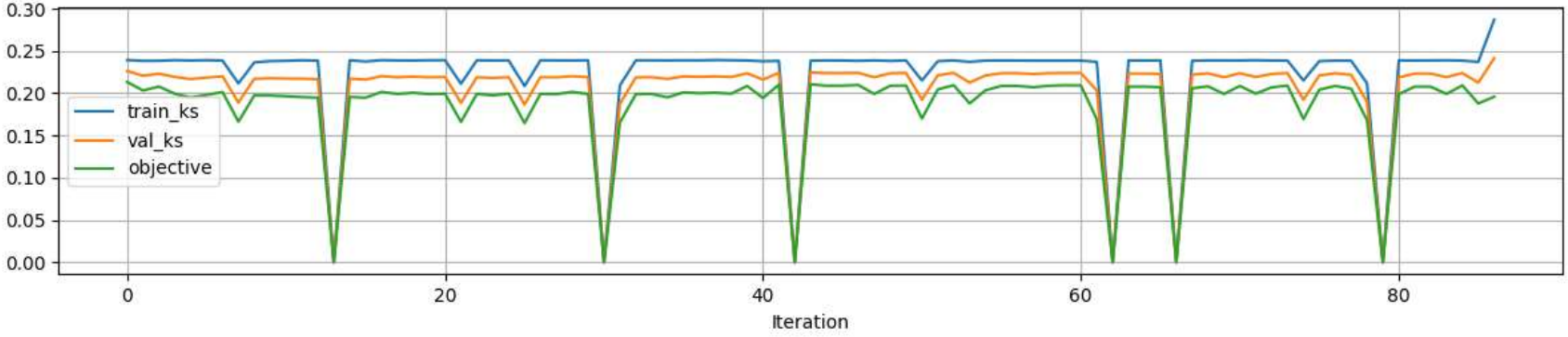
```
    with open(f'{filename}', 'rb') as file:
        model = pickle.load(file)
    return model
```

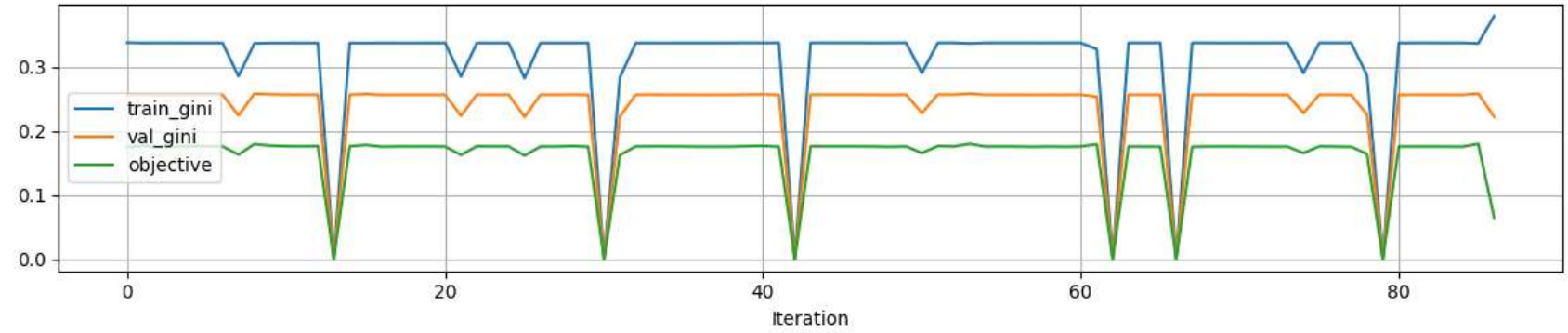
Load selected model hyperparams

In [ ]:

```
JSON = f'../data/models/train/hyperparams/MOD4_RL.json'
with open(JSON, 'r') as archivo:
    datos_json = [json.loads(line) for line in archivo]

params_df = pd.DataFrame(datos_json)
best_params = params_df.loc[params_df.target == params_df.target.max(), 'params'][47]

best_model, best_result = train_and_evaluate_rl(**best_params)
```

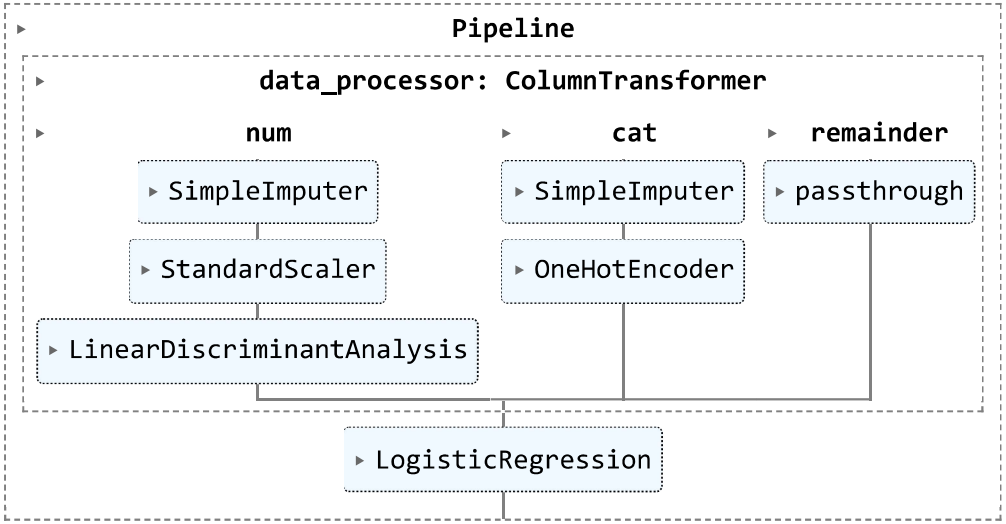






In [ ]:
```
best_model
```

Out[ ]:



In [ ]:

```
MODEL_PATH = '../model/'
save_model(f'{MODEL_PATH}/rl_model.pkl', best_model)
```

Model Evaluation and Selection

Load Libreries

In [ ]:
```
# Librerias base
import os
import re
import json
from joblib import load
import warnings

# Manipulacion de datos
import pandas as pd
import numpy as np

# Visualizacion
import matplotlib.pyplot as plt
import seaborn as sns
# from IPython.display import clear_output
import collections
from pprint import pprint
%matplotlib inline

# Pipeline
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer, KNNImputer
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# Modelamiento
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
# import xgboost as xgb
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
```

```
# import shap

# Metricas
from sklearn.metrics import confusion_matrix, f1_score, fbeta_score, recall_score, precision_score, roc_auc_score, rc

pd.options.display.max_columns = None
warnings.filterwarnings('ignore')


FILE_PATH = f'../data/models'
```

Auxiliary Functions

In [ ]:

```
def roc_plot(train, test, x_thres=None, y_thres=None, fig_title=None):
    """ Function for plotting roc curve
    """
    fig, axs = plt.subplots(1, 2, figsize=(14, 6))
    fig.suptitle(f"ROC Curve - {fig_title} ", fontsize=20)
    axs[0].plot(train.fpr, train.tpr)
    axs[0].set_title("Training Data", fontdict={'fontname': 'Arial', 'fontsize': 18})
    axs[0].plot([0, 1], [0, 1], 'k--')
    axs[0].set_xlabel('False Positive Rate', fontdict={'fontname': 'Arial', 'fontsize': 18})
    axs[0].set_ylabel('True Positive Rate', fontdict={'fontname': 'Arial', 'fontsize': 18})
    axs[0].set_xticklabels(axs[0].get_xticklabels(), fontdict={'fontname': 'Arial',
                                                               'fontsize': 16})
    axs[0].set_yticklabels(axs[0].get_yticklabels(), fontdict={'fontname': 'Arial',
                                                               'fontsize': 16})
    axs[1].plot(test.fpr, test.tpr, color='orange')
    axs[1].set_title("Test Data", fontdict={'fontname': 'Arial', 'fontsize': 18})
    axs[1].plot([0, 1], [0, 1], 'k--')
    axs[1].scatter(x_thres, y_thres, color='red')
    axs[1].set_xlabel('False Positive Rate', fontdict={'fontname': 'Arial', 'fontsize': 18})
    axs[1].set_ylabel('True Positive Rate', fontdict={'fontname': 'Arial', 'fontsize': 18})
    axs[1].set_xticklabels(axs[1].get_xticklabels(), fontdict={'fontname': 'Arial',
                                                               'fontsize': 16})
    axs[1].set_yticklabels(axs[1].get_yticklabels(), fontdict={'fontname': 'Arial',
                                                               'fontsize': 16})
    plt.show()
```

```python
def fpr_tpr_plot(data, title=None, x_thres=None, y_thres=None):
    df = data.loc[data.threshold <= 1, :].copy()
    fig, ax = plt.subplots(figsize=(7, 5))
    ax.scatter(df.threshold,np.abs(df.fpr + df.tpr - 1))
    ax.scatter(x_thres, y_thres, color='red')
    ax.set_xlabel("Threshold")
    ax.set_ylabel("|FPR + TPR - 1|")
    ax.set_title(f"FPR - TPR Curve- {title}")
    plt.show()


def precision_recall_plot(train, test, x_thres=None, y_thres=None, fig_title=None):
    """Function for plotting precision - recall plot
    """
    fig, axs = plt.subplots(1, 2, figsize=(14, 6))
    fig.suptitle(f"Precision-Recall Curve - {fig_title}")
    axs[0].plot(train.recall, train.precision)
    axs[0].set_title("Training Data")
    # axs[0].plot([0, 1], [0, 1], 'k--')
    axs[0].set_xlabel('Recall')
    axs[0].set_ylabel('Precision')
    axs[1].plot(test.recall, test.precision, color='orange')
    axs[1].set_title("Test Data")
    axs[1].scatter(x_thres, y_thres, color='red')
    # axs[1].plot([0, 1], [0, 1], 'k--')
    axs[1].set_xlabel('Recall')
    axs[1].set_ylabel('Precision')
    plt.show()


def fscore_approve_plot(prec_rec_data, pred_data, beta=None,
                        base_thres=None, alt_thres=None, fig_title=None):
    """ Function for plotting Fbeta-score and approval curve
    """
    approved = []
    reject = []
    for p in prec_rec_data.threshold:
        y_test_preds = []
        for prob in pred_data.predicted_bad:
            if prob > p:
                y_test_preds.append(1)
            else:
```

```
            y_test_preds.append(0)

        # Rechazados
        reject.append(np.mean(y_test_preds))
        # Aprobados
        approved.append(1 - np.mean(y_test_preds))

    beta = 2 if beta is None else beta
    fscore = ((1 + beta**2) * prec_rec_data.precision * prec_rec_data.recall) / (beta**2 * prec_rec_data.precision +
    index = np.argmax(fscore)
    opt_thres = prec_rec_data.threshold[index]
    opt_f = fscore[index]
    opt_prec = prec_rec_data.precision[index]
    opt_rec = prec_rec_data.recall[index]

    fig, axes = plt.subplots(1, 2, figsize=(14,6))
    axes[0].plot(prec_rec_data.threshold, fscore, label=f'f{int(beta)}_score')
    axes[0].plot(prec_rec_data.threshold, prec_rec_data.precision, label='Precision')
    axes[0].plot(prec_rec_data.threshold, prec_rec_data.recall, label='Recall')
    axes[0].axvline(opt_thres, linestyle='--', color='black', label='Threshold')
    if base_thres is not None:
        axes[0].axvline(base_thres, linestyle='--', color='red', label='Base Thresh') # |fpr + tpr - 1|
    if alt_thres is not None:
        axes[0].axvline(alt_thres, linestyle='--', color='b', label='Alt Thresh')
    axes[0].set_xlabel('Probability Threshold', fontdict={'fontname': 'Arial', 'fontsize': 18})
    axes[0].set_ylabel('f-beta score', fontdict={'fontname': 'Arial', 'fontsize': 18})
    axes[0].set_title(f"F{int(beta)}_score Curve - {fig_title}", fontdict={'fontname': 'Arial', 'fontsize': 20})
    axes[0].set_xticklabels(axes[0].get_xticklabels(), fontdict={'fontname': 'Arial',
                                                                 'fontsize': 16})
    axes[0].set_yticklabels(axes[0].get_yticklabels(), fontdict={'fontname': 'Arial',
                                                                 'fontsize': 16})
    axes[0].legend(loc='upper right', bbox_to_anchor=(1, 1))

    axes[1].plot(prec_rec_data.threshold, approved, label='Preaprobados')
    axes[1].axvline(opt_thres, linestyle='--', color='black', label='Threshold')
    if base_thres is not None:
        axes[1].axvline(base_thres, linestyle='--', color='red', label='Base Thresh') # |fpr + tpr - 1|
    if alt_thres is not None:
        axes[1].axvline(alt_thres, linestyle='--', color='b', label='Alt Thresh')
    axes[1].set_xlabel('Probability Threshold', fontdict={'fontname': 'Arial', 'fontsize': 18})
    axes[1].set_ylabel('% Approvals', fontdict={'fontname': 'Arial', 'fontsize': 18})
    axes[1].set_title(f"Approvals Curve - {fig_title}", fontdict={'fontname': 'Arial', 'fontsize': 20})
```

```
    axes[1].set_xticklabels(axes[1].get_xticklabels(), fontdict={'fontname': 'Arial',
                                                                 'fontsize': 16})
    axes[1].set_yticklabels(axes[1].get_yticklabels(), fontdict={'fontname': 'Arial',
                                                                 'fontsize': 16})
    axes[1].legend(loc='upper left')

    print("threshold: ", opt_thres)
    print("F-Score: ", opt_f)
    print("Precision: ", opt_prec)
    print("Recall: ", opt_rec)

    plt.show()


def best_beta(data):
    """Function for finding best beta
    """
    betas = np.linspace(0.1, 10, 100)
    betas_df = data.copy()
    for beta in betas:
        betas_df[f'f_{round(beta, 4)}'] = ((1 + beta**2) * data.precision * data.recall) / (beta**2 * data.precision

    betas_df = betas_df.fillna(0)

    betas_series = betas_df.loc[:, 'f_0.1': 'f_10.0'].apply(lambda col: col.max(), axis=0)
    best_beta = betas_series.index[np.argmax(betas_series)]
    best_beta_val = betas_series[np.argmax(betas_series)]
    # betas_df.loc[betas_df[best_beta] == best_beta_val, 'threshold']
    return best_beta, best_beta_val


def proba_to_predictions(data, threshold):
    """Function for converting probabilities into predictions based on
    a defined threshold
    """
    pred = []
    for prob in data.predicted_bad:
        if prob > threshold:
            pred.append(1)
        else:
            pred.append(0)
    return pred
```

```python
def metrics_report(y_test, predictions, beta, threshold, model_name):
    """Function for creating metrics report
    """
    # Confusion matrix
    tn, fp, fn, tp = confusion_matrix(y_test, predictions).ravel()
    # Precision
    prec = precision_score(y_test, predictions)
    # Recall
    rec = recall_score(y_test, predictions)
    # F-Beta
    fscore = fbeta_score(y_test, predictions, beta=beta)
    # ROC-AUC
    roc_auc = roc_auc_score(y_test, predictions)
    # GINI and KS
    fpr, tpr, _ = roc_curve(y_test, predictions)
    ks = np.max(abs(fpr - tpr))
    gini = 2 * roc_auc - 1
    # Specificity
    spec = tn / (tn + tp)
    # Predictions
    pred_bad = sum(predictions)
    pred_good = len(predictions) - pred_bad
    # Actual Goods and bads
    act_bad = sum(y_test)
    act_good = len(y_test) - act_bad
    total = int(act_bad + act_good)
    # Approval
    pre_aprob = pred_good / len(predictions)

    metrics = {'Model': [model_name],
               'Threshold': [threshold],
               f'F{beta}-Score': [fscore],
               'Recall': [rec],
               'Precision': [prec],
               'Specificity': [spec],
               'Roc-Auc': [roc_auc],
               'KS': [ks],
               'GINI': [gini],
               'TN': [tn],
               'FP': [fp],
```

```
                'FN': [fn],
                'TP': [tp],
                'Act_good': [act_good],
                'Act_bad': [act_bad],
                'Total': [total],
                'Pred_good': [pred_good],
                'Pred_bad': [pred_bad],
                'Approval': [pre_aprob]
                }

    metrics_df = pd.DataFrame(metrics)
    return metrics_df
```

Models

In this section we will evaluate models performance using roc curve and precision-recall curves.

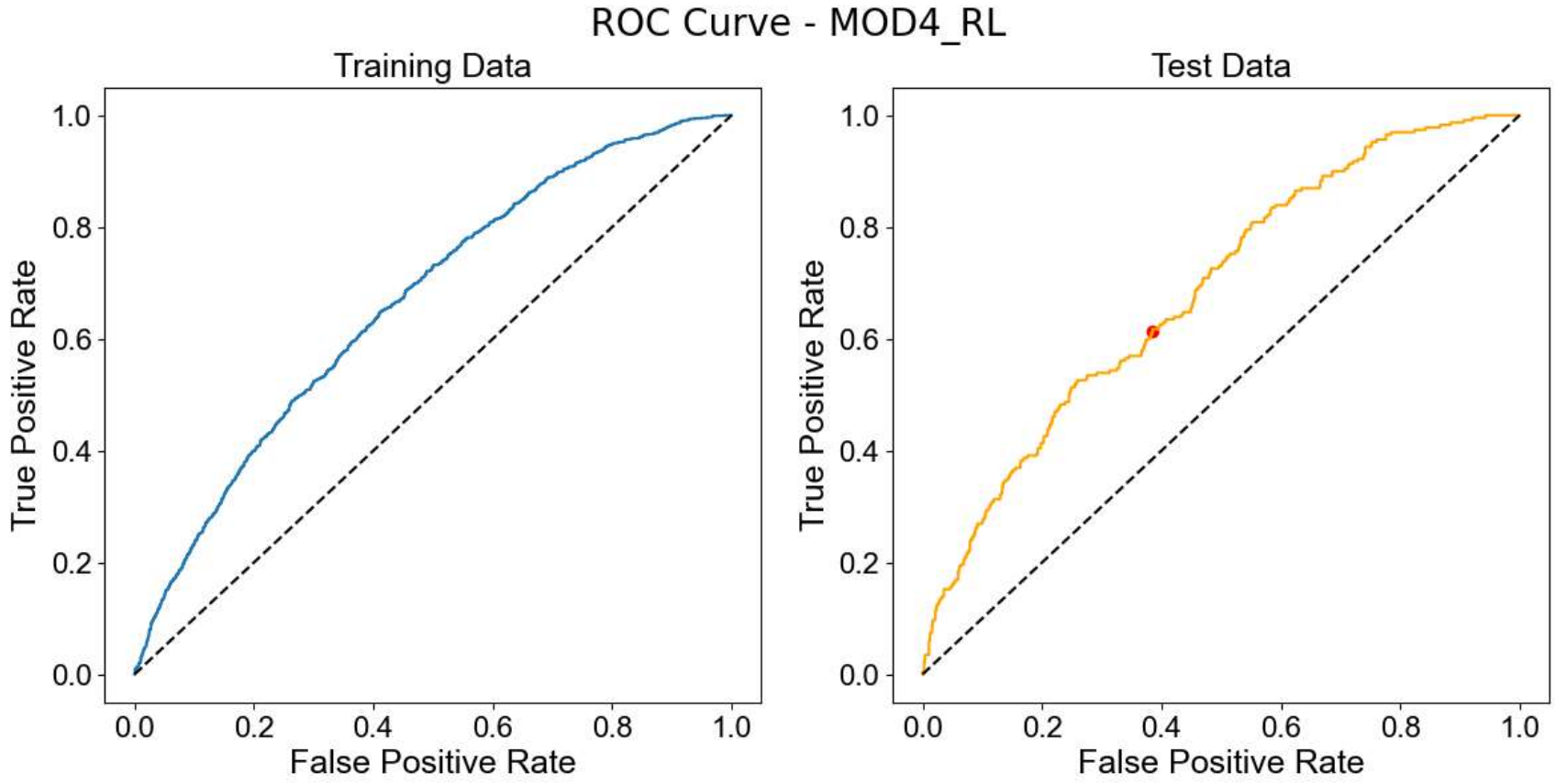
As the main metrics we will focus on the Fbeta-score. So we will find the classification threshold based on the max value of the Fbeta-score. We will also check what is the percentage of approval based on the threshold selected

Logistic Regression

In [ ]:
```
mod4_rl_train_pred = pd.read_csv(f'{FILE_PATH}/train/predictions/MOD_4_RL.csv', sep=',')
mod4_rl_test_pred = pd.read_csv(f'{FILE_PATH}/test/predictions/MOD_4_RL.csv', sep=',')

mod4_rl_train_roc = pd.read_csv(f'{FILE_PATH}/train/roc/ROC_MOD_4_RL.csv', sep=',')
mod4_rl_test_roc = pd.read_csv(f'{FILE_PATH}/test/roc/ROC_MOD_4_RL.csv', sep=',')

mod4_rl_train_prec_rec = pd.read_csv(f'{FILE_PATH}/train/roc/PREC_REC_MOD_4_RL.csv', sep=',')
mod4_rl_test_prec_rec = pd.read_csv(f'{FILE_PATH}/test/roc/PREC_REC_MOD_4_RL.csv', sep=',')
```

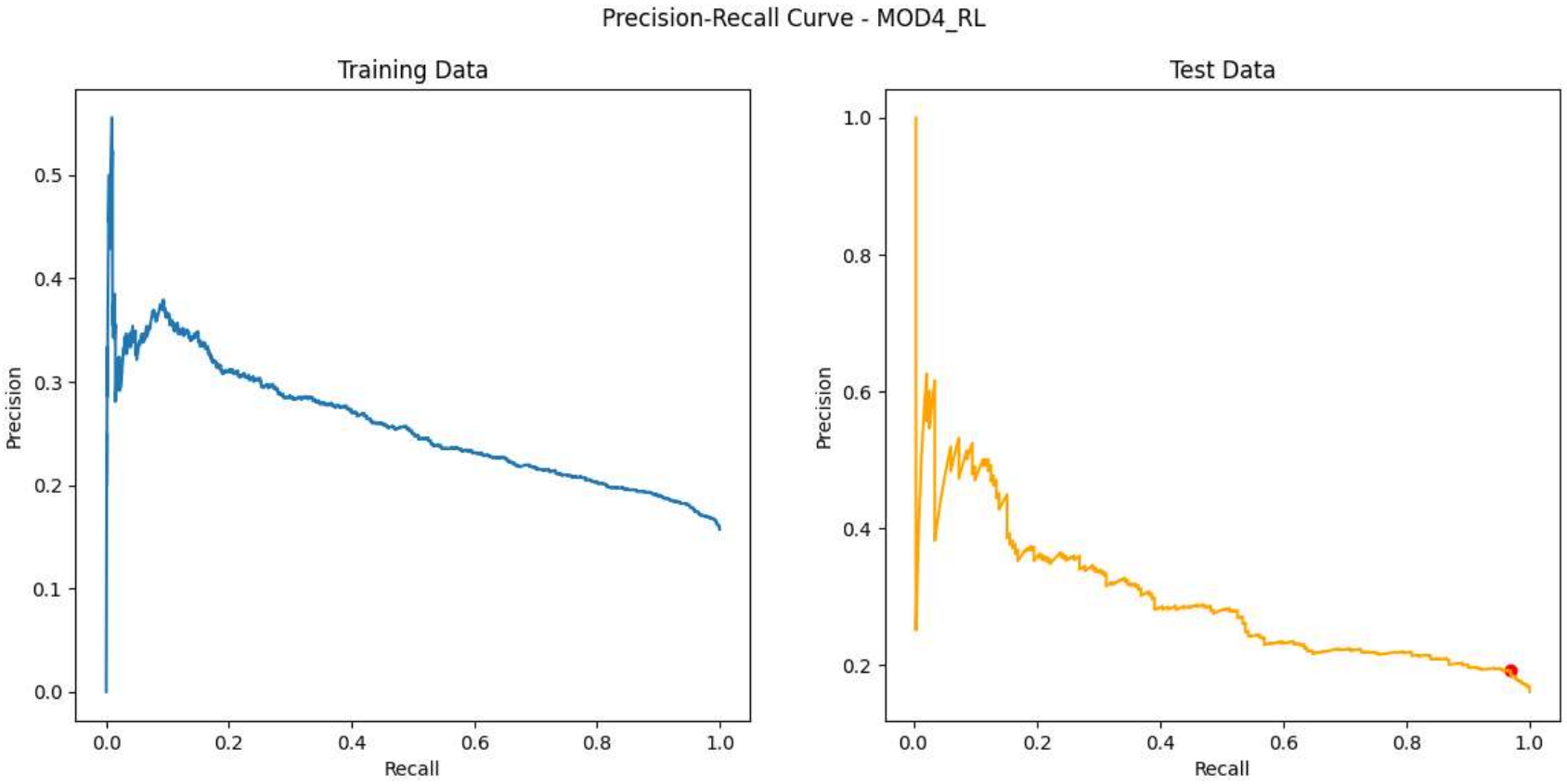
In [ ]:
```
threshold = mod4_rl_test_roc.threshold[np.argmin(np.abs(mod4_rl_test_roc.fpr + mod4_rl_test_roc.tpr - 1))]
fpr, tpr = mod4_rl_test_roc.loc[mod4_rl_test_roc.threshold == threshold, :].values[:, :2][0]
print("threshold: ", threshold)
roc_plot(mod4_rl_train_roc, mod4_rl_test_roc, x_thres=fpr, y_thres=tpr, fig_title='MOD4_RL')
```

```
threshold:  0.9414687847542648
```



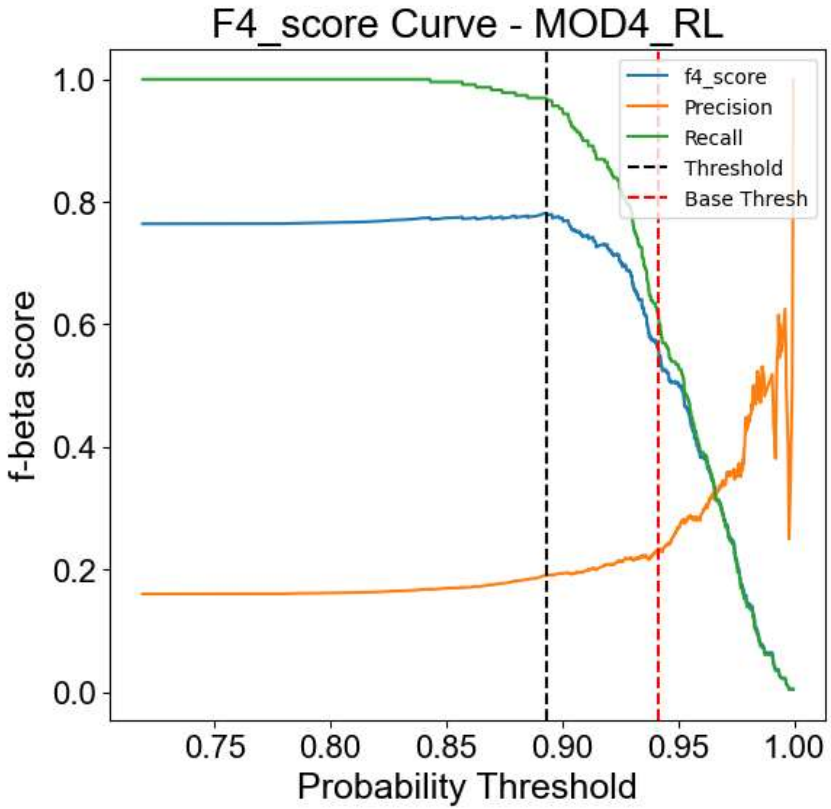
In [ ]:
```
beta = 4
f2_score = ((1 + beta**2) * mod4_rl_test_prec_rec.precision * mod4_rl_test_prec_rec.recall) / (beta**2 * mod4_rl_test
index = np.argmax(f2_score)
opt_thres = mod4_rl_test_prec_rec.threshold[index]
opt_f2 = f2_score[index]
opt_prec = mod4_rl_test_prec_rec.precision[index]
opt_rec = mod4_rl_test_prec_rec.recall[index]

print("threshold: ", opt_thres)
print("F-Score: ", opt_f2)
print("Precision: ", opt_prec)
print("Recall: ", opt_rec)

precision_recall_plot(mod4_rl_train_prec_rec, mod4_rl_test_prec_rec, x_thres=opt_rec, y_thres=opt_prec, fig_title='MO
```

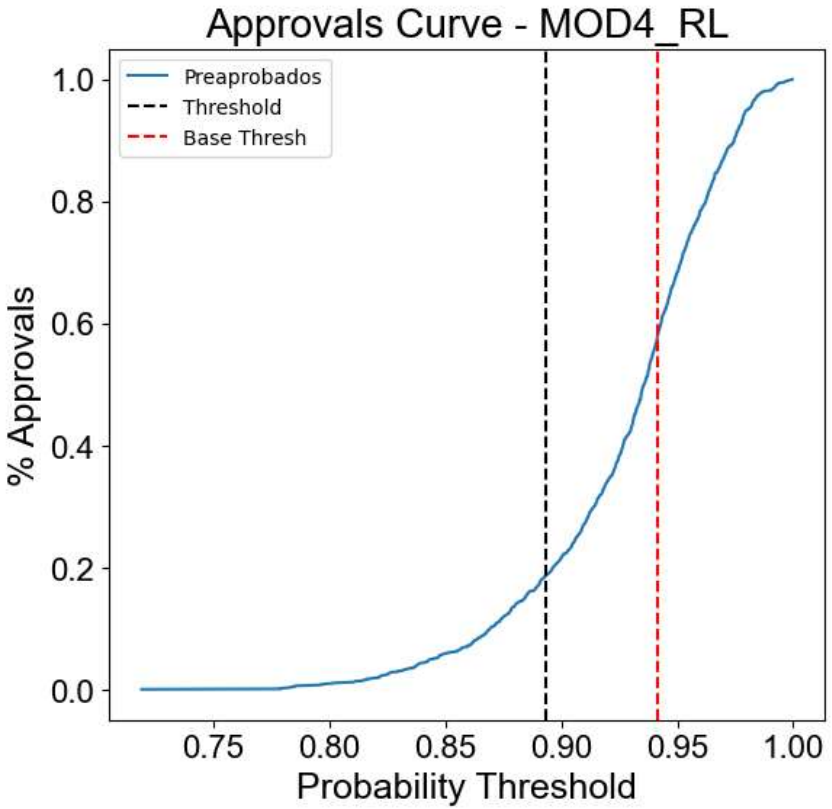
```
threshold:  0.8931466239541553
F-Score:  0.7816494845360826
Precision:  0.1905982905982906
Recall:  0.9695652173913044
```



In [ ]:
```
_ , _, x_thres = mod4_rl_test_roc.loc[mod4_rl_test_roc.threshold == mod4_rl_test_roc.threshold[np.argmin(np.abs(mod4_
fscore_approve_plot(mod4_rl_test_prec_rec, mod4_rl_test_pred, beta=4, base_thres=x_thres, alt_thres=None, fig_title=
```

```
Beta:  10.0
threshold:  0.8931466239541553
F-Score:  0.7816494845360826
Precision:  0.1905982905982906
Recall:  0.9695652173913044
```



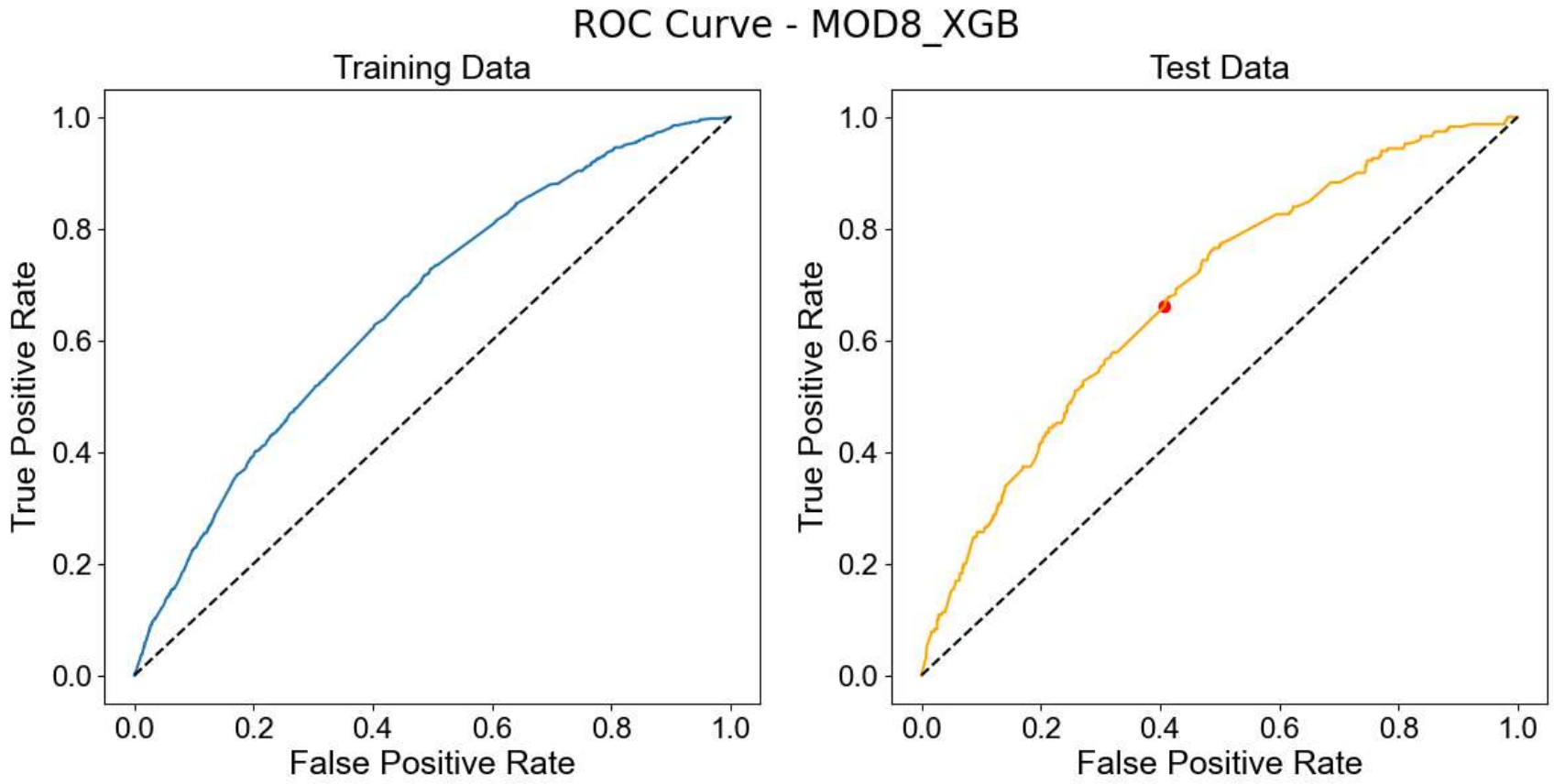


XGBoost

In [ ]:
```
mod8_xgb_train_pred = pd.read_csv(f'{FILE_PATH}/train/predictions/MOD_8_XGB.csv', sep=',')
mod8_xgb_test_pred = pd.read_csv(f'{FILE_PATH}/test/predictions/MOD_8_XGB.csv', sep=',')

mod8_xgb_train_roc = pd.read_csv(f'{FILE_PATH}/train/roc/ROC_MOD_8_XGB.csv', sep=',')
mod8_xgb_test_roc = pd.read_csv(f'{FILE_PATH}/test/roc/ROC_MOD_8_XGB.csv', sep=',')

mod8_xgb_train_prec_rec = pd.read_csv(f'{FILE_PATH}/train/roc/PREC_REC_MOD_8_XGB.csv', sep=',')
mod8_xgb_test_prec_rec = pd.read_csv(f'{FILE_PATH}/test/roc/PREC_REC_MOD_8_XGB.csv', sep=',')
```

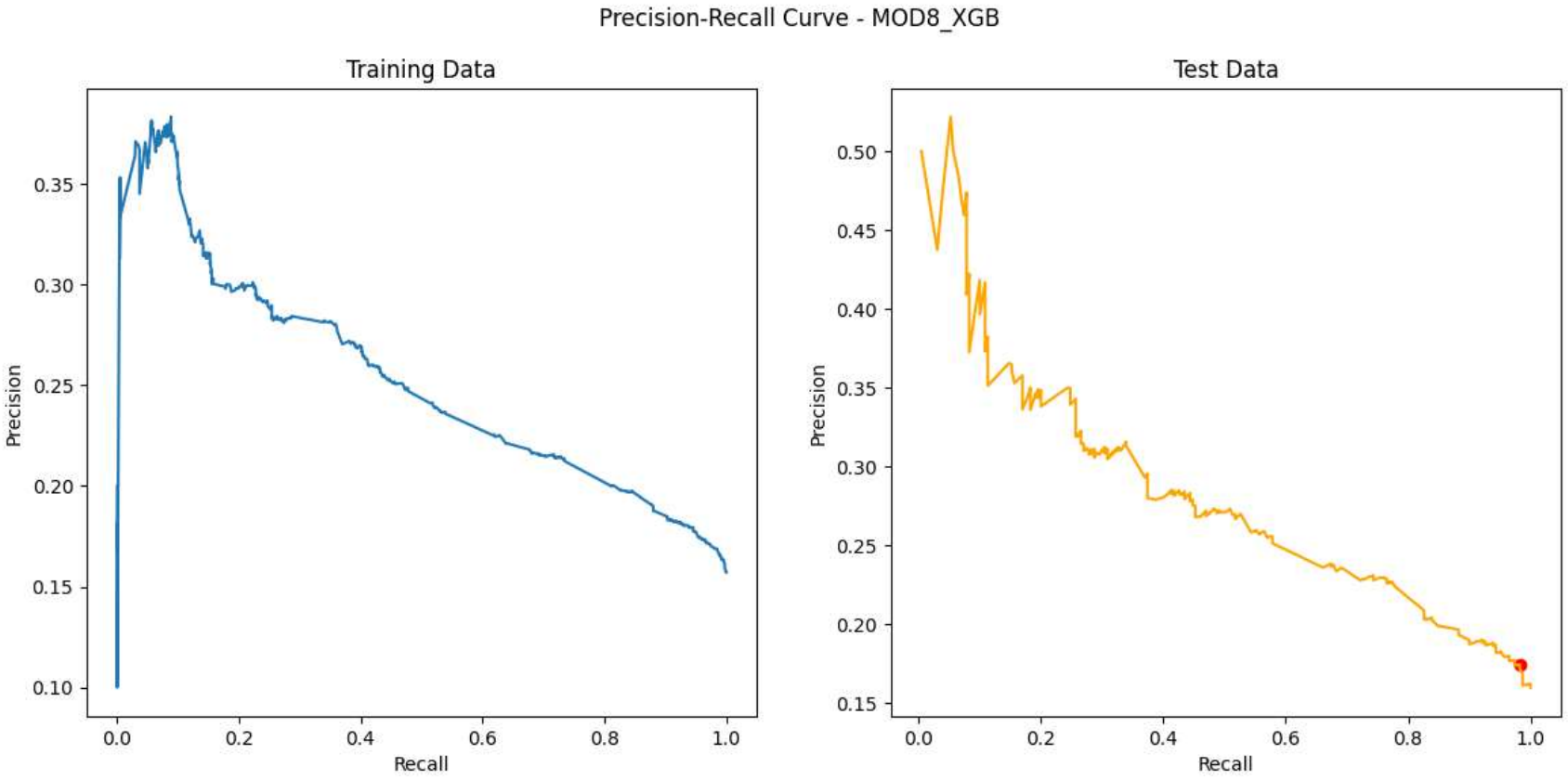
In [ ]:
```
threshold = mod8_xgb_test_roc.threshold[np.argmin(np.abs(mod8_xgb_test_roc.fpr + mod8_xgb_test_roc.tpr - 1))]
fpr, tpr = mod8_xgb_test_roc.loc[mod8_xgb_test_roc.threshold == threshold, :].values[:, :2][0]
print("threshold: ", threshold)
roc_plot(mod8_xgb_train_roc, mod8_xgb_test_roc, x_thres=fpr, y_thres=tpr, fig_title='MOD8_XGB')
```

```
threshold:  0.85104173
```



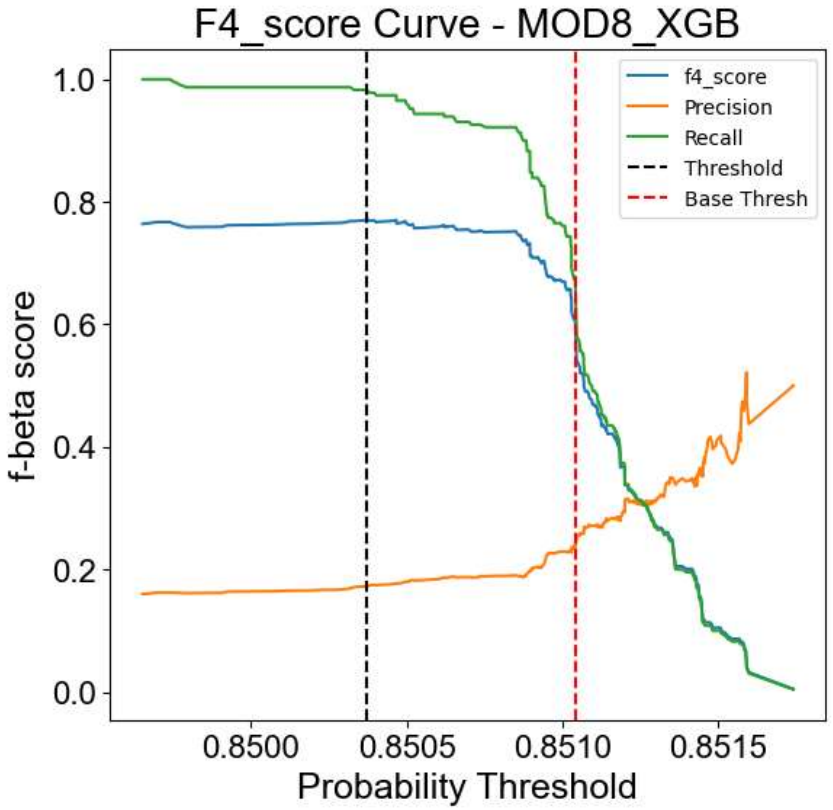
In [ ]:
```
beta = 4
f2_score = ((1 + beta**2) * mod8_xgb_test_prec_rec.precision * mod8_xgb_test_prec_rec.recall) / (beta**2 * mod8_xgb_t
index = np.argmax(f2_score)
opt_thres = mod8_xgb_test_prec_rec.threshold[index]
opt_f2 = f2_score[index]
opt_prec = mod8_xgb_test_prec_rec.precision[index]
opt_rec = mod8_xgb_test_prec_rec.recall[index]

print("threshold: ", opt_thres)
print("F-Score: ", opt_f2)
print("Precision: ", opt_prec)
print("Recall: ", opt_rec)

precision_recall_plot(mod8_xgb_train_prec_rec, mod8_xgb_test_prec_rec, x_thres=opt_rec, y_thres=opt_prec, fig_title=
```

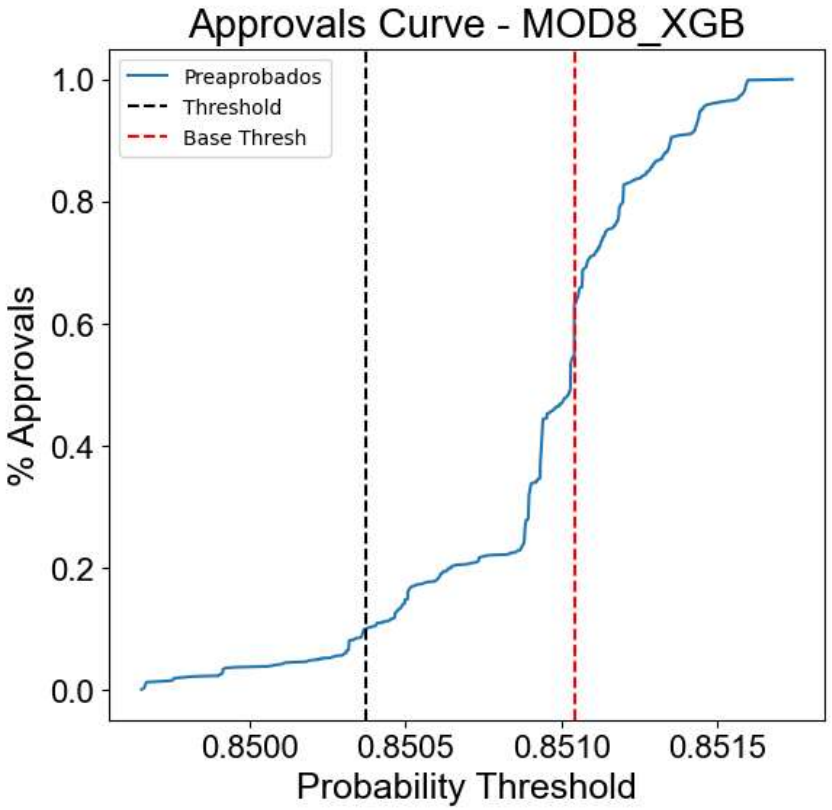
```
threshold:  0.8503717
F-Score:  0.7724165661439485
Precision:  0.1746522411128284
Recall:  0.982608695652174
```



In [ ]:
```
_ , _, x_thres = mod8_xgb_test_roc.loc[mod8_xgb_test_roc.threshold == mod8_xgb_test_roc.threshold[np.argmin(np.abs(mo
fscore_approve_plot(mod8_xgb_test_prec_rec, mod8_xgb_test_pred, beta=4, base_thres=x_thres, alt_thres=None, fig_title
```

```
threshold:  0.8503717
F-Score:  0.7724165661439485
Precision:  0.1746522411128284
Recall:  0.982608695652174
```





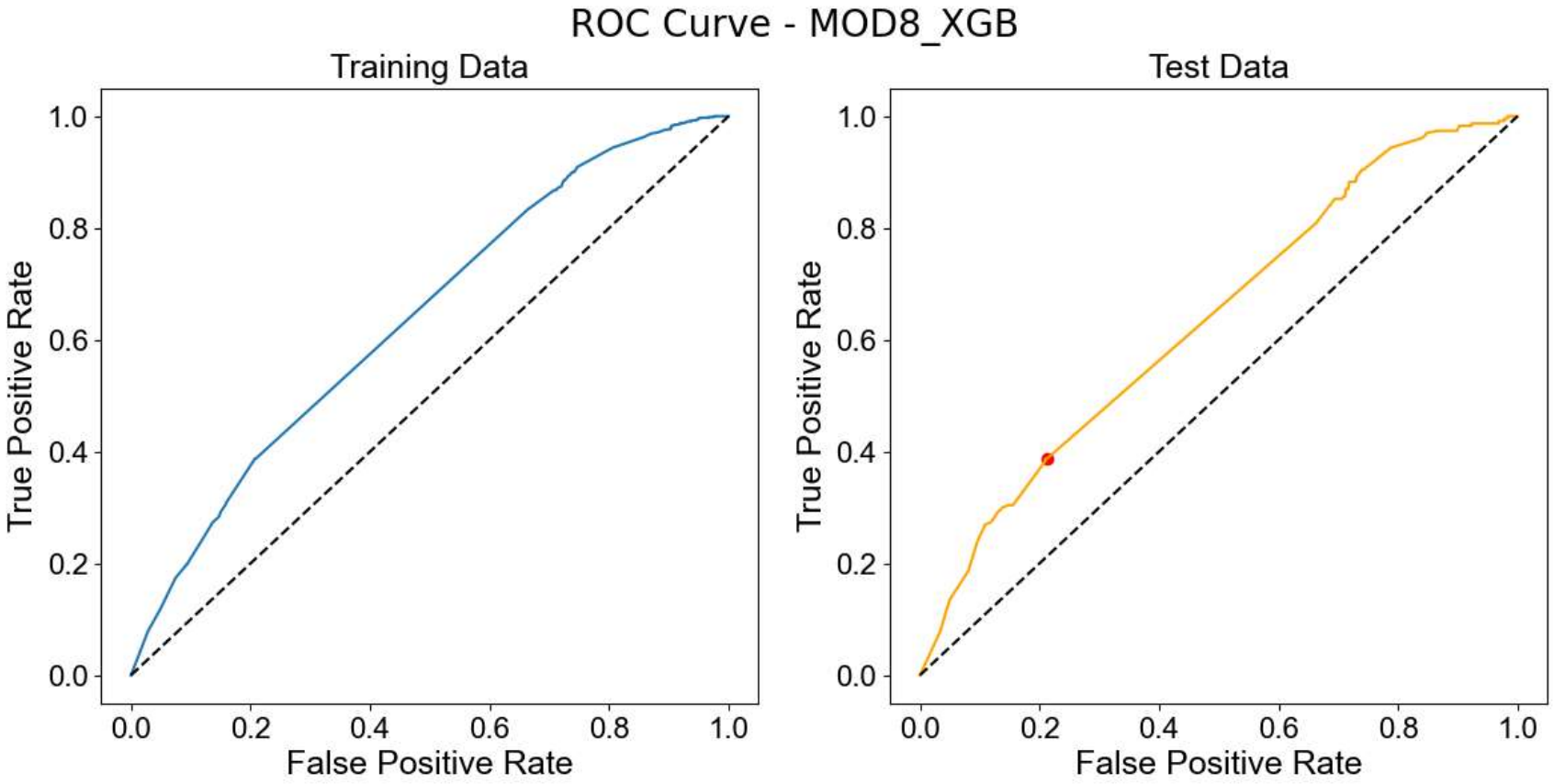
Random Forest

In [ ]:
```
mod11_rf_train_pred = pd.read_csv(f'{FILE_PATH}/train/predictions/MOD_11_RF.csv', sep=',')
mod11_rf_test_pred = pd.read_csv(f'{FILE_PATH}/test/predictions/MOD_11_RF.csv', sep=',')

mod11_rf_train_roc = pd.read_csv(f'{FILE_PATH}/train/roc/ROC_MOD_11_RF.csv', sep=',')
mod11_rf_test_roc = pd.read_csv(f'{FILE_PATH}/test/roc/ROC_MOD_11_RF.csv', sep=',')

mod11_rf_train_prec_rec = pd.read_csv(f'{FILE_PATH}/train/roc/PREC_REC_MOD_11_RF.csv', sep=',')
mod11_rf_test_prec_rec = pd.read_csv(f'{FILE_PATH}/test/roc/PREC_REC_MOD_11_RF.csv', sep=',')
```

In [ ]:
```
threshold = mod11_rf_test_roc.threshold[np.argmin(np.abs(mod11_rf_test_roc.fpr + mod11_rf_test_roc.tpr - 1))]
fpr, tpr = mod11_rf_test_roc.loc[mod11_rf_test_roc.threshold == threshold, :].values[:, :2][0]
```

```
print("threshold: ", threshold)
roc_plot(mod11_rf_train_roc, mod11_rf_test_roc, x_thres=fpr, y_thres=tpr, fig_title='MOD8_XGB')
```

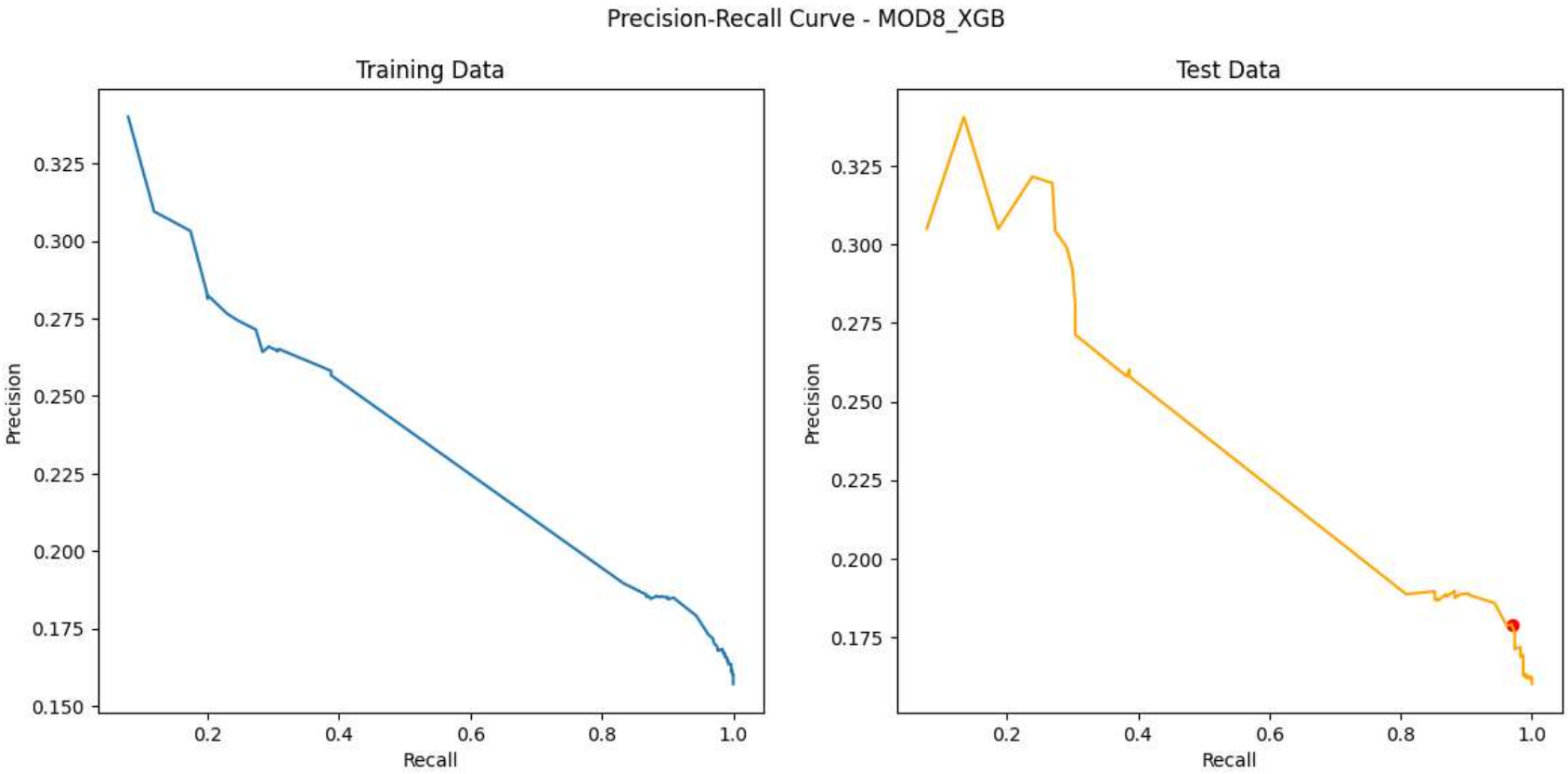
```
threshold:  0.9550156700915294
```



In [ ]:
```
beta = 4
f2_score = ((1 + beta**2) * mod11_rf_test_prec_rec.precision * mod11_rf_test_prec_rec.recall) / (beta**2 * mod11_rf_t
index = np.argmax(f2_score)
opt_thres = mod11_rf_test_prec_rec.threshold[index]
opt_f2 = f2_score[index]
opt_prec = mod11_rf_test_prec_rec.precision[index]
opt_rec = mod11_rf_test_prec_rec.recall[index]

print("threshold: ", opt_thres)
print("F-Score: ", opt_f2)
print("Precision: ", opt_prec)
```

```
print("Recall: ", opt_rec)

precision_recall_plot(mod11_rf_train_prec_rec, mod11_rf_test_prec_rec, x_thres=opt_rec, y_thres=opt_prec, fig_title=
```

```
threshold:  0.920715811606211
F-Score:  0.7695899309784815
Precision:  0.1789727126805778
Recall:  0.9695652173913044
```



Confusion Matrix

In [ ]:
```
models = {
    'MOD4_RL':[mod4_rl_test_prec_rec, mod4_rl_test_pred, mod4_rl_test_roc],
    'MOD8_XGB':[mod8_xgb_test_prec_rec, mod8_xgb_test_pred, mod8_xgb_test_roc],
```

```
    'MOD11_RF':[mod11_rf_test_prec_rec, mod11_rf_test_pred, mod11_rf_test_roc]
}
```

In [ ]:

```
use_base_thresh = False
beta = 4
cols = ['Model', 'Threshold', f'F{beta}-Score', 'Recall',       'Precision',
        'Specificity', 'Roc-Auc', 'KS', 'GINI', 'TN', 'FP',     'FN', 'TP',
        'Act_good', 'Act_bad', 'Total', 'Pred_good', 'Pred_bad',
        'Approval']
models_resume = pd.DataFrame(columns=cols)
for key, values in models.items():
    f_score = ((1 + beta**2) * values[0].precision * values[0].recall) / (beta**2 * values[0].precision + values[0].r
    index = np.argmax(f_score)

    if use_base_thresh:
        opt_thres = values[2].threshold[np.argmin(np.abs(values[2].fpr + values[2].tpr - 1))]
    else:
        opt_thres = values[0].threshold[index]

    y_test = values[1].actual_bad.values
    model_preds = proba_to_predictions(values[1], threshold=opt_thres)
    df = metrics_report(y_test=y_test, predictions=model_preds, beta=beta, threshold=opt_thres, model_name=key)

    models_resume = pd.concat([models_resume, df], axis=0, ignore_index=True)

models_resume = models_resume.reset_index(drop=True)

"""
     A1 |  A0
---------------
P1|  TP | FP
---------------
P0|  FN | TN

"""

(
models_resume
.style
.background_gradient(subset=['Recall', 'Precision', 'Roc-Auc', 'KS', f'F{beta}-Score', 'TN', 'FN', 'FP', 'TP', 'Appro
```

```
.hide(axis='index')
 )
```

Out[ ]:

Model	Threshold	F4-Score	Recall	Precision	Specificity	Roc-Auc	KS	GINI	TN	FP	FN	TP	Act_good
MOD4_RL	0.893147	0.778305	0.965217	0.189906	0.539419	0.590314	0.180627	0.180627	260	947	8	222	1207
MOD8_XGB	0.850372	0.769153	0.978261	0.174014	0.381868	0.546711	0.093422	0.093422	139	1068	5	225	1207
MOD11_RF	0.920716	0.766761	0.965217	0.178744	0.457213	0.560073	0.120147	0.120147	187	1020	8	222	1207