loan eligibility

August 25, 2025

1 Loan Eligibility with Ensemble Boosting

1.1 Table of Contents

pointer to the dataset

1.2 1. Abstract

Banks and all sort of credit institutes which provide *loans* face great challenges when a new potential customer asks for a loan.
Those challenges translate into *risks* if not well balanced may end up with catastrophic consequences for all clients in the financial-chain and may have a big impact in the outside world too.
Technology can mitigate these risks.

By gathering a huge amout of *data*, and by applying some analytics we can see the *hidden shape* of the risks that credit institutes take.
Then, thanks to some state-of-the-art algotithms we can predict the potential risk that a new client may represent for a bank.

At the end of the day, the aim is to provide a robust model to support diffuclt *decision-making* scenarios.

The purpose of the notebook is to show and manipulate a huge dataset collected from Kaggle, then after some *feature engineering* we apply and compare the three State-of-the-Art *ensemble learning* algorithms: - AdaBoost - GradientBoost - XGBoost

Based on statistics we decide which the best one is.
Further conlusions and possible expansions are presented.

[1]:
```
#import of all libraries and packages
import kagglehub #for downloading automatically the dataset from Kaggle IF it␣
↪has changed
import pandas as pd #data manipulation
import numpy as np #numerical manipulation
import matplotlib.pyplot as plt #data visualization
import matplotlib.patches as mpatches #for distinguishing the dots in the␣
↪scatter matrix plots
import sklearn #ML
from sklearn.preprocessing import OrdinalEncoder, OneHotEncoder #handling␣
↪categorical data
import sys #get the real size of the DataFrames
```

1.3 2. Dataset Import

[2]:
```
#Code to fetch the data
try:
    #Download latest version
    path = kagglehub.dataset_download("yasserh/loan-default-dataset")

    #to check where the dataset has been located
    #print("Path to dataset files:", path)

    loan_data = pd.read_csv(path + '\Loan_Default.csv')

#if something goes wrong (e.g.: connection, wrong path, ...), use the copy of␣
↪the dataset in the directory
except:
    loan_data = pd.read_csv('Loan_Default.csv')
```

[3]:
```
pd.set_option('display.max_columns', None) #display all columns
loan_data.head()
```

[3]:
```
      ID  year loan_limit             Gender approv_in_adv loan_type  \
0  24890  2019         cf  Sex Not Available         nopre     type1
1  24891  2019         cf               Male         nopre     type2
2  24892  2019         cf               Male           pre     type1
3  24893  2019         cf               Male         nopre     type1
```

```
4  24894  2019         cf              Joint           pre     type1

  loan_purpose Credit_Worthiness open_credit business_or_commercial  \
0           p1                l1        nopc                  nob/c
1           p1                l1        nopc                    b/c
2           p1                l1        nopc                  nob/c
3           p4                l1        nopc                  nob/c
4           p1                l1        nopc                  nob/c

   loan_amount  rate_of_interest  Interest_rate_spread  Upfront_charges  \
0       116500               NaN                   NaN              NaN
1       206500               NaN                   NaN              NaN
2       406500              4.56                0.2000            595.0
3       456500              4.25                0.6810              NaN
4       696500              4.00                0.3042              0.0

    term Neg_ammortization interest_only lump_sum_payment  property_value  \
0  360.0           not_neg      not_int         not_lpsm        118000.0
1  360.0           not_neg      not_int             lpsm             NaN
2  360.0           neg_amm      not_int         not_lpsm        508000.0
3  360.0           not_neg      not_int         not_lpsm        658000.0
4  360.0           not_neg      not_int         not_lpsm        758000.0

  construction_type occupancy_type Secured_by total_units   income  \
0                sb             pr       home          1U   1740.0
1                sb             pr       home          1U   4980.0
2                sb             pr       home          1U   9480.0
3                sb             pr       home          1U  11880.0
4                sb             pr       home          1U  10440.0

  credit_type  Credit_Score co-applicant_credit_type    age  \
0         EXP           758                      CIB  25-34
1        EQUI           552                      EXP  55-64
2         EXP           834                      CIB  35-44
3         EXP           587                      CIB  45-54
4        CRIF           602                      EXP  25-34

  submission_of_application        LTV Region Security_Type  Status  dtir1
0                   to_inst  98.728814  south        direct       1   45.0
1                   to_inst        NaN  North        direct       1    NaN
2                   to_inst  80.019685  south        direct       0   46.0
3                  not_inst  69.376900  North        direct       0   42.0
4                  not_inst  91.886544  North        direct       0   39.0
```

1.3.1 2.1. Dataset Description

This description may be very helpful through all the notebook's reading, so try to keep it in sight.

- **ID:** client loan application id
- **year:** year of loan application
- **loan limit:** indicates whether the loan is conforming (cf) or non-conforming (ncf)
- **Gender:** gender of the applicant (male, female, joint, sex not available)
- **approv_in_adv:** indicates whether the loan was approved in advance (pre, nopre)
- **loan_type:** type of loan (type1, type2, type3) :
  - *Type 1 (Conventional Loans):* Characterized by higher loan amounts, lower LTV ratios, and stronger credit scores, making them a preferred option for well-qualified, lower-risk borrowers.
  - *Type 2 (Government-Backed Loans):* Typically involve lower loan amounts, higher LTV ratios, and moderate credit scores, indicating they are used by borrowers with smaller down payments who benefit from government-backed programs.
  - *Type 3 (Non-Conventional Loans):* Feature moderate loan amounts, the highest LTV ratios, and lower credit scores, often associated with higher-risk products such as jumbo loans or adjustable-rate mortgages.
- **loan_purpose:** purpose of the loan (p1, p2, p3, p4):
  - *p1 (Home Purchase):* Represents loans taken out for primary residences, often displaying moderate credit scores and higher LTV ratios.
  - *p2 (Home Improvement):* Smaller loan amounts used for property renovations, with lower LTV ratios suggesting homeowners are leveraging built-up equity.
  - *p3 (Refinancing):* Applies to homeowners replacing an existing mortgage, characterized by moderate loan amounts and lower LTV ratios, indicating financial stability.
  - *p4 (Investment Property):* Involves larger loan amounts and higher risk profiles, primarily financed through conventional loans due to restrictions on Government-backed funding for investment properties.
- **Credit_Worthiness:** credit worthiness (l1, l2)
- **open_credit:** indicates whether the applicant has any open credit accounts (opc, nopc)
- **business_or_commercial:** indicates whether the loan is for business/commercial purposes (ob/c - business/commercial, nob/c - personal)
- **loan_amount:** amount of money being borrowed
- **rate_of_interest:** interest rate charged on the loan
- **Interest_rate_spread:** difference between the interest rate on the loan and a benchmark interest rate
- **Upfront_charges:** initial charges associated with securing the loan
- **term:** duration of the loan in months

- **Neg_ammortization:** indicates whether the loan allows for negative amortization (neg_amm, not_neg)
- **interest_only:** indicates whether the loan has an interest-only payment option (int_only, not_int)
- **lump_sum_payment:** indicates if a lump sum payment is required at the end of the loan term (lpsm, not_lpsm)
- **property_value:** value of the property being financed
- **construction_type:** type of construction (sb - site built, mh - manufactured home)
- **occupancy_type:** occupancy type (pr - primary residence, sr - secondary residence, ir - investment property)
- **Secured_by:** specifies the type of collateral securing the loan (home, land)
- **total_units:** number of units in the property being financed (1U, 2U, 3U, 4U)
- **income * :** applicant's annual income
- **credit_type:** applicant's type of credit (CIB - credit information bureau, CRIF - CIRF credit information bureau, EXP - experian, EQUI - equifax)
- **Credit_Score:** applicant's credit score
- **co-applicant_credit_type:** co-applicant's type of credit (CIB - credit information bureau, EXP - experian)
- **age:** the age of the applicant
- **submission_of_application:** indicates how the application was submitted (to_inst - to institution, not_inst - not to institution)
- **LTV:** loan-to-value ratio, calculated as the loan amount divided by the property value
- **Region:** geographic region where the property is located (North, South, Central, North-East)
- **Security_Type:** type of security or collateral backing the loan (direct, indirect)
- **Status:** indicates whether the loan has been defaulted (1) or not (0)
- **dtir1:** debt-to-income ratio

* The annual income is in thousands of dollars.

[4]:
```
loan_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 148670 entries, 0 to 148669
Data columns (total 34 columns):
 #   Column                     Non-Null Count   Dtype
---  ------                     --------------   -----
 0   ID                         148670 non-null  int64
 1   year                       148670 non-null  int64
 2   loan_limit                 145326 non-null  object
 3   Gender                     148670 non-null  object
 4   approv_in_adv              147762 non-null  object
 5   loan_type                  148670 non-null  object
 6   loan_purpose               148536 non-null  object
 7   Credit_Worthiness          148670 non-null  object
 8   open_credit                148670 non-null  object
 9   business_or_commercial     148670 non-null  object
 10  loan_amount                148670 non-null  int64
 11  rate_of_interest           112231 non-null  float64
 12  Interest_rate_spread       112031 non-null  float64
 13  Upfront_charges            109028 non-null  float64
 14  term                       148629 non-null  float64
 15  Neg_ammortization          148549 non-null  object
 16  interest_only              148670 non-null  object
 17  lump_sum_payment           148670 non-null  object
 18  property_value             133572 non-null  float64
 19  construction_type          148670 non-null  object
 20  occupancy_type             148670 non-null  object
 21  Secured_by                 148670 non-null  object
 22  total_units                148670 non-null  object
 23  income                     139520 non-null  float64
 24  credit_type                148670 non-null  object
 25  Credit_Score               148670 non-null  int64
 26  co-applicant_credit_type   148670 non-null  object
 27  age                        148470 non-null  object
 28  submission_of_application  148470 non-null  object
 29  LTV                        133572 non-null  float64
 30  Region                     148670 non-null  object
 31  Security_Type              148670 non-null  object
 32  Status                     148670 non-null  int64
 33  dtir1                      124549 non-null  float64
dtypes: float64(8), int64(5), object(21)
memory usage: 38.6+ MB
```

Notice: above the `info()` command, gives us a memory usage size that is actually a bit misleading, see.
That is the reason why we decided to use: `sys.getsizeof()`, since this data will be quite interesting especially for the next sections.

[5]:
```
sys.getsizeof(loan_data)
```

[5]: 207281850

[6]:
```
loan_data.describe()
```

[6]:
```
                  ID      year   loan_amount  rate_of_interest  \
count  148670.000000  148670.0  1.486700e+05     112231.000000
mean    99224.500000    2019.0  3.311177e+05          4.045476
std     42917.476598       0.0  1.839093e+05          0.561391
min     24890.000000    2019.0  1.650000e+04          0.000000
25%     62057.250000    2019.0  1.965000e+05          3.625000
50%     99224.500000    2019.0  2.965000e+05          3.990000
75%    136391.750000    2019.0  4.365000e+05          4.375000
max    173559.000000    2019.0  3.576500e+06          8.000000

       Interest_rate_spread  Upfront_charges           term  property_value  \
count         112031.000000    109028.000000  148629.000000    1.335720e+05
mean               0.441656      3224.996127     335.136582    4.978935e+05
std                0.513043      3251.121510      58.409084    3.599353e+05
min               -3.638000         0.000000      96.000000    8.000000e+03
25%                0.076000       581.490000     360.000000    2.680000e+05
50%                0.390400      2596.450000     360.000000    4.180000e+05
75%                0.775400      4812.500000     360.000000    6.280000e+05
max                3.357000     60000.000000     360.000000    1.650800e+07

              income   Credit_Score            LTV         Status  \
count  139520.000000  148670.000000  133572.000000  148670.000000
mean     6957.338876     699.789103      72.746457       0.246445
std      6496.586382     115.875857      39.967603       0.430942
min         0.000000     500.000000       0.967478       0.000000
25%      3720.000000     599.000000      60.474860       0.000000
50%      5760.000000     699.000000      75.135870       0.000000
75%      8520.000000     800.000000      86.184211       0.000000
max    578580.000000     900.000000    7831.250000       1.000000

               dtir1
count  124549.000000
mean       37.732932
std        10.545435
min         5.000000
25%        31.000000
50%        39.000000
75%        45.000000
max        61.000000
```

Since the columns: **ID** and **year** are not meaningful for any purposes, we can drop them since the very beginning and still be compliant to the preprocessing best practices.

[7]:
```
#since they are not meaningful columns at all, we can drop them for the whole
 ↪dataset
loan_data = loan_data.drop('ID', axis=1)
loan_data = loan_data.drop('year', axis=1)
```

[8]:
```
#to get insights on the categorical data
categorical_values_counts = list()
for cat in loan_data.select_dtypes(include=['object']).columns: #extract the
 ↪list of all categorical features
    categorical_values_counts.append(loan_data[f"{cat}"].value_counts())
    categorical_values_counts.append('-'*45) #for spacing them
categorical_values_counts
```

[8]:
```
[cf     135348
 ncf      9978
 Name: loan_limit, dtype: int64,
 '---------------------------------------------',
 Male                 42346
 Joint                41399
 Sex Not Available    37659
 Female               27266
 Name: Gender, dtype: int64,
 '---------------------------------------------',
 nopre    124621
 pre       23141
 Name: approv_in_adv, dtype: int64,
 '---------------------------------------------',
 type1    113173
 type2     20762
 type3     14735
 Name: loan_type, dtype: int64,
 '---------------------------------------------',
 p3    55934
 p4    54799
 p1    34529
 p2     3274
 Name: loan_purpose, dtype: int64,
 '---------------------------------------------',
 l1    142344
 l2      6326
 Name: Credit_Worthiness, dtype: int64,
 '---------------------------------------------',
 nopc    148114
 opc        556
 Name: open_credit, dtype: int64,
 '---------------------------------------------',
 nob/c    127908
```

```
b/c       20762
Name: business_or_commercial, dtype: int64,
'----------------------------------------------',
not_neg    133420
neg_amm     15129
Name: Neg_ammortization, dtype: int64,
'----------------------------------------------',
not_int     141560
int_only      7110
Name: interest_only, dtype: int64,
'----------------------------------------------',
not_lpsm    145286
lpsm          3384
Name: lump_sum_payment, dtype: int64,
'----------------------------------------------',
sb    148637
mh        33
Name: construction_type, dtype: int64,
'----------------------------------------------',
pr    138201
ir      7340
sr      3129
Name: occupancy_type, dtype: int64,
'----------------------------------------------',
home    148637
land        33
Name: Secured_by, dtype: int64,
'----------------------------------------------',
1U    146480
2U      1477
3U       393
4U       320
Name: total_units, dtype: int64,
'----------------------------------------------',
CIB     48152
CRIF    43901
EXP     41319
EQUI    15298
Name: credit_type, dtype: int64,
'----------------------------------------------',
CIB    74392
EXP    74278
Name: co-applicant_credit_type, dtype: int64,
'----------------------------------------------',
45-54    34720
35-44    32818
55-64    32534
```

```
65-74    20744
25-34    19142
>74       7175
<25       1337
Name: age, dtype: int64,
'---------------------------------------------',
to_inst     95814
not_inst    52656
Name: submission_of_application, dtype: int64,
'---------------------------------------------',
North         74722
south         64016
central        8697
North-East     1235
Name: Region, dtype: int64,
'---------------------------------------------',
direct      148637
Indriect        33
Name: Security_Type, dtype: int64,
'---------------------------------------------']
```

1.3.2 2.2. Missing Values

We can see that the biggest issues in terms of missing values come from: **`Upfront_charges`**, **`Interest_rate_spread`**, **`rate_of_interest`**. we could have some criticalities also on other features but still negligible, even though we have to keep in mind that the NaN are spread out as:

Numerical

- 39642 Upfront_charges
- 36639 Interest_rate_spread
- 36439 rate_of_interest
- 24121 dtir1
- 15098 LTV
- 15098 property_value
- 9150 income
- 41 term

Categorical

- 3344 loan_limit
- 908 approv_in_adv
- 200 age
- 200 submission_of_application
- 136 loan_purpose
- 121 Neg_ammortization

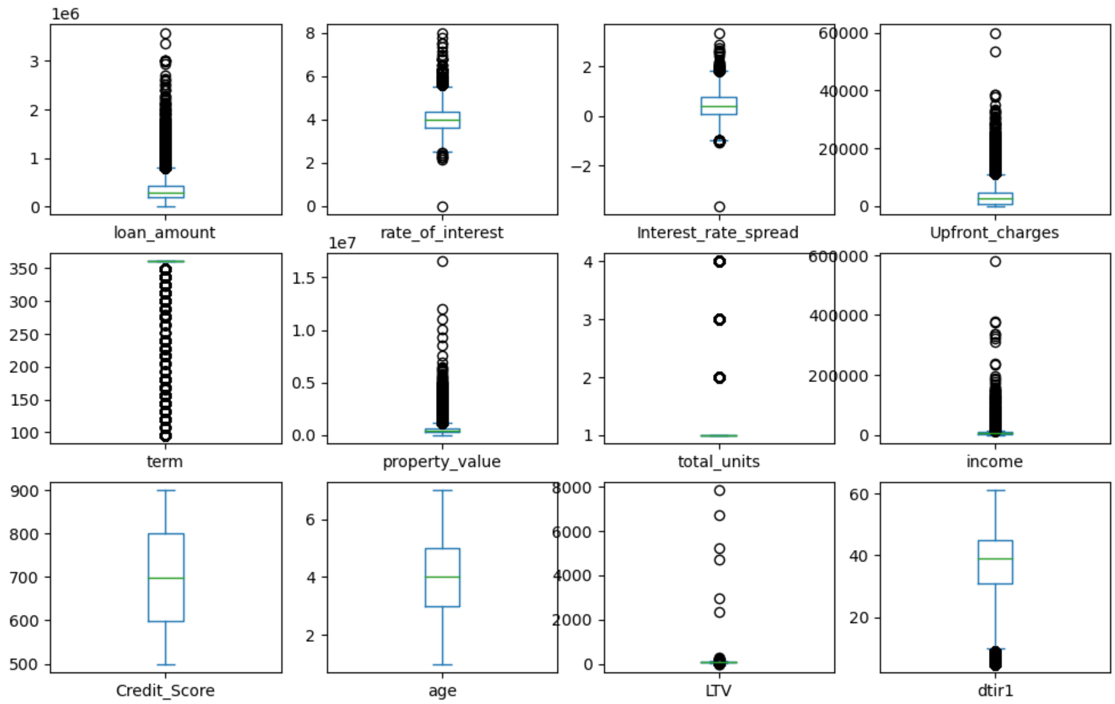
In the next block codes we will see how to approach this problem.

1.3.3 2.3. Numerical and Ordinal data

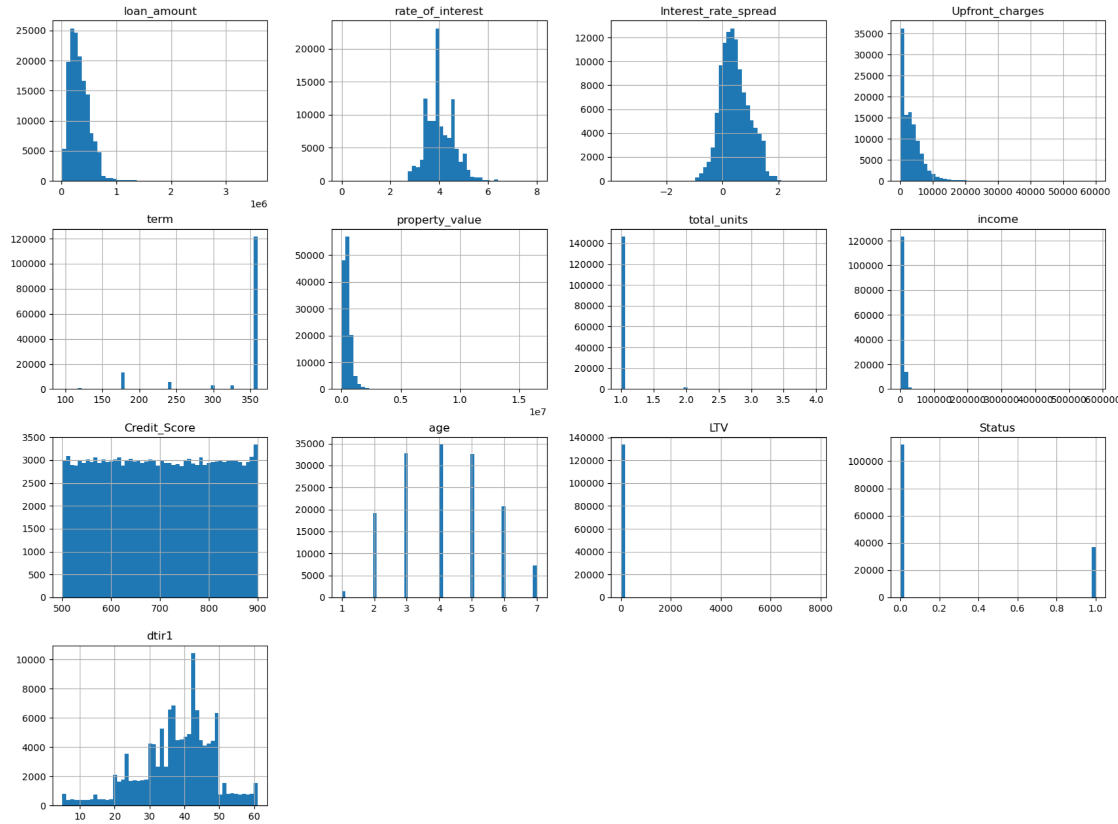
[9]:
```
#converting 'total_units' and 'age' categorical data into ordinal
ordinal_encoder = OrdinalEncoder()

#just for the total_units and age features, since it makes sense a concept of ordinal data we transform them this way
total_units_cat = loan_data[['total_units']] #take the column we want to encode
total_units_encoded = ordinal_encoder.fit_transform(total_units_cat) + 1 #we shift the numbers, since we represent the units
loan_data['total_units'] = total_units_encoded #replace the column in the dataset

age_mapping = {
    '<25':1,
    '25-34':2,
    '35-44':3,
    '45-54':4,
    '55-64':5,
    '65-74':6,
    '>74':7
}
loan_data['age'] = loan_data['age'].map(age_mapping) #replace the column in the dataset
```

[10]:
```
#list containing all the ordinal features plus the target
numerical_ordinal_columns = [col for col in loan_data.columns if col not in loan_data.select_dtypes(include=['category', 'object']).columns]
#list with only the ordinal features
numerical_ordinal_features = numerical_ordinal_columns.copy()
numerical_ordinal_features.remove('Status')
```

[11]:
```
#boxplots
loan_data[numerical_ordinal_features].plot(kind='box',
                              subplots=True, #distinct box plots for each feature
                              layout= (int(len(numerical_ordinal_features) ** 0.5) + 1, int(len(numerical_ordinal_features) ** 0.5) + 1), #make the plots readable
                              figsize=((int(len(numerical_ordinal_features) ** 0.5) + 1)*3, (int(len(numerical_ordinal_features) ** 0.5) + 1)*2.5),
                              sharex=False, #each subplot uses a different x scale
                              sharey=False #each subplot uses a different y scale
                             )
plt.show()
```



[12]:
```
#plot the histograms
loan_data.hist(bins=50, #number of bins to encapsulate the data
               figsize=(20,15)
              );
```



Notice: There are some capped values, specifically concerning: - term - dtir1 - Credit_Score

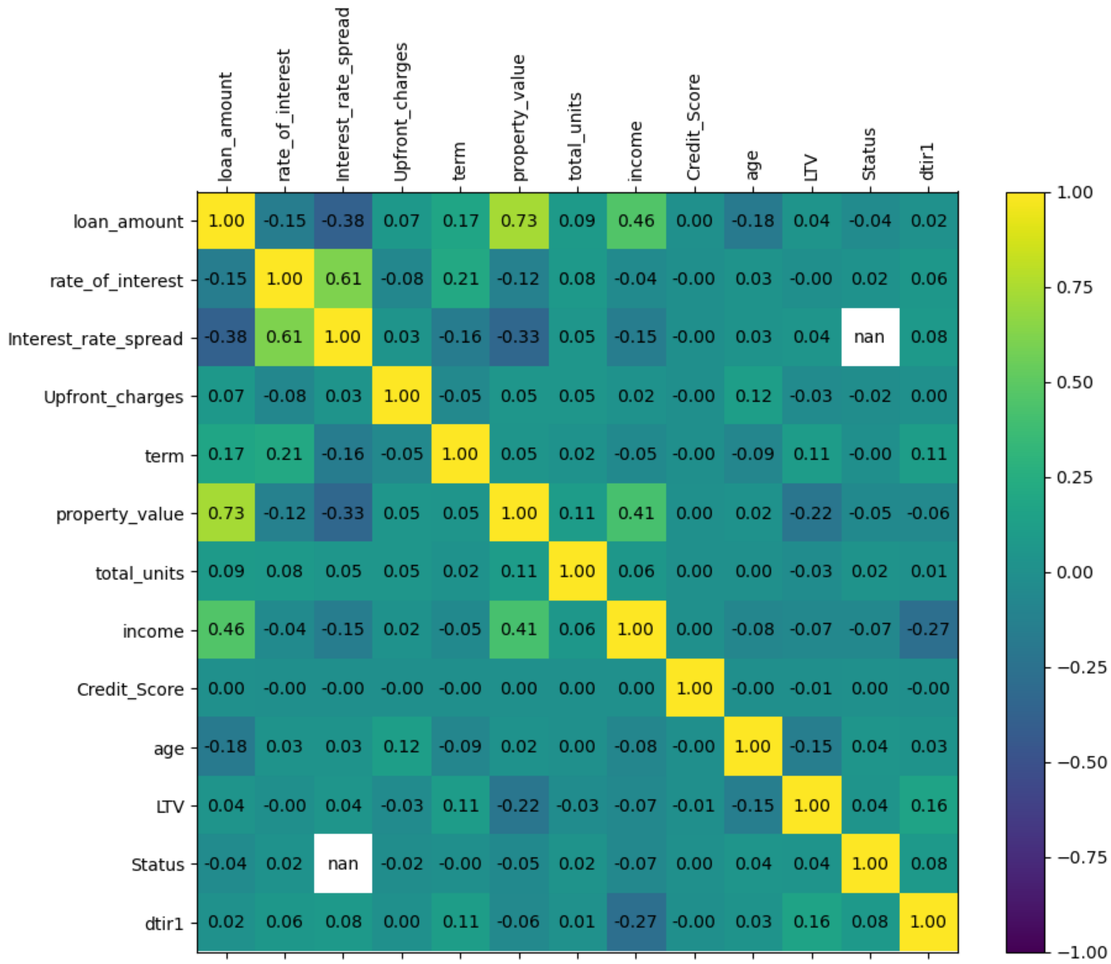
`term` feature for sure is capped at: 360, probably also: `dtir1` and `Credit_Sore`, but with a lower impact.
In any case, since they are not our target attribute, this is not a big issue and we can actually ignore it.

[13]:
```
loan_data.corr()["Status"].sort_values(ascending=False) #correlation related to
↪out target
```

[13]:
```
Status              1.000000
dtir1               0.078083
age                 0.044600
LTV                 0.038895
total_units         0.023800
rate_of_interest    0.022957
Credit_Score        0.004004
term               -0.000240
Upfront_charges    -0.019138
loan_amount        -0.036825
property_value     -0.048864
income             -0.065119
```

```
Interest_rate_spread         NaN
Name: Status, dtype: float64
```

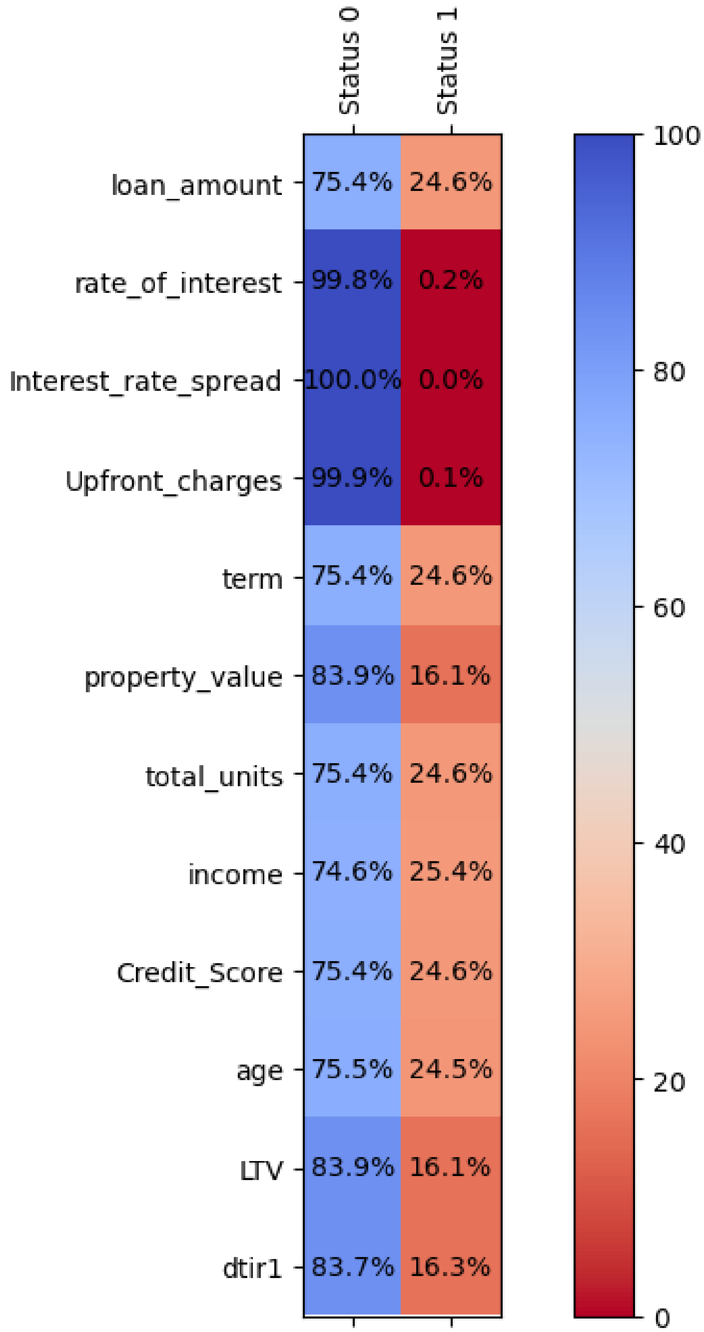
[14]:
```
#compute the correlations
correlations = loan_data.corr()

#plot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111)
cax = ax.matshow(correlations,
                 vmin=-1,
                 vmax=1)
fig.colorbar(cax)
ticks = np.arange(len(numerical_ordinal_columns))
ax.set_xticks(ticks)
ax.set_yticks(ticks)
ax.set_xticklabels(numerical_ordinal_columns,
                   rotation=90) #to see the column lables clearly
ax.set_yticklabels(numerical_ordinal_columns)

# Annotate cells with correlation values
for i in range(len(correlations)):
    for j in range(len(correlations)):
        value = correlations.iloc[i, j]
        ax.text(j,
                i,
                f'{value:.2f}',
                va='center',
                ha='center',
                color='black')

plt.show()
```



Notice: the correlation coefficient between **Status** and **interest_rate_spread** returns not a number, this is due to the fact that the latter feature is actually compromised, (we will see in few block codes how), furthermore the two standard deviations are too low.

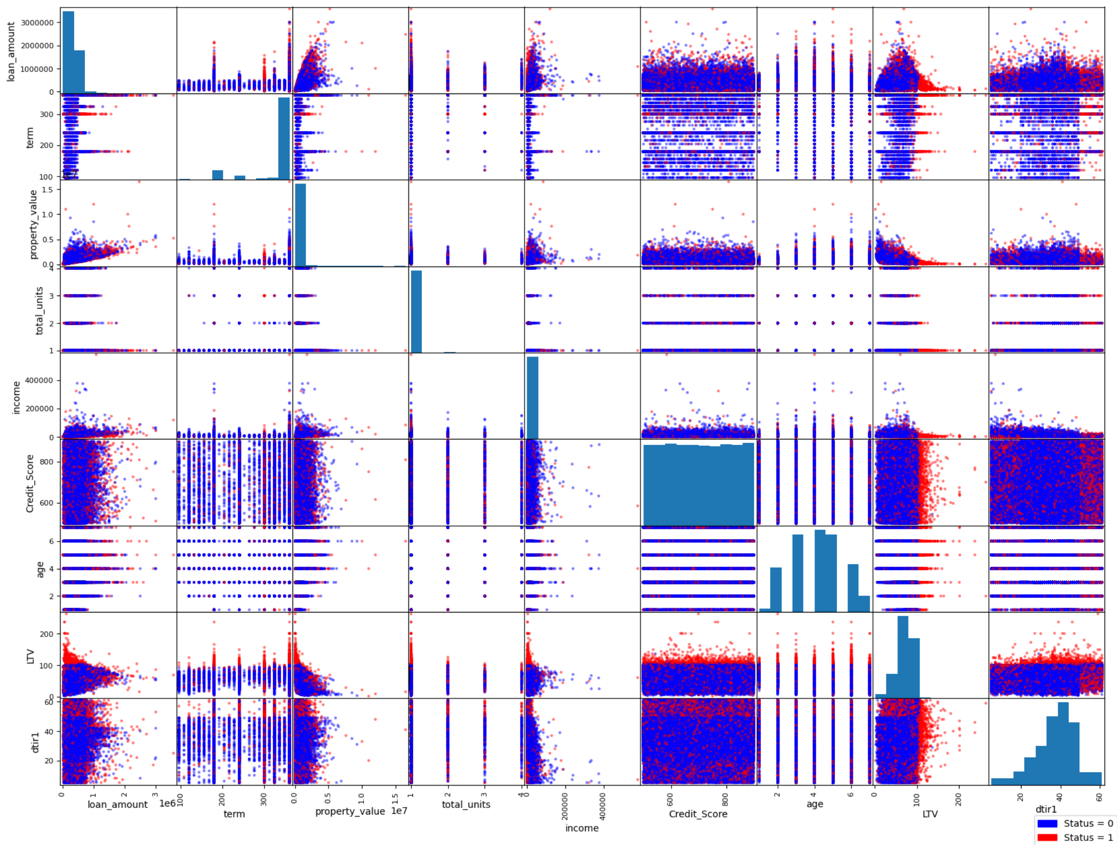
[15]:
```python
#showing the percentages of data grouped by target values
def compute_feature_target_percentages(df, features, target='Status'):
    rows = []
    for feat in features:
        #Discharge NaN values for feature and target
        feature_data = df[[feat, target]].dropna()

        #total valid values for this feature
        total = len(feature_data)

        #How many of these are target=0 and target=1
        count_0 = feature_data[feature_data[target] == 0].shape[0]
        count_1 = feature_data[feature_data[target] == 1].shape[0]
```

```python
        #we translate the above data into percentages
        pct_0 = (count_0 / total) * 100 if total > 0 else 0
        pct_1 = (count_1 / total) * 100 if total > 0 else 0

        rows.append({'feature': feat,
                     'target=0 %': pct_0,
                     'target=1 %': pct_1})
    return pd.DataFrame(rows)

result = compute_feature_target_percentages(loan_data,␣
 ↪numerical_ordinal_features)

#Extract data and labels
data_matrix = result[['target=0 %',
                      'target=1 %']].to_numpy()
feature_labels = result['feature'].tolist()
target_labels = ['Status 0',
                 'Status 1']

#Plot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111)
cax = ax.matshow(data_matrix,
                 vmin=0,
                 vmax=100,
                 cmap='coolwarm_r')
fig.colorbar(cax)

#Ticks
ax.set_xticks(np.arange(len(target_labels)))
ax.set_yticks(np.arange(len(feature_labels)))
ax.set_xticklabels(target_labels,
                   rotation=90)
ax.set_yticklabels(feature_labels)

#Values
for i in range(data_matrix.shape[0]):
    for j in range(data_matrix.shape[1]):
        value = data_matrix[i, j]
        ax.text(j,
                i, f'{value:.1f}%',
                va='center',
                ha='center',
                color='black')
plt.show()
```



Due to what we saw in section: 2.2. Missing Values, we can say that: **`rate_of_interest`**, **`Interest_rate_spread`** and **`Upfront_charges`**, when they are not missing they are always with **`Status=0`**, hence we must drop them entirely from the whole dataset, because they are no meaningful.

[16]:
```
#discharge the not desired columns
data = loan_data.drop(['rate_of_interest',
                       'Interest_rate_spread',
                       'Upfront_charges'],
                      axis=1)

#discharge the rows without elemnts
data = data.dropna()

#prepare the colors based on the Status (0 o 1)
color_map = {0: 'blue',
             1: 'red'
            }
colors = data['Status'].map(color_map)

# Plot dello scatter_matrix
pd.plotting.scatter_matrix(data.drop(columns=['Status']),
                            figsize=(20, 15),
                            diagonal='hist',
                            color=colors,
                            alpha=0.5
            )

#patch for the legend
legend_handles = [mpatches.Patch(color='blue',
                                  label='Status = 0'
                                 ),
                  mpatches.Patch(color='red',
                                  label='Status = 1'
                                 )
                ]

plt.legend(handles=legend_handles,
           loc='upper right',
           bbox_to_anchor=(1.15, -0.3)
          )
plt.show()
```



Notice: grouping dots based on the the target condition is very helpful for data visualization, there is a Python library for data visualization Seaborn * that allows to plot directly these types of graphs in a very straightforward way, but it is a more computational expensive tool, that in cases of big datasets can slow down the process a lot, this is the reason why we implemented the solution in `matplotlib` in a longer way.

*: Here is a project of mine with an implention using Seaborn

1.3.4 General categorical and binary categorical data

Notice: in the following block of code, we decided to transform manually the binary categorical data in binary ordinal data, assigning 0 or 1 with the following concept: 0 whether the associated variable has lower probability of deafult, 1 viceversa. (where possible).

[17]:
```
#to convert all the cathegorical binary columns to numerical binary
loan_data['loan_limit'] = loan_data['loan_limit'].map({'cf': 1,'ncf': 0})
loan_data['approv_in_adv'] = loan_data['approv_in_adv'].map({'nopre': 1,'pre': 0})
loan_data['Credit_Worthiness'] = loan_data['Credit_Worthiness'].map({'l2': 1,'l1': 0})
loan_data['open_credit'] = loan_data['open_credit'].map({'opc': 1,'nopc': 0})
```

```
loan_data['business_or_commercial'] = loan_data['business_or_commercial'].
 ↪map({'b/c': 1,'nob/c': 0})
loan_data['Neg_ammortization'] = loan_data['Neg_ammortization'].map({'not_neg':␣
 ↪1,'neg_amm': 0})
loan_data['interest_only'] = loan_data['interest_only'].map({'not_int':␣
 ↪1,'int_only': 0})
loan_data['lump_sum_payment'] = loan_data['lump_sum_payment'].map({'not_lpsm':␣
 ↪1,'lpsm': 0})
loan_data['construction_type'] = loan_data['construction_type'].map({'sb':␣
 ↪1,'mh': 0})
loan_data['Secured_by'] = loan_data['Secured_by'].map({'home': 1,'land': 0})
loan_data['co-applicant_credit_type'] = loan_data['co-applicant_credit_type'].
 ↪map({'CIB': 1,'EXP': 0})
loan_data['submission_of_application'] = loan_data['submission_of_application'].
 ↪map({'to_inst': 1,'not_inst': 0})
loan_data['Security_Type'] = loan_data['Security_Type'].map({'direct':␣
 ↪1,'Indriect': 0})
```

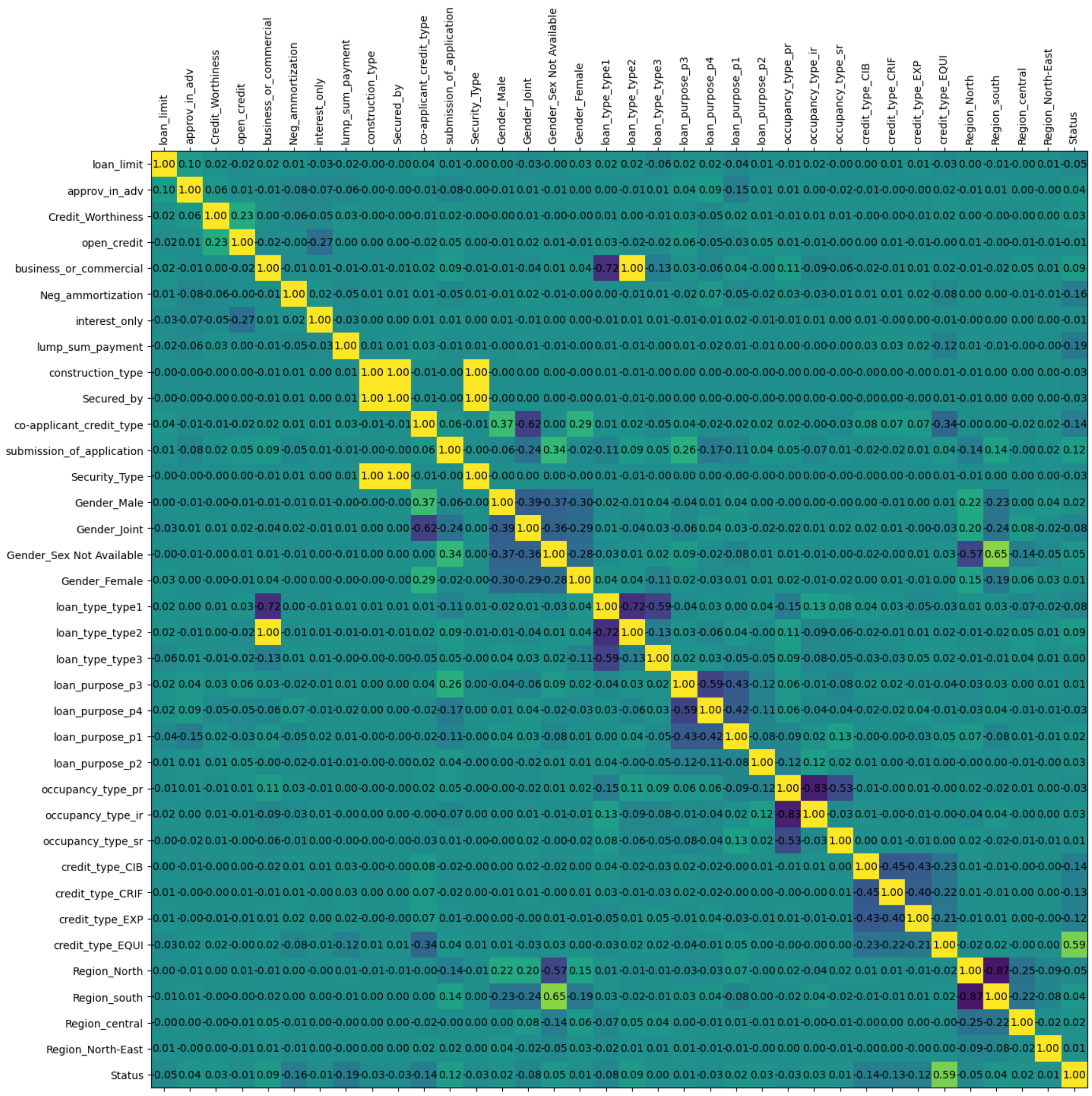
[18]:
```
#list of categorical non-binary columns
cat_columns = ['Gender',
               'loan_type',
               'loan_purpose',
               'occupancy_type',
               'credit_type',
               'Region']

# Loop through each column
for col in cat_columns:
    encoder = OneHotEncoder(sparse=False,
                            handle_unknown='ignore')  #new encoder for each
    encoded_array = encoder.fit_transform(loan_data[[col]])
    encoded_df = pd.DataFrame(encoded_array,
                              columns=encoder.get_feature_names_out([col]),
                              index=loan_data.index)

    # Drop original column and concatenate the encoded DataFrame
    loan_data.drop(columns=[col],
                   inplace=True)
    loan_data = pd.concat([loan_data,
                           encoded_df
                          ],
                          axis=1)
```

[19]:
```
#list of all categorical columns
categorical_columns = ['loan_limit',
                       'approv_in_adv',
```

```
                    'Credit_Worthiness',
                    'open_credit',
                    'business_or_commercial',
                    'Neg_ammortization',
                    'interest_only',
                    'lump_sum_payment',
                    'construction_type',
                    'Secured_by',
                    'co-applicant_credit_type',
                    'submission_of_application',
                    'Security_Type',
                    'Gender_Male', 'Gender_Joint', 'Gender_Sex Not Available',␣
↪'Gender_Female',
                    'loan_type_type1', 'loan_type_type2', 'loan_type_type3',
                    'loan_purpose_p3', 'loan_purpose_p4', 'loan_purpose_p1',␣
↪'loan_purpose_p2',
                    'occupancy_type_pr', 'occupancy_type_ir', 'occupancy_type_sr',
                    'credit_type_CIB', 'credit_type_CRIF', 'credit_type_EXP',␣
↪'credit_type_EQUI',
                    'Region_North', 'Region_south', 'Region_central',␣
↪'Region_North-East',
                    'Status'
]


categorical_features = categorical_columns.copy()
#list of all binary features
categorical_features.remove('Status')
```

[20]:
```
binary_correlation = loan_data[categorical_columns].corr()

#plot
fig = plt.figure(figsize=(20, 16))
ax = fig.add_subplot(111)
cax = ax.matshow(binary_correlation,
                 vmin=-1,
                 vmax=1)

ticks = np.arange(len(categorical_columns))
ax.set_xticks(ticks)
ax.set_yticks(ticks)
ax.set_xticklabels(categorical_columns,
                   rotation=90) #to see the column lables clearly
ax.set_yticklabels(categorical_columns)

#annotate cells with correlation values
for i in range(len(binary_correlation)):
```

```python
    for j in range(len(binary_correlation)):
        value = binary_correlation.iloc[i, j]
        ax.text(j,
                i,
                f'{value:.2f}',
                va='center',
                ha='center',
                color='black')

plt.show()
```



Pandas `dataframe.corr()` method computes the Pearson's correlation coefficient:

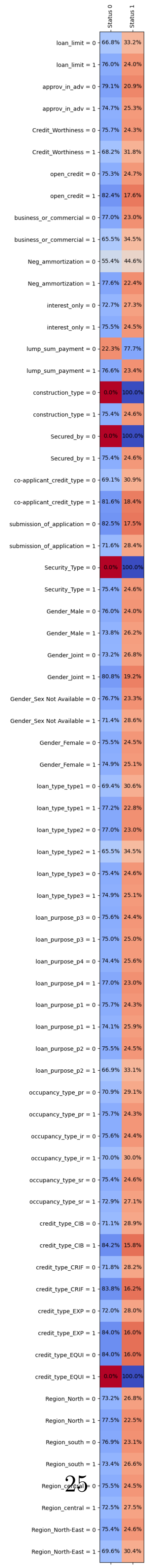
$$\rho_{x,y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y}$$

where: - cov is the covariance computed as: $cov(X,Y) = \mathbb{E}[(X-\mu_X)(Y-\mu_Y)]$ - σ_X is the standard deviation of X - σ_Y is the sandard deviation of Y

In the case of binary variables that assume only values in $\{0,1\}$, using Pearson's coefficient is a possible way to gain an overview of the correlation between variables, but keep in mind that in literature there many other methods to achieve so.

[21]:
```python
def compute_binary_feature_target_distribution(df, binary_features,␣
    ↪target='Status'):
    rows = []
    for feat in binary_features:
        for val in [0, 1]:
            #filter for feature value
            subset = df[(df[feat] == val) & (df[target].isin([0, 1]))]

            total = len(subset)
            count_0 = subset[subset[target] == 0].shape[0]
            count_1 = subset[subset[target] == 1].shape[0]

            pct_0 = (count_0 / total) * 100 if total > 0 else 0
            pct_1 = (count_1 / total) * 100 if total > 0 else 0

            rows.append({
                'feature': feat,
                'value': val,
                'Status 0 %': pct_0,
                'Status 1 %': pct_1
            })
    return pd.DataFrame(rows)


result_bin = compute_binary_feature_target_distribution(loan_data,␣
    ↪categorical_features)

#extract data for heatmap
data_matrix = result_bin[['Status 0 %', 'Status 1 %']].to_numpy()
row_labels = [f"{f} = {v}" for f, v in zip(result_bin['feature'],␣
    ↪result_bin['value'])]
col_labels = ['Status 0', 'Status 1']

#plot
fig = plt.figure(figsize=(15, len(row_labels) * 0.7))
ax = fig.add_subplot(111)
cax = ax.matshow(data_matrix,
```

```
                 vmin=0,
                 vmax=100,
                 cmap='coolwarm_r')
# ticks
ax.set_xticks(np.arange(len(col_labels)))
ax.set_yticks(np.arange(len(row_labels)))
ax.set_xticklabels(col_labels, rotation=90)
ax.set_yticklabels(row_labels)

#annotate values
for i in range(data_matrix.shape[0]):
    for j in range(data_matrix.shape[1]):
        value = data_matrix[i, j]
        ax.text(j,
                i,
                f'{value:.1f}%',
                va='center',
                ha='center',
                color='black')

plt.show()
```


Status 0
Status 1
loan_limit = 0 66.8% 33.2%
loan_limit = 1 76.0% 24.0%
approv_in_adv = 0 79.1% 20.9%
approv_in_adv = 1 74.7% 25.3%
Credit_Worthiness = 0 75.7% 24.3%
Credit_Worthiness = 1 68.2% 31.8%
open_credit = 0 75.3% 24.7%
open_credit = 1 82.4% 17.6%
business_or_commercial = 0 77.0% 23.0%
business_or_commercial = 1 65.5% 34.5%
Neg_ammortization = 0 55.4% 44.6%
Neg_ammortization = 1 77.6% 22.4%
interest_only = 0 72.7% 27.3%
interest_only = 1 75.5% 24.5%
lump_sum_payment = 0 22.3% 77.7%
lump_sum_payment = 1 76.6% 23.4%
construction_type = 0 0.0% 100.0%
construction_type = 1 75.4% 24.6%
Secured_by = 0 0.0% 100.0%
Secured_by = 1 75.4% 24.6%
co-applicant_credit_type = 0 69.1% 30.9%
co-applicant_credit_type = 1 81.6% 18.4%
submission_of_application = 0 82.5% 17.5%
submission_of_application = 1 71.6% 28.4%
Security_Type = 0 0.0% 100.0%
Security_Type = 1 75.4% 24.6%
Gender_Male = 0 76.0% 24.0%
Gender_Male = 1 73.8% 26.2%
Gender_Joint = 0 73.2% 26.8%
Gender_Joint = 1 80.8% 19.2%
Gender_Sex Not Available = 0 76.7% 23.3%
Gender_Sex Not Available = 1 71.4% 28.6%
Gender_Female = 0 75.5% 24.5%
Gender_Female = 1 74.9% 25.1%
loan_type_type1 = 0 69.4% 30.6%
loan_type_type1 = 1 77.2% 22.8%
loan_type_type2 = 0 77.0% 23.0%
loan_type_type2 = 1 65.5% 34.5%
loan_type_type3 = 0 75.4% 24.6%
loan_type_type3 = 1 74.9% 25.1%
loan_purpose_p3 = 0 75.6% 24.4%
loan_purpose_p3 = 1 75.0% 25.0%
loan_purpose_p4 = 0 74.4% 25.6%
loan_purpose_p4 = 1 77.0% 23.0%
loan_purpose_p1 = 0 75.7% 24.3%
loan_purpose_p1 = 1 74.1% 25.9%
loan_purpose_p2 = 0 75.5% 24.5%
loan_purpose_p2 = 1 66.9% 33.1%
occupancy_type_pr = 0 70.9% 29.1%
occupancy_type_pr = 1 75.7% 24.3%
occupancy_type_ir = 0 75.6% 24.4%
occupancy_type_ir = 1 70.0% 30.0%
occupancy_type_sr = 0 75.4% 24.6%
occupancy_type_sr = 1 72.9% 27.1%
credit_type_CIB = 0 71.1% 28.9%
credit_type_CIB = 1 84.2% 15.8%
credit_type_CRIF = 0 71.8% 28.2%
credit_type_CRIF = 1 83.8% 16.2%
credit_type_EXP = 0 72.0% 28.0%
credit_type_EXP = 1 84.0% 16.0%
credit_type_EQUI = 0 84.0% 16.0%
credit_type_EQUI = 1 0.0% 100.0%
Region_North = 0 73.2% 26.8%
Region_North = 1 77.5% 22.5%
Region_south = 0 76.9% 23.1%
Region_south = 1 73.4% 26.6%
Region_central = 0 75.5% 24.5%
Region_central = 1 72.5% 27.5%
Region_North-East = 0 75.4% 24.6%
Region_North-East = 1 69.6% 30.4%

possiamo porre a zero le feature quando assumono tali valori

1.4 3. Data Preprocessing

[22]:
```
#import of packages and libraries useful for fitting, transforming hence␣
↪pipelines
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import FunctionTransformer
from sklearn.impute import SimpleImputer
import warnings #for ignoring the warnings while the models run
```

[23]:
```
#while running some code blocks we encounter some useless warnings that in␣
↪production we can ignore for a tidier code
#ignore the FutureWarning
warnings.filterwarnings("ignore", category=FutureWarning)
#ignore the UserWarning
#warnings.filterwarnings("ignore", category=UserWarning)
```

[24]:
```
#features are now only numerical and ordinal, the number of them increased, as␣
↪the memory usage increased by: 17+ MB
loan_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 148670 entries, 0 to 148669
Data columns (total 49 columns):
 #   Column                     Non-Null Count   Dtype
---  ------                     --------------   -----
 0   loan_limit                 145326 non-null  float64
 1   approv_in_adv              147762 non-null  float64
 2   Credit_Worthiness          148670 non-null  int64
 3   open_credit                148670 non-null  int64
 4   business_or_commercial     148670 non-null  int64
 5   loan_amount                148670 non-null  int64
 6   rate_of_interest           112231 non-null  float64
 7   Interest_rate_spread       112031 non-null  float64
 8   Upfront_charges            109028 non-null  float64
 9   term                       148629 non-null  float64
 10  Neg_ammortization          148549 non-null  float64
 11  interest_only              148670 non-null  int64
 12  lump_sum_payment           148670 non-null  int64
 13  property_value             133572 non-null  float64
 14  construction_type          148670 non-null  int64
 15  Secured_by                 148670 non-null  int64
 16  total_units                148670 non-null  float64
```

```
 17  income                     139520 non-null  float64
 18  Credit_Score               148670 non-null  int64
 19  co-applicant_credit_type   148670 non-null  int64
 20  age                        148470 non-null  float64
 21  submission_of_application  148470 non-null  float64
 22  LTV                        133572 non-null  float64
 23  Security_Type              148670 non-null  int64
 24  Status                     148670 non-null  int64
 25  dtir1                      124549 non-null  float64
 26  Gender_Female              148670 non-null  float64
 27  Gender_Joint               148670 non-null  float64
 28  Gender_Male                148670 non-null  float64
 29  Gender_Sex Not Available   148670 non-null  float64
 30  loan_type_type1            148670 non-null  float64
 31  loan_type_type2            148670 non-null  float64
 32  loan_type_type3            148670 non-null  float64
 33  loan_purpose_p1            148670 non-null  float64
 34  loan_purpose_p2            148670 non-null  float64
 35  loan_purpose_p3            148670 non-null  float64
 36  loan_purpose_p4            148670 non-null  float64
 37  loan_purpose_nan           148670 non-null  float64
 38  occupancy_type_ir          148670 non-null  float64
 39  occupancy_type_pr          148670 non-null  float64
 40  occupancy_type_sr          148670 non-null  float64
 41  credit_type_CIB            148670 non-null  float64
 42  credit_type_CRIF           148670 non-null  float64
 43  credit_type_EQUI           148670 non-null  float64
 44  credit_type_EXP            148670 non-null  float64
 45  Region_North               148670 non-null  float64
 46  Region_North-East          148670 non-null  float64
 47  Region_central             148670 non-null  float64
 48  Region_south               148670 non-null  float64
dtypes: float64(37), int64(12)
memory usage: 55.6 MB
```

Notice: the memory usage given directly by Pandas has increased by 17+ MB, due to the transformation into ordinal data.
In order to have a clearer idea of the actual memory used, as already mentioned in section 2.1. Dataset Description, we should use `sys.gestsizeof()` as below.

[25]:
```
#the result is in bytes
sys.getsizeof(loan_data)
```

[25]: 58278784

The number shows that actually the memory used is way less with respect to the beginning, this is due to the fact that integer objects take less memory compared to string objects.

[26]:
```
try:
    loan_data = pd.read_csv(path + '\Loan_Default.csv')

except:
    loan_data = pd.read_csv('Loan_Default.csv') #if something goes wrong. use␣
 ↪the copy of the dataset in the folder
```

In the following code block we apply column removal, based on the characteristics that emerged in sections 2.3. and 2.4..
It is important to point out that the column removal has been applied on the whole dataset, because the criticalities on those features were so strong that allowed us to procede without compromising the effectiveness of the test set.
In order to better explain how the decisions have been taken we divide the data into numerical and categorical:

- *Numerical*:
  - **`ID`**, **`year`**: have been removed since they did not bring any relevant information for predictive purposes.
  - **`rate_of_interest`**,**`Interest_rate_spread`**,**`Upfront_charges`**: have been removed, because they were strongly compromised, they have the highest missing values and furthermore whenever they are not missing, the target is nearly always: **`Status=0`**
- *Categorical*:
  - **`business_or_commercial`**: has been removed because from the correlation matrix emerges that is strongly correlated to **`loan_type_2`**
  - **`Secured_by`**, **`Security_Type`**: have been removed because from the correlation matrix emerged that they were strongly correlated to each other and to **`construction_type`**, furthermore whenever **`Secured_by=land`** or **`Security_Type=indirect`** the target always assumes: **'Status=1'**
  - The feature **`credit_type`** when assumes the value **`EQUI`** the target is always **`Status=1`**, hence it does not bring any relevant information. This column has been transformed with one-hot encoding, because it can assume four different values, hence we decided to set it to **`NaN`** whenever it assumes **`EQUI`**, this is due to the way we handle missing values (we will see it in few code blocks).

It is important to point out that the threshold of correlation by which we decided to remove columns has been set to 0.90 since decision trees and even better ensemble learning algorithms are quite robust when they face correlation.

[27]:
```
#removing the non-relevant features
loan_data = loan_data.drop([#numerical
                           'ID',
                           'year',
                           'rate_of_interest',
                           'Interest_rate_spread',
                           'Upfront_charges',
                            #categorical
                           'Security_Type',
                           'Secured_by',
                           'business_or_commercial',
```

```
                             ],
                             axis=1
                            )

#setting to NaN the non-relevant columns
loan_data.loc[loan_data['credit_type'] == 'EQUI', 'credit_type'] = np.nan
```

[28]:
```
#map for the ordinal mapping of age groups
age_mapping = {'<25': 1,
                '25-34': 2,
                '35-44': 3,
                '45-54': 4,
                '55-64': 5,
                '65-74': 6,
                '>74': 7
                }
#map for the ordinal mapping of the total_units
total_units_mapping = {'1U': 1,
                    '2U': 2,
                    '3U': 3,
                    '4U': 4
                }

#list of the names of the ordinal features
ordinal_features = ['age',
                'total_units',
               ]
#list of the names of the categorical features where we will apply␣
 ↪OneHotEncoding
onehot_features = ['loan_limit',
               'approv_in_adv',
               'Credit_Worthiness',
               'open_credit',
               'Neg_ammortization',
               'interest_only',
               'lump_sum_payment',
               'construction_type',
               'co-applicant_credit_type',
               'submission_of_application',
               'Gender',
               'loan_type',
               'loan_purpose',
               'occupancy_type',
               'credit_type',
               'Region',
              ]
#list of the numerical features
```

```
numerical_features = ['loan_amount',
                      'term',
                      'property_value',
                      'income',
                      'Credit_Score',
                      'LTV',
                      'dtir1'
                     ]
```

[29]:

```
# Categorical one-hot
cat_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore')) #we ignore the unknown␣
 ↪cathegory setting to zeros, best-practice
])

# Numerical
num_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='median'))
])

# Ordinal
#age ordinal pipeline
age_pipeline = Pipeline([
    ('mapper', FunctionTransformer(
        lambda x: x.iloc[:, 0].map(age_mapping).to_frame(), #anonymous␣
 ↪function, thanks to which we map the 'age' column to
        validate=False)),                                   #the mapping we␣
 ↪have previously defined, in a DataFrame format
    ('imputer', SimpleImputer(strategy='most_frequent'))
])
#total_units ordinal pipeline
total_units_pipeline = Pipeline([
    ('mapper', FunctionTransformer(
        lambda x: x.iloc[:, 0].map(total_units_mapping).to_frame(),
        validate=False)), #since we want to work with DataFrame we must set␣
 ↪validate to False
    ('imputer', SimpleImputer(strategy='most_frequent'))
])

#Full pipeline, merging all type of data
preprocessor = ColumnTransformer([
    ('cat_onehot', cat_pipeline, onehot_features),
    ('num', num_pipeline, numerical_features),
    ('age_ord', age_pipeline, ['age']), #since ColumnTransformer apply␣
 ↪transformations on list of columns we must give list
    ('units_ord', total_units_pipeline, ['total_units']),
```

```
])
```

[30]:
```
#memory usage in bytes of the processed dataset with the pipeline
sys.getsizeof(preprocessor.fit_transform(loan_data))
```

[30]: 59468120

Notice: now that we have applied the **preprocessor** (that is a pipeline for column transformation) to the original dataset, we see that the memory used is again way less with respect to the original one, buth slightly more compared to the manually transformed dataset obtained at the end of section 2. (that contained 49 columns), even though now we have removed nine features.
This is simply due to the fact that with the **preprocessor** the one-hot encoding is fully applied to all categorical data (except **age**, **total_units**), resulting with 50 columns.

1.5 4. Training the Models

[31]:
```
#import of all libraries and packages
from sklearn.model_selection import GridSearchCV, StratifiedKFold,␣
 ↪train_test_split
from xgboost import XGBClassifier #XGBoostClassifier
from sklearn.ensemble import AdaBoostClassifier #AdaBoost
from sklearn.tree import DecisionTreeClassifier #for AdaBoost
from sklearn.ensemble import GradientBoostingClassifier #GradientBoost
#instead of Joblib, we use Cloudpickle, since it is able to save also the␣
 ↪lambda functions
import cloudpickle #for saving the models
#for getting all the metrics and displaying them
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,␣
 ↪average_precision_score, PrecisionRecallDisplay, accuracy_score,␣
 ↪precision_score, recall_score, f1_score, precision_recall_curve
import time #for counting the amount of time it takes for the algorithm to run
```

[32]:
```
#for converting seconds into readable format, for the runtime
def convert_time(seconds):
    seconds = seconds % (24 * 3600)
    hour = seconds // 3600
    seconds %= 3600
    minutes = seconds // 60
    seconds %= 60

    return "%d:%02d:%02d" % (hour, minutes, seconds)
```

[33]:
```
#Separate features and target
X = loan_data.drop(columns=['Status'])
y = loan_data['Status']
```

```
#split the dataset into train and test sets, with a proportion of 90-10%, since
↪we have 148670 samples.
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.10,
                                                    random_state=42)



#stratified K-Fold for balanced class distribution over the target
strat_kfold = StratifiedKFold(n_splits=5, #number of folds
                              shuffle=True, #before creating the fold, it
↪shuffles the rows (samples)
                              random_state=42, #footnote following block
                             )
```

Notice: our dataset is a bit unbalanced in terms of target values, since the samples with `Status=0` are way more than the ones with `Status=1`, this can lead to slighlty biased models. In order to overcome this issue we tweak some models' paramaters accordingly.

This practice is especially useful for improving the `recall` of our models.

$$Recall = \frac{\text{True Positives}}{\text{True Positives} \ + \ \text{False Negatives}}$$

As opposed to:

$$Precision = \frac{\text{True Positives}}{\text{True Positives} \ + \ \text{False Positives}}$$

Since credit institutes tend to be risk adverse in front of potential loan clients, they ***prefer recall over precision***, and this is the way we approach our three following models.

`random_state = 42` is an inside joke of machine learning base on the famous book: *The Hitchhiker's Guide to the Galaxy, Douglas Adams*, see

1.5.1 4.1. eXtreme Gradient Boosting

[34]:
```
#counting the number of positive and negative samples for balancing the model
num_negatives = loan_data['Status'].value_counts()[0]
num_positives = loan_data['Status'].value_counts()[1]
```

[35]:
```
#setting the XGBoost classifier
xgb_model = XGBClassifier(eval_metric='logloss',
                          random_state=42, #to control the internal random
↪components of the classifier
                          scale_pos_weight = num_negatives / num_positives #it
↪gives more weight to the positive samples
                         )
↪#since they are less, in order to overcome
```

```
↪#unbalanced dataset
#adding to the pipeline the algorithm
full_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', xgb_model)
])

#parameter grid for GridSearch
param_grid = {
    'classifier__n_estimators': [50, 100],
    'classifier__max_depth': [3, 6],
    'classifier__learning_rate': [0.01, 0.1],
    'classifier__subsample': [0.8, 1],  #fraction of samples used for fitting␣
↪each tree (classic XGBoosting Vs Stochastic)
}

# Grid search setup
grid_search_XGBoost = GridSearchCV(full_pipeline,
                           param_grid,
                           cv=strat_kfold,
                           n_jobs=-1, #it uses all the CPU cores availbale in␣
↪parallel
                           verbose=2, #it shows all details for each␣
↪combination
                           #scoring = 'recall' #to find hyperparameters to␣
↪maximize recall but it lowers too much accuracy
                          )

start_time = time.time() #for counting the time
# Fit GridSearchCV on training data
grid_search_XGBoost.fit(X_train,
                y_train
               )
end_time = time.time() #end time of execution
elapsed_time = convert_time(end_time - start_time)
print(f"XGBoosting training + grid search took {elapsed_time}")

# Best params and score
print("Best parameters found:", grid_search_XGBoost.best_params_)
print("Best cross-validation accuracy:", grid_search_XGBoost.best_score_)

# Optional: Evaluate on the test set
test_score = grid_search_XGBoost.score(X_test, y_test)
print(f"Test set accuracy: {test_score:.4f}") #we print the accuracy with the␣
↪four decimals
```

```
Fitting 5 folds for each of 16 candidates, totalling 80 fits
XGBoosting training + grid search took 0:03:32
Best parameters found: {'classifier__learning_rate': 0.1,
'classifier__max_depth': 6, 'classifier__n_estimators': 100,
'classifier__subsample': 1}
Best cross-validation accuracy: 0.8728429363860769
Test set accuracy: 0.8743
```

[36]:
```
#try to see whether the above cell has been run
try:
    grid_search_XGBoost
    recovered = False #boolean variable to understand wheter the model has been
 ↪recovered from serialization or not

#otherwise open the serialized model
except:
    with open("XGBoost_loan.pkl", mode="rb") as f: #rb stands for reading binary
        grid_search_XGBoost = cloudpickle.load(f)
    recovered = True
```

[37]:
```
pd.DataFrame(grid_search_XGBoost.cv_results_)
```

[37]:
```
    mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0        7.082961      0.098872         0.571366        0.044118
1        6.194776      0.187028         0.585241        0.050450
2       10.809194      0.276659         0.626966        0.045916
3        8.409936      0.284035         0.620777        0.038486
4        7.901922      0.134044         0.642341        0.045532
5        7.112858      0.199286         0.623072        0.045224
6       11.852906      0.135451         0.777327        0.044953
7       10.045854      0.190022         0.772979        0.062497
8        7.180450      0.297617         0.575607        0.048334
9        5.938304      0.099533         0.610159        0.055767
10       9.330348      0.360060         0.632164        0.042070
11       7.719397      0.068763         0.637710        0.051803
12       7.693387      0.095195         0.652083        0.068410
13       7.047542      0.167268         0.653615        0.046432
14      10.941994      0.083474         0.735098        0.047523
15       9.639021      0.407220         0.681991        0.096792

   param_classifier__learning_rate param_classifier__max_depth  \
0                             0.01                           3
1                             0.01                           3
2                             0.01                           3
3                             0.01                           3
4                             0.01                           6
5                             0.01                           6
```

```
6                             0.01                           6
7                             0.01                           6
8                              0.1                           3
9                              0.1                           3
10                             0.1                           3
11                             0.1                           3
12                             0.1                           6
13                             0.1                           6
14                             0.1                           6
15                             0.1                           6

   param_classifier__n_estimators param_classifier__subsample  \
0                              50                         0.8
1                              50                           1
2                             100                         0.8
3                             100                           1
4                              50                         0.8
5                              50                           1
6                             100                         0.8
7                             100                           1
8                              50                         0.8
9                              50                           1
10                            100                         0.8
11                            100                           1
12                             50                         0.8
13                             50                           1
14                            100                         0.8
15                            100                           1

                                               params  split0_test_score  \
0   {'classifier__learning_rate': 0.01, 'classifie…           0.857105
1   {'classifier__learning_rate': 0.01, 'classifie…           0.857330
2   {'classifier__learning_rate': 0.01, 'classifie…           0.860917
3   {'classifier__learning_rate': 0.01, 'classifie…           0.860917
4   {'classifier__learning_rate': 0.01, 'classifie…           0.867531
5   {'classifier__learning_rate': 0.01, 'classifie…           0.868839
6   {'classifier__learning_rate': 0.01, 'classifie…           0.872613
7   {'classifier__learning_rate': 0.01, 'classifie…           0.872725
8   {'classifier__learning_rate': 0.1, 'classifier…           0.862300
9   {'classifier__learning_rate': 0.1, 'classifier…           0.862449
10  {'classifier__learning_rate': 0.1, 'classifier…           0.864691
11  {'classifier__learning_rate': 0.1, 'classifier…           0.862823
12  {'classifier__learning_rate': 0.1, 'classifier…           0.870595
13  {'classifier__learning_rate': 0.1, 'classifier…           0.871679
14  {'classifier__learning_rate': 0.1, 'classifier…           0.871231
15  {'classifier__learning_rate': 0.1, 'classifier…           0.873585
```

```
    split1_test_score  split2_test_score  split3_test_score  \
0            0.862374           0.859871           0.861061   
1            0.857218           0.859796           0.861584   
2            0.858675           0.857965           0.864499   
3            0.858189           0.857965           0.863789   
4            0.871941           0.867270           0.874327   
5            0.872987           0.867008           0.872907   
6            0.873921           0.870819           0.874514   
7            0.871791           0.870035           0.871375   
8            0.863981           0.863495           0.867115   
9            0.864093           0.866223           0.867003   
10           0.862636           0.863757           0.866218   
11           0.865140           0.862337           0.867414   
12           0.869960           0.871268           0.872496   
13           0.870334           0.871006           0.873019   
14           0.873996           0.870334           0.874701   
15           0.872650           0.868876           0.875747   

    split4_test_score  mean_test_score  std_test_score  rank_test_score  
0            0.861435         0.860369        0.001819               15  
1            0.860987         0.859383        0.001816               16  
2            0.862220         0.860855        0.002377               13  
3            0.862033         0.860579        0.002240               14  
4            0.867750         0.869764        0.002857                8  
5            0.871824         0.870713        0.002386                7  
6            0.871114         0.872596        0.001469                3  
7            0.871487         0.871483        0.000866                6  
8            0.865321         0.864443        0.001650               12  
9            0.862593         0.864472        0.001857               11  
10           0.866779         0.864816        0.001530               10  
11           0.867377         0.865018        0.002160                9  
12           0.873767         0.871617        0.001364                5  
13           0.874552         0.872118        0.001506                4  
14           0.873244         0.872701        0.001658                2  
15           0.873356         0.872843        0.002237                1  
```

[38]:
```
#we encapsulate just the best model
XGBoost_whole_model = grid_search_XGBoost.best_estimator_ #here we save the␣
 ↪whole best model full pipeline (prepocessor + classifier)
XGBoost_classifier = XGBoost_whole_model.named_steps['classifier'] #here we␣
 ↪save just the best classifier
```

[39]:
```
#recover just the preprocessor from the whole model
preprocessor = XGBoost_whole_model.named_steps['preprocessor']
feature_names = []

for name, transformer, cols in preprocessor.transformers_:
```

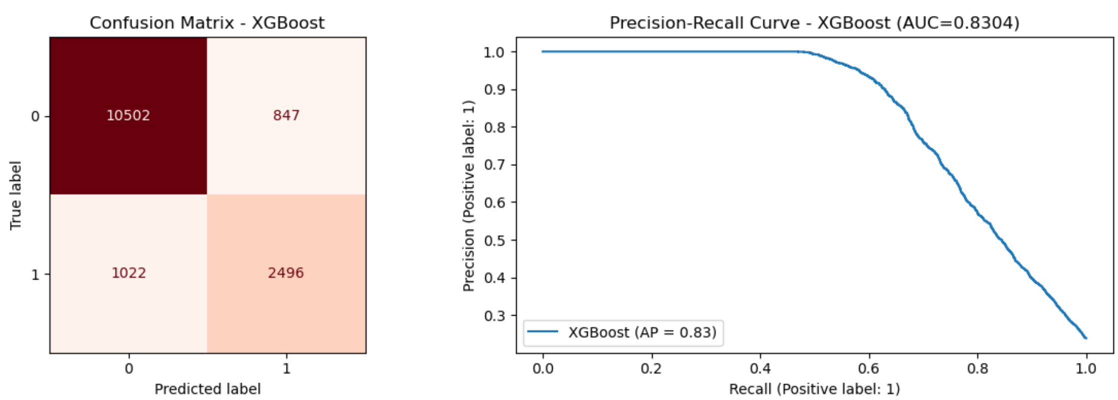
```python
    #if the transformer is a drop operator
    if transformer == 'drop':
        continue
    #if the transformer is actually a pipeline object, so that has inside other␣
 ↪transformers
    if isinstance(transformer, Pipeline):
        #get the last step of the pipeline, because the last step is the one␣
 ↪that finally transforms the data and generates new names
        last_step = transformer.steps[-1][1]
        if hasattr(last_step, 'get_feature_names_out'):
            names = last_step.get_feature_names_out(cols)
        else:
            names = cols
    #if the transformer is directly a transformer
    else:
        if hasattr(transformer, 'get_feature_names_out'):
            names = transformer.get_feature_names_out(cols)
        else:
            names = cols

    feature_names.extend(names)

feat_imp = pd.DataFrame({
                        'feature': feature_names,
                        'importance': XGBoost_classifier.feature_importances_
                        }).sort_values(
                                        by="importance",
                                        ascending=False
                                        )

print(feat_imp)
```

```
                               feature  importance
43                      property_value    0.110603
12                   lump_sum_payment_lpsm    0.105424
46                                 LTV    0.101989
8            Neg_ammortization_neg_amm    0.073284
34                     credit_type_CIB    0.073172
18  submission_of_application_not_inst    0.055749
4                  Credit_Worthiness_l1    0.043453
24                     loan_type_type1    0.043202
47                               dtir1    0.042422
0                        loan_limit_cf    0.028888
25                     loan_type_type2    0.026857
26                     loan_type_type3    0.026608
27                     loan_purpose_p1    0.020804
44                              income    0.019346
```

```
32                    occupancy_type_pr    0.019235
10               interest_only_int_only    0.017363
2                   approv_in_adv_nopre    0.016886
37                         Region_North    0.016690
29                      loan_purpose_p3    0.016467
49                          total_units    0.013241
21                         Gender_Joint    0.013088
28                      loan_purpose_p2    0.012605
30                      loan_purpose_p4    0.011667
41                          loan_amount    0.010784
31                    occupancy_type_ir    0.010308
16          co-applicant_credit_type_CIB    0.010225
14                 construction_type_mh    0.009525
42                                 term    0.007672
22                          Gender_Male    0.007316
48                                  age    0.005338
6                      open_credit_nopc    0.005161
33                    occupancy_type_sr    0.004921
40                         Region_south    0.003721
23             Gender_Sex Not Available    0.003419
35                     credit_type_CRIF    0.003211
45                         Credit_Score    0.002650
36                      credit_type_EXP    0.002287
39                       Region_central    0.002118
20                        Gender_Female    0.001518
38                    Region_North-East    0.000784
15                 construction_type_sb    0.000000
7                       open_credit_opc    0.000000
17          co-applicant_credit_type_EXP    0.000000
9              Neg_ammortization_not_neg    0.000000
5                  Credit_Worthiness_l2    0.000000
19   submission_of_application_to_inst    0.000000
3                     approv_in_adv_pre    0.000000
13             lump_sum_payment_not_lpsm    0.000000
1                        loan_limit_ncf    0.000000
11                interest_only_not_int    0.000000
```

[40]:
```
# Predict on the test set
y_pred = grid_search_XGBoost.predict(X_test)

# Compute confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Get predicted probabilities for the positive class
y_scores = grid_search_XGBoost.predict_proba(X_test)[:, 1]

# Compute average precision (AUC-PR)
```

```
auc_pr = average_precision_score(y_test, y_scores)

# Create a figure
fig, axes = plt.subplots(1, # one row
                         2, # two columns
                         figsize=(12, 4)
                        )

# --- Confusion Matrix ---
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=grid_search_XGBoost.classes_
                             )
disp.plot(cmap="Reds",
          values_format='d',
          colorbar=False, #don't show the legend colormap
          ax=axes[0])
axes[0].set_title("Confusion Matrix - XGBoost")

# --- Precision-Recall Curve ---
PrecisionRecallDisplay.from_predictions(y_test,
                                        y_scores,
                                        name="XGBoost",
                                        ax=axes[1]
                                       )
axes[1].set_title(f"Precision-Recall Curve - XGBoost (AUC={auc_pr:.4f})")

plt.tight_layout()
plt.show()
```



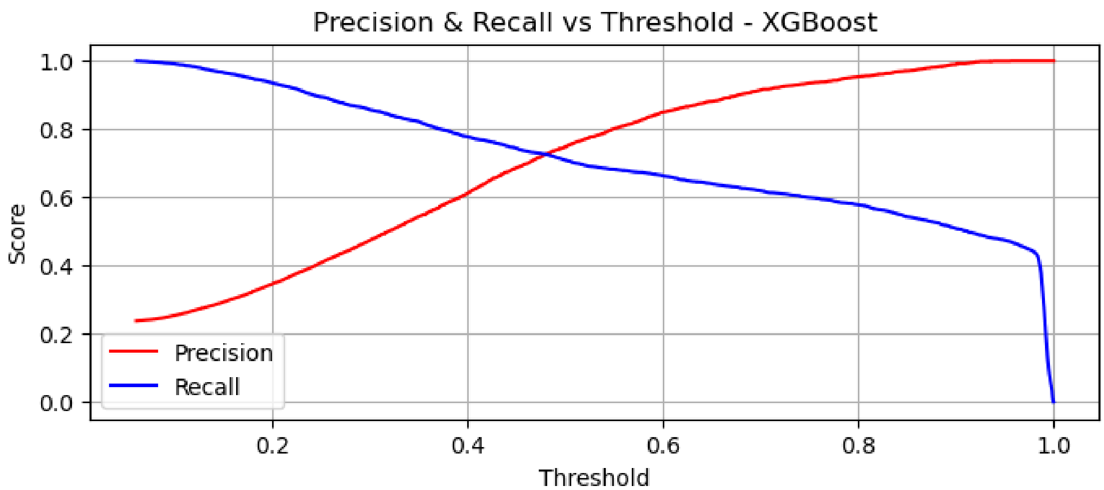
[41]:
```
# Predict on test set
y_pred = grid_search_XGBoost.predict(X_test)
```

```python
# Accuracy
acc = accuracy_score(y_test, y_pred)
# Precision (by default, for positive class in binary classification)
prec = precision_score(y_test, y_pred)
# Recall
rec = recall_score(y_test, y_pred)
# F1-score
f1 = f1_score(y_test, y_pred)

print(f"Accuracy: {acc:.4f}")
print(f"Precision: {prec:.4f}")
print(f"Recall: {rec:.4f}")
print(f"F1-score: {f1:.4f}")
```

```
Accuracy: 0.8743
Precision: 0.7466
Recall: 0.7095
F1-score: 0.7276
```

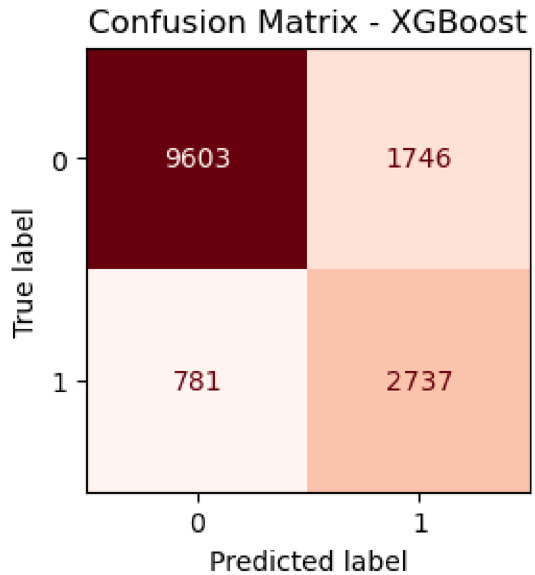
[42]:

```python
# Get predicted probabilities for the positive class
y_scores = grid_search_XGBoost.predict_proba(X_test)[:, 1]

# Compute precision, recall, thresholds
precision, recall, thresholds = precision_recall_curve(y_test, y_scores)

# Plot Precision and Recall vs Threshold
plt.figure(figsize=(8, 3))
plt.plot(thresholds, precision[:-1], label='Precision', color='red')
plt.plot(thresholds, recall[:-1], label='Recall', color='blue')
plt.xlabel('Threshold')
plt.ylabel('Score')
plt.title('Precision & Recall vs Threshold - XGBoost')
plt.legend()
plt.grid(True)
plt.show()
```



[43]:
```
y_scores = grid_search_XGBoost.predict_proba(X_test)[:, 1]
threshold = 0.4  # lower than 0.5, where it is centered
y_pred_adjusted = (y_scores >= threshold).astype(int)
```

[44]:
```
# Compute confusion matrix
cm = confusion_matrix(y_test,
                      y_pred_adjusted
                     )

# Display confusion matrix as a heatmap
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=grid_search_XGBoost.classes_
                             )
plt.figure(figsize=(3, 3)) #create a specific figure object in order to better␣
 ↪manipulate the dimensions

disp.plot(cmap="Reds",
          values_format='d', #show numbers as integers
          colorbar = False, #colorbar as legend disactivated
          ax=plt.gca() #plot inn the figure created
         )

plt.title("Confusion Matrix - XGBoost")
plt.show()
```



[45]:
```
# Accuracy
acc = accuracy_score(y_test, y_pred_adjusted)
# Precision (by default, for positive class in binary classification)
prec = precision_score(y_test, y_pred_adjusted)
# Recall
rec = recall_score(y_test, y_pred_adjusted)
# F1-score
f1 = f1_score(y_test, y_pred_adjusted)

print(f"Accuracy: {acc:.4f}")
print(f"Precision: {prec:.4f}")
print(f"Recall: {rec:.4f}")
print(f"F1-score: {f1:.4f}")
```

```
Accuracy: 0.8300
Precision: 0.6105
Recall: 0.7780
F1-score: 0.6842
```

The following code block performs *serialization*, it is a very helpful practice that allows to save a model once has been trained, so it saves the training time every time you open the notebook by simply opening directly the saved model.
There are many libraries that apply serialization and probably one of the best-known is **pickle**, even though it is not able to save **lambda** functions when present.
Since this practice can also save the full pipeline, and in our case there are **lambda** functions, we ended up using **cloudpickle**, that also saves the anonymous functions.
The storing format is binary.

[46]:
```
# --- Saving ---
if recovered is False: #if the model currently used has not been recovered from␣
↪serialization:
    with open("XGBoost_loan.pkl", mode="wb") as f: #wb stands for Writing Binary
        cloudpickle.dump(grid_search_XGBoost, f)

# --- Loading ---
"""
with open("XGBoost_loan.pkl", mode="rb") as f: #rb stands for reading binary
    XGBoost_recovered = cloudpickle.load(f)
"""
```

[46]:
```
'\nwith open("XGBoost_loan.pkl", mode="rb") as f: #rb stands for reading
binary\n    XGBoost_recovered = cloudpickle.load(f)\n'
```

1.5.2 4.2. Adaptive Boosting

[ ]:
```
#Setting AdaBoost classifier (with Decision Tree as base estimator)
ada_model = AdaBoostClassifier(
                                ␣
↪base_estimator=DecisionTreeClassifier(class_weight="balanced", #it balances␣
↪the dataset
                                                                    ),
                                random_state=42
                            )

#adding to the pipeline the algorithm
full_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', ada_model)
])

# Parameter grid for GridSearch
param_grid = {
    'classifier__base_estimator__max_depth': [2,3],       #Maximum depth of␣
↪decision trees
    'classifier__n_estimators': [50, 100, 200],            #Number of boosting␣
↪stages(how many weak learners to add)
    'classifier__learning_rate': [0.01, 0.1, 0.5, 1.0],    #Shrinkage␣
↪(learning) rate
    'classifier__algorithm': [#'SAMME', # Boosting algorithms, for our case␣
↪SAMME.R performs better
                              'SAMME.R']
}

# Grid search setup
grid_search_AdaBoost = GridSearchCV(
```

```
                            full_pipeline,
                            param_grid,
                            cv=strat_kfold,
                            n_jobs=-1,
                            verbose=2,
                        )

start_time = time.time() #for counting the time
# Fit GridSearchCV on training data
grid_search_AdaBoost.fit(X_train,
                y_train
               )
end_time = time.time() #end time of execution
elapsed_time = convert_time(end_time - start_time)
print(f"AdaBoost training + grid search took {elapsed_time}")

# Best params and score
print("Best parameters found:", grid_search_AdaBoost.best_params_)
print("Best cross-validation accuracy:", grid_search_AdaBoost.best_score_)

# Evaluate on the test set
test_score = grid_search_AdaBoost.score(X_test, y_test)
print(f"Test set accuracy: {test_score:.4f}")
```

[47]:
```
#try to see whether the above cell has been run
try:
    grid_search_AdaBoost
    recovered = False #boolean variable to understand wheter the model has been␣
 ↪recovered from serialization or not

#otherwise open the serialized model
except:
    with open("AdaBoost_loan.pkl", mode="rb") as f: #rb stands for reading␣
 ↪binary
        grid_search_AdaBoost = cloudpickle.load(f)
    recovered = True
```

[48]:
```
pd.DataFrame(grid_search_AdaBoost.cv_results_)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_14124\2811122147.py in <cell line: 1>()
----> 1 pd.DataFrame(grid_search_AdaBoost.cv_results_)

AttributeError: 'Pipeline' object has no attribute 'cv_results_'
```

[51]:
```
#we encapsulate just the best model
AdaBoost_whole_model = grid_search_AdaBoost.best_estimator_ #here we save the␣
↪whole best model full pipeline (prepocessor + classifier)
AdaBoost_classifier = AdaBoost_whole_model.named_steps['classifier'] #here we␣
↪save just the best classifier
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_14124\1057454025.py in <cell line: 2>()
      1 #we encapsulate just the best model
----> 2 AdaBoost_whole_model = grid_search_AdaBoost.best_estimator_ #here we␣
↪save the whole best model full pipeline (prepocessor + classifier)
      3 AdaBoost_classifier = AdaBoost_whole_model.named_steps['classifier']␣
↪#here we save just the best classifier

AttributeError: 'Pipeline' object has no attribute 'best_estimator_'
```

[52]:
```
#recover just the preprocessor from the whole model
preprocessor = AdaBoost_whole_model.named_steps['preprocessor']
feature_names = []

for name, transformer, cols in preprocessor.transformers_:
    #if the transformer is a drop operator
    if transformer == 'drop':
        continue
    #if the transformer is actually a pipeline object, so that has inside other␣
↪transformers
    if isinstance(transformer, Pipeline):
        #get the last step of the pipeline, because the last step is the one␣
↪that finally transforms the data and generates new names
        last_step = transformer.steps[-1][1]
        if hasattr(last_step, 'get_feature_names_out'):
            names = last_step.get_feature_names_out(cols)
        else:
            names = cols
    #if the transformer is directly a transformer
    else:
        if hasattr(transformer, 'get_feature_names_out'):
            names = transformer.get_feature_names_out(cols)
        else:
            names = cols

    feature_names.extend(names)

feat_imp = pd.DataFrame({
                        'feature': feature_names,
```

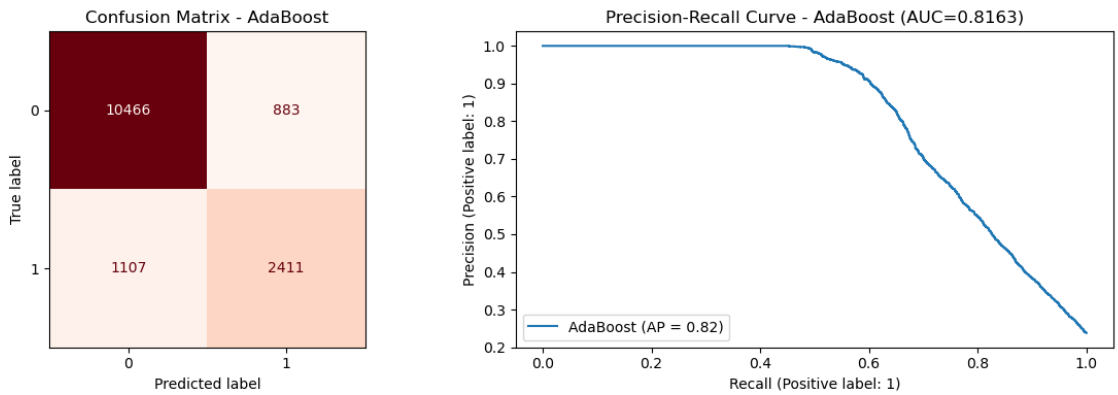
```
                        'importance': AdaBoost_classifier.feature_importances_
                        }).sort_values(
                                        by="importance",
                                        ascending=False
                                        )

print(feat_imp)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_14124\2077480982.py in <cell line: 2>()
      1 #recover just the preprocessor from the whole model
----> 2 preprocessor = AdaBoost_whole_model.named_steps['preprocessor']
      3 feature_names = []
      4 
      5 for name, transformer, cols in preprocessor.transformers_:

NameError: name 'AdaBoost_whole_model' is not defined
```

[53]:

```
# Predict on the test set
y_pred = grid_search_AdaBoost.predict(X_test)

# Compute confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Get predicted probabilities for the positive class
y_scores = grid_search_AdaBoost.predict_proba(X_test)[:, 1]

# Compute average precision (AUC-PR)
auc_pr = average_precision_score(y_test, y_scores)

# Create a figure
fig, axes = plt.subplots(1, # one row
                         2, # two columns
                         figsize=(12, 4)
                        )

# --- Confusion Matrix ---
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=grid_search_AdaBoost.classes_
                             )
disp.plot(cmap="Reds",
          values_format='d',
          colorbar=False, #don't show the legend colormap
          ax=axes[0])
axes[0].set_title("Confusion Matrix - AdaBoost")
```

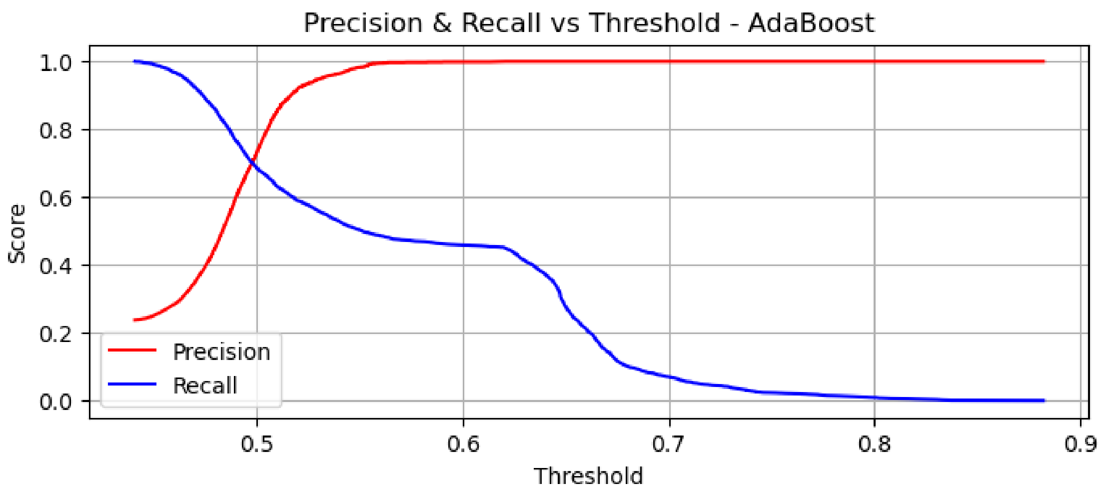
```
# --- Precision-Recall Curve ---
PrecisionRecallDisplay.from_predictions(y_test,
                                        y_scores,
                                        name="AdaBoost",
                                        ax=axes[1]
                                       )
axes[1].set_title(f"Precision-Recall Curve - AdaBoost (AUC={auc_pr:.4f})")

plt.tight_layout()
plt.show()
```



[54]:
```
# Predict on test set
y_pred = grid_search_AdaBoost.predict(X_test)

# Accuracy
acc = accuracy_score(y_test, y_pred)
# Precision (by default, for positive class in binary classification)
prec = precision_score(y_test, y_pred)
# Recall
rec = recall_score(y_test, y_pred)
# F1-score
f1 = f1_score(y_test, y_pred)

print(f"Accuracy: {acc:.4f}")
print(f"Precision: {prec:.4f}")
print(f"Recall: {rec:.4f}")
print(f"F1-score: {f1:.4f}")
```

```
Accuracy: 0.8661
Precision: 0.7319
Recall: 0.6853
F1-score: 0.7079
```

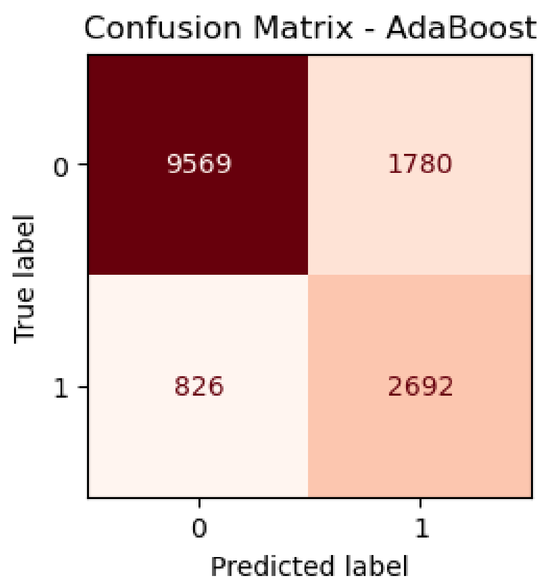
[55]:
```
# Get predicted probabilities for the positive class
y_scores = grid_search_AdaBoost.predict_proba(X_test)[:, 1]

# Compute precision, recall, thresholds
precision, recall, thresholds = precision_recall_curve(y_test, y_scores)

# Plot Precision and Recall vs Threshold
plt.figure(figsize=(8, 3))
plt.plot(thresholds, precision[:-1], label='Precision', color='red')
plt.plot(thresholds, recall[:-1], label='Recall', color='blue')
plt.xlabel('Threshold')
plt.ylabel('Score')
plt.title('Precision & Recall vs Threshold - AdaBoost')
plt.legend()
plt.grid(True)
plt.show()
```



[56]:
```
y_scores = grid_search_AdaBoost.predict_proba(X_test)[:, 1]
threshold = 0.49  # lower than 0.5, where it is centered
y_pred_adjusted = (y_scores >= threshold).astype(int)
```

[57]:
```
# Compute confusion matrix
cm = confusion_matrix(y_test,
                      y_pred_adjusted
                     )

# Display confusion matrix as a heatmap
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=grid_search_AdaBoost.classes_
                             )
```

```
plt.figure(figsize=(3, 3)) #create a specific figure object in order to better␣
↪manipulate the dimensions

disp.plot(cmap="Reds",
          values_format='d', #show numbers as integers
          colorbar = False, #colorbar as legend disactivated
          ax=plt.gca() #plot inn the figure created
         )

plt.title("Confusion Matrix - AdaBoost")
plt.show()
```



[58]:
```
# Accuracy
acc = accuracy_score(y_test, y_pred_adjusted)
# Precision (by default, for positive class in binary classification)
prec = precision_score(y_test, y_pred_adjusted)
# Recall
rec = recall_score(y_test, y_pred_adjusted)
# F1-score
f1 = f1_score(y_test, y_pred_adjusted)

print(f"Accuracy: {acc:.4f}")
print(f"Precision: {prec:.4f}")
print(f"Recall: {rec:.4f}")
print(f"F1-score: {f1:.4f}")
```

```
Accuracy: 0.8247
Precision: 0.6020
```

```
Recall: 0.7652
F1-score: 0.6738
```

[59]:
```
# --- Saving ---
if recovered is False: #if the model currently used has not been recovered from␣
↪serialization:
    with open("AdaBoost_loan.pkl", mode="wb") as f: #wb stands for Writing␣
↪Binary
        cloudpickle.dump(grid_search_AdaBoost, f)

# --- Loading ---
"""
with open("AdaBoost_loan.pkl", mode="rb") as f: #rb stands for reading binary
    AdaBoost_recovered = cloudpickle.load(f)
"""
```

[59]:
```
'\nwith open("AdaBoost_loan.pkl", mode="rb") as f: #rb stands for reading
binary\n    AdaBoost_recovered = cloudpickle.load(f)\n'
```

1.5.3 4.3. Gradient Boosting

[ ]:
```
#setting the GradientBoosting model
gb_model = GradientBoostingClassifier(
                                    random_state=42
                                )

#adding to the pipeline the algorithm
full_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', gb_model)
])

# Parameter grid for GridSearch
param_grid = {
    'classifier__n_estimators': [50, 100, 200],          #Number of boosting␣
↪stages(how many weak learners to add)
    'classifier__learning_rate': [0.01, 0.1, 0.2],       #Shrinkage (learning)␣
↪rate
    'classifier__max_depth': [2, 3],                   #Maximum depth of␣
↪decision trees
    'classifier__subsample': [0.8, 1.0],                 #fraction of samples␣
↪used for fitting each tree (classic gradient boosting Vs Stochastic)
    'classifier__min_samples_split': [2, 5]              #minimum samples␣
↪required to split a node
}

# Grid search setup
```

```
grid_search_GradientBoost = GridSearchCV(
                            full_pipeline,
                            param_grid,
                            cv=strat_kfold,
                            n_jobs=-1,
                            verbose=2
                        )

start_time = time.time() #for counting the time
# Fit GridSearchCV on training data
grid_search_GradientBoost.fit(X_train,
                                y_train
                               )
end_time = time.time() #end time of execution
elapsed_time = convert_time(end_time - start_time)
print(f"GradientBoost training + grid search took {elapsed_time}")

# Best params and score
print("Best parameters found:", grid_search_GradientBoost.best_params_)
print("Best cross-validation accuracy:", grid_search_GradientBoost.best_score_)

# Optional: Evaluate on the test set
test_score = grid_search_GradientBoost.score(X_test, y_test)
print(f"Test set accuracy: {test_score:.4f}")
```

[60]:
```
#try to see whether the above cell has been run
try:
    grid_search_GradientBoost
    recovered = False #boolean variable to understand wheter the model has been␣
 ↪recovered from serialization or not

#otherwise open the serialized model
except:
    with open("GradientBoost_loan.pkl", mode="rb") as f: #rb stands for reading␣
 ↪binary
        grid_search_GradientBoost = cloudpickle.load(f)
    recovered = True
```

[61]:
```
pd.DataFrame(grid_search_GradientBoost.cv_results_)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_14124\3663306907.py in <cell line: 1>()
----> 1 pd.DataFrame(grid_search_GradientBoost.cv_results_)

AttributeError: 'Pipeline' object has no attribute 'cv_results_'
```

```
[62]: #we encapsulate just the best model
GradientBoost_whole_model = grid_search_GradientBoost.best_estimator_ #here we␣
↪save the whole best model full pipeline (prepocessor + classifier)
GradientBoost_classifier = GradientBoost_whole_model.named_steps['classifier']␣
↪#here we save just the best classifier
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_14124\244217999.py in <cell line: 2>()
      1 #we encapsulate just the best model
----> 2 GradientBoost_whole_model = grid_search_GradientBoost.best_estimator_␣
↪#here we save the whole best model full pipeline (prepocessor + classifier)
      3 GradientBoost_classifier = GradientBoost_whole_model.
↪named_steps['classifier'] #here we save just the best classifier

AttributeError: 'Pipeline' object has no attribute 'best_estimator_'
```

```
[63]: #recover just the preprocessor from the whole model
preprocessor = GradientBoost_whole_model.named_steps['preprocessor']
feature_names = []

for name, transformer, cols in preprocessor.transformers_:
    #if the transformer is a drop operator
    if transformer == 'drop':
        continue
    #if the transformer is actually a pipeline object, so that has inside other␣
↪transformers
    if isinstance(transformer, Pipeline):
        #get the last step of the pipeline, because the last step is the one␣
↪that finally transforms the data and generates new names
        last_step = transformer.steps[-1][1]
        if hasattr(last_step, 'get_feature_names_out'):
            names = last_step.get_feature_names_out(cols)
        else:
            names = cols
    #if the transformer is directly a transformer
    else:
        if hasattr(transformer, 'get_feature_names_out'):
            names = transformer.get_feature_names_out(cols)
        else:
            names = cols

    feature_names.extend(names)

feat_imp = pd.DataFrame({
                        'feature': feature_names,
```

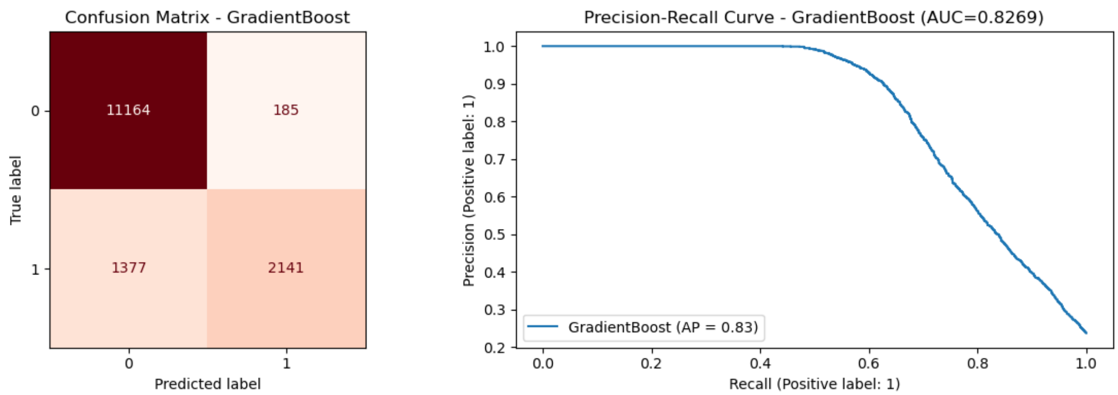
```
                         'importance': AdaBoost_classifier.feature_importances_
                         }).sort_values(
                                         by="importance",
                                         ascending=False
                                         )

print(feat_imp)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_14124\2084709128.py in <cell line: 2>()
      1 #recover just the preprocessor from the whole model
----> 2 preprocessor = GradientBoost_whole_model.named_steps['preprocessor']
      3 feature_names = []
      4
      5 for name, transformer, cols in preprocessor.transformers_:

NameError: name 'GradientBoost_whole_model' is not defined
```

[64]:

```
# Predict on the test set
y_pred = grid_search_GradientBoost.predict(X_test)

# Compute confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Get predicted probabilities for the positive class
y_scores = grid_search_GradientBoost.predict_proba(X_test)[:, 1]

# Compute average precision (AUC-PR)
auc_pr = average_precision_score(y_test, y_scores)

# Create a figure
fig, axes = plt.subplots(1, # one row
                         2, # two columns
                         figsize=(12, 4)
                        )

# --- Confusion Matrix ---
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=grid_search_GradientBoost.classes_
                             )
disp.plot(cmap="Reds",
          values_format='d',
          colorbar=False, #don't show the legend colormap
          ax=axes[0])
axes[0].set_title("Confusion Matrix - GradientBoost")
```

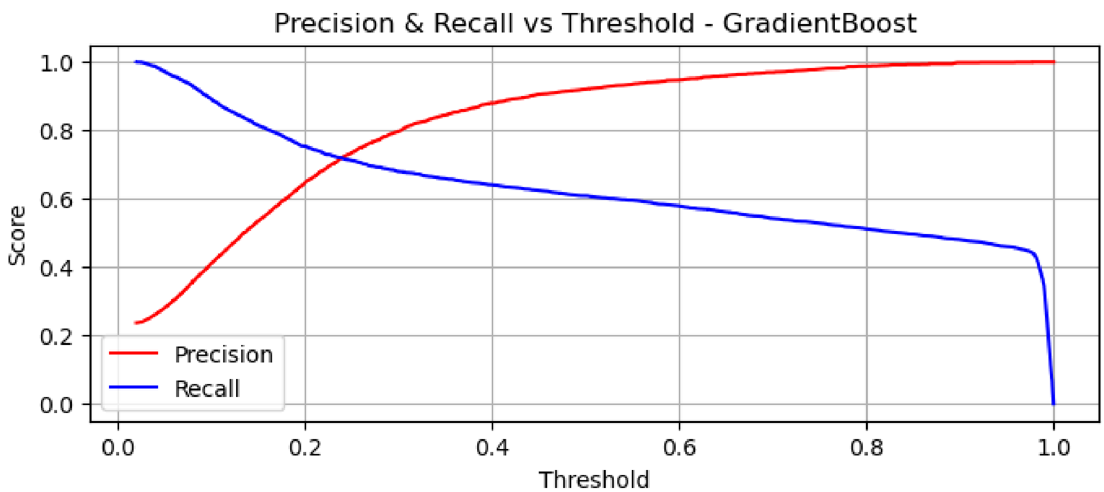
```
# --- Precision-Recall Curve ---
PrecisionRecallDisplay.from_predictions(y_test,
                                        y_scores,
                                        name="GradientBoost",
                                        ax=axes[1]
                                       )
axes[1].set_title(f"Precision-Recall Curve - GradientBoost (AUC={auc_pr:.4f})")

plt.tight_layout()
plt.show()
```



[65]:
```
# Predict on test set
y_pred = grid_search_GradientBoost.predict(X_test)

# Accuracy
acc = accuracy_score(y_test, y_pred)
# Precision (by default, for positive class in binary classification)
prec = precision_score(y_test, y_pred)
# Recall
rec = recall_score(y_test, y_pred)
# F1-score
f1 = f1_score(y_test, y_pred)

print(f"Accuracy: {acc:.4f}")
print(f"Precision: {prec:.4f}")
print(f"Recall: {rec:.4f}")
print(f"F1-score: {f1:.4f}")
```

```
Accuracy: 0.8949
Precision: 0.9205
Recall: 0.6086
F1-score: 0.7327
```

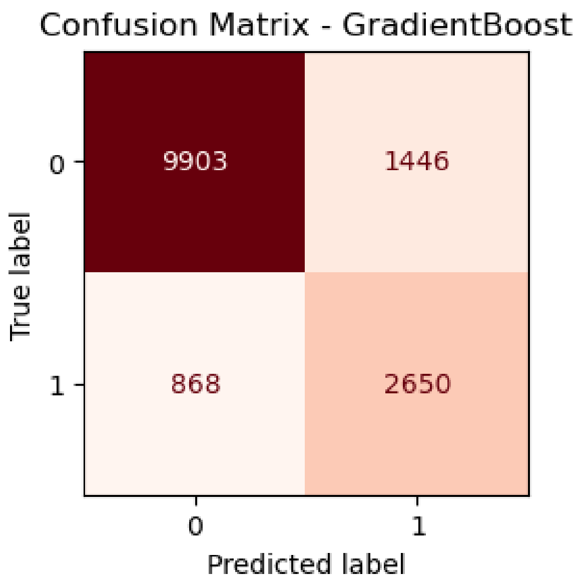
[66]:
```python
# Get predicted probabilities for the positive class
y_scores = grid_search_GradientBoost.predict_proba(X_test)[:, 1]

# Compute precision, recall, thresholds
precision, recall, thresholds = precision_recall_curve(y_test, y_scores)

# Plot Precision and Recall vs Threshold
plt.figure(figsize=(8, 3))
plt.plot(thresholds, precision[:-1], label='Precision', color='red')
plt.plot(thresholds, recall[:-1], label='Recall', color='blue')
plt.xlabel('Threshold')
plt.ylabel('Score')
plt.title('Precision & Recall vs Threshold - GradientBoost')
plt.legend()
plt.grid(True)
plt.show()
```



[67]:
```python
y_scores = grid_search_GradientBoost.predict_proba(X_test)[:, 1]
threshold = 0.2  # lower than 0.5, where it is centered
y_pred_adjusted = (y_scores >= threshold).astype(int)
```

[68]:
```python
# Compute confusion matrix
cm = confusion_matrix(y_test,
                      y_pred_adjusted
                     )

# Display confusion matrix as a heatmap
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=grid_search_GradientBoost.classes_
                             )
```

```
plt.figure(figsize=(3, 3)) #create a specific figure object in order to better␣
 ↪manipulate the dimensions

disp.plot(cmap="Reds",
          values_format='d', #show numbers as integers
          colorbar = False, #colorbar as legend disactivated
          ax=plt.gca() #plot inn the figure created
         )

plt.title("Confusion Matrix - GradientBoost")
plt.show()
```



[69]:
```
# Accuracy
acc = accuracy_score(y_test, y_pred_adjusted)
# Precision (by default, for positive class in binary classification)
prec = precision_score(y_test, y_pred_adjusted)
# Recall
rec = recall_score(y_test, y_pred_adjusted)
# F1-score
f1 = f1_score(y_test, y_pred_adjusted)

print(f"Accuracy: {acc:.4f}")
print(f"Precision: {prec:.4f}")
print(f"Recall: {rec:.4f}")
print(f"F1-score: {f1:.4f}")
```

```
Accuracy: 0.8444
Precision: 0.6470
```

```
Recall: 0.7533
F1-score: 0.6961
```

[70]:
```python
# --- Saving ---
if recovered is False: #if the model currently used has not been recovered from␣
↪serialization:
    with open("GradientBoost_loan.pkl", mode="wb") as f: #wb stands for Writing␣
↪Binary
        cloudpickle.dump(grid_search_GradientBoost, f)

# --- Loading ---
"""
with open("GradientBoost_loan.pkl", mode="rb") as f: #rb stands for reading␣
↪binary
    GradientBoost_recovered = cloudpickle.load(f)
"""
```

[70]:
```
'\nwith open("GradientBoost_loan.pkl", mode="rb") as f: #rb stands for reading
binary\n    GradientBoost_recovered = cloudpickle.load(f)\n'
```

1.6 5. Comparisons

Now that all three models have been run we can make comparisons and draw conclusions.

Since in our dataset the positive class is rare, as a rule of thumb we should prefer the Precision-Recall curves for comparison, hence we will combine them with the best scores found after tweaking the precision-recall threshold, in order to choose the best model.

```
  XGBoost
<img src="PR-curve XGBoost.png" style="padding:1%"><br>
Accuracy: 0.8300 <br>
Precision: 0.6105 <br>
Recall: 0.7780 <br>
F1-score: 0.6842
```

```
  AdaBoost
<img src="PR-curve AdaBoost.png" style="padding:1%"><br>
Accuracy: 0.8247 <br>
Precision: 0.6020 <br>
Recall: 0.7652 <br>
F1-score: 0.6738
```

```
  GradientBoost
<img src="PR-curve GradientBoost.png" style="padding:1%"><br>
Accuracy: 0.8444 <br>
Precision: 0.6470 <br>
Recall: 0.7533 <br>
F1-score: 0.6961
```

All three models are quite good, and have similar scores, even though AdaBoost is clearly the one

with the lowest scores.
On the other hand XGBoost and GradientBoost have different scores each one with its own pros and cons, if we take into account the overall F1-score that is basically the harmonic mean between precision and recall:

$$\text{F1-score} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} \ + \ \text{recall}}$$

we would choose GradientBoost, since this metric is comprehensive of both Precision and Recall.
On the other hand if we take into account the general principle that for this project we want to prioritize recall over precision and even further we take a look at the training time:

XGBoost	AdaBoost	GradientBoost
00:03:48	01:36:11	03:15:27

XGBoost is the one that performs better.

1.7 6. Conclusion and Possible Expansions

The project here can have many further expansions…

1.8 7. Appendix

The following part contains some tips and interesting tricks…

1.8.1 7.A Python stay-awake module

At the end of section 5. we higlighted the training time for the three models. If we sum all training times we quickly understand that in total, running this notebook from scratch requires approximately five hours.
If we do not want to change the settings of our computer in order to prevent sleep mode, we can use a python script that runs in a different thread as a background daemon, since it requires very few CPU.
Online there are many modules from which we can take advantage of.

For example this one here is very easy and straighforward to implement, even though if you run it on Windows 10 or newer versions you need to adjust it a bit, because you need to automatic mouse moving does not prevent sleep mode anymore see.
For example once you have found out the location of the script in your computer with: **`pip show stay-awake`** you can change the **`__init__.py`** inside the `stay-wake` folder script so that instead of moving the mouse it activates/deactives the ScrollLock button.

Once the script is ready, it is very easy to use it, just open a Shell prompt ad type: **`python -m stay-awake`**

2 TODO!

- ho mixato numerici e ordinari;
- possibili sviluppi futuri e riadattamenti;
- frase d'impatto;
- usa delle curve e grafici per mostrare la qualità del modello:
  - learning curves;
  - feature importance, gini…;
  - learning rate;
  - early stopping?;
  - Parametri specifici dei due boosting;
  - features importance for the choices made by the trees;
- take a look at the jupyter notebook files of: Classificators, Decision Trees and Ensemble Learning;
- adjusting other models for Recall;

[78]:
```
import nbconvert
print(nbconvert.__version__)
```

```
6.4.4
```

[82]:
```
nbconvert "loan eligibity.ipynb" --to webpdf
```

```
  File "C:\Users\Francesco\AppData\Local\Temp\ipykernel_14124\3101380777.py",␣
↪line 1
    nbconvert "loan eligibity.ipynb" --to webpdf
              ^

SyntaxError: invalid syntax
```

[ ]:
```

```