project_working_file copy

April 26, 2024

1 Summary Report

2 A mini project : Feature and Model Selection

2.1 Table of contents

0. Imports Introduction
1. Overview of the dataset
2. Data splitting
3. EDA
4. Preprocessing and transformations
5. Baseline model
6. Linear models
7. Different models
8. Hyperparameter optimization
9. Interpretation and feature importances
10. Results on the test set
11. Summary of the results

2.2 Imports

[1]:
```
import os

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from sklearn import datasets
from sklearn.compose import ColumnTransformer, make_column_transformer
from sklearn.dummy import DummyRegressor, DummyClassifier
from sklearn.linear_model import LogisticRegression, Ridge
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.impute import SimpleImputer
from sklearn.metrics import (
    accuracy_score,
    auc,
    average_precision_score,
    classification_report,
```

```
    confusion_matrix,
    f1_score,
    make_scorer,
    precision_score,
    recall_score,
)

from sklearn.model_selection import (
    cross_val_score,
    cross_validate,
    train_test_split,
)

from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder, StandardScaler
from sklearn.tree import DecisionTreeRegressor, export_graphviz

%matplotlib inline
```

2.3 Introduction

This is a mini-project where my group consolidated all the various concepts we learned in the Supervised Learning and Model Selection course during the Master's Program at UBC to address an interesting problem.

2.4 Problem:

- A classification problem of predicting whether a credit card client will default or not. For this problem, you will use Default of Credit Card Clients Dataset. In this data set, there are 30,000 examples and 24 features, and the goal is to estimate whether a person will default (fail to pay) their credit card bills; this column is labeled "default.payment.next.month" in the data.

2.5 1. Overview of the dataset

- The data set is about credit transactions of credit card clients in Taiwan from April 2005 to September 2005.
- The problem is to predict whether a credit card client will default (fail to pay) the credit card bills.
- The target column is `default.payment.next.month` with 2 values: 1 = yes, 0 = no.
- The following 23 features can be used as explanatory variables:
  - LIMIT_BAL: Amount of the given credit (NT dollar)
  - SEX: Gender (1 = male, 2 = female)
  - EDUCATION: Education level (1 = graduate school, 2 = university, 3 = high school, 4 = others, 5 or 6 = unknown)
  - MARRIAGE: Marital status (1 = married, 2 = single, 3 = others)
  - AGE: Age (years)

- PAY_0 – PAY_6: Status of past monthly payment (-1 = pay duly, 1 = payment delay for one month,..., 9 = payment delay for nine months and above), where PAY_0 = repayment status in September 2005,..., PAY_6 = repayment status in April 2005.
- BILL_AMT1 – BILL_AMT6: Amount of bill statement (NT dollar) from September 2005 to April 2005, respectively.
- PAY_AMT1 – PAY_AMT6: Amount of previous statement (NT dollar) from September 2005 to April 2005, respectively.

[2]:
```
# 2. Read in the data
credit_card_df = pd.read_csv("UCI_Credit_Card.csv")
credit_card_df.sort_index()
```

[2]:
```
          ID  LIMIT_BAL  SEX  EDUCATION  MARRIAGE  AGE  PAY_0  PAY_2  PAY_3  \
0          1    20000.0    2          2         1   24      2      2     -1
1          2   120000.0    2          2         2   26     -1      2      0
2          3    90000.0    2          2         2   34      0      0      0
3          4    50000.0    2          2         1   37      0      0      0
4          5    50000.0    1          2         1   57     -1      0     -1
…        …         … …          …         … …      …     …     …
29995  29996   220000.0    1          3         1   39      0      0      0
29996  29997   150000.0    1          3         2   43     -1     -1     -1
29997  29998    30000.0    1          2         2   37      4      3      2
29998  29999    80000.0    1          3         1   41      1     -1      0
29999  30000    50000.0    1          2         1   46      0      0      0

       PAY_4  …  BILL_AMT4  BILL_AMT5  BILL_AMT6  PAY_AMT1  PAY_AMT2  \
0         -1  …        0.0        0.0        0.0       0.0     689.0
1          0  …     3272.0     3455.0     3261.0       0.0    1000.0
2          0  …    14331.0    14948.0    15549.0    1518.0    1500.0
3          0  …    28314.0    28959.0    29547.0    2000.0    2019.0
4          0  …    20940.0    19146.0    19131.0    2000.0   36681.0
…        … …         …         …          …        …       …
29995      0  …    88004.0    31237.0    15980.0    8500.0   20000.0
29996     -1  …     8979.0     5190.0        0.0    1837.0    3526.0
29997     -1  …    20878.0    20582.0    19357.0       0.0       0.0
29998      0  …    52774.0    11855.0    48944.0   85900.0    3409.0
29999      0  …    36535.0    32428.0    15313.0    2078.0    1800.0

       PAY_AMT3  PAY_AMT4  PAY_AMT5  PAY_AMT6  default.payment.next.month
0           0.0       0.0       0.0       0.0                           1
1        1000.0    1000.0       0.0    2000.0                           1
2        1000.0    1000.0    1000.0    5000.0                           0
3        1200.0    1100.0    1069.0    1000.0                           0
4       10000.0    9000.0     689.0     679.0                           0
…           …        …      …       …                           …
29995    5003.0    3047.0    5000.0    1000.0                           0
29996    8998.0     129.0       0.0       0.0                           0
```

```
29997   22000.0    4200.0    2000.0    3100.0                            1
29998    1178.0    1926.0   52964.0    1804.0                            1
29999    1430.0    1000.0    1000.0    1000.0                            1

[30000 rows x 25 columns]
```

Preliminary Preprocessing:
Based on the results of `.info()` and `.describe()` below, we can see that there are no missing values in the data set, and feature names are quite standard except that the repayment status columns are `PAY_0`, `PAY_2`, etc. with no `PAY_1`). Hence, besides renaming column `PAY_0` to `PAY_1`, there is no need to do any other preliminary preprocessing.

[3]:
```
credit_card_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30000 entries, 0 to 29999
Data columns (total 25 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   ID                          30000 non-null  int64
 1   LIMIT_BAL                   30000 non-null  float64
 2   SEX                         30000 non-null  int64
 3   EDUCATION                   30000 non-null  int64
 4   MARRIAGE                    30000 non-null  int64
 5   AGE                         30000 non-null  int64
 6   PAY_0                       30000 non-null  int64
 7   PAY_2                       30000 non-null  int64
 8   PAY_3                       30000 non-null  int64
 9   PAY_4                       30000 non-null  int64
 10  PAY_5                       30000 non-null  int64
 11  PAY_6                       30000 non-null  int64
 12  BILL_AMT1                   30000 non-null  float64
 13  BILL_AMT2                   30000 non-null  float64
 14  BILL_AMT3                   30000 non-null  float64
 15  BILL_AMT4                   30000 non-null  float64
 16  BILL_AMT5                   30000 non-null  float64
 17  BILL_AMT6                   30000 non-null  float64
 18  PAY_AMT1                    30000 non-null  float64
 19  PAY_AMT2                    30000 non-null  float64
 20  PAY_AMT3                    30000 non-null  float64
 21  PAY_AMT4                    30000 non-null  float64
 22  PAY_AMT5                    30000 non-null  float64
 23  PAY_AMT6                    30000 non-null  float64
 24  default.payment.next.month  30000 non-null  int64
dtypes: float64(13), int64(12)
memory usage: 5.7 MB
```

[4]:
```
credit_card_df.describe().T
```

[4]:
```
                              count           mean            std       min  \
ID                          30000.0   15000.500000    8660.398374       1.0
LIMIT_BAL                   30000.0  167484.322667  129747.661567   10000.0
SEX                         30000.0       1.603733       0.489129       1.0
EDUCATION                   30000.0       1.853133       0.790349       0.0
MARRIAGE                    30000.0       1.551867       0.521970       0.0
AGE                         30000.0      35.485500       9.217904      21.0
PAY_0                       30000.0      -0.016700       1.123802      -2.0
PAY_2                       30000.0      -0.133767       1.197186      -2.0
PAY_3                       30000.0      -0.166200       1.196868      -2.0
PAY_4                       30000.0      -0.220667       1.169139      -2.0
PAY_5                       30000.0      -0.266200       1.133187      -2.0
PAY_6                       30000.0      -0.291100       1.149988      -2.0
BILL_AMT1                   30000.0   51223.330900   73635.860576 -165580.0
BILL_AMT2                   30000.0   49179.075167   71173.768783  -69777.0
BILL_AMT3                   30000.0   47013.154800   69349.387427 -157264.0
BILL_AMT4                   30000.0   43262.948967   64332.856134 -170000.0
BILL_AMT5                   30000.0   40311.400967   60797.155770  -81334.0
BILL_AMT6                   30000.0   38871.760400   59554.107537 -339603.0
PAY_AMT1                    30000.0    5663.580500   16563.280354       0.0
PAY_AMT2                    30000.0    5921.163500   23040.870402       0.0
PAY_AMT3                    30000.0    5225.681500   17606.961470       0.0
PAY_AMT4                    30000.0    4826.076867   15666.159744       0.0
PAY_AMT5                    30000.0    4799.387633   15278.305679       0.0
PAY_AMT6                    30000.0    5215.502567   17777.465775       0.0
default.payment.next.month  30000.0       0.221200       0.415062       0.0

                                 25%       50%        75%        max
ID                           7500.75   15000.5   22500.25    30000.0
LIMIT_BAL                   50000.00  140000.0  240000.00  1000000.0
SEX                             1.00       2.0       2.00        2.0
EDUCATION                       1.00       2.0       2.00        6.0
MARRIAGE                        1.00       2.0       2.00        3.0
AGE                            28.00      34.0      41.00       79.0
PAY_0                          -1.00       0.0       0.00        8.0
PAY_2                          -1.00       0.0       0.00        8.0
PAY_3                          -1.00       0.0       0.00        8.0
PAY_4                          -1.00       0.0       0.00        8.0
PAY_5                          -1.00       0.0       0.00        8.0
PAY_6                          -1.00       0.0       0.00        8.0
BILL_AMT1                    3558.75   22381.5   67091.00   964511.0
BILL_AMT2                    2984.75   21200.0   64006.25   983931.0
BILL_AMT3                    2666.25   20088.5   60164.75  1664089.0
BILL_AMT4                    2326.75   19052.0   54506.00   891586.0
BILL_AMT5                    1763.00   18104.5   50190.50   927171.0
```

```
BILL_AMT6                      1256.00    17071.0    49198.25    961664.0
PAY_AMT1                       1000.00     2100.0     5006.00    873552.0
PAY_AMT2                        833.00     2009.0     5000.00   1684259.0
PAY_AMT3                        390.00     1800.0     4505.00    896040.0
PAY_AMT4                        296.00     1500.0     4013.25    621000.0
PAY_AMT5                        252.50     1500.0     4031.50    426529.0
PAY_AMT6                        117.75     1500.0     4000.00    528666.0
default.payment.next.month        0.00        0.0        0.00         1.0
```

[5]:
```
credit_card_df = credit_card_df.rename(columns={"PAY_0": "PAY_1"})
credit_card_df.sort_index()
```

[5]:
```
          ID  LIMIT_BAL  SEX  EDUCATION  MARRIAGE  AGE  PAY_1  PAY_2  PAY_3  \
0          1    20000.0    2          2         1   24      2      2     -1
1          2   120000.0    2          2         2   26     -1      2      0
2          3    90000.0    2          2         2   34      0      0      0
3          4    50000.0    2          2         1   37      0      0      0
4          5    50000.0    1          2         1   57     -1      0     -1
…        …        … …     …        … …     …     …     …
29995  29996   220000.0    1          3         1   39      0      0      0
29996  29997   150000.0    1          3         2   43     -1     -1     -1
29997  29998    30000.0    1          2         2   37      4      3      2
29998  29999    80000.0    1          3         1   41      1     -1      0
29999  30000    50000.0    1          2         1   46      0      0      0

       PAY_4  …  BILL_AMT4  BILL_AMT5  BILL_AMT6  PAY_AMT1  PAY_AMT2  \
0         -1  …        0.0        0.0        0.0       0.0     689.0
1          0  …     3272.0     3455.0     3261.0       0.0    1000.0
2          0  …    14331.0    14948.0    15549.0    1518.0    1500.0
3          0  …    28314.0    28959.0    29547.0    2000.0    2019.0
4          0  …    20940.0    19146.0    19131.0    2000.0   36681.0
…        …  …          …          …          …         …         …
29995      0  …    88004.0    31237.0    15980.0    8500.0   20000.0
29996     -1  …     8979.0     5190.0        0.0    1837.0    3526.0
29997     -1  …    20878.0    20582.0    19357.0       0.0       0.0
29998      0  …    52774.0    11855.0    48944.0   85900.0    3409.0
29999      0  …    36535.0    32428.0    15313.0    2078.0    1800.0

       PAY_AMT3  PAY_AMT4  PAY_AMT5  PAY_AMT6  default.payment.next.month
0           0.0       0.0       0.0       0.0                           1
1        1000.0    1000.0       0.0    2000.0                           1
2        1000.0    1000.0    1000.0    5000.0                           0
3        1200.0    1100.0    1069.0    1000.0                           0
4       10000.0    9000.0     689.0     679.0                           0
…           …        …        …        …                         …
29995    5003.0    3047.0    5000.0    1000.0                           0
29996    8998.0     129.0       0.0       0.0                           0
```

```
29997    22000.0     4200.0     2000.0     3100.0                            1
29998     1178.0     1926.0    52964.0     1804.0                            1
29999     1430.0     1000.0     1000.0     1000.0                            1

[30000 rows x 25 columns]
```

2.6 2. Data splitting

Split the data into train and test portions.
`test_size` determines the portion of the data which will go into test sets.
`random_state`: the purpose of setting a random seed (using random_state) is to ensure reproducibility

Answer:

[6]:
```
train_df, test_df = train_test_split(credit_card_df, test_size=0.3,
 ↪random_state=123)
```

[7]:
```
train_df.head()
```

[7]:
```
          ID  LIMIT_BAL  SEX  EDUCATION  MARRIAGE  AGE  PAY_1  PAY_2  PAY_3  \
16395  16396   320000.0    2          1         2   36      0      0      0
21448  21449   440000.0    2          1         2   30     -1     -1     -1
20034  20035   160000.0    2          3         1   44     -2     -2     -2
25755  25756   120000.0    2          2         1   30      0      0      0
1438    1439    50000.0    1          2         2   54      1      2      0

       PAY_4  …  BILL_AMT4  BILL_AMT5  BILL_AMT6  PAY_AMT1  PAY_AMT2  \
16395      0  …    19370.0    10155.0     3788.0    5000.0    5018.0
21448      0  …   171244.0   150897.0   117870.0     612.0   87426.0
20034     -2  …      -18.0      -18.0      -18.0       0.0       0.0
25755      0  …   103058.0    71095.0    47379.0    3706.0    5502.0
1438       0  …    27585.0    27910.0    27380.0       0.0    1400.0

       PAY_AMT3  PAY_AMT4  PAY_AMT5  PAY_AMT6  default.payment.next.month
16395    1000.0    3000.0       0.0    7013.0                           0
21448  130007.0    3018.0   15000.0   51663.0                           0
20034       0.0       0.0       0.0       0.0                           0
25755    4204.0    3017.0    2005.0    1702.0                           0
1438     1200.0    1500.0    1000.0    1500.0                           0

[5 rows x 25 columns]
```

[8]:
```
train_df.shape
```

[8]:
```
(21000, 25)
```

[9]:
```
X_train, y_train = train_df.drop(columns=["default.payment.next.month"]),␣
↪train_df["default.payment.next.month"]
X_test, y_test = test_df.drop(columns=["default.payment.next.month"]),␣
↪test_df["default.payment.next.month"]
```

2.7 3. Exploratory Data Analysis

The analysis is performed on the train set including two summary statistics and two visualizations to summarize our observations about the data. Then, appropriate metric(s) will be picked for further assessment.

Summary Statistics & Visualization 1: Correlation among features

[10]:
```
cor_all = train_df.corr()
cor_all
```

[10]:
```
                                 ID  LIMIT_BAL       SEX  EDUCATION  \
ID                         1.000000   0.028419  0.019014   0.040633
LIMIT_BAL                  0.028419   1.000000  0.027466  -0.223207
SEX                        0.019014   0.027466  1.000000   0.012307
EDUCATION                  0.040633  -0.223207  0.012307   1.000000
MARRIAGE                  -0.024071  -0.115202 -0.033413  -0.142499
AGE                        0.021795   0.146419 -0.091890   0.175042
PAY_1                     -0.029574  -0.271686 -0.061038   0.111222
PAY_2                     -0.011899  -0.299924 -0.073214   0.125907
PAY_3                     -0.017471  -0.289222 -0.068192   0.118096
PAY_4                     -0.000293  -0.269399 -0.063772   0.110732
PAY_5                     -0.022719  -0.249030 -0.055062   0.101603
PAY_6                     -0.022829  -0.236218 -0.041594   0.088186
BILL_AMT1                  0.020447   0.283635 -0.035212   0.026108
BILL_AMT2                  0.019669   0.277334 -0.031960   0.020668
BILL_AMT3                  0.028270   0.283969 -0.023333   0.016967
BILL_AMT4                  0.042005   0.297468 -0.022471   0.003286
BILL_AMT5                  0.020323   0.299353 -0.015973  -0.005203
BILL_AMT6                  0.019477   0.293757 -0.015158  -0.005595
PAY_AMT1                   0.013488   0.191669  0.001324  -0.039769
PAY_AMT2                   0.013389   0.183705  0.000908  -0.028295
PAY_AMT3                   0.036544   0.206416 -0.008136  -0.039621
PAY_AMT4                   0.010153   0.204308  0.001473  -0.038918
PAY_AMT5                  -0.000093   0.215244 -0.004470  -0.031589
PAY_AMT6                   0.001252   0.215337 -0.001600  -0.038563
default.payment.next.month -0.017861  -0.149247 -0.046320   0.026558

                           MARRIAGE       AGE     PAY_1     PAY_2     PAY_3  \
ID                        -0.024071  0.021795 -0.029574 -0.011899 -0.017471
LIMIT_BAL                 -0.115202  0.146419 -0.271686 -0.299924 -0.289222
SEX                       -0.033413 -0.091890 -0.061038 -0.073214 -0.068192
EDUCATION                 -0.142499  0.175042  0.111222  0.125907  0.118096
```

```
MARRIAGE                    1.000000 -0.414446  0.016416  0.023994  0.035001
AGE                        -0.414446  1.000000 -0.032232 -0.045343 -0.050597
PAY_1                       0.016416 -0.032232  1.000000  0.670967  0.571947
PAY_2                       0.023994 -0.045343  0.670967  1.000000  0.770190
PAY_3                       0.035001 -0.050597  0.571947  0.770190  1.000000
PAY_4                       0.031905 -0.047465  0.534071  0.664641  0.779639
PAY_5                       0.035830 -0.050073  0.504219  0.622672  0.685032
PAY_6                       0.029353 -0.041689  0.470939  0.575450  0.631932
BILL_AMT1                  -0.027264  0.064703  0.186433  0.235992  0.207658
BILL_AMT2                  -0.024688  0.061367  0.187401  0.236222  0.235810
BILL_AMT3                  -0.029866  0.062484  0.179065  0.226122  0.227556
BILL_AMT4                  -0.028532  0.061071  0.172808  0.218033  0.221998
BILL_AMT5                  -0.031878  0.059023  0.175548  0.216657  0.219870
BILL_AMT6                  -0.027376  0.058003  0.173661  0.216919  0.218656
PAY_AMT1                   -0.001337  0.023255 -0.076790 -0.076280  0.002073
PAY_AMT2                   -0.005287  0.023572 -0.071744 -0.057781 -0.068541
PAY_AMT3                   -0.002401  0.034136 -0.074217 -0.055477 -0.054288
PAY_AMT4                   -0.014206  0.025197 -0.065880 -0.052057 -0.052475
PAY_AMT5                   -0.000819  0.028544 -0.053086 -0.033231 -0.035079
PAY_AMT6                   -0.007532  0.017527 -0.063282 -0.039994 -0.043431
default.payment.next.month -0.021735  0.010715  0.325102  0.265160  0.240503

                               PAY_4  …  BILL_AMT4  BILL_AMT5  BILL_AMT6  \
ID                         -0.000293  …   0.042005   0.020323   0.019477
LIMIT_BAL                  -0.269399  …   0.297468   0.299353   0.293757
SEX                        -0.063772  …  -0.022471  -0.015973  -0.015158
EDUCATION                   0.110732  …   0.003286  -0.005203  -0.005595
MARRIAGE                    0.031905  …  -0.028532  -0.031878  -0.027376
AGE                        -0.047465  …   0.061071   0.059023   0.058003
PAY_1                       0.534071  …   0.172808   0.175548   0.173661
PAY_2                       0.664641  …   0.218033   0.216657   0.216919
PAY_3                       0.779639  …   0.221998   0.219870   0.218656
PAY_4                       1.000000  …   0.240648   0.236905   0.234309
PAY_5                       0.817452  …   0.265262   0.263856   0.257387
PAY_6                       0.713851  …   0.262129   0.287259   0.282981
BILL_AMT1                   0.201714  …   0.861671   0.830888   0.805667
BILL_AMT2                   0.224515  …   0.892482   0.858302   0.832519
BILL_AMT3                   0.245064  …   0.931703   0.892319   0.858671
BILL_AMT4                   0.240648  …   1.000000   0.941142   0.902447
BILL_AMT5                   0.236905  …   0.941142   1.000000   0.944748
BILL_AMT6                   0.234309  …   0.902447   0.944748   1.000000
PAY_AMT1                   -0.005518  …   0.243395   0.227985   0.211163
PAY_AMT2                    0.000193  …   0.223203   0.199445   0.175053
PAY_AMT3                   -0.069715  …   0.298209   0.248918   0.230284
PAY_AMT4                   -0.048923  …   0.138020   0.292033   0.244266
PAY_AMT5                   -0.032754  …   0.166755   0.149058   0.309427
PAY_AMT6                   -0.028226  …   0.169249   0.156680   0.105011
```

```
default.payment.next.month  0.219692  …  -0.012313  -0.007868  -0.004944

                            PAY_AMT1  PAY_AMT2  PAY_AMT3  PAY_AMT4  PAY_AMT5  \
ID                          0.013488  0.013389  0.036544  0.010153 -0.000093
LIMIT_BAL                   0.191669  0.183705  0.206416  0.204308  0.215244
SEX                         0.001324  0.000908 -0.008136  0.001473 -0.004470
EDUCATION                  -0.039769 -0.028295 -0.039621 -0.038918 -0.031589
MARRIAGE                   -0.001337 -0.005287 -0.002401 -0.014206 -0.000819
AGE                         0.023255  0.023572  0.034136  0.025197  0.028544
PAY_1                      -0.076790 -0.071744 -0.074217 -0.065880 -0.053086
PAY_2                      -0.076280 -0.057781 -0.055477 -0.052057 -0.033231
PAY_3                       0.002073 -0.068541 -0.054288 -0.052475 -0.035079
PAY_4                      -0.005518  0.000193 -0.069715 -0.048923 -0.032754
PAY_5                      -0.002720  0.000538  0.008884 -0.058547 -0.032159
PAY_6                       0.002340 -0.001092  0.005805  0.016991 -0.045211
BILL_AMT1                   0.146775  0.108425  0.150079  0.150477  0.172609
BILL_AMT2                   0.282783  0.113772  0.146473  0.139343  0.165163
BILL_AMT3                   0.252786  0.285372  0.119375  0.134327  0.171117
BILL_AMT4                   0.243395  0.223203  0.298209  0.138020  0.166755
BILL_AMT5                   0.227985  0.199445  0.248918  0.292033  0.149058
BILL_AMT6                   0.211163  0.175053  0.230284  0.244266  0.309427
PAY_AMT1                    1.000000  0.359611  0.263792  0.217311  0.147038
PAY_AMT2                    0.359611  1.000000  0.268024  0.212814  0.134298
PAY_AMT3                    0.263792  0.268024  1.000000  0.224648  0.142094
PAY_AMT4                    0.217311  0.212814  0.224648  1.000000  0.126775
PAY_AMT5                    0.147038  0.134298  0.142094  0.126775  1.000000
PAY_AMT6                    0.177324  0.173510  0.146699  0.149169  0.141182
default.payment.next.month -0.071563 -0.060730 -0.060868 -0.061005 -0.050943

                            PAY_AMT6  default.payment.next.month
ID                          0.001252                   -0.017861
LIMIT_BAL                   0.215337                   -0.149247
SEX                        -0.001600                   -0.046320
EDUCATION                  -0.038563                    0.026558
MARRIAGE                   -0.007532                   -0.021735
AGE                         0.017527                    0.010715
PAY_1                      -0.063282                    0.325102
PAY_2                      -0.039994                    0.265160
PAY_3                      -0.043431                    0.240503
PAY_4                      -0.028226                    0.219692
PAY_5                      -0.025219                    0.208726
PAY_6                      -0.029144                    0.194787
BILL_AMT1                   0.170210                   -0.020632
BILL_AMT2                   0.169213                   -0.015301
BILL_AMT3                   0.180129                   -0.014718
BILL_AMT4                   0.169249                   -0.012313
BILL_AMT5                   0.156680                   -0.007868
```

```
BILL_AMT6                   0.105011                     -0.004944
PAY_AMT1                    0.177324                     -0.071563
PAY_AMT2                    0.173510                     -0.060730
PAY_AMT3                    0.146699                     -0.060868
PAY_AMT4                    0.149169                     -0.061005
PAY_AMT5                    0.141182                     -0.050943
PAY_AMT6                    1.000000                     -0.056093
default.payment.next.month -0.056093                      1.000000

[25 rows x 25 columns]
```

The correlation table above matches with our intuition that limiting balance and previous repayment records seem to be most correlated to the target variable. Hence, we will take a closer look into these possibly most relevant features.

[11]:
```
possibly_most_relevant = [
    "LIMIT_BAL",
    "PAY_1",
    "PAY_2",
    "PAY_3",
    "PAY_4",
    "PAY_5",
    "PAY_6",
    "default.payment.next.month",
]
cor_core = train_df[possibly_most_relevant].corr()
cor_core
```

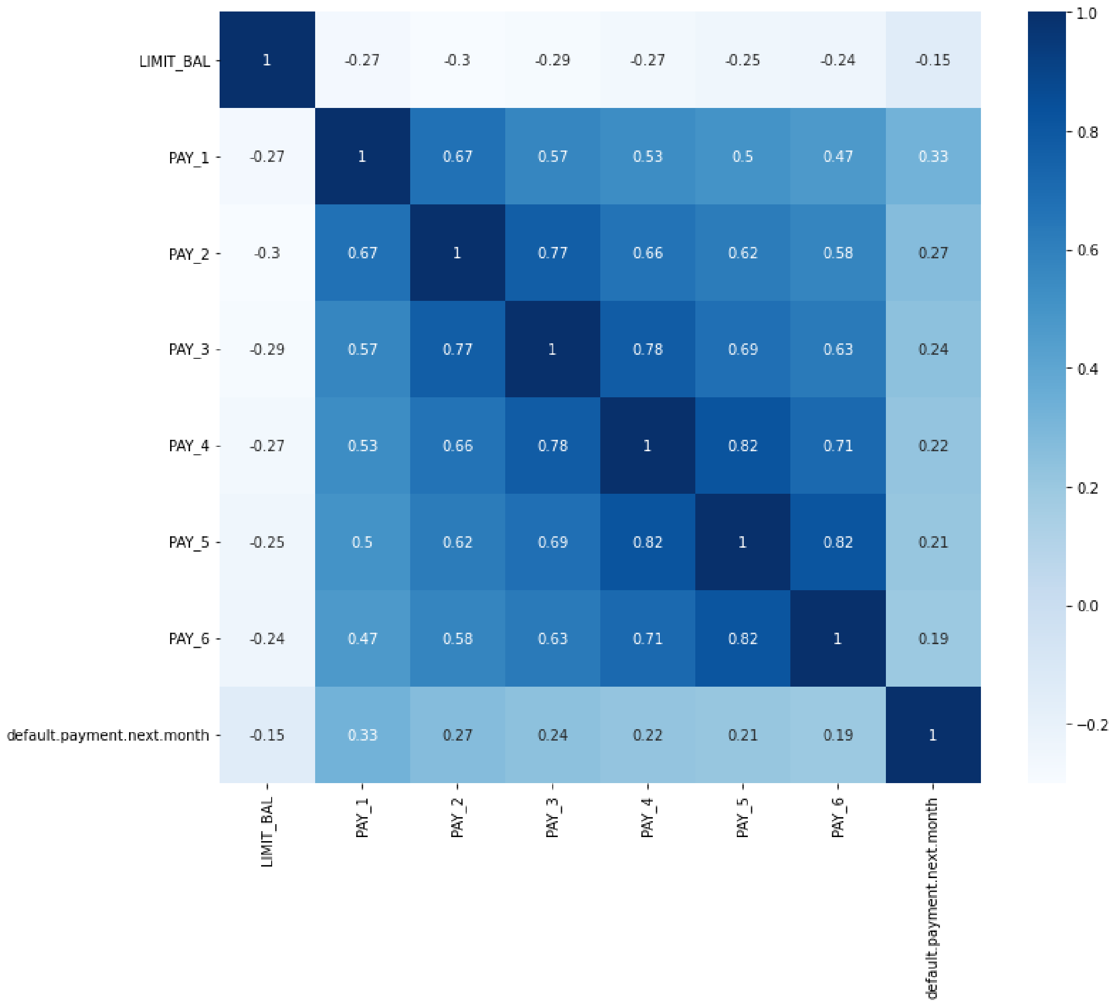
[11]:
```
                            LIMIT_BAL     PAY_1     PAY_2     PAY_3     PAY_4  \
LIMIT_BAL                    1.000000 -0.271686 -0.299924 -0.289222 -0.269399
PAY_1                       -0.271686  1.000000  0.670967  0.571947  0.534071
PAY_2                       -0.299924  0.670967  1.000000  0.770190  0.664641
PAY_3                       -0.289222  0.571947  0.770190  1.000000  0.779639
PAY_4                       -0.269399  0.534071  0.664641  0.779639  1.000000
PAY_5                       -0.249030  0.504219  0.622672  0.685032  0.817452
PAY_6                       -0.236218  0.470939  0.575450  0.631932  0.713851
default.payment.next.month  -0.149247  0.325102  0.265160  0.240503  0.219692

                               PAY_5     PAY_6  default.payment.next.month
LIMIT_BAL                  -0.249030 -0.236218                   -0.149247
PAY_1                       0.504219  0.470939                    0.325102
PAY_2                       0.622672  0.575450                    0.265160
PAY_3                       0.685032  0.631932                    0.240503
PAY_4                       0.817452  0.713851                    0.219692
PAY_5                       1.000000  0.815793                    0.208726
PAY_6                       0.815793  1.000000                    0.194787
default.payment.next.month  0.208726  0.194787                    1.000000
```

[12]:

```
import seaborn as sns

plt.figure(figsize=(12, 10))
sns.heatmap(cor_core, annot=True, cmap=plt.cm.Blues)
plt.show()
```



The correlation plot suggests the Repayment Status features are highly correlated. This makes sense because these features are lag features.

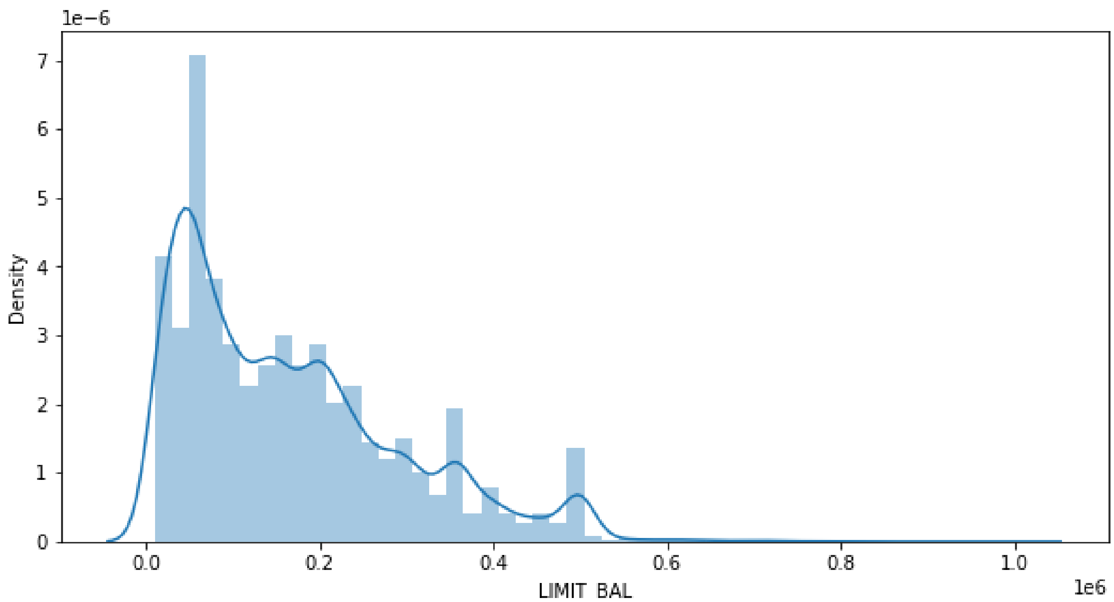
Summary Statistics & Visualization 2: Distribution of LIMIT_BAL

[13]:

```
plt.figure(figsize=(10, 5))
sns.distplot(train_df.LIMIT_BAL)
plt.show()
```

```
/opt/anaconda3/envs/573/lib/python3.9/site-
packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a
```

```
deprecated function and will be removed in a future version. Please adapt your
code to use either `displot` (a figure-level function with similar flexibility)
or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```



The distribution plot shows that the limit balance is right-skewed, and the majority of clients are given the credit line ranging from 0 to 200,000 (NT dollar).

Metric Selection

Lastly, we look at the distribution of the target variable.

[14]:
```
train_df["default.payment.next.month"].value_counts(normalize=True)
```

[14]:
```
0    0.776762
1    0.223238
Name: default.payment.next.month, dtype: float64
```

We do have a class imbalance in the data set since only 22% of the examples in the training set belong to the "default" class (class 1). We are more interested in the "default" class because it is more important to catch as many credit card clients who will default as possible so that the bank can stop offering them credit lines. Therefore, we decided to pick the f1-score as our most important metric.

[15]:
```
scoring = ["accuracy", "recall", "precision", "f1", "average_precision"]
```

2.8 4. Preprocessing and transformations

In this part, we focus on:

1. Identify different feature types and the transformations we apply on each feature type.
2. Define a column transformer

[16]:
```
# 1. Identify feature types

drop_features = ["ID"]

categorical_features = ["MARRIAGE", "EDUCATION"]

binary_features = ["SEX"]

numeric_features = list(
    set(X_train.columns)
    - set(categorical_features)
    - set(binary_features)
    - set(drop_features)
)
```

Rationality - drop_features: Drop ID as it is a unique identifier for each row that is unlikely to be useful. - categorical_features: MARRIAGE and EDUCATION are numbers to begin with but have categorical meanings. Note: EDUCATION looks like an already encoded ordinal column. However, the undefined/unknown values are problematic, and the documentation does not provide enough information on how to deal with them properly. Hence, we would encode this feature with OHE. - binary_features: SEX is binary. - numeric_features: Treat the rest as numeric and standardize them. Note: PAY_1 - PAY_6 look like already encoded ordinal features. Moreover, even though these features are collinear, it should not be a problem because we are using regularized models. Hence, I would keep all these features and treat them as numeric to apply scaling on.

[17]:
```
# 2. Define a column transformer

numeric_transformer = make_pipeline(StandardScaler())

binary_transformer = make_pipeline(OneHotEncoder(drop="if_binary", dtype=int))

categorical_transformer = make_pipeline(OneHotEncoder(handle_unknown="ignore",␣
↪sparse=False))

preprocessor = make_column_transformer(
    (numeric_transformer, numeric_features),
    (binary_transformer, binary_features),
    (categorical_transformer, categorical_features),
    ("drop", drop_features),
)
```

[18]:
```
# 3. Transform training set
preprocessor.fit(X_train)
```

```
[18]: ColumnTransformer(transformers=[('pipeline-1',
                                  Pipeline(steps=[('standardscaler',
                                                   StandardScaler())]),
                                  ['BILL_AMT1', 'PAY_AMT4', 'BILL_AMT5',
                                   'BILL_AMT2', 'PAY_AMT3', 'LIMIT_BAL', 'PAY_6',
                                   'PAY_AMT2', 'PAY_1', 'PAY_2', 'PAY_AMT6',
                                   'PAY_AMT5', 'PAY_4', 'BILL_AMT6', 'BILL_AMT4',
                                   'AGE', 'BILL_AMT3', 'PAY_3', 'PAY_5',
                                   'PAY_AMT1']),
                                 ('pipeline-2',
                                  Pipeline(steps=[('onehotencoder',
OneHotEncoder(drop='if_binary',
                                                                 dtype=<class
'int'>))]),
                                  ['SEX']),
                                 ('pipeline-3',
                                  Pipeline(steps=[('onehotencoder',
OneHotEncoder(handle_unknown='ignore',
sparse=False))]),
                                  ['MARRIAGE', 'EDUCATION']),
                                 ('drop', 'drop', ['ID'])])
```

2.9 5. Baseline model

In this part we will try baseline model as a starting benchmark.

We use `scikit-learn`'s baseline model and report results.

Since this is a classification problem, we will use DummyClassifier as the baseline model.

```
[19]: results = {}
```

```
[20]: # The code is adapted from lectures and previous labs:

def mean_std_cross_val_scores(model, X_train, y_train, **kwargs):
    """
    Returns mean and std of cross validation

    Parameters
    ----------
    model :
        scikit-learn model
    X_train : numpy array or pandas DataFrame
        X in the training data
    y_train :
        y in the training data

    Returns
```

```
    ----------
        pandas Series with mean scores from cross_validation
    """

    scores = cross_validate(model, X_train, y_train, **kwargs)

    mean_scores = pd.DataFrame(scores).mean()
    std_scores = pd.DataFrame(scores).std()
    out_col = []

    for i in range(len(mean_scores)):
        out_col.append((f"%0.3f (+/- %0.3f)" % (mean_scores[i], std_scores[i])))

    return pd.Series(data=out_col, index=mean_scores.index)
```

[21]:
```
dummy_model = DummyClassifier(strategy="stratified")
results["Dummy"] = mean_std_cross_val_scores(
    dummy_model, X_train, y_train, return_train_score=True, scoring=scoring
)
pd.DataFrame(results)
```

[21]:

	Dummy
fit_time	0.001 (+/- 0.000)
score_time	0.004 (+/- 0.000)
test_accuracy	0.650 (+/- 0.005)
train_accuracy	0.655 (+/- 0.004)
test_recall	0.227 (+/- 0.010)
train_recall	0.225 (+/- 0.012)
test_precision	0.223 (+/- 0.009)
train_precision	0.226 (+/- 0.009)
test_f1	0.225 (+/- 0.009)
train_f1	0.225 (+/- 0.010)
test_average_precision	0.224 (+/- 0.001)
train_average_precision	0.223 (+/- 0.001)

Result The test scores are very low. F1=0.225 only.

2.10 6. Linear models

Next, we want to try a linear model as a first real attempt. Also, we carry out hyperparameter tuning to explore different values for the regularization hyperparameter. After that, we report cross-validation scores along with standard deviation and summarize our results.

Since this is a classification problem,we will use LogisticRegression as our linear model.

Note: As seen above, we have a class imbalance in the data set. Hence, we apply `class_weight="balanced"` before hyperparameter optimization to deal with the class imbalance. Since doing so results in much better scores, we will fix this in hyperparamter optimization and search for other hyperparameters.

[22]:
```python
# 1. Run logistic regression without hyperparameter optimization

pipe_lr = make_pipeline(
    preprocessor, LogisticRegression(max_iter=1000, class_weight="balanced",␣
 ↪random_state=123)
)
results["Logistic Regression"] = mean_std_cross_val_scores(
    pipe_lr, X_train, y_train, return_train_score=True, scoring=scoring
)
pd.DataFrame(results)
```

[22]:
```
                                      Dummy Logistic Regression
fit_time                 0.001 (+/- 0.000)   0.324 (+/- 0.061)
score_time               0.004 (+/- 0.000)   0.013 (+/- 0.002)
test_accuracy            0.650 (+/- 0.005)   0.683 (+/- 0.007)
train_accuracy           0.655 (+/- 0.004)   0.686 (+/- 0.003)
test_recall              0.227 (+/- 0.010)   0.646 (+/- 0.021)
train_recall             0.225 (+/- 0.012)   0.649 (+/- 0.005)
test_precision           0.223 (+/- 0.009)   0.378 (+/- 0.007)
train_precision          0.226 (+/- 0.009)   0.381 (+/- 0.003)
test_f1                  0.225 (+/- 0.009)   0.477 (+/- 0.009)
train_f1                 0.225 (+/- 0.010)   0.480 (+/- 0.003)
test_average_precision   0.224 (+/- 0.001)   0.507 (+/- 0.016)
train_average_precision  0.223 (+/- 0.001)   0.509 (+/- 0.004)
```

[23]:
```python
# 2. Carry out hyperparameter optimization

from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import loguniform

param_dist_lr = {
    "logisticregression__C": loguniform(1e-3, 1e3),
}

search_lr = RandomizedSearchCV(
    pipe_lr,
    param_dist_lr,
    n_iter=50,
    verbose=1,
    n_jobs=-1,
    return_train_score=True,
    scoring="f1",
    random_state=123,
)

search_lr.fit(X_train, y_train);
```

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
```

[24]:
```python
search_lr.best_params_
```

[24]:
```
{'logisticregression__C': 0.011290431413903904}
```

[25]:
```python
search_lr.best_score_
```

[25]:
```
0.47983419889173823
```

[26]:
```python
# 3. Report scores

results["Tuned Logistic Regression"] = mean_std_cross_val_scores(
    search_lr.best_estimator_, X_train, y_train, return_train_score=True,␣
 ↪scoring=scoring
)
pd.DataFrame(results)
```

[26]:
```
                                     Dummy Logistic Regression  \
fit_time                 0.001 (+/- 0.000)   0.324 (+/- 0.061)
score_time               0.004 (+/- 0.000)   0.013 (+/- 0.002)
test_accuracy            0.650 (+/- 0.005)   0.683 (+/- 0.007)
train_accuracy           0.655 (+/- 0.004)   0.686 (+/- 0.003)
test_recall              0.227 (+/- 0.010)   0.646 (+/- 0.021)
train_recall             0.225 (+/- 0.012)   0.649 (+/- 0.005)
test_precision           0.223 (+/- 0.009)   0.378 (+/- 0.007)
train_precision          0.226 (+/- 0.009)   0.381 (+/- 0.003)
test_f1                  0.225 (+/- 0.009)   0.477 (+/- 0.009)
train_f1                 0.225 (+/- 0.010)   0.480 (+/- 0.003)
test_average_precision   0.224 (+/- 0.001)   0.507 (+/- 0.016)
train_average_precision  0.223 (+/- 0.001)   0.509 (+/- 0.004)

                        Tuned Logistic Regression
fit_time                        0.068 (+/- 0.005)
score_time                      0.012 (+/- 0.002)
test_accuracy                   0.689 (+/- 0.007)
train_accuracy                  0.690 (+/- 0.003)
test_recall                     0.642 (+/- 0.021)
train_recall                    0.643 (+/- 0.004)
test_precision                  0.383 (+/- 0.008)
train_precision                 0.384 (+/- 0.004)
test_f1                         0.480 (+/- 0.009)
train_f1                        0.481 (+/- 0.004)
test_average_precision          0.507 (+/- 0.015)
train_average_precision         0.508 (+/- 0.004)
```

Summarize the results: - The best hyperparameter found by our random search is C = 0.01 with a validation f1-score of 0.48. - The tuned logistic regression model seems to not improve much compared to the model without hyperparameter optimization. In fact, recall decreases a bit while accuracy, precision, and f1 slightly increase.

2.11 7. Different machine learning models

We will try at least 3 other models aside from a linear model then summarize the results in terms of overfitting/underfitting and fit and score times. From the results, it is interesting that we can figure out if machine learning model can beat a linear model.

Machine Learning Models

We decide to choose RandomForest, KNeighbors, and LGBM as 3 models for this classification purpose.

[27]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from lightgbm.sklearn import LGBMClassifier

pipe_rf = make_pipeline(preprocessor,␣
 ↪RandomForestClassifier(class_weight="balanced", random_state=123))
pipe_knn = make_pipeline(preprocessor, KNeighborsClassifier())
pipe_lgbm = make_pipeline(preprocessor, LGBMClassifier(class_weight="balanced",␣
 ↪random_state=123))

models = {
    "Random Forest": pipe_rf,
    "KNN": pipe_knn,
    "LightGBM": pipe_lgbm
}
```

[28]:
```
for (name, model) in models.items():
    results[name] = mean_std_cross_val_scores(
        model, X_train, y_train, return_train_score=True, scoring=scoring
    )
pd.DataFrame(results)
```

[28]:
```
                                    Dummy Logistic Regression  \
fit_time                 0.001 (+/- 0.000)   0.324 (+/- 0.061)
score_time               0.004 (+/- 0.000)   0.013 (+/- 0.002)
test_accuracy            0.650 (+/- 0.005)   0.683 (+/- 0.007)
train_accuracy           0.655 (+/- 0.004)   0.686 (+/- 0.003)
test_recall              0.227 (+/- 0.010)   0.646 (+/- 0.021)
train_recall             0.225 (+/- 0.012)   0.649 (+/- 0.005)
test_precision           0.223 (+/- 0.009)   0.378 (+/- 0.007)
train_precision          0.226 (+/- 0.009)   0.381 (+/- 0.003)
test_f1                  0.225 (+/- 0.009)   0.477 (+/- 0.009)
train_f1                 0.225 (+/- 0.010)   0.480 (+/- 0.003)
test_average_precision   0.224 (+/- 0.001)   0.507 (+/- 0.016)
train_average_precision  0.223 (+/- 0.001)   0.509 (+/- 0.004)
```

```
                          Tuned Logistic Regression      Random Forest  \
fit_time                          0.068 (+/- 0.005)  2.706 (+/- 0.034)
score_time                        0.012 (+/- 0.002)  0.132 (+/- 0.003)
test_accuracy                     0.689 (+/- 0.007)  0.814 (+/- 0.005)
train_accuracy                    0.690 (+/- 0.003)  0.999 (+/- 0.000)
test_recall                       0.642 (+/- 0.021)  0.348 (+/- 0.013)
train_recall                      0.643 (+/- 0.004)  1.000 (+/- 0.000)
test_precision                    0.383 (+/- 0.008)  0.659 (+/- 0.024)
train_precision                   0.384 (+/- 0.004)  0.997 (+/- 0.000)
test_f1                           0.480 (+/- 0.009)  0.455 (+/- 0.015)
train_f1                          0.481 (+/- 0.004)  0.998 (+/- 0.000)
test_average_precision            0.507 (+/- 0.015)  0.541 (+/- 0.017)
train_average_precision           0.508 (+/- 0.004)  1.000 (+/- 0.000)

                                       KNN           LightGBM
fit_time                 0.017 (+/- 0.004)  0.187 (+/- 0.013)
score_time               2.457 (+/- 0.198)  0.027 (+/- 0.001)
test_accuracy            0.793 (+/- 0.005)  0.765 (+/- 0.007)
train_accuracy           0.844 (+/- 0.001)  0.824 (+/- 0.003)
test_recall              0.355 (+/- 0.012)  0.615 (+/- 0.014)
train_recall             0.471 (+/- 0.004)  0.775 (+/- 0.009)
test_precision           0.559 (+/- 0.017)  0.480 (+/- 0.012)
train_precision          0.733 (+/- 0.004)  0.580 (+/- 0.005)
test_f1                  0.434 (+/- 0.013)  0.539 (+/- 0.013)
train_f1                 0.573 (+/- 0.003)  0.664 (+/- 0.004)
test_average_precision   0.418 (+/- 0.009)  0.562 (+/- 0.019)
train_average_precision  0.643 (+/- 0.004)  0.739 (+/- 0.003)
```

Summarize the results: - Regarding score, LightGBM has the highest validation f1 score while KNN has the lowest. Hence, we can see that not all non-linear model can beat the linear model. - Regarding overfitting/ underfitting, Random Forest is overfitting badly because the F1 score on the train set is 0.998 while on test set is only 0.455. - Regarding fit and score time, most models are quite quick, except Random Forest takes a while to fit and KNN takes a while to score.

Overall, `LightGBM` is the best performing model as it achieves the best score, is fast, and does not overfit/underfit.

2.12 8. Hyperparameter optimization

We perform hyperparameter optimization on the best-performing model, which is `LightGBM` and summarize the results. We use `sklearn`'s methods for hyperparameter optimization.
- RandomizedSearchCV

[29]:
```
import numpy as np
param_dist_lgbm = {
    "lgbmclassifier__max_depth": np.arange(1, 20, 2),
    "lgbmclassifier__num_leaves": np.arange(20, 80, 5),
```

```
    "lgbmclassifier__max_bin": np.arange(200, 300, 20),
}

search_lgbm = RandomizedSearchCV(
    pipe_lgbm,
    param_dist_lgbm,
    n_iter=50,
    verbose=1,
    n_jobs=-1,
    return_train_score=True,
    scoring="f1",
    random_state=123,
)

search_lgbm.fit(X_train, y_train);
```

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
```

[30]:
```
results["Tuned LGBMClassification"] = mean_std_cross_val_scores(
    search_lgbm.best_estimator_, X_train, y_train, return_train_score=True,␣
 ↪scoring=scoring
)
pd.DataFrame(results)
```

[30]:
```
                                      Dummy Logistic Regression  \
fit_time                 0.001 (+/- 0.000)    0.324 (+/- 0.061)
score_time               0.004 (+/- 0.000)    0.013 (+/- 0.002)
test_accuracy            0.650 (+/- 0.005)    0.683 (+/- 0.007)
train_accuracy           0.655 (+/- 0.004)    0.686 (+/- 0.003)
test_recall              0.227 (+/- 0.010)    0.646 (+/- 0.021)
train_recall             0.225 (+/- 0.012)    0.649 (+/- 0.005)
test_precision           0.223 (+/- 0.009)    0.378 (+/- 0.007)
train_precision          0.226 (+/- 0.009)    0.381 (+/- 0.003)
test_f1                  0.225 (+/- 0.009)    0.477 (+/- 0.009)
train_f1                 0.225 (+/- 0.010)    0.480 (+/- 0.003)
test_average_precision   0.224 (+/- 0.001)    0.507 (+/- 0.016)
train_average_precision  0.223 (+/- 0.001)    0.509 (+/- 0.004)

                        Tuned Logistic Regression      Random Forest  \
fit_time                        0.068 (+/- 0.005)  2.706 (+/- 0.034)
score_time                      0.012 (+/- 0.002)  0.132 (+/- 0.003)
test_accuracy                   0.689 (+/- 0.007)  0.814 (+/- 0.005)
train_accuracy                  0.690 (+/- 0.003)  0.999 (+/- 0.000)
test_recall                     0.642 (+/- 0.021)  0.348 (+/- 0.013)
train_recall                    0.643 (+/- 0.004)  1.000 (+/- 0.000)
test_precision                  0.383 (+/- 0.008)  0.659 (+/- 0.024)
train_precision                 0.384 (+/- 0.004)  0.997 (+/- 0.000)
```

```
test_f1                         0.480 (+/- 0.009)  0.455 (+/- 0.015)
train_f1                        0.481 (+/- 0.004)  0.998 (+/- 0.000)
test_average_precision          0.507 (+/- 0.015)  0.541 (+/- 0.017)
train_average_precision         0.508 (+/- 0.004)  1.000 (+/- 0.000)

                                      KNN           LightGBM  \
fit_time                 0.017 (+/- 0.004)  0.187 (+/- 0.013)
score_time               2.457 (+/- 0.198)  0.027 (+/- 0.001)
test_accuracy            0.793 (+/- 0.005)  0.765 (+/- 0.007)
train_accuracy           0.844 (+/- 0.001)  0.824 (+/- 0.003)
test_recall              0.355 (+/- 0.012)  0.615 (+/- 0.014)
train_recall             0.471 (+/- 0.004)  0.775 (+/- 0.009)
test_precision           0.559 (+/- 0.017)  0.480 (+/- 0.012)
train_precision          0.733 (+/- 0.004)  0.580 (+/- 0.005)
test_f1                  0.434 (+/- 0.013)  0.539 (+/- 0.013)
train_f1                 0.573 (+/- 0.003)  0.664 (+/- 0.004)
test_average_precision   0.418 (+/- 0.009)  0.562 (+/- 0.019)
train_average_precision  0.643 (+/- 0.004)  0.739 (+/- 0.003)

                        Tuned LGBMClassification
fit_time                       0.133 (+/- 0.003)
score_time                     0.022 (+/- 0.001)
test_accuracy                  0.768 (+/- 0.008)
train_accuracy                 0.801 (+/- 0.004)
test_recall                    0.625 (+/- 0.016)
train_recall                   0.698 (+/- 0.006)
test_precision                 0.485 (+/- 0.014)
train_precision                0.543 (+/- 0.008)
test_f1                        0.546 (+/- 0.014)
train_f1                       0.611 (+/- 0.005)
test_average_precision         0.567 (+/- 0.019)
train_average_precision        0.677 (+/- 0.005)
```

[31]:
```
search_lgbm.best_params_
```

[31]:
```
{'lgbmclassifier__num_leaves': 60,
 'lgbmclassifier__max_depth': 5,
 'lgbmclassifier__max_bin': 240}
```

[32]:
```
search_lgbm.best_score_
```

[32]:
```
0.5459147030871935
```

Result: The best hyperparameters found by our random search are: num_leaves = 0.01129, max_depth=5, max_bin=240 with a validation f1-score of 0.546.

2.13 9. Interpretation and feature importances

We use the `shap` methods to examine the most important features of the `LightGBM` models. 2. Summarize your observations.

[33]:
```
import shap
```

[34]:
```
preprocessor.fit(X_train, y_train)
ohe_feature_names = (
    preprocessor
    .named_transformers_["pipeline-3"]
    .named_steps["onehotencoder"]
    .get_feature_names_out(categorical_features)
    .tolist()
)
feature_names = numeric_features + binary_features + ohe_feature_names
```

[35]:
```
X_train_enc = pd.DataFrame(
    data=preprocessor.transform(X_train),
    columns=feature_names,
    index=X_train.index,
)
X_train_enc.head()
```

[35]:
```
       BILL_AMT1  PAY_AMT4  BILL_AMT5  BILL_AMT2  PAY_AMT3  LIMIT_BAL  \
16395  -0.300665 -0.114944  -0.494781  -0.293394 -0.234603   1.168355
21448  -0.685307 -0.113778   1.805461  -0.679495  6.785208   2.090017
20034  -0.696132 -0.309323  -0.661045  -0.688319 -0.289017  -0.060527
25755   0.687456 -0.113843   0.501203   0.752583 -0.060260  -0.367748
1438   -0.040230 -0.212134  -0.204599  -0.031399 -0.223720  -0.905384

          PAY_6  PAY_AMT2     PAY_1     PAY_2  …  MARRIAGE_1  MARRIAGE_2  \
16395  0.257059 -0.040229  0.013770  0.114774  …         0.0         1.0
21448  0.257059  3.739796 -0.878738 -0.722412  …         0.0         1.0
20034 -1.485154 -0.270403 -1.771246 -1.559598  …         1.0         0.0
25755  0.257059 -0.018028  0.013770  0.114774  …         1.0         0.0
1438   0.257059 -0.206185  0.906278  1.789147  …         0.0         1.0

       MARRIAGE_3  EDUCATION_0  EDUCATION_1  EDUCATION_2  EDUCATION_3  \
16395         0.0          0.0          1.0          0.0          0.0
21448         0.0          0.0          1.0          0.0          0.0
20034         0.0          0.0          0.0          0.0          1.0
25755         0.0          0.0          0.0          1.0          0.0
1438          0.0          0.0          0.0          1.0          0.0

       EDUCATION_4  EDUCATION_5  EDUCATION_6
16395          0.0          0.0          0.0
21448          0.0          0.0          0.0
```

```
20034          0.0          0.0          0.0
25755          0.0          0.0          0.0
1438           0.0          0.0          0.0

[5 rows x 32 columns]
```

[36]:
```
X_test_enc = pd.DataFrame(
    data=preprocessor.transform(X_test),
    columns=feature_names,
    index=X_test.index,
)
X_test_enc.head()
```

[36]:
```
       BILL_AMT1  PAY_AMT4  BILL_AMT5  BILL_AMT2  PAY_AMT3  LIMIT_BAL  \
25665  -0.301142  1.140290   0.058763  -0.346448 -0.289017  -0.982189
16464   0.334336 -0.205460   0.162513   0.293371 -0.180189  -0.674969
22386   1.427002  0.532986   2.086523   1.536341 -0.289017   0.016278
10149  -0.374955 -0.309323  -0.660751  -0.677772 -0.289017   0.246693
8729   -0.584044 -0.287229  -0.506842  -0.575543 -0.271006  -0.905384

          PAY_6  PAY_AMT2     PAY_1     PAY_2  …  MARRIAGE_1  MARRIAGE_2  \
25665  0.257059 -0.224533 -0.878738  0.114774  …         0.0         1.0
16464  0.257059 -0.173801  0.013770  0.114774  …         1.0         0.0
22386  1.999273  0.027750  1.798787  1.789147  …         0.0         1.0
10149 -1.485154 -0.270403 -1.771246 -1.559598  …         1.0         0.0
8729   0.257059 -0.217653  0.013770  0.114774  …         1.0         0.0

       MARRIAGE_3  EDUCATION_0  EDUCATION_1  EDUCATION_2  EDUCATION_3  \
25665         0.0          0.0          0.0          1.0          0.0
16464         0.0          0.0          0.0          0.0          1.0
22386         0.0          0.0          1.0          0.0          0.0
10149         0.0          0.0          0.0          1.0          0.0
8729          0.0          0.0          0.0          1.0          0.0

       EDUCATION_4  EDUCATION_5  EDUCATION_6
25665          0.0          0.0          0.0
16464          0.0          0.0          0.0
22386          0.0          0.0          0.0
10149          0.0          0.0          0.0
8729           0.0          0.0          0.0

[5 rows x 32 columns]
```

[37]:
```
pipe_lgbm.fit(X_train, y_train);
```

[38]:
```
lgbm_explainer = shap.TreeExplainer(pipe_lgbm.named_steps["lgbmclassifier"])
train_lgbm_shap_values = lgbm_explainer.shap_values(X_train_enc)
```

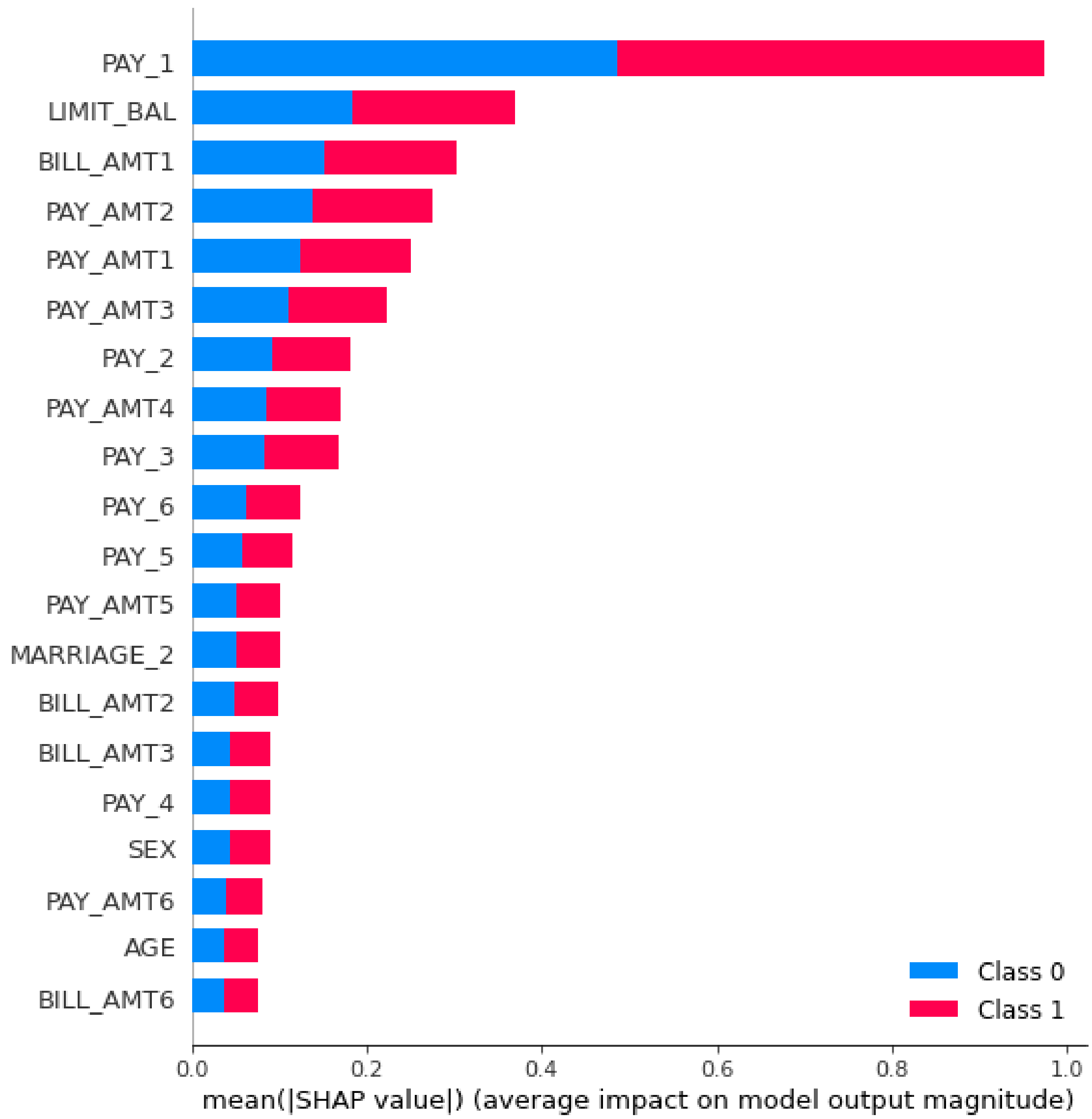
LightGBM binary classifier with TreeExplainer shap values output has changed to a list of ndarray

[39]:
```
# We are only extracting shapely values for the first 100 test examples for␣
↪speed.
test_lgbm_shap_values = lgbm_explainer.shap_values(X_test_enc[:100])
```

[40]:
```
shap.initjs()
```

<IPython.core.display.HTML object>

[41]:
```
shap.summary_plot(train_lgbm_shap_values, X_train_enc)
```



Summary of Observations: - The plot shows global feature importances, where the features are ranked in descending order of feature importances. - Colour shows the class of feature (red for default payment and blue for non-default payment) - `PAY_1` is likely the most important feature while `BILL_AMT6` is likely the least important one.

2.14 10. Results on the test set

We try your best performing model `LightGBM` on the test data and report test scores to answer following questions:
1. Do the test scores agree with the validation scores from before? 2. To what extent do we trust our results? 3. Is there any optimization bias? After that we take one or two test predictions and explain them with SHAP force plots.

Step 1

1. Try on test data and report test scores.

[42]:
```
best_lgbm = search_lgbm.best_estimator_
```

[43]:
```
best_lgbm.predict(X_test)
```

[43]:
```
array([0, 0, 1, …, 1, 1, 1])
```

[44]:
```
f1_score(y_test,best_lgbm.predict(X_test))
```

[44]:
```
0.5299528831052278
```

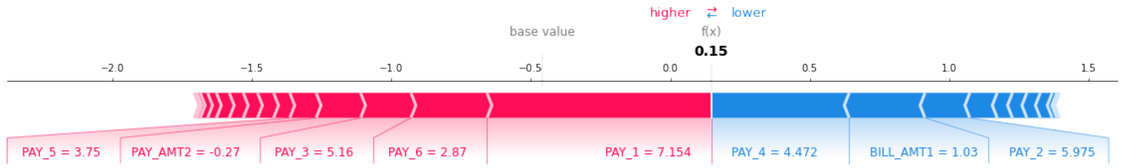
Answer questions The test score agrees with the validation score from `Section 8`. I would trust the result because the test score of 0.53 is just slightly lower than validation score of 0.546. Therefore, I think there's no issue with optimization bias in this case.

Step 2

2. Test predictions and explain with SHAP force plots.

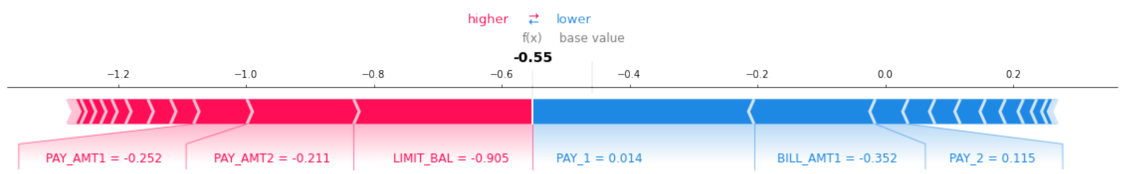
[45]:
```
X_train_enc = X_train_enc.round(3)
X_test_enc = X_test_enc.round(3)
```

[46]:
```
shap.force_plot(
    lgbm_explainer.expected_value[1],
    test_lgbm_shap_values[1][31,:],
    X_test_enc.iloc[31, :],
    matplotlib=True,
)
```



- The raw model score is higher than the base value so the prediction is default (1) because this example was pushed higher by all the factors shown in red such as `PAY_1`, `PAY_6`.
- Meanwhile, `PAY_4`, `BILL_AMT1` are pushing the prediction towards lower score.

[47]:
```
shap.force_plot(
    lgbm_explainer.expected_value[1],
    test_lgbm_shap_values[1][6,:],
    X_test_enc.iloc[6, :],
    matplotlib=True,
)
```



- The raw model score is lower than the base value so the prediction is non-default (0) because this example was pushed lower by all the factors shown in blue such as `PAY_1`, `BILL_AMT1`.
- Meanwhile, `LIMIT_BAL`, `PAY_AMT2` are pushing the prediction towards higher score.

2.15 11. Summary of results

Here is the summary of these results to our readers.

Summary Table

[48]:
```
models = {
    "Dummy": dummy_model,
    "Logistic Regression": pipe_lr,
    "Tuned Logistic Regression": search_lr.best_estimator_,
    "Random Forest": pipe_rf,
    "KNN": pipe_knn,
    "LGBM": pipe_lgbm,
```

```
    "Best LightGBM": best_lgbm
}
```

[51]:
```
important_scores = ["f1"]
final_result={}
for (name, model) in models.items():
    final_result[name] = mean_std_cross_val_scores(
        model, X_train, y_train,
        return_train_score=True,
        scoring=important_scores
    )

pd.DataFrame(final_result)
```

[51]:
```
                         Dummy Logistic Regression Tuned Logistic Regression  \
fit_time    0.003 (+/- 0.001)   0.269 (+/- 0.021)         0.065 (+/- 0.004)
score_time  0.002 (+/- 0.001)   0.005 (+/- 0.000)         0.005 (+/- 0.000)
test_f1     0.221 (+/- 0.016)   0.477 (+/- 0.009)         0.480 (+/- 0.009)
train_f1    0.225 (+/- 0.005)   0.480 (+/- 0.003)         0.481 (+/- 0.004)

                Random Forest                KNN               LGBM  \
fit_time    2.665 (+/- 0.024)  0.016 (+/- 0.003)  0.230 (+/- 0.054)
score_time  0.064 (+/- 0.001)  1.248 (+/- 0.121)  0.012 (+/- 0.001)
test_f1     0.455 (+/- 0.015)  0.434 (+/- 0.013)  0.539 (+/- 0.013)
train_f1    0.998 (+/- 0.000)  0.573 (+/- 0.003)  0.664 (+/- 0.004)

                Best LightGBM
fit_time    0.167 (+/- 0.076)
score_time  0.010 (+/- 0.001)
test_f1     0.546 (+/- 0.014)
train_f1    0.611 (+/- 0.005)
```

Concluding remarks: - Best and worst performing models: > With default hyperparameters for all models, the LGBM model seems to be performing best, whereas KNN seems to be performing worst. > With hyper parameters optimization, the best hyperparameters found by our random search for LGBM model are: num_leaves = 0.01129, max_depth=5, max_bin=240 with a validation f1-score of 0.546.

- Overfitting/underfitting: > Random Forest model seems to overfit; the training score is high and the gap between train and validation score f1 is big compared to other models. (Of course, our baseline model, dummy regressor, is also underfitting.) > All other models seem to underfit; the training score is low and the gap between train and validation score is not that big.
- Fit time > Random Forest model is much slower compared to other models. > KNN performs worst but it fits much faster than other models.
- Score time >Scoring is fast for almost all models except KNN.

- Stability of scores >The scores look more or less stable with std in the range 0.009 to 0.016 for f1 score.

Further development

Due to time limit, there are shortcomings in our mini project; hence the f1 score of the best model is not quite satisfactory. If we are able to try different models (such as SVC, SVM, tree models) and implement feature engineering such as polinomial, it is possible that we can improve the performance/interpretability of this project.

Result

TEST SCORE: 0.53, METRIC: F1

Takeaway

The biggest takeaway of our group from this project is: - To define and use evaluation metrics for classification and regression, - To learn the importance of feature engineering in building machine learning models. - To learn the importance of interpretability in Machine Learning.