hw6

December 5, 2021

1 CPSC 330 - Applied Machine Learning

1.1 Homework 6: Putting it all together

1.1.1 Associated lectures: All material till lecture 13

Due date: Monday, November 15, 2021 at 11:59pm

1.2 Table of contents

1.3 Imports

[1]:
```
import os

%matplotlib inline
import sys

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import xgboost as xgb
from sklearn.compose import ColumnTransformer, make_column_transformer
```

```python
from sklearn.dummy import DummyClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    classification_report,
    confusion_matrix,
    f1_score,
    make_scorer,
    plot_confusion_matrix,
)
from sklearn.model_selection import (
    GridSearchCV,
    RandomizedSearchCV,
    cross_val_score,
    cross_validate,
    train_test_split,
)
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder, StandardScaler
from sklearn.svm import SVC
```

[2]:
```python
# Custom imports
from pandas_profiling import ProfileReport
from sklearn.tree import DecisionTreeClassifier
from lightgbm.sklearn import LGBMClassifier
from sklearn.feature_selection import RFECV
import eli5
import shap
```

1.4 Instructions

rubric={points:2}

Follow the homework submission instructions.

You may work on this homework in a group and submit your assignment as a group. Below are some instructions on working as a group.
- The maximum group size is 3. - Use group work as an opportunity to collaborate and learn new things from each other. - Be respectful to each other and make sure you understand all the concepts in the assignment well. - It's your responsibility to make sure that the assignment is submitted by one of the group members before the deadline. - You can find the instructions on how to do group submission on Gradescope here.

1.5 Introduction

At this point we are at the end of supervised machine learning part of the course. So in this homework, you will be working on an open-ended mini-project, where you will put all the different things you have learned so far together to solve an interesting problem.

A few notes and tips when you work on this mini-project:

Tips

1. This mini-project is open-ended, and while working on it, there might be some situations where you'll have to use your own judgment and make your own decisions (as you would be doing when you work as a data scientist). Make sure you explain your decisions whenever necessary.
2. **Do not include everything you ever tried in your submission** – it's fine just to have your final code. That said, your code should be reproducible and well-documented. For example, if you chose your hyperparameters based on some hyperparameter optimization experiment, you should leave in the code for that experiment so that someone else could re-run it and obtain the same hyperparameters, rather than mysteriously just setting the hyperparameters to some (carefully chosen) values in your code.
3. If you realize that you are repeating a lot of code try to organize it in functions. Clear presentation of your code, experiments, and results is the key to be successful in this lab. You may use code from lecture notes or previous lab solutions with appropriate attributions.
4. If you are having trouble running models on your laptop because of the size of the dataset, you can create your train/test split in such a way that you have less data in the train split. If you end up doing this, please write a note to the grader in the submission explaining why you are doing it.

Assessment We plan to grade fairly and leniently. We don't have some secret target score that you need to achieve to get a good grade. **You'll be assessed on demonstration of mastery of course topics, clear presentation, and the quality of your analysis and results.** For example, if you just have a bunch of code and no text or figures, that's not good. If you do a bunch of sane things and get a lower accuracy than your friend, don't sweat it.

A final note Finally, this style of this "project" question is different from other assignments. It'll be up to you to decide when you're "done" – in fact, this is one of the hardest parts of real projects. But please don't spend WAY too much time on this… perhaps "a few hours" (2-8 hours???) is a good guideline for a typical submission. Of course if you're having fun you're welcome to spend as much time as you want! But, if so, try not to do it out of perfectionism or getting the best possible grade. Do it because you're learning and enjoying it. Students from the past cohorts have found such kind of labs useful and fun and I hope you enjoy it as well.

NB: Echoing the above, the goal of this notebook is really not to get a perfect score on the test set, but rather to demonstrate mastery of concepts in CPSC 330. Our focus is to show a good understanding on core ML fundamentals, but sometimes we make decisions we wouldn't probably do "in real life" to show mastery of course content. Where this occurs, we will comment and make this clear.

1.6 1. Understanding the problem

rubric={points:4}

In this mini project, you will be working on 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. The rest of the columns can be used as features. You may take some ideas and compare your results with the associated research paper, which is available through the UBC library.

Your tasks:

1. Spend some time understanding the problem and what each feature means. You can find this information in the documentation on the dataset page on Kaggle. Write a few sentences on your initial thoughts on the problem and the dataset.
2. Download the dataset and read it as a pandas dataframe.

Problem

The problem were given is one of binary classification. The goal is to, from an existing set of data about past credit card payments, predict whether or not a set of individuals will default on their next payment.

Columns

Here's the definitions of each of the available columns, verbatim from Kaggle. They're pasted here for ease-of-access. Additional comments and analysis are added as sub-points.

- `ID` ID of each client
  - This can serve as our index
- `LIMIT_BAL` Amount of given credit in NT dollars (includes individual and family/supplementary credit
- `SEX` Gender (1=male, 2=female)
  - It is probably worth dropping this feature; we don't want our model to predict based on sex as that's unethical
- `EDUCATION` (1=graduate school, 2=university, 3=high school, 4=others, 5=unknown, 6=unknown)
  - It turns out education also has 0; we'll want to consolidate 0, 5, and 6 to some common category
- `MARRIAGE` Marital status (1=married, 2=single, 3=others)
  - It turns out marriage also has a 0, so we'll put this in "others" as well
- `AGE` Age in years
- `PAY_0` Repayment status in September, 2005 (-1=pay duly, 1=payment delay for one month, 2=payment delay for two months, … 8=payment delay for eight months, 9=payment delay for nine months and above)
  - There's a mistake in the original write-up; per this comment on Kaggle, there's also -2, which indicates "no consumption" and 0 which is use of revolving credit
  - This should definitely be named `PAY1` to be consistent with the other columns
- `PAY_2` Repayment status in August, 2005 (scale same as above)
  - Rename to `PAY2` to be consistent, etc.
- `PAY_3` Repayment status in July, 2005 (scale same as above)
- `PAY_4` Repayment status in June, 2005 (scale same as above)
- `PAY_5` Repayment status in May, 2005 (scale same as above)
- `PAY_6` Repayment status in April, 2005 (scale same as above)
- `BILL_AMT1` Amount of bill statement in September, 2005 (NT dollar)
- `BILL_AMT2` Amount of bill statement in August, 2005 (NT dollar)

- `BILL_AMT3` Amount of bill statement in July, 2005 (NT dollar)
- `BILL_AMT4` Amount of bill statement in June, 2005 (NT dollar)
- `BILL_AMT5` Amount of bill statement in May, 2005 (NT dollar)
- `BILL_AMT6` Amount of bill statement in April, 2005 (NT dollar)
- `PAY_AMT1` Amount of previous payment in September, 2005 (NT dollar)
- `PAY_AMT2` Amount of previous payment in August, 2005 (NT dollar)
- `PAY_AMT3` Amount of previous payment in July, 2005 (NT dollar)
- `PAY_AMT4` Amount of previous payment in June, 2005 (NT dollar)
- `PAY_AMT5` Amount of previous payment in May, 2005 (NT dollar)
- `PAY_AMT6` Amount of previous payment in April, 2005 (NT dollar)
- `default.payment.next.month` Default payment (1=yes, 0=no)
  - This will be our target

[3]:
```
# Define a fixed random state to use for the entire notebook

RANDOM_STATE = 123
np.random.seed(RANDOM_STATE)
TODO = NotImplementedError()
```

[4]:
```
# Read in the data set

file = "UCI_Credit_Card.csv"

# Read in df; treat column `ID` as index since it's unique
credit_df = pd.read_csv(f"./{file}", index_col=0)
credit_df.head()
```

[4]:
```
    LIMIT_BAL  SEX  EDUCATION  MARRIAGE  AGE  PAY_0  PAY_2  PAY_3  PAY_4  \
ID
1     20000.0    2          2         1   24      2      2     -1     -1
2    120000.0    2          2         2   26     -1      2      0      0
3     90000.0    2          2         2   34      0      0      0      0
4     50000.0    2          2         1   37      0      0      0      0
5     50000.0    1          2         1   57     -1      0     -1      0

    PAY_5  …  BILL_AMT4  BILL_AMT5  BILL_AMT6  PAY_AMT1  PAY_AMT2  PAY_AMT3  \
ID         …
1      -2  …        0.0        0.0        0.0       0.0     689.0       0.0
2       0  …     3272.0     3455.0     3261.0       0.0    1000.0    1000.0
3       0  …    14331.0    14948.0    15549.0    1518.0    1500.0    1000.0
4       0  …    28314.0    28959.0    29547.0    2000.0    2019.0    1200.0
5       0  …    20940.0    19146.0    19131.0    2000.0   36681.0   10000.0

    PAY_AMT4  PAY_AMT5  PAY_AMT6  default.payment.next.month
ID
1        0.0       0.0       0.0                           1
2     1000.0       0.0    2000.0                           1
```

```
3      1000.0     1000.0     5000.0                           0
4      1100.0     1069.0     1000.0                           0
5      9000.0      689.0      679.0                           0

[5 rows x 24 columns]
```

Wrangling

Before we continue, let's do preliminary data wrangling. Note this is *before* we split the data into train and test sets, but it should be okay to do despite without violating the Golden Rule meaningfully. We will

- rename the columns to be more consistent in formatting
- group undefined values from certain columns (e.g., value 0 is marriage is meaningless, so group with 3)

The undefined values were found from EDA on the training data; we went back and decided to remove them from the get-go, since that's a bit cleaner and does not impact the validity of our test score.

[5]:
```
# Preliminary data wrangling

# Clean-up the column names
credit_df = credit_df.rename(columns={
    "PAY_0": "PAY1",
    "PAY_2": "PAY2",
    "PAY_3": "PAY3",
    "PAY_4": "PAY4",
    "PAY_5": "PAY5",
    "PAY_6": "PAY6",
    "default.payment.next.month": "DEFAULT"
})


# Remove "bad" values from marriage, education (https://piazza.com/class/
 ↪kt60nrdhu53454?cid=348)
# This was caught in EDA; we went back and removed it from ALL data (including␣
 ↪test data)
credit_df["MARRIAGE"] = credit_df["MARRIAGE"].replace([0], 3)
credit_df["EDUCATION"] = credit_df["EDUCATION"].replace([0, 6], 5)

credit_df.head()
```

[5]:
```
    LIMIT_BAL  SEX  EDUCATION  MARRIAGE  AGE  PAY1  PAY2  PAY3  PAY4  PAY5  \
ID
1     20000.0    2          2         1   24     2     2    -1    -1    -2
2    120000.0    2          2         2   26    -1     2     0     0     0
3     90000.0    2          2         2   34     0     0     0     0     0
4     50000.0    2          2         1   37     0     0     0     0     0
```

```
5     50000.0    1          2         1   57    -1     0    -1     0     0

    …  BILL_AMT4  BILL_AMT5  BILL_AMT6  PAY_AMT1  PAY_AMT2  PAY_AMT3  \
ID  …
1   …        0.0        0.0        0.0       0.0     689.0       0.0
2   …     3272.0     3455.0     3261.0       0.0    1000.0    1000.0
3   …    14331.0    14948.0    15549.0    1518.0    1500.0    1000.0
4   …    28314.0    28959.0    29547.0    2000.0    2019.0    1200.0
5   …    20940.0    19146.0    19131.0    2000.0   36681.0   10000.0

    PAY_AMT4  PAY_AMT5  PAY_AMT6  DEFAULT
ID
1        0.0       0.0       0.0        1
2     1000.0       0.0    2000.0        1
3     1000.0    1000.0    5000.0        0
4     1100.0    1069.0    1000.0        0
5     9000.0     689.0     679.0        0

[5 rows x 24 columns]
```

1.7 2. Data splitting

rubric={points:2}

Your tasks:

1. Split the data into train and test portions.

Let's create our train and test splits, after reading in the data frame.

[6]:
```
# See how many rows and columns are present in the original data to inform split

credit_df.shape
```

[6]:
```
(30000, 24)
```

[7]:
```
# Create X, y

target = "DEFAULT"
X = credit_df.drop(target, axis=1)
y = credit_df[target]
```

[8]:
```
# Split our data into a training set and a testing set

test_size = 0.25
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size,␣
 ↪random_state=RANDOM_STATE)

display(X_train.head())
display(pd.DataFrame(y_train).head())
```

```
       LIMIT_BAL  SEX  EDUCATION  MARRIAGE  AGE  PAY1  PAY2  PAY3  PAY4  PAY5  \
ID
16096   140000.0    2          2         1   36     1     2     3     2     0
28549   210000.0    2          2         2   33     0     0     0    -2    -2
25097    20000.0    1          3         2   53    -1     0    -1    -1    -1
12261    90000.0    2          2         2   23     2     4     4     3     4
21550    50000.0    2          3         2   22    -2    -2    -2    -2    -2

       …  BILL_AMT3  BILL_AMT4  BILL_AMT5  BILL_AMT6  PAY_AMT1  PAY_AMT2  \
ID     …
16096  …    61459.0    59798.0    61287.0     8383.0    5200.0       0.0
28549  …        0.0        0.0        0.0        0.0    1000.0       0.0
25097  …      390.0    18280.0     2880.0     1600.0    1105.0     390.0
12261  …    37825.0    40299.0    39093.0    38167.0    2000.0       0.0
21550  …     1697.0        0.0        0.0     5000.0       0.0    1699.0

       PAY_AMT3  PAY_AMT4  PAY_AMT5  PAY_AMT6
ID
16096       0.0    3009.0    1000.0   94000.0
28549       0.0       0.0       0.0       0.0
25097   18280.0    2880.0    1600.0       0.0
12261    3400.0       0.0       0.0    1000.0
21550       0.0       0.0    5000.0       0.0

[5 rows x 23 columns]

       DEFAULT
ID
16096        0
28549        0
25097        1
12261        0
21550        0
```

We're now finished creating the needed data frames for our analysis. In creating our model, we'll use only `X_train` and `y_train`. Once our final model is created, we'll score it again `X_test` and `y_test`.

1.8 3. EDA

rubric={points:10}

Your tasks:

1. Perform exploratory data analysis on the train set.
2. Include at least two summary statistics and two visualizations that you find useful, and accompany each one with a sentence explaining it.
3. Summarize your initial observations about the data.

4. Pick appropriate metric/metrics for assessment.

```
[9]: # Let's begin by generating a profile report courtesy of pandas_profiling

     profile = ProfileReport(X_train, minimal=True)
     profile.to_widgets()

     # Export summary to HTML file
     profile.to_file("credit_df.html")
```

```
Summarize dataset:   0%|          | 0/32 [00:00<?, ?it/s]

Generate report structure:   0%|          | 0/1 [00:00<?, ?it/s]

Render widgets:   0%|          | 0/1 [00:00<?, ?it/s]

VBox(children=(Tab(children=(Tab(children=(GridBox(children=(VBox(children=(GridspecLayout(chi

Render HTML:   0%|          | 0/1 [00:00<?, ?it/s]

Export report to file:   0%|          | 0/1 [00:00<?, ?it/s]
```

```
[10]: # Generate basic summary statistics

      X_train.describe()
```

```
[10]:          LIMIT_BAL           SEX     EDUCATION      MARRIAGE           AGE  \
      count     22500.000000  22500.000000  22500.000000  22500.000000  22500.000000
      mean     167912.608000      1.601689      1.852533      1.558400     35.477867
      std      130144.258317      0.489561      0.784837      0.520791      9.206516
      min       10000.000000      1.000000      1.000000      1.000000     21.000000
      25%       50000.000000      1.000000      1.000000      1.000000     28.000000
      50%      140000.000000      2.000000      2.000000      2.000000     34.000000
      75%      240000.000000      2.000000      2.000000      2.000000     41.000000
      max     1000000.000000      2.000000      5.000000      3.000000     79.000000

                     PAY1          PAY2          PAY3         PAY4          PAY5  \
      count  22500.000000  22500.000000  22500.000000  22500.00000  22500.000000
      mean      -0.017867     -0.137111     -0.172133     -0.22520     -0.265378
      std        1.120041      1.195946      1.196880      1.16945      1.137946
      min       -2.000000     -2.000000     -2.000000     -2.00000     -2.000000
      25%       -1.000000     -1.000000     -1.000000     -1.00000     -1.000000
      50%        0.000000      0.000000      0.000000      0.00000      0.000000
      75%        0.000000      0.000000      0.000000      0.00000      0.000000
      max        8.000000      8.000000      8.000000      8.00000      8.000000

             …     BILL_AMT3      BILL_AMT4      BILL_AMT5      BILL_AMT6  \
      count  …  22500.000000   22500.000000   22500.000000    22500.00000
      mean   …  46961.377200   43389.790578   40363.350933    38763.81320
      std    …  68802.264441   64599.545694   61009.524485    59457.26977
```

```
min    … -157264.000000  -65167.000000  -61372.000000 -339603.00000
25%    …    2680.750000    2293.750000    1727.750000    1200.00000
50%    …   20089.000000   19067.500000   18043.500000   16855.00000
75%    …   60016.250000   54603.000000   50398.500000   49276.25000
max    …  855086.000000  891586.000000  927171.000000  961664.00000

            PAY_AMT1      PAY_AMT2       PAY_AMT3       PAY_AMT4  \
count   22500.000000  2.250000e+04   22500.000000   22500.000000
mean     5673.348800  5.926811e+03    5287.000667    4785.724978
std     16916.734372  2.151420e+04   18146.139695   15301.390285
min         0.000000  0.000000e+00       0.000000       0.000000
25%      1000.000000  8.240000e+02     390.000000     264.750000
50%      2100.000000  2.011000e+03    1812.000000    1500.000000
75%      5011.000000  5.000000e+03    4625.250000    4030.750000
max    873552.000000  1.227082e+06  896040.000000  621000.000000

            PAY_AMT5       PAY_AMT6
count   22500.000000   22500.000000
mean     4806.159911    5261.697022
std     15316.094460   18162.612241
min         0.000000       0.000000
25%       234.000000     113.750000
50%      1500.000000    1500.000000
75%      4012.000000    4000.000000
max    426529.000000  528666.000000

[8 rows x 23 columns]
```

[11]:
```
# More summary statistics

X_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 22500 entries, 16096 to 19967
Data columns (total 23 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   LIMIT_BAL  22500 non-null  float64
 1   SEX        22500 non-null  int64
 2   EDUCATION  22500 non-null  int64
 3   MARRIAGE   22500 non-null  int64
 4   AGE        22500 non-null  int64
 5   PAY1       22500 non-null  int64
 6   PAY2       22500 non-null  int64
 7   PAY3       22500 non-null  int64
 8   PAY4       22500 non-null  int64
 9   PAY5       22500 non-null  int64
 10  PAY6       22500 non-null  int64
 11  BILL_AMT1  22500 non-null  float64
 12  BILL_AMT2  22500 non-null  float64
 13  BILL_AMT3  22500 non-null  float64
 14  BILL_AMT4  22500 non-null  float64
 15  BILL_AMT5  22500 non-null  float64
 16  BILL_AMT6  22500 non-null  float64
 17  PAY_AMT1   22500 non-null  float64
 18  PAY_AMT2   22500 non-null  float64
 19  PAY_AMT3   22500 non-null  float64
 20  PAY_AMT4   22500 non-null  float64
 21  PAY_AMT5   22500 non-null  float64
 22  PAY_AMT6   22500 non-null  float64
dtypes: float64(13), int64(10)
memory usage: 4.1 MB
```

[12]:
```
# Pertinent summary statistics
#  - The observation of the individual with the highest limit balance (#2198)

pd.DataFrame(X_train.loc[X_train["LIMIT_BAL"].idxmax()])
```

[12]:
```
                2198
LIMIT_BAL  1000000.0
SEX              2.0
EDUCATION        1.0
MARRIAGE         1.0
AGE             47.0
PAY1             0.0
PAY2             0.0
PAY3             0.0
PAY4            -1.0
PAY5             0.0
PAY6             0.0
BILL_AMT1   964511.0
BILL_AMT2   983931.0
BILL_AMT3   535020.0
BILL_AMT4   891586.0
BILL_AMT5   927171.0
BILL_AMT6   961664.0
PAY_AMT1     50784.0
PAY_AMT2     50723.0
PAY_AMT3    896040.0
PAY_AMT4     50000.0
PAY_AMT5     50000.0
PAY_AMT6     50256.0
```

```
[13]: #  - The average age
      display(X_train["AGE"].mean())
      display(X_train["AGE"].median())
```

```
35.477866666666664

34.0
```

```
[14]: # It might help to see value counts of MARRIAGE, EDUCATION, and PAY_1 to ensure␣
       ↪the values that were missing from our legend are indeed gone

      X_train["MARRIAGE"].value_counts()
```

```
[14]: 2    12010
      1    10213
      3      277
      Name: MARRIAGE, dtype: int64
```

```
[15]: X_train["EDUCATION"].value_counts()
```

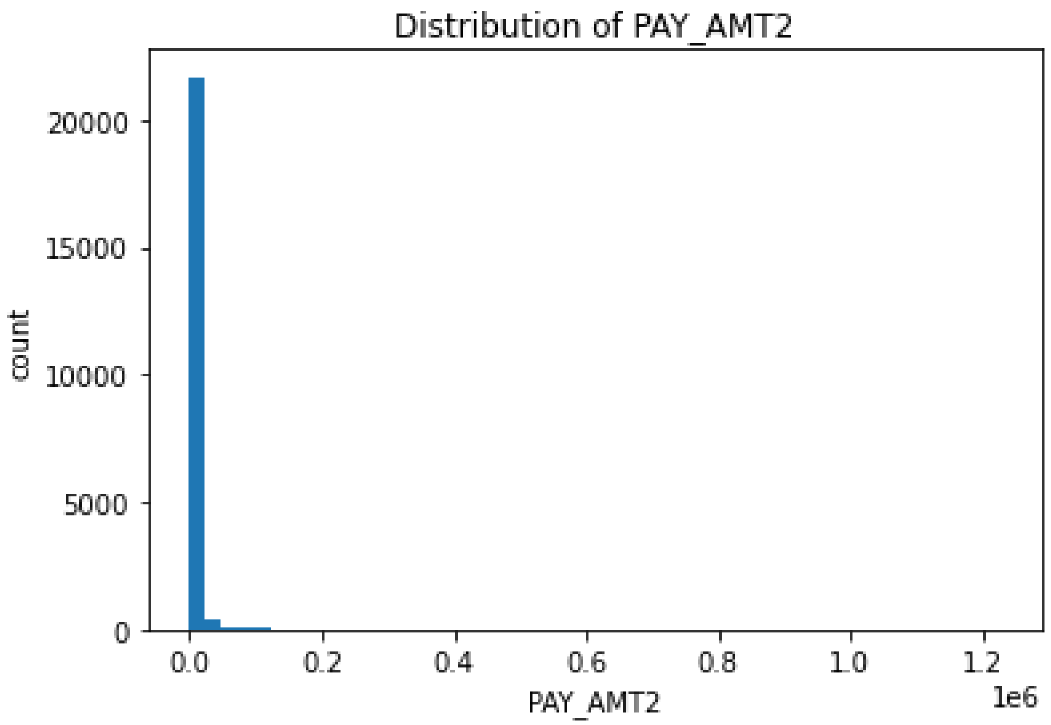
```
[15]: 2    10521
      1     7955
      3     3676
      5      265
      4       83
      Name: EDUCATION, dtype: int64
```

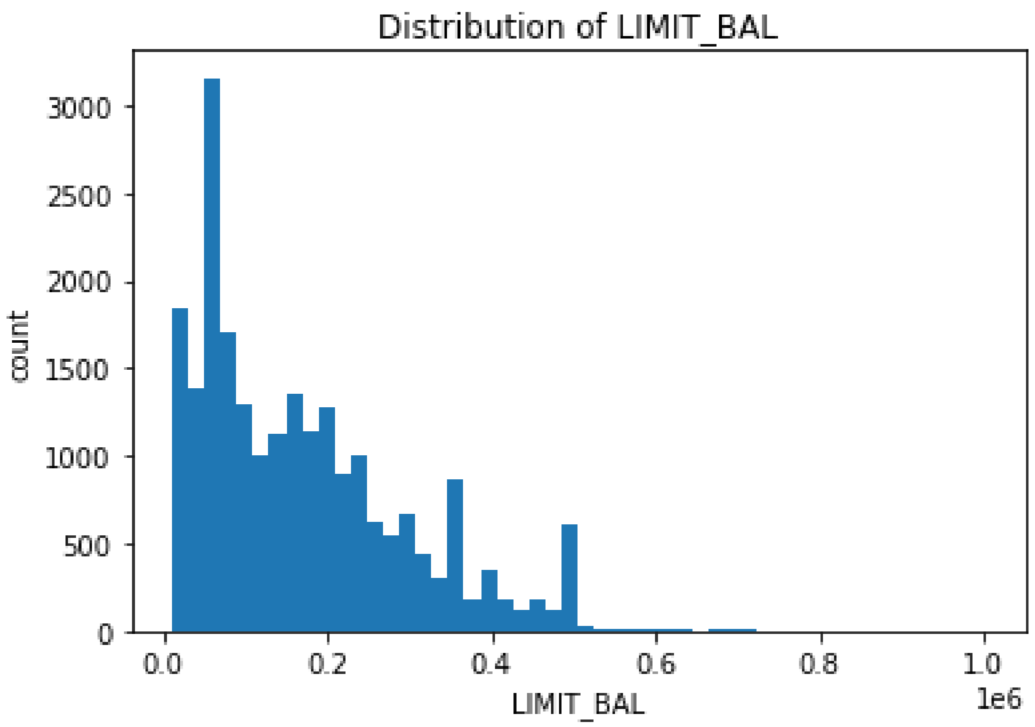
```
[16]: X_train["PAY1"].value_counts()
```

```
[16]:  0    11065
      -1     4313
       1     2755
      -2     2036
       2     1980
       3      247
       4       55
       5       20
       8       12
       7        9
       6        8
      Name: PAY1, dtype: int64
```

```
[17]: # Pertinent visualizations
      #  - According to PandasProfilier, PAY_AMT2 is highly skewed; let's take a look!
      #  - LIMIT_BAL is very pertinent to our analysis, so let's also understand it's␣
       ↪distribution

      # Code adapted from HW2
```

```python
features = ["PAY_AMT2", "LIMIT_BAL"]

for feature in features:
    plt.hist(X_train[feature], alpha=1, bins=50)

    plt.xlabel(feature)
    plt.ylabel("count")
    plt.title(f"Distribution of {feature}")
    plt.show()
```

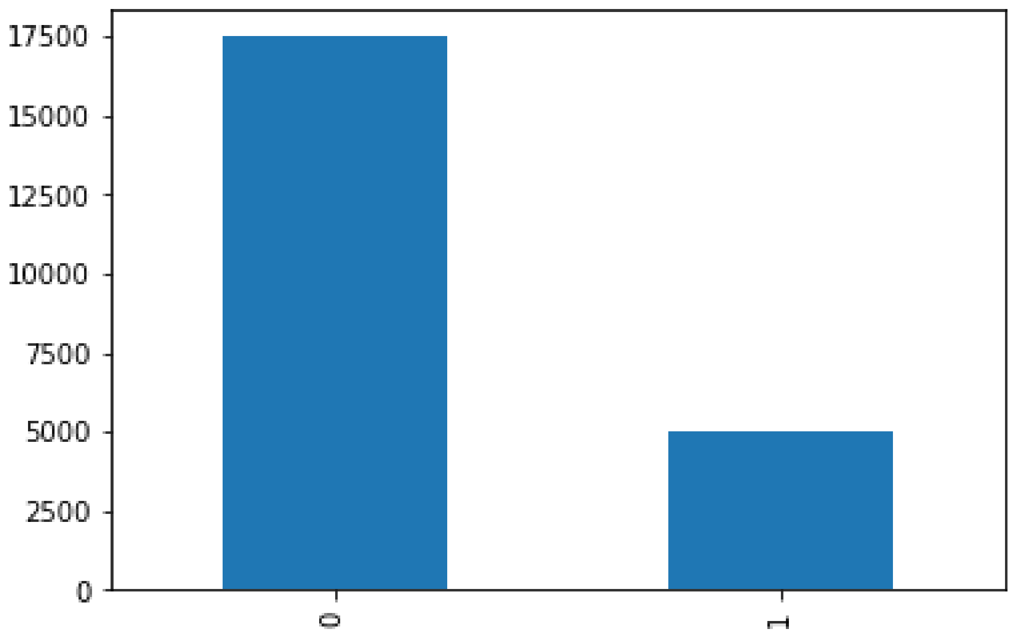




Let's also look at our class imbalance. Note the following plot does not have a title and is otherwise not well-formed, but what it represents should be fairly obvious despite. We're counting the number of "1"s and "0"s in our training data, to get a sense of any imbalance.

[18]:
```
# It's definitely helpful to visualize our class imbalance as well

y_train.value_counts().plot(kind = "bar")
```

[18]:
```
<AxesSubplot:>
```



Exploration

We've now completed our exploration of this data set. In our exploration, we showed numerous summary statistics and visualized the distributions of two key features.

Looking at the customer with the highest credit, we can see repeatedly huge `BILL_AMT` values with correspondingly small `PAY_AMT`s. Our intuition tells us this kind of behaviour leads to a credit default. The average age in our data is ~35 and the median is 34. This gives us a rough sense of the age range we are working with; it seems most rows are "middle aged" individuals.

Our value counts show the undefined values from `MARRIAGE`, `EDUCATION` were successfully re-grouped as desired. Again, we did this *before* splitting our data, but since it won't meaningfully impact our results, it's okay. Taking the step before the split greatly simplifies our code, which is valuable for code maintenance.

The distribution of `PAY_AMT2` is highly skewed, as suggested by `pandas_profiler`. It seems almost all values are 0 or close to 0 for this month. I wonder why bill repayments were so low in the month previous to August 2005? Perhaps some understanding of events within that time frame could help. The distribution for `LIMIT_BAL` is also skewed to the left, but less dramatically so. Most people have smaller balances. This seems reasonable given our intuition about wealth disparity in the 21st century.

Finally, we can see we have a bit of class imbalance in this dataset; there is an approximately 3-to-1 ratio of 0s to 1s, meaning there are many more examples of individuals not defaulting versus defaulting. This will motivate our chosen metric.

Metrics

The problem we're working with here is one of binary classification. This leads to a natural question—is accuracy the best metric?

To come at this question, we'll assume that the goal of this analysis is to predict when clients will default ahead of time, thus avoiding the default in the first place. Therefore, we want to minimize instances where we miss a default that would've occurred. (It should be safe to assume having extra "will-default" predictions is okay.)

Since "will-default" is our positive class, missing a "will-default" is considered a "false negative" (i.e., the absence of a condition when it is present). Given that we want to minimize false negatives, we should focus on the recall metric to see the best performance.

As a final caveat though, we'll be performing automatic hyperparameter optimization at numerous points in this analysis. Therefore, instead of letting various models automatically tune on recall, we'll take a more measured approach and use the F1-score instead. (Otherwise, we'll get an abysmal precision score; a harmonic mean is a better, more balanced approach.) This is also in lieu of an operating point, which would probably come from a hypothetical CEO or team lead.

1.9 (Optional) 4. Feature engineering

rubric={points:1}

Your tasks:

1. Carry out feature engineering. In other words, extract new features relevant for the problem and work with your new feature set in the following exercises. You may have to go back and forth between feature engineering and preprocessing.

The goal is feature engineering is to use domain-specific knowledge, existing literature, and cross-validation to device more effective features for a given problem, that are of course not already present in the data set.

The paper linked at the start of the notebook evaluates a number of different "data mining techniques" for this binary classification problem. Crucially, in the paper, the authors *do not* perform any form of feature engineering, so there's nothing we can directly mimic. However, he authors conclude that artificial neural networks (ANNs) outperform all other models, which is interesting, and can provides some benchmarks we can refer to later.

The question here does not specify how many features should be created nor how we should evaluate them. As a rough 'guess,' let's see if adding a feature that assesses the relative gap between the amount of the bill paid in one month and the amount of bill "generated" in one month.

Perhaps customers that have a consistently decreasing this gap are unlikely to default, and vice versa. We'd need to do a pairwise comparison between `BILL_AMT` and `PAY_AMT` for each month `1...6`. Our model, with only access to the absolute values, may miss out on the trends in the relative 'spacing' between these values.

We define a helper function that computes this column, and apply it to both data frames. Again, this is okay to do to `X_test` because it does not artificially enhance our test score in some way.

[19]:
```python
def relative(x1, x2):
    """
    Compute the relative difference between columns.
    """
    if x1 == 0 and x2 == 0:
        return 0 # 0% change
    if x1 == 0:
        return 1 # 100% change
    return abs(x2 - x1) / x1
```

[20]:
```
# Visualize the newly added columns on train data
X_train_extended = X_train.copy()
X_test_extended = X_test.copy()

amts = [1, 2, 3, 4, 5, 6]
for amt in amts:
    X_train_extended.loc[:, f"BILL_PAY_RATIO{amt}"] = X_train_extended.
 ↪apply(lambda row: relative(row[f"BILL_AMT{amt}"], row[f"PAY_AMT{amt}"]),␣
 ↪axis = 1)
    X_test_extended.loc[:, f"BILL_PAY_RATIO{amt}"]  = X_test_extended.
 ↪apply(lambda row: relative(row[f"BILL_AMT{amt}"], row[f"PAY_AMT{amt}"]),␣
 ↪axis = 1)

X_train_extended[["BILL_PAY_RATIO1", "BILL_PAY_RATIO2", "BILL_PAY_RATIO3",␣
 ↪"BILL_PAY_RATIO4", "BILL_PAY_RATIO5", "BILL_PAY_RATIO6"]].head()
```

[20]:
```
       BILL_PAY_RATIO1  BILL_PAY_RATIO2  BILL_PAY_RATIO3  BILL_PAY_RATIO4  \
ID
16096         0.912427         1.000000         1.000000         0.949681
28549         0.976770         1.000000         0.000000         0.000000
25097         0.918516         0.970115        45.871795         0.842451
12261         0.946864         1.000000         0.910112         1.000000
21550         1.000000      -850.500000         1.000000         0.000000

       BILL_PAY_RATIO5  BILL_PAY_RATIO6
ID
16096         0.983683        10.213170
28549         0.000000         0.000000
25097         0.444444         1.000000
12261         1.000000         0.973799
21550         1.000000         1.000000
```

1.10 5. Preprocessing and transformations

rubric={points:10}

Your tasks:

1. Identify different feature types and the transformations you would apply on each feature type.
2. Define a column transformer, if necessary.

We don't need to impute our data at all, we just need to encode or scale. The features `LIMIT_BAL`, `AGE`, `BILL_AMT1...6`, `PAY_AMT1...6`, and `BILL_PAY_RATIO1...6` should be scaled, as they are all numeric. Sex, education, marital status, and `PAY1...6` should all be encoded somehow. (Upon further reflection though, it's like more ethical to drop sex entirely.) We can almost apply an ordinal encoding to `education`, but the "others" category makes this impossible (could be more or less than what's given). We cannot apply an ordinal encoding to `PAY1...6` either since there is no way to rank some of the categories; we cannot tell whether value -2 (no consumption) is better or worse than value -1 (pay duly) or value 0 (use of rotating credit) or how these should compare with payment delay values 1 through 9. We will therefore apply a one-hot encoding to the all the categorical features.

We now define our column transformer.

[21]:
```
feats = X_train_extended.columns.tolist()

# Apply scaling to numeric features
numeric_feats = [
 'LIMIT_BAL',
 'AGE',
 'BILL_AMT1',
 'BILL_AMT2',
 'BILL_AMT3',
 'BILL_AMT4',
 'BILL_AMT5',
 'BILL_AMT6',
 'PAY_AMT1',
 'PAY_AMT2',
 'PAY_AMT3',
 'PAY_AMT4',
 'PAY_AMT5',
 'PAY_AMT6',
 'BILL_PAY_RATIO1',
 'BILL_PAY_RATIO2',
 'BILL_PAY_RATIO3',
 'BILL_PAY_RATIO4',
 'BILL_PAY_RATIO5',
 'BILL_PAY_RATIO6'
]

# Apply a one-hot encoding
categorical_feats = [
 'EDUCATION',
 'MARRIAGE',
 'PAY1',
 'PAY2',
 'PAY3',
 'PAY4',
 'PAY5',
 'PAY6'
]
```

```
# Remove sex; unethical to consider
drop_feats = [
 'SEX',
]

assert(len(feats) == len(numeric_feats + categorical_feats + drop_feats))
```

[22]:
```
ct = ColumnTransformer(
    [
        ("scaling", StandardScaler(), numeric_feats),
        ("onehot",  OneHotEncoder(sparse=False, handle_unknown='ignore'),␣
 ↪categorical_feats),
        ("drop", "drop", drop_feats),
    ],
)

# ct
```

[23]:
```
X_train_transformed = ct.fit_transform(X_train_extended)
```

[24]:
```
column_names = (
    numeric_feats
    # Passing in categorical_feats gives our OHE nice names :-)
    + ct.named_transformers_["onehot"].get_feature_names(categorical_feats).
 ↪tolist()
)

# column_names
```

[25]:
```
pd.DataFrame(X_train_transformed, columns=column_names).head()
```

[25]:
```
   LIMIT_BAL       AGE  BILL_AMT1  BILL_AMT2  BILL_AMT3  BILL_AMT4  BILL_AMT5  \
0  -0.214479  0.056715   0.113698   0.196343   0.210719   0.254004   0.342965
1   0.323397 -0.269149  -0.109391  -0.557621  -0.682571  -0.671688  -0.661606
2  -1.136553  1.903274  -0.512158  -0.505997  -0.676902  -0.388708  -0.614399
3  -0.598677 -1.355360  -0.183262  -0.144782  -0.132795  -0.047846  -0.020823
4  -0.906035 -1.463981  -0.663274  -0.689493  -0.657905  -0.671688  -0.661606

   BILL_AMT6  PAY_AMT1  PAY_AMT2  …  PAY6_-2  PAY6_-1  PAY6_0  PAY6_2  \
0  -0.510980 -0.027982 -0.275490  …      0.0      0.0     1.0     0.0
1  -0.651975 -0.276262 -0.275490  …      1.0      0.0     0.0     0.0
2  -0.625065 -0.270055 -0.257362  …      0.0      1.0     0.0     0.0
3  -0.010038 -0.217148 -0.275490  …      0.0      0.0     0.0     0.0
4  -0.567879 -0.335376 -0.196517  …      1.0      0.0     0.0     0.0

   PAY6_3  PAY6_4  PAY6_5  PAY6_6  PAY6_7  PAY6_8
0     0.0     0.0     0.0     0.0     0.0     0.0
```

```
1     0.0     0.0     0.0     0.0     0.0     0.0
2     0.0     0.0     0.0     0.0     0.0     0.0
3     1.0     0.0     0.0     0.0     0.0     0.0
4     0.0     0.0     0.0     0.0     0.0     0.0

[5 rows x 92 columns]
```

1.11 6. Baseline model

rubric={points:2}

Your tasks: 1. Try `scikit-learn`'s baseline model and report results.

[26]:
```
# Adapted from CPSC 330, but modified to break mean, std into separate columns
def mean_std_cross_val_scores(model, X_train, y_train, name=None, raw=False,␣
↪flatten=False, **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 DataFrame with mean and std scores from cross_validation
    """
    # Could pass in cv as kwargs; defaults to 5
    # Always return train score
    kwargs["return_train_score"] = True

    scoring = [
        "accuracy",
        "f1",
        "recall",
    ]
    scores = cross_validate(model, X_train, y_train, scoring=scoring, **kwargs)

    mean_scores = pd.DataFrame(scores).mean()
    std_scores = pd.DataFrame(scores).std()

    mean_col = []
    std_col = []
    for mean, std in zip(mean_scores, std_scores):
        mean_col.append(float(f"%0.3f" % mean))
        std_col.append(float(f"%0.3f" % std))

    data = {}
    if flatten:
        # Having the data "flat" will be useful in hyperparameter tuning
        data = {
            "mean_fit_time"       : mean_col[0],
            "std_fit_time"        : std_col[0],
            "mean_score_time"     : mean_col[1],
            "std_score_time"      : std_col[1],
            "mean_test_accuracy"  : mean_col[2],
            "std_test_accuracy"   : std_col[2],
            "mean_train_accuracy" : mean_col[3],
            "std_train_accuracy"  : std_col[3],
            "mean_test_f1"        : mean_col[4],
            "std_test_f1"         : std_col[4],
            "mean_train_f1"       : mean_col[5],
            "std_train_f1"        : std_col[5],
            "mean_test_recall"    : mean_col[6],
            "std_test_recall"     : std_col[6],
            "mean_train_recall"   : mean_col[7],
            "std_train_recall"    : std_col[7],
        }
    else:
        # This is easier to read in a table
        data = {
            "mean": mean_col,
            "std" : std_col
        }

    if raw:
        # Don't convert to data frame
        return data
    else:
        return pd.DataFrame(data=data, index=[name or 0] if flatten else␣
↪mean_scores.index)
```

```
[27]: def create_blank_data():
    """
    Create a starter frame for doing hyperparameter optimization.
    """
    return {
            "mean_fit_time"      : list(),
            "std_fit_time"       : list(),
            "mean_score_time"    : list(),
            "std_score_time"     : list(),
            "mean_test_accuracy" : list(),
            "std_test_accuracy"  : list(),
            "mean_train_accuracy": list(),
            "std_train_accuracy" : list(),
            "mean_test_f1"       : list(),
            "std_test_f1"        : list(),
            "mean_train_f1"      : list(),
            "std_train_f1"       : list(),
            "mean_test_recall"   : list(),
            "std_test_recall"    : list(),
            "mean_train_recall"  : list(),
            "std_train_recall"   : list(),
        }
```

[28]:
```
dc = DummyClassifier()
pipe_dc = make_pipeline(ct, dc)

dc_results = mean_std_cross_val_scores(pipe_dc, X_train_extended, y_train)
dc_df = pd.DataFrame(dc_results)
display(dc_df)
```

```
                 mean    std
fit_time        0.059  0.007
score_time      0.022  0.004
test_accuracy   0.777  0.000
train_accuracy  0.777  0.000
test_f1         0.000  0.000
train_f1        0.000  0.000
test_recall     0.000  0.000
train_recall    0.000  0.000
```

[29]:
```
# Example of a flattened result dataframe
dc_results = mean_std_cross_val_scores(pipe_dc, X_train_extended, y_train,␣
 ↪name="Default", flatten=True)
dc_df_flat = pd.DataFrame(dc_results)
display(dc_df_flat)
```

```
         mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
Default          0.054         0.008             0.02           0.003

         mean_test_accuracy  std_test_accuracy  mean_train_accuracy  \
Default               0.777                0.0                0.777

         std_train_accuracy  mean_test_f1  std_test_f1  mean_train_f1  \
Default                 0.0           0.0          0.0            0.0
```

```
         std_train_f1  mean_test_recall  std_test_recall  mean_train_recall  \
Default           0.0               0.0              0.0                0.0

         std_train_recall
Default               0.0
```

This will serve as our baseline for this prediction problem. Note this score reflects the class imbalance.

1.12 7. Linear models

rubric={points:12}

Your tasks:

1. Try logistic regression as a first real attempt.
2. Carry out hyperparameter tuning to explore different values for the complexity hyperparameter C.
3. Report validation scores along with standard deviation.
4. Summarize your results.

[30]:
```
lr_scores_dict = {
    "C": 10.0 ** np.arange(-4, 6, 1),
} | create_blank_data()

for C in lr_scores_dict["C"]:
    # Create pipeline
    lr = LogisticRegression(C=C, max_iter=10000)
    pipe_lr = make_pipeline(ct, lr)

    lr_result = mean_std_cross_val_scores(pipe_lr, X_train_extended, y_train,␣
 ↪name=f"C={C}", raw=True, flatten=True)
    for k, v in lr_result.items():
        lr_scores_dict[k].append(v)
```

[31]:
```
lr_result_df = pd.DataFrame(data=lr_scores_dict)
lr_result_df
```

[31]:
```
            C  mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0      0.0001          0.070         0.007            0.015           0.001
1      0.0010          0.089         0.002            0.015           0.001
2      0.0100          0.154         0.013            0.016           0.001
3      0.1000          0.371         0.066            0.019           0.003
4      1.0000          0.692         0.074            0.017           0.007
5     10.0000          1.319         0.150            0.015           0.002
6    100.0000          2.487         0.532            0.016           0.001
7   1000.0000          3.569         0.586            0.018           0.003
8  10000.0000          2.928         0.845            0.020           0.005
```

```
9  100000.0000          3.312         0.822            0.015            0.001

   mean_test_accuracy  std_test_accuracy  mean_train_accuracy  \
0               0.777              0.000                0.777
1               0.799              0.003                0.800
2               0.815              0.003                0.816
3               0.819              0.005                0.821
4               0.819              0.005                0.822
5               0.820              0.005                0.822
6               0.820              0.005                0.822
7               0.820              0.005                0.822
8               0.820              0.005                0.822
9               0.820              0.005                0.822

   std_train_accuracy  mean_test_f1  std_test_f1  mean_train_f1  std_train_f1  \
0               0.000         0.000        0.000          0.000         0.000
1               0.001         0.268        0.016          0.269         0.008
2               0.001         0.426        0.013          0.431         0.009
3               0.002         0.464        0.019          0.469         0.007
4               0.002         0.465        0.019          0.472         0.006
5               0.001         0.466        0.019          0.473         0.006
6               0.001         0.465        0.020          0.473         0.006
7               0.001         0.465        0.020          0.473         0.006
8               0.001         0.466        0.020          0.473         0.006
9               0.001         0.465        0.020          0.473         0.006

   mean_test_recall  std_test_recall  mean_train_recall  std_train_recall
0             0.000            0.000              0.000             0.000
1             0.165            0.011              0.166             0.005
2             0.309            0.012              0.312             0.009
3             0.352            0.018              0.355             0.007
4             0.352            0.018              0.358             0.006
5             0.353            0.018              0.359             0.006
6             0.353            0.019              0.358             0.006
7             0.353            0.019              0.358             0.006
8             0.353            0.019              0.358             0.006
9             0.353            0.019              0.359             0.006
```

The columns in the table we're most interested in are `mean_test_f1` and `mean_test_recall`. These are the validation scores we're interested in.

We see the scores plateau at about `C` equals 0.1 or 1; I'd probably lean to 0.1 since simpler models *generally* are more effective on unseen data. This achieves a test accuracy of 0.819, a test F1 of 0.464, and a test recall of 0.352. Hopefully we can improve on these scores later on!

1.13 8. Different classifiers

rubric={points:15}

Your tasks: 1. Try at least 3 other models aside from logistic regression. At least one of these models should be a tree-based ensemble model (e.g., lgbm, random forest, xgboost). 2. Summarize your results. Can you beat logistic regression?

1.13.1 Decision Tree

For a first attempt, let's try a basic decision tree.

```
[32]: dt_scores_dict = {
          "max_depth": [1] + list(range(0, 50, 5))[1:] + [None],
      } | create_blank_data()

      for max_depth in dt_scores_dict["max_depth"]:
          # Create pipeline
          dt = DecisionTreeClassifier(max_depth=max_depth)
          pipe_dt = make_pipeline(ct, dt)

          dt_result = mean_std_cross_val_scores(pipe_dt, X_train_extended, y_train,␣
       ↪name=f"max_depth={max_depth}", raw=True, flatten=True)
          for k, v in dt_result.items():
              dt_scores_dict[k].append(v)
```

```
[33]: dt_result_df = pd.DataFrame(data=dt_scores_dict)

      # Clean-up NaN; convert to None as expected
      dt_result_df.loc[[10], ["max_depth"]] = "None"

      dt_result_df
```

```
[33]:    max_depth  mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
      0        1.0          0.102         0.015            0.019           0.003
      1        5.0          0.288         0.029            0.019           0.001
      2       10.0          0.493         0.034            0.019           0.005
      3       15.0          0.659         0.034            0.016           0.003
      4       20.0          0.736         0.056            0.016           0.001
      5       25.0          0.778         0.050            0.018           0.004
      6       30.0          0.740         0.015            0.015           0.001
      7       35.0          0.742         0.024            0.015           0.001
      8       40.0          0.730         0.027            0.015           0.000
      9       45.0          0.802         0.064            0.017           0.003
      10      None          0.786         0.043            0.017           0.002

          mean_test_accuracy  std_test_accuracy  mean_train_accuracy  \
      0                0.812              0.004                0.812
      1                0.819              0.006                0.824
      2                0.809              0.004                0.850
      3                0.783              0.006                0.897
      4                0.755              0.005                0.944
```

```
5                 0.735              0.007                0.975
6                 0.723              0.010                0.991
7                 0.723              0.012                0.997
8                 0.721              0.011                0.999
9                 0.719              0.010                0.999
10                0.723              0.009                0.999

    std_train_accuracy  mean_test_f1  std_test_f1  mean_train_f1  \
0                0.001         0.394        0.017          0.394
1                0.002         0.449        0.023          0.466
2                0.002         0.449        0.020          0.568
3                0.003         0.434        0.016          0.729
4                0.007         0.405        0.014          0.861
5                0.007         0.402        0.015          0.943
6                0.003         0.394        0.017          0.980
7                0.001         0.396        0.018          0.994
8                0.000         0.393        0.013          0.997
9                0.000         0.395        0.012          0.998
10               0.000         0.400        0.013          0.998

    std_train_f1  mean_test_recall  std_test_recall  mean_train_recall  \
0          0.004             0.275            0.015              0.275
1          0.012             0.332            0.020              0.346
2          0.013             0.351            0.025              0.442
3          0.007             0.374            0.016              0.621
4          0.019             0.375            0.014              0.782
5          0.018             0.400            0.014              0.906
6          0.008             0.404            0.016              0.968
7          0.003             0.408            0.014              0.988
8          0.001             0.404            0.008              0.994
9          0.000             0.411            0.011              0.995
10         0.000             0.415            0.011              0.996

    std_train_recall
0              0.004
1              0.014
2              0.020
3              0.007
4              0.025
5              0.025
6              0.011
7              0.005
8              0.001
9              0.001
10             0.001
```

A bit shockingly, the shorter decision trees were very effective. A depth of 1 and 2 yielded test F1 scores of 0.449, which is nearly as good as our `LogisticRegression` model from the previous part. We also see a test recall score of 0.351 and a test accuracy of 0.809 for a `max_depth` equal to 2.

1.13.2 Random Forest

For a second attempt, let's try a random forest. We won't tune the hyperparameters yet, because we'll do that in the next section!

[34]:
```
rf_scores_dict = {
    "hps": ["Default"], # for RF, let's just leave the parameters as default␣
 ↪for now
} | create_blank_data()

for hp in rf_scores_dict["hps"]:
    # Create pipeline
    rf = RandomForestClassifier()
    pipe_rf = make_pipeline(ct, rf)

    rf_result = mean_std_cross_val_scores(pipe_rf, X_train_extended, y_train,␣
 ↪name=f"hp={hp}", raw=True, flatten=True)
    for k, v in rf_result.items():
        rf_scores_dict[k].append(v)
```

[35]:
```
rf_result_df = pd.DataFrame(data=rf_scores_dict)
rf_result_df
```

[35]:
```
       hps  mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0  Default          5.335         0.467            0.138           0.016

   mean_test_accuracy  std_test_accuracy  mean_train_accuracy  \
0               0.815              0.006                0.999

   std_train_accuracy  mean_test_f1  std_test_f1  mean_train_f1  std_train_f1  \
0                 0.0          0.47        0.018          0.998           0.0

   mean_test_recall  std_test_recall  mean_train_recall  std_train_recall
0             0.368            0.017              0.997             0.001
```

Our forest is overfitting, but we haven't tuned yet. Our test accuracy was 0.815, test F1 was 0.47 (beating `LogisticRegression`), and our test recall was 0.368 (also winning out over `LogisticRegression`).

1.13.3 LightGBM Classifier

Finally, let's try LightGBM. Again, no tuning.

[36]:
```
lg_scores_dict = {
    "hps": ["Default"], # for LGBM, let's just leave the parameters as default␣
 ↪for now
} | create_blank_data()

for hp in lg_scores_dict["hps"]:
    # Create pipeline
    lg = LGBMClassifier()
    pipe_lg = make_pipeline(ct, lg)

    lg_result = mean_std_cross_val_scores(pipe_lg, X_train_extended, y_train,␣
↪name=f"hp={hp}", raw=True, flatten=True)
    for k, v in lg_result.items():
        lg_scores_dict[k].append(v)
```

[37]:
```
lg_result_df = pd.DataFrame(data=lg_scores_dict)
lg_result_df
```

[37]:
```
       hps  mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0  Default          0.294         0.075            0.023           0.005

   mean_test_accuracy  std_test_accuracy  mean_train_accuracy  \
0               0.819              0.005                0.852

   std_train_accuracy  mean_test_f1  std_test_f1  mean_train_f1  std_train_f1  \
0               0.001         0.474        0.017          0.573         0.005

   mean_test_recall  std_test_recall  mean_train_recall  std_train_recall
0             0.366            0.014              0.448             0.007
```

Our `LGBMClassifier` only did marginally better, but again, has yet to be tuned. Our test accuracy was 0.819, test F1 was 0.474, and test recall was 0.366.

1.13.4 Summary

There are comments in each section reflecting on the individual scores. In sum, unsurprisingly, `RandomForestClassifier` and `LightGBM`, without any tuning, were able to beat our tuned `LogisticRegression` validation scores on the whole, especially with respect to our metric of interest, recall.

`LightGBM` and `RandomForestClassifier` were about on par with one another. We'll see if feature selection and additional tuning can improve either of these models in subsequent sections.

1.14 (Optional) 9. Feature selection

rubric={points:1}

Your tasks:

Make some attempts to select relevant features. You may try `RFECV` or forward selection. Do the results improve with feature selection? Summarize your results. If you see improvements in the results, keep feature selection in your pipeline. If not, you may abandon it in the next exercises.

Let's arbitrarily choose `RandomForestClassifier` and run `RFECV` against it.

[38]:
```
# Adapted from lecture code
rfe_pipe = make_pipeline(
    ct,
    # CV has to be 3 to complete in a reasonable amount of time on my device;␣
 ↪ideally, this would be >=5
    RFECV(LogisticRegression(max_iter=2000), cv=3),
    RandomForestClassifier(),
)

rfe_result_df = mean_std_cross_val_scores(rfe_pipe, X_train_extended, y_train)
```

[39]:
```
rfe_result_df
```

[39]:
```
                  mean    std
fit_time        75.878  5.442
score_time       0.126  0.047
test_accuracy    0.799  0.020
train_accuracy   0.903  0.076
test_f1          0.450  0.016
train_f1         0.730  0.228
test_recall      0.369  0.014
train_recall     0.656  0.274
```

There's no major improvement in this case using `RandomForestClassifier`, and given how expensive this operation is, we'll abandon it in the next exercises. (In fact, for this run, our F1 and accuracy scores were even worse, and our test recall value was higher by merely 0.001.)

1.15 10. Hyperparameter optimization

rubric={points:15}

Your tasks:

Make some attempts to optimize hyperparameters for the models you've tried and summarize your results. You may pick one of the best performing models from the previous exercise and tune hyperparameters only for that model. You may use `sklearn`'s methods for hyperparameter optimization or fancier Bayesian optimization methods. - GridSearchCV
- RandomizedSearchCV - scikit-optimize

We again, arbitrarily choose `RandomForestClassifier` and try to tune it's three major hyperparameters. `LightGBM` would've also been a very valid choice, but we don't have the computational ability to try both in a reasonable amount of time.

In this case, we also need to be particular about `scoring` being equal to `"f1"`, since the `RandomizedSearchCV` relies on it to determine the best score.

[40]:
```
param_grid = {
    "randomforestclassifier__n_estimators": [1, 2, 5, 10, 25, 50, 100, 250],
    "randomforestclassifier__max_depth":    [1, 2, 5, 10, 25, 50, 100, 250],
    "randomforestclassifier__max_features": [1, 2, 5, 10, None]
}

pipe_rf = make_pipeline(ct, rf)

random_search = RandomizedSearchCV(pipe_rf, param_distributions=param_grid,␣
↪scoring="f1", n_jobs=-1, n_iter=10, cv=5, refit=True,␣
↪random_state=RANDOM_STATE)
random_search.fit(X_train_extended, y_train)

results = pd.DataFrame(random_search.cv_results_)
results.T
```

[40]:

```
                        0  \
mean_fit_time
0.272867
std_fit_time
0.004259
mean_score_time
0.037516
std_score_time
0.002434
param_randomforestclassifier__n_estimators
25
param_randomforestclassifier__max_features
5
param_randomforestclassifier__max_depth
1
params
{'randomforestclassifier__n_estimators': 25, '…
split0_test_score
0.0
split1_test_score
0.0
split2_test_score
0.0
split3_test_score
0.0
split4_test_score
0.0
mean_test_score
0.0
std_test_score
0.0
rank_test_score
8
```

```
           1  \
mean_fit_time
0.142272
std_fit_time
0.008735
mean_score_time
0.026674
std_score_time
0.003429
param_randomforestclassifier__n_estimators
10
param_randomforestclassifier__max_features
2
param_randomforestclassifier__max_depth
1
params
{'randomforestclassifier__n_estimators': 10, '…
split0_test_score
0.0
split1_test_score
0.0
split2_test_score
0.0
split3_test_score
0.0
split4_test_score
0.0
mean_test_score
0.0
std_test_score
0.0
rank_test_score
8

           2  \
mean_fit_time
1.2098
std_fit_time
0.013383
mean_score_time
0.103303
std_score_time
0.004641
param_randomforestclassifier__n_estimators
100
param_randomforestclassifier__max_features
```

```
2
param_randomforestclassifier__max_depth
5
params
{'randomforestclassifier__n_estimators': 100, …
split0_test_score
0.019704
split1_test_score
0.046602
split2_test_score
0.019724
split3_test_score
0.033301
split4_test_score
0.011893
mean_test_score
0.026245
std_test_score
0.012292
rank_test_score
7

            3  \
mean_fit_time
11.312225
std_fit_time
0.170636
mean_score_time
0.053441
std_score_time
0.000803
param_randomforestclassifier__n_estimators
50
param_randomforestclassifier__max_features
None
param_randomforestclassifier__max_depth
5
params
{'randomforestclassifier__n_estimators': 50, '…
split0_test_score
0.437122
split1_test_score
0.446809
split2_test_score
0.449832
split3_test_score
0.492084
```

```
split4_test_score
0.44339
mean_test_score
0.453847
std_test_score
0.01958
rank_test_score
2

            4  \
mean_fit_time
0.856569
std_fit_time
0.024228
mean_score_time
0.036105
std_score_time
0.001068
param_randomforestclassifier__n_estimators
10
param_randomforestclassifier__max_features
10
param_randomforestclassifier__max_depth
25
params
{'randomforestclassifier__n_estimators': 10, '…
split0_test_score
0.406692
split1_test_score
0.408759
split2_test_score
0.423001
split3_test_score
0.452611
split4_test_score
0.418605
mean_test_score
0.421934
std_test_score
0.016488
rank_test_score
3

            5  \
mean_fit_time
23.775367
std_fit_time
```

```
1.051513
mean_score_time
0.118577
std_score_time
0.02916
param_randomforestclassifier__n_estimators
250
param_randomforestclassifier__max_features
None
param_randomforestclassifier__max_depth
2
params
{'randomforestclassifier__n_estimators': 250, …
split0_test_score
0.386913
split1_test_score
0.381022
split2_test_score
0.378022
split3_test_score
0.417441
split4_test_score
0.389088
mean_test_score
0.390497
std_test_score
0.014044
rank_test_score
6

            6  \
mean_fit_time
0.203254
std_fit_time
0.013148
mean_score_time
0.029575
std_score_time
0.001456
param_randomforestclassifier__n_estimators
5
param_randomforestclassifier__max_features
1
param_randomforestclassifier__max_depth
25
params
{'randomforestclassifier__n_estimators': 5, 'r…
```

```
split0_test_score
0.387516
split1_test_score
0.378025
split2_test_score
0.400251
split3_test_score
0.392005
split4_test_score
0.415216
mean_test_score
0.394603
std_test_score
0.012559
rank_test_score
5

            7  \
mean_fit_time
2.769991
std_fit_time
0.101174
mean_score_time
0.150838
std_score_time
0.006858
param_randomforestclassifier__n_estimators
250
param_randomforestclassifier__max_features
10
param_randomforestclassifier__max_depth
1
params
{'randomforestclassifier__n_estimators': 250, …
split0_test_score
0.0
split1_test_score
0.0
split2_test_score
0.0
split3_test_score
0.0
split4_test_score
0.0
mean_test_score
0.0
std_test_score
```

```
0.0
rank_test_score
8

            8  \
mean_fit_time
2.779718
std_fit_time
0.03423
mean_score_time
0.215598
std_score_time
0.029421
param_randomforestclassifier__n_estimators
100
param_randomforestclassifier__max_features
1
param_randomforestclassifier__max_depth
25
params
{'randomforestclassifier__n_estimators': 100, …
split0_test_score
0.410641
split1_test_score
0.411846
split2_test_score
0.401394
split3_test_score
0.434193
split4_test_score
0.412457
mean_test_score
0.414106
std_test_score
0.010816
rank_test_score
4

            9
mean_fit_time
3.849024
std_fit_time
0.258476
mean_score_time
0.095423
std_score_time
0.009311
```

```
param_randomforestclassifier__n_estimators
50
param_randomforestclassifier__max_features
10
param_randomforestclassifier__max_depth
250
params
{'randomforestclassifier__n_estimators': 50, '…
split0_test_score
0.434048
split1_test_score
0.458974
split2_test_score
0.466495
split3_test_score
0.502203
split4_test_score
0.449804
mean_test_score
0.462305
std_test_score
0.022694
rank_test_score
1
```

[41]:
```
random_search.best_params_
```

[41]:
```
{'randomforestclassifier__n_estimators': 50,
 'randomforestclassifier__max_features': 10,
 'randomforestclassifier__max_depth': 250}
```

[42]:
```
random_search.best_score_
```

[42]:
```
0.46230487270663206
```

Our best performing parameters earned an F1-score of 0.461, which is close to the best score we've seen thus far, but still falls short of `LGBM`. The parameter used were `n_estimators=50`, `max_features=10`, and `max_depth=250`, so it created a rather complex forest. Something surprisingly, it's worse than our un-tuned random forest.

Since it stills falls short of `LGBM`'s performance, we'll return to `LGBM` in subsequent parts.

1.16 11. Interpretation and feature importances

rubric={points:15}

Your tasks:

1. Use the methods we saw in class (e.g., `eli5`, `shap`) (or any other methods of your choice) to

explain feature importances of one of the best performing models. Summarize your observations.

Here we arbitrarily jump back to LGBM, just to get more practice with different models, since just needed to choose "one of the best performing models."

[43]:
```
# Let's go back to LGBM!

pipe_lgbm = make_pipeline(ct, LGBMClassifier())
pipe_lgbm.fit(X_train_extended, y_train)

eli5.explain_weights(
    pipe_lgbm.named_steps["lgbmclassifier"], feature_names=column_names
)
```

[43]:
```
Explanation(estimator='LGBMClassifier()', description='\nLightGBM feature
importances; values are numbers 0 <= x <= 1;\nall values sum to 1.\n',
error=None, method='feature importances', is_regression=False, targets=None, fea
ture_importances=FeatureImportances(importances=[FeatureWeight(feature='PAY1_2',
weight=0.2674962228629025, std=None, value=None),
FeatureWeight(feature='PAY2_2', weight=0.09279469703957027, std=None,
value=None), FeatureWeight(feature='PAY_AMT2', weight=0.05178526818032497,
std=None, value=None), FeatureWeight(feature='LIMIT_BAL',
weight=0.04313299101121964, std=None, value=None),
FeatureWeight(feature='PAY_AMT1', weight=0.03814855658312894, std=None,
value=None), FeatureWeight(feature='BILL_AMT1', weight=0.03130686087060737,
std=None, value=None), FeatureWeight(feature='AGE', weight=0.029395644320057878,
std=None, value=None), FeatureWeight(feature='PAY3_2',
weight=0.02840320758841829, std=None, value=None),
FeatureWeight(feature='BILL_PAY_RATIO1', weight=0.02612967679246415, std=None,
value=None), FeatureWeight(feature='PAY_AMT3', weight=0.025576572610516577,
std=None, value=None), FeatureWeight(feature='PAY_AMT6',
weight=0.024752646264134755, std=None, value=None),
FeatureWeight(feature='PAY_AMT4', weight=0.021024037813521076, std=None,
value=None), FeatureWeight(feature='BILL_PAY_RATIO6',
weight=0.020740976226685415, std=None, value=None),
FeatureWeight(feature='PAY_AMT5', weight=0.020599191723264326, std=None,
value=None), FeatureWeight(feature='BILL_PAY_RATIO5',
weight=0.020243784125652572, std=None, value=None),
FeatureWeight(feature='BILL_PAY_RATIO3', weight=0.020075569813608914, std=None,
value=None), FeatureWeight(feature='BILL_PAY_RATIO2',
weight=0.019893061198509077, std=None, value=None),
FeatureWeight(feature='BILL_AMT2', weight=0.019249685498300535, std=None,
value=None), FeatureWeight(feature='BILL_PAY_RATIO4', weight=0.018391137194284,
std=None, value=None), FeatureWeight(feature='PAY4_2',
weight=0.018029659931611854, std=None, value=None)], remaining=72),
decision_tree=None, highlight_spaces=None, transition_features=None, image=None)
```

We find PAY1 to carry the most impact; since it was one-hot encoded, and in particular, when it took on a value of 2. Indeed, it seems “2” as a value for any of the PAYX features carried a significant amount of weight. This corresponds to a “payment delay [of] two months” which I find rather interesting; perhaps a domain-expert would be able to provide some insight of why 2 months in particular becomes unrecoverable, whereas presumably 1 month could be okay.

Looking at other meaningful columns, we see PAY_AMT2 and PAY_AMT1 ranking highly, which makes intuitive sense; if a customer is on track from the beginning, we’d expect them to stay on track until the end. On the other hand, starting on the wrong foot could have lasting consequences.

AGE is higher than I expected it to be; we can’t know if this is a linear impact or not, but I’d expect it to be non-linear. It’s like young people and old people are the most likely to default, with middle-aged folks who have stable jobs to be the least likely to default. This is all speculation, though.

Again, it’s important to note this importances are unsigned and not necessarily linear, so we have to be careful in our interpretation of them. They’re just “important somehow,” and it’s up to us to interpret in what direction (or, rely on SHAP force plots, as we do below).

1.17 12. Results on the test set

rubric={points:5}

Your tasks:

1. Try your best performing model on the test data and report test scores.
2. Do the test scores agree with the validation scores from before? To what extent do you trust your results? Do you think you’ve had issues with optimization bias?

Our best model was LGBMClassifier, despite attempting to tune RandomForestClassifier. Let’s now try it on the test set.

[44]:
```
lg = LGBMClassifier()
pipe_lg = make_pipeline(ct, lg)

pipe_lg.fit(X_train_extended, y_train)

y_train_pred = pipe_lg.predict(X_train_extended)
print(classification_report(y_train, y_train_pred, target_names=["non-default",␣
↪"default"]))
```

```
              precision    recall  f1-score   support

 non-default       0.86      0.96      0.91     17491
     default       0.77      0.43      0.55      5009

    accuracy                           0.84     22500
   macro avg       0.81      0.70      0.73     22500
weighted avg       0.84      0.84      0.83     22500
```

[45]:
```
y_test_pred = pipe_lg.predict(X_test_extended)
print(classification_report(y_test, y_test_pred, target_names=["non-default",␣
 ↪"default"]))
```

```
              precision    recall  f1-score   support

 non-default       0.85      0.94      0.89      5873
     default       0.65      0.38      0.48      1627

    accuracy                           0.82      7500
   macro avg       0.75      0.66      0.69      7500
weighted avg       0.80      0.82      0.80      7500
```

We find an F1-score of 0.48, an accuracy of 0.82, and a recall of 0.38. They're on par with our validation scores from earlier, though lower than I would've hoped. In particular, a 0.38 recall score is quite good, and our F1-score generalized well to the test set; our validation set score was 0.474. (So, surprisingly, our model did even better in testing than in validation!)

There was perhaps *opportunity* for optimization bias here, but we don't think it occurred in our analysis. We did tune on hyperparameters, the number of features, and even the kind of model we're using (switching between `LGBM` and `RandomForestClassifier` in this case). Given how well our model generalized, I don't believe we in fact over-optimized for our hyperparameters; we would've seen a much worse F1-score if we had done so.

1.18 (Optional) 13. Explaining predictions

rubric={points:1}

Your tasks

1. Take one or two test predictions and explain them with SHAP force plots.

[46]:
```
shap.initjs()
```

```
<IPython.core.display.HTML object>
```

[47]:
```
X_train_enc = pd.DataFrame(
    data=ct.transform(X_train_extended),
    columns=column_names,
    index=X_train_extended.index,
)

X_train_enc.shape
```

[47]:
```
(22500, 92)
```

[48]:
```
X_test_enc = pd.DataFrame(
    data=ct.transform(X_test_extended),
    columns=column_names,
    index=X_test_extended.index,
)

X_test_enc.shape
```

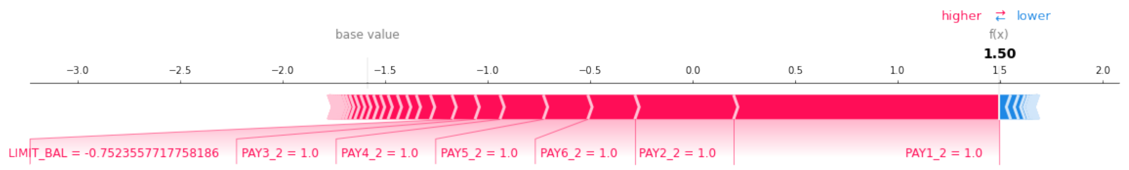
[48]:
```
(7500, 92)
```

[49]:
```
# This cell takes a VERY LONG time to run (https://github.com/slundberg/shap/
↪issues/2002)
# It may not be able to output plots in time for the HW submission; however,␣
↪this is the general idea for SHAP force plots
# Limit to 100 values as a fix (https://github.com/slundberg/shap/issues/838)

lg_explainer = shap.TreeExplainer(pipe_lg.named_steps["lgbmclassifier"])
train_lgbm_shap_values = lg_explainer.shap_values(X_train_enc[:100])
test_lgbm_shap_values  = lg_explainer.shap_values(X_test_enc[:100])
```

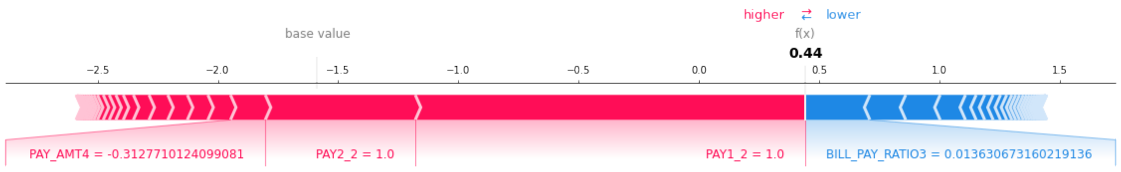
LightGBM binary classifier with TreeExplainer shap values output has changed to a list of ndarray

[50]:
```
# Sample values to test
y_test_reset = y_test.reset_index(drop=True)

ind_0 = y_test_reset[y_test_reset == 0].index.tolist()
ind_1 = y_test_reset[y_test_reset == 1].index.tolist()

sample_ind_0 = ind_0[10]
sample_ind_1 = ind_1[10]
```

[51]:
```
# Try an example on X_train
shap.force_plot(
    lg_explainer.expected_value[1],
    train_lgbm_shap_values[1][sample_ind_1, :],
    X_train_enc.iloc[sample_ind_1, :],
    matplotlib=True,
)
```



[52]:
```python
# Try an example on X_test
shap.force_plot(
    lg_explainer.expected_value[1],
    test_lgbm_shap_values[1][sample_ind_1, :],
    X_test_enc.iloc[sample_ind_1, :],
    matplotlib=True,
)
```



1.19 14. Summary of results

rubric={points:10}

Your tasks:

1. Report your final test score along with the metric you used.
2. Write concluding remarks.
3. Discuss other ideas that you did not try but could potentially improve the performance/interpretability .

1. Report your final test score along with the metric you used.

We're most interested in the recall score (since we wanted to minimize false negatives), but since we used `RandomizedSearchCV` and didn't want to ruin our precision score (in lieu of an operating point), we used the F1-score. The best performing model was of type `LightGBM`.

Our final F1-score was 0.48. We also saw an accuracy of 0.82 and a recall of 0.38.

2. Write concluding remarks.

In this analysis, we created a model to predict whether or not an individual would default on their next credit card payment given their relevant credit history and other demographic information. We began by performing some initial data wrangling and exploratory data analysis. After getting an understanding of the relevant data, we create a number of different models and performed cross-validation on them to get a sense of how they would perform on our test data. Linear models did more poorly on average in comparison to our tree-based models; in particular, the random forest classifier became an immediate candidate for further experimentation. We chose to tune on that model.

After hyperparameter tuning and attempted feature selection, we produced a random forest classification model that still fell short of our LightGBM attempt (that went without much tuning at all). Our LightGBM model achieved an accuracy of 0.82, F1-score of 0.48, and recall of 0.38. These results are promising, as they are on par with what we saw in the validation stage of this analysis; our results should generalize well on "deployment data" since the test and validation scores are similar.

3. Discuss other ideas that you did not try but could potentially improve the performance/interpretability.

For performance, we could always try and continue to tune our model. In this case, we had our `RandomForestClassifier` still fall short of `LightGBM`, despite tuning, so perhaps we could continue to engineer `LightGBM` for even better results. The parameters look complicated though, so tuning them would require a bit of extra learning (that's beyond the scope of what we've seen in this course). However, we run the risk of overoptimizing to our hyperparameters and creating a model that does not generalize well. Instead, we could try to do more work with feature engineering; none of the columns that were engineered appeared in our feature importance table, which suggests there is room for improvement. Feature engineering would require some more domain knowledge as well, so perhaps it'd be worth performing some research in this area before continuing.

For interpretability, a simpler model could help. `LightGBM` classifiers are not easy to interpret, given the sheer mathematical complexity of the model; it's hard for an individual to get a good grasp of how to weigh the importance of certain features. Instead, we could aim to achieve a similar result with a simpler model (either in terms of `LightGBM` classifier's hyperparameters or just a separate, simpler model altogether). Finally, we could also do more work in terms of producing various SHAP plots and visualizing feature importances. This helps a domain expert get a better feel for what the model is looking at without needing a specialization in data science itself.

1.20 Submission instructions

PLEASE READ: When you are ready to submit your assignment do the following:

1. Run all cells in your notebook to make sure there are no errors by doing `Kernel -> Restart Kernel and Clear All Outputs` and then `Run -> Run All Cells`.
2. Notebooks with cell execution numbers out of order or not starting from "1" will have marks deducted. Notebooks without the output displayed may not be graded at all (because we need to see the output in order to grade your work).
3. Upload the assignment using Gradescope's drag and drop tool. Check out this Gradescope Student Guide if you need help with Gradescope submission.