lab4

December 13, 2025

[1]:
```
# Initialize Otter
import otter
grader = otter.Notebook("lab4.ipynb")
```

1 Lab 4: Putting it all together in a mini project

For this lab, **you can choose to work alone or in a group of up to three students**. You are in charge of how you want to work and who you want to work with. Maybe you really want to go through all the steps of the ML process yourself or maybe you want to practice your collaboration skills, it is up to you! Just remember to indicate who your group members are (if any) when you submit on Gradescope. If you choose to work in a group, you only need to use one GitHub repo (you can create one on github.ubc.ca and set the visibility to "public"). If it takes a prohibitively long time to run any of the steps on your laptop, it is OK if you sample the data to reduce the runtime, just make sure you write a note about this.

1.1 Submission instructions

rubric={mechanics}

You receive marks for submitting your lab correctly, please follow these instructions:

Follow the general lab instructions.

Click here to view a description of the rubrics used to grade the questions

Make at least three commits.

Push your .ipynb file to your GitHub repository for this lab and upload it to Gradescope.

Before submitting, make sure you restart the kernel and rerun all cells.

Make sure to only make one gradescope submission per group, and to assign all group members on gradescope at submission time.

Also upload a .pdf export of the notebook to facilitate grading of manual questions (preferably WebPDF, you can select two files when uploading to gradescope)

Don't change any variable names that are given to you, don't move cells around, and don't include any code to install packages in the notebook.

The data you download for this lab SHOULD NOT BE PUSHED TO YOUR REPOSITORY (there is also a .gitignore in the repo to prevent this).

Include a clickable link to your GitHub repo for the lab just below this cell

It should look something like this https://github.ubc.ca/MDS-2020-21/DSCI_531_labX_yourcwl.

Points: 2

https://github.com/will-chh/573-lab4-creditcard-default-predictor#

1.2 Introduction

In this lab you will be working on an open-ended mini-project, where you will put all the different things you have learned so far in 571 and 573 together to solve an interesting problem.

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

Tips

1. Since this mini-project is open-ended 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.

Assessment 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 instead try several reasonable approaches and you have clearly motivated your choices, but still get lower model performance than your friend, don't sweat it.

A final note Finally, the style of this "project" 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 "several hours" but not "many hours" is a good guideline for a high quality 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 we hope you enjoy it as well.

1.3 1. Pick your problem and explain the prediction problem

rubric={reasoning}

In this mini project, you will pick one of the following problems:

1. 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.

OR

2. A regression problem of predicting `reviews_per_month`, as a proxy for the popularity of the listing with New York City Airbnb listings from 2019 dataset. Airbnb could use this sort of model to predict how popular future listings might be before they are posted, perhaps to help guide hosts create more appealing listings. In reality they might instead use something like vacancy rate or average rating as their target, but we do not have that available here.

Your tasks:

1. Spend some time understanding the problem and what each feature means. 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.
3. Carry out any preliminary preprocessing, if needed (e.g., changing feature names, handling of NaN values etc.)

Points: 3

Our problem of interest is predicting whether a credit card client will default on their payment next month. Given a dataset of 30,000 examples and 24 features, we aim to build a binary classification model that can accurately predict default payment behavior. This can be used by financial institutions in areas such as risk assessment and decision-making regarding credit issuance.

- ID: row identifier
- LIMIT_BAL: amount of given credit in NT dollars (includes individual and family/supplementary credit)
- SEX: gender (1=male, 2=female)
- EDUCATION: education level (1=graduate school; 2=university; 3=high school; 4=others)
- MARRIAGE: marital status (1=married; 2=single; 3=others)
- AGE: age in years

- PAY_0 to PAY_6: history of past monthly payment (from September 2005 to April 2005). Values are -1=pay duly (no delay), 1=payment delay for one month, 2=payment delay for two months, and so on.
- PAY_AMT1 to PAY_AMT6: amount of previous payment (payment status from the last 6 months starting from September)
- BILL_AMT1 to BILL_AMT6: amount of bill statement (from September 2005 to April 2005)
- default.payment.next.month: default payment (1=yes, 0=no). This is our target variable

There are a mix of categorical and numerical features in the dataset, demographic features include SEX, EDUCATION, MARRIAGE, and AGE. Financial features include LIMIT_BAL, PAY_0 to PAY_6, PAY_AMT1 to PAY_AMT6, and BILL_AMT1 to BILL_AMT6. Behavioural features could include payment history (PAY_0 to PAY_6) and previous payment amounts (PAY_AMT1

to PAY_AMT6). An initial thought is that PAY_0 to PAY_6 may be most predictive of default payment next month, as they likely reflect the client's payment behaviour over the past six months.

Package Importation

[2]:
```
# Data & plotting
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import altair as alt
from ydata_profiling import ProfileReport

# Scikit-learn utilities
from sklearn.model_selection import (
    train_test_split,
    cross_val_score,
    cross_validate,
    GridSearchCV,
    RandomizedSearchCV,
)

# Preprocessing / transformations
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder,␣
 ↪StandardScaler, KBinsDiscretizer
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer, make_column_transformer
from sklearn.pipeline import Pipeline, make_pipeline

# Models - classification & regression
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor,␣
 ↪GradientBoostingClassifier
from sklearn.dummy import DummyClassifier, DummyRegressor

# Additional things to import
from scipy.stats import loguniform, randint
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.inspection import permutation_importance

from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score, roc_auc_score,
    confusion_matrix, ConfusionMatrixDisplay
)
```

Muting Warnings

[3]:
```
import warnings
warnings.filterwarnings(
    "ignore",
    message=r"You passed a .* to `is_pandas_dataframe`",
    module="altair.utils.data",
    category=UserWarning,
)
```

Data Loading

[4]:
```
cr_card_df = pd.read_csv('data/UCI_Credit_Card.csv', header=0)
#cr_card_df = cr_card_df.sample(frac=0.25, random_state=123)
```

[5]:
```
#viewing the first 5 rows of data
cr_card_df.head()
```

[5]:

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	…	BILL_AMT4	BILL_AMT5	BILL_AMT6	PAY_AMT1	PAY_AMT2	PAY_AMT3	PAY_AMT4	PAY_AMT5	PAY_AMT6	default.payment.next.month
0	1	20000.0	2	2	1	24	2	2	-1	-1	…	0.0	0.0	0.0	0.0	689.0	0.0	0.0	0.0	0.0	1
1	2	120000.0	2	2	2	26	-1	2	0	0	…	3272.0	3455.0	3261.0	0.0	1000.0	1000.0	1000.0	0.0	2000.0	1
2	3	90000.0	2	2	2	34	0	0	0	0	…	14331.0	14948.0	15549.0	1518.0	1500.0	1000.0	1000.0	1000.0	5000.0	0
3	4	50000.0	2	2	1	37	0	0	0	0	…	28314.0	28959.0	29547.0	2000.0	2019.0	1200.0	1100.0	1069.0	1000.0	0
4	5	50000.0	1	2	1	57	-1	0	-1	0	…	20940.0	19146.0	19131.0	2000.0	36681.0	10000.0	9000.0	689.0	679.0	0

[5 rows x 25 columns]

[6]:
```
#checking for null values
cr_card_df.info() #checking datatypes and non-null counts
cr_card_df.isna().sum()
```

```
<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
```

[6]:
```
ID                          0
LIMIT_BAL                   0
SEX                         0
EDUCATION                   0
MARRIAGE                    0
AGE                         0
PAY_0                       0
PAY_2                       0
PAY_3                       0
PAY_4                       0
PAY_5                       0
PAY_6                       0
BILL_AMT1                   0
BILL_AMT2                   0
BILL_AMT3                   0
BILL_AMT4                   0
BILL_AMT5                   0
BILL_AMT6                   0
PAY_AMT1                    0
PAY_AMT2                    0
```

```
PAY_AMT3                      0
PAY_AMT4                      0
PAY_AMT5                      0
PAY_AMT6                      0
default.payment.next.month    0
dtype: int64
```

[7]:
```
#cleaning column names so that they are more consistent and readable (in␣
 ↪snake_case)
cr_card_df.columns = cr_card_df.columns.str.lower().str.replace('.','_')
print(cr_card_df.columns)
```

```
Index(['id', 'limit_bal', 'sex', 'education', 'marriage', 'age', 'pay_0',
       'pay_2', 'pay_3', 'pay_4', 'pay_5', 'pay_6', 'bill_amt1', 'bill_amt2',
       'bill_amt3', 'bill_amt4', 'bill_amt5', 'bill_amt6', 'pay_amt1',
       'pay_amt2', 'pay_amt3', 'pay_amt4', 'pay_amt5', 'pay_amt6',
       'default_payment_next_month'],
      dtype='object')
```

[8]:
```
#dropping irrelevant columns
cr_card_df = cr_card_df.drop(columns=['id'])
```

[9]:
```
#according to the research paper, there are some extra categories that are not␣
 ↪represented correctly
print(cr_card_df['education'].value_counts().sort_index()) #5,6 are undefined/
 ↪unknown, can be treated as 'others'
print(cr_card_df['marriage'].value_counts().sort_index()) #0 is undefined, can␣
 ↪be treated as 'others'

cr_card_df['education'] = cr_card_df['education'].replace({5:4, 6:4, 0:4})
cr_card_df['marriage'] = cr_card_df['marriage'].replace({0:3})
```

```
education
0       14
1    10585
2    14030
3     4917
4      123
5      280
6       51
Name: count, dtype: int64
marriage
0       54
1    13659
2    15964
3      323
Name: count, dtype: int64
```

[10]:
```
#checking values after they are grouped to 'others' categories
print(cr_card_df['education'].value_counts().sort_index())
print(cr_card_df['marriage'].value_counts().sort_index())
```

```
education
1    10585
2    14030
3     4917
4      468
Name: count, dtype: int64
marriage
1    13659
2    15964
3      377
Name: count, dtype: int64
```

[11]:
```
#checking class imbalance
cr_card_df['default_payment_next_month'].value_counts(normalize=True)
```

[11]:
```
default_payment_next_month
0    0.7788
1    0.2212
Name: proportion, dtype: float64
```

1.4 2. Data splitting

rubric={reasoning}

Your tasks:

1. Split the data into train and test portions.

   Make the decision on the `test_size` based on the capacity of your laptop.

Points: 1

[12]:
```
# splitting into 70% train and 30% test portions
cr_card_train, cr_card_test = train_test_split(
    cr_card_df,
    test_size=0.3,
    random_state=123
    )

# Below could be dropped
X_train = cr_card_train.drop(columns=['default_payment_next_month'])
y_train = cr_card_train['default_payment_next_month']
X_test = cr_card_test.drop(columns=['default_payment_next_month'])
y_test = cr_card_test['default_payment_next_month']
```

1.5 3. EDA

rubric={viz,reasoning}

Perform exploratory data analysis on the train set.

Your tasks:

1. Include at least two summary statistics and two visualizations that you find useful, and accompany each one with a sentence explaining it.
2. Summarize your initial observations about the data.
3. Pick appropriate metric/metrics for assessment.

Points: 6

2. The profile report shows that there are no missing values in the dataset. The target variable, default_payment_next_month, has a class imbalance with 22.3% of instances being 1 (default) and 77.7% being 0 (no default). This indicates that the model might need to account for this imbalance during training.
   In the correlations section, we can see that pay_0 has the highest positive correlation with the target variable, suggesting that recent payment history is a strong predictor of default payment next month. BILL_AMT1 also shows a moderate positive correlation, indicating that higher bill amounts may be associated with a higher likelihood of default.
   Many financial features like bill_amt, pay_amt are highly skewed to the right, indicating that most clients have lower amounts while a few have very high amounts. This suggests that transformations like log transformation might be beneficial for these features.

3. Based on the class imbalance in the target variable, accuracy may not be sufficient to evaluate model performance. Therefore, we will use recall, which indicates how many actual defaults were correctly identified by the model, since missed defaults are the most expensive and important to identify. We can consider changing the decision threshold to optimize recall. Additionally, we will also report precision and F1-score to provide a more comprehensive evaluation of the model's performance (including a confusion matrix), as well as the ROC-AUC score to assess the model's ability to discriminate between classes across different thresholds.

[13]:
```
profile = ProfileReport(cr_card_train, title='Credit card default training set␣
↪EDA', explorative=True)
profile
```

```
Summarize dataset:   0%|          | 0/5 [00:00<?, ?it/s]

100%|      | 24/24 [00:00<00:00, 135.01it/s]

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

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

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

[13]:

[14]:
```
cr_card_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 21000 entries, 16395 to 19966
Data columns (total 24 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   limit_bal                   21000 non-null  float64
 1   sex                         21000 non-null  int64
 2   education                   21000 non-null  int64
 3   marriage                    21000 non-null  int64
 4   age                         21000 non-null  int64
 5   pay_0                       21000 non-null  int64
 6   pay_2                       21000 non-null  int64
 7   pay_3                       21000 non-null  int64
 8   pay_4                       21000 non-null  int64
 9   pay_5                       21000 non-null  int64
 10  pay_6                       21000 non-null  int64
 11  bill_amt1                   21000 non-null  float64
 12  bill_amt2                   21000 non-null  float64
 13  bill_amt3                   21000 non-null  float64
 14  bill_amt4                   21000 non-null  float64
 15  bill_amt5                   21000 non-null  float64
 16  bill_amt6                   21000 non-null  float64
 17  pay_amt1                    21000 non-null  float64
 18  pay_amt2                    21000 non-null  float64
 19  pay_amt3                    21000 non-null  float64
 20  pay_amt4                    21000 non-null  float64
 21  pay_amt5                    21000 non-null  float64
 22  pay_amt6                    21000 non-null  float64
 23  default_payment_next_month  21000 non-null  int64
dtypes: float64(13), int64(11)
memory usage: 4.5 MB
```

[15]:
```
#Creating a Correlation Table for all numerical features against our target column

num_cols0 = cr_card_train.select_dtypes(include=['float64', 'int64']).columns

corr_w_target = (cr_card_train[num_cols0]
                 .corrwith(cr_card_train['default_payment_next_month'].
 astype(int))
                 .sort_values(ascending=False))

table = corr_w_target.reset_index()
table.columns = (['feature', 'correlation with default'])

table
```

[15]:

	feature	correlation with default
0	default_payment_next_month	1.000000
1	pay_0	0.325102
2	pay_2	0.265160
3	pay_3	0.240503
4	pay_4	0.219692
5	pay_5	0.208726
6	pay_6	0.194787
7	education	0.032647
8	age	0.010715
9	bill_amt6	-0.004944
10	bill_amt5	-0.007868
11	bill_amt4	-0.012313
12	bill_amt3	-0.014718
13	bill_amt2	-0.015301
14	bill_amt1	-0.020632
15	marriage	-0.024121
16	sex	-0.046320
17	pay_amt5	-0.050943
18	pay_amt6	-0.056093
19	pay_amt2	-0.060730
20	pay_amt3	-0.060868
21	pay_amt4	-0.061005
22	pay_amt1	-0.071563
23	limit_bal	-0.149247

Brief interpretation on Correlation outcome

1. With pay_0 at 0.3251, customers with worse repayment status last month (higher PAY_0) are more likely to default next month.
2. With education at 0.0326, age at 0.0107, and sex at -0.0463, these 3 fields do not clearly increase or decrease default risk, therefore, are considered to be dropped.
3. With limit_bal at -0.1492, customers with higher credit limit has lower chance of defaulting, which makes sense in real life.
4. All bill_amt* fields have small negative values, indicating a higher billed amount are associated with lower default risks.
5. All pay_amt* fields have negatives values, with pay_amt1 = -0.7156, it indicates higher and more recent payments are associated with lower default risks.

[16]:
```
#Creating a summary table of numeric features for each class (of our target)
num_feats = cr_card_train.select_dtypes(include=['float64']).columns

cr_card_train[['default_payment_next_month'] + list(num_feats)].melt(
    id_vars='default_payment_next_month',
    var_name='feature',
    value_name='value'
```

```
).pivot_table(
    index='feature', columns='default_payment_next_month',
    values='value',
    aggfunc=['mean', 'min', 'max', 'std'], observed=False
).round(2)
```

[16]:

	mean		min		max		std	
default_payment_next_month	0	1	0	1	0	1	0	1
feature								
bill_amt1	51919.88	48281.11	-15308.0	-6676.0	964511.0	610723.0	73510.24	73151.24
bill_amt2	49712.48	47089.01	-67526.0	-17710.0	983931.0	572677.0	71536.64	70892.54
bill_amt3	47555.09	45115.19	-157264.0	-46127.0	855086.0	578971.0	69335.80	67954.75
bill_amt4	43914.62	41997.34	-34503.0	-50616.0	891586.0	548020.0	65087.41	63971.24
bill_amt5	40686.59	39530.54	-61372.0	-53007.0	927171.0	547880.0	61203.11	61129.84
bill_amt6	38925.15	38217.63	-209051.0	-339603.0	961664.0	514975.0	59592.71	59573.31
limit_bal	178297.94	131633.46	10000.0	10000.0	1000000.0	720000.0	132225.85	115831.12
pay_amt1	6327.04	3399.86	0.0	0.0	873552.0	244500.0	18613.93	9351.22
pay_amt2	6604.80	3425.35	0.0	0.0	1227082.0	358689.0	23898.22	11579.43
pay_amt3	5911.11	3224.84	0.0	0.0	896040.0	319494.0	19999.34	10755.38
pay_amt4	5278.77	3017.73	0.0	0.0	621000.0	432130.0	16628.10	10053.39
pay_amt5	5167.72	3304.81	0.0	0.0	426529.0	332000.0	15709.33	13321.02
pay_amt6	5782.54	3342.18	0.0	0.0	528666.0	345293.0	19333.95	12846.71

Brief interpretation on the summary table:

1. People who do NOT default pay more and have higher limits.
2. Defaulters have: lower credit limits, lower bill amounts, and lower historical payment amounts.

[17]:
```
#given that pay_* are our strongest features, visualizing the distribution of␣
↪values within each pay_* feature can help us differentiate and understand it
```

```
alt.data_transformers.disable_max_rows()

pay_cols = ['pay_0', 'pay_2', 'pay_3', 'pay_4', 'pay_5', 'pay_6']

pay_long = cr_card_train[pay_cols + ['default_payment_next_month']].melt(
    id_vars='default_payment_next_month',
    var_name='pay_month',
    value_name='delay_status')

alt.Chart(pay_long).mark_bar().encode(
    x=alt.X('delay_status:O'),
    y=alt.Y('count():Q', title='Count of Clients'),
    color=alt.Color('default_payment_next_month:N', title='default')
).facet('pay_month:N', columns=3).properties(title='Payment Status distribution␣
 ↪by month')
```

[17]: alt.FacetChart(…)

[18]:
```
categorical_feats = ['sex', 'education', 'marriage']

cat_data = cr_card_train[categorical_feats + ['default_payment_next_month']].
 ↪melt(
    id_vars='default_payment_next_month',
    var_name='feature',
    value_name='value')

alt.Chart(cat_data).mark_bar().encode(
    x=alt.X('value:N'),
    y=alt.Y('count():Q', title='Count of Clients'),
    color=alt.Color('default_payment_next_month:N', title="default")
).facet('feature:N', columns=3).properties(title='Categorical feature␣
 ↪distribution')
```

[18]: alt.FacetChart(…)

[19]:
```
#A deeper dive into the correlation between amount of bill statement and␣
 ↪repayment status
num_feats = cr_card_train.select_dtypes(include=['float64']).columns

corr = cr_card_train[num_feats].corr()

corr_long = corr.reset_index().melt(
    id_vars='index',
    var_name='feature',
    value_name='correlation')

heatmap = alt.Chart(corr_long).mark_rect().encode(
```

```
    x=alt.X('feature:O', title='feature'),
    y=alt.Y('index:O', title='feature'),
    color=alt.Color('correlation:Q')
)

heatmap + heatmap.mark_text().encode(
    text=alt.Text('correlation:Q', format='.1f'),
    color=alt.value('black')
)
```

[19]: alt.LayerChart(…)

Brief Interpretation on the Correlation Heat Map Key Info only: 1. Bill_amt1 - 6 are highly correlated, leading to multicollinearity issue. 2. Pay_amt1 - 6 are somewhat correlated, multicollinearity is still a concern. 3. Limit_bal has some correlation with other fields, it could be a very useful feature.

[20]:
```
import altair as alt

# Ensure default is categorical with readable labels
cr_card_train['default_label'] = cr_card_train['default_payment_next_month'].
 ↪map({0: 'No Default', 1: 'Default'})

box = (
    alt.Chart(cr_card_train)
    .mark_boxplot(size=60)
    .encode(
        x=alt.X('default_label:N', title='Default Status'),
        y=alt.Y('limit_bal:Q', title='Credit Limit'),
        color=alt.Color('default_label:N', title='Default Status')
    )
    .properties(
        width=300,
        height=350,
        title='Boxplot of Credit Limit by Default Status'
    )
)

box
```

[20]: alt.Chart(…)

1.6 4. Feature engineering (Challenging)

rubric={reasoning}

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.

Points: 0.5

Feature Engineering Reasoning: The following 3 new features are created as a sniff test:

1. payment difference: pay_0 - pay_6 captures the change in payment behaviour from 6months ago to this month. A negative value indicates improvement (since pay_6 was bad and pay_0 was good), while a positive value indicates worsening behaviour, which may be predictive of default risk (behavioural trend over the 6 mo)
2. average pay amount: avg_pay_amt = captures the average payment amount over the past 6 months, which may indicate the client's ability to pay the next one
3. standard deviation of pay: captures payment volatility over the past 6 months, which may indicate inconsistent payment behaviour (for example: high standard deviation 0,1,3,5,0,6 -> erratic payment behaviour -> higher default risk). Low std (1,1,1,1,1,1) indicates consistent payment behaviour -> lower default risk

[21]:
```
#aggregate features for pay*

#Creating copies on the cr_card_train before having new features
cr_card_train_fe = cr_card_train.copy()
cr_card_test_fe = cr_card_test.copy()

#updating this line to have pay_0 as its own feature as pay_0 deviates from the␣
 ↪rest of the pay_* features
#averaging only from pay_2 to pay_6
hist_pay_cols = ['pay_2', 'pay_3', 'pay_4', 'pay_5', 'pay_6']

# Train
cr_card_train_fe['pay_avg_2_6'] = cr_card_train_fe[hist_pay_cols].mean(axis=1)
cr_card_train_fe['pay_past_std'] = cr_card_train_fe[hist_pay_cols].std(axis=1)
cr_card_train_fe['pay_diff'] = cr_card_train_fe['pay_0'] -␣
 ↪cr_card_train_fe['pay_6']

# Test
cr_card_test_fe['pay_avg_2_6'] = cr_card_test_fe[hist_pay_cols].mean(axis=1)
cr_card_test_fe['pay_past_std'] = cr_card_test_fe[hist_pay_cols].std(axis=1)
cr_card_test_fe['pay_diff'] = cr_card_test_fe['pay_0'] -␣
 ↪cr_card_test_fe['pay_6']


cr_card_train_fe.head()
```

[21]:

	limit_bal	sex	education	marriage	age	pay_0	pay_2	pay_3	pay_4	\
16395	320000.0	2	1	2	36	0	0	0	0	

```
21448   440000.0    2          1          2   30     -1     -1     -1      0
20034   160000.0    2          3          1   44     -2     -2     -2     -2
25755   120000.0    2          2          1   30      0      0      0      0
1438     50000.0    1          2          2   54      1      2      0      0

       pay_5  …  pay_amt2  pay_amt3  pay_amt4  pay_amt5  pay_amt6  \
16395      0  …    5018.0    1000.0    3000.0       0.0    7013.0
21448      0  …   87426.0  130007.0    3018.0   15000.0   51663.0
20034     -2  …       0.0       0.0       0.0       0.0       0.0
25755      0  …    5502.0    4204.0    3017.0    2005.0    1702.0
1438       0  …    1400.0    1200.0    1500.0    1000.0    1500.0

       default_payment_next_month  default_label  pay_avg_2_6  pay_past_std  \
16395                           0     No Default          0.0      0.000000
21448                           0     No Default         -0.4      0.547723
20034                           0     No Default         -2.0      0.000000
25755                           0     No Default          0.0      0.000000
1438                            0     No Default          0.4      0.894427

       pay_diff
16395         0
21448        -1
20034         0
25755         0
1438          1

[5 rows x 28 columns]
```

[22]:
```
eng_cols = ['pay_avg_2_6', 'pay_past_std', 'pay_diff']

eng_long = cr_card_train_fe[eng_cols + ['default_payment_next_month']].melt(
    id_vars='default_payment_next_month',
    var_name='feature',
    value_name='value')

alt.Chart(eng_long).mark_boxplot(size=40).encode(
    x=alt.X('default_payment_next_month:O'),
    y=alt.Y('value:Q'),
    color=alt.Color('default_payment_next_month:N'),
    facet=alt.Facet('feature:N', columns=3)
).properties(width=180, height=240, title='Distribution of engineered payment␣
 ↪features')
```

[22]:
```
alt.Chart(…)
```

[23]:
```
eng_long = cr_card_train_fe[eng_cols + ['default_payment_next_month']].melt(
    id_vars='default_payment_next_month',
```

```
    var_name='feature',
    value_name='value')

alt.Chart(eng_long).mark_bar(opacity=0.8).encode(
    x=alt.X('value:Q', bin=alt.Bin(maxbins=30)),
    y=alt.Y('count():Q'),
    color=alt.Color('default_payment_next_month:N'),
    facet=alt.Facet('feature:N', columns=3)
).properties(width=180, height=240, title='Distribution of engineered payment␣
 ↪features')
```

[23]:
```
alt.Chart(…)
```

1.7 5. Preprocessing and transformations

rubric={accuracy,reasoning}

Your tasks:

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

Points: 4

Preprocessing Set up In Q4, it explicitly mentioned to work with the newly created features in the following exercises, therefore we will be using using the transformed data sets for the rest of the assignment.

[24]:
```
cr_card_test_fe.head()
```

[24]:

	limit_bal	sex	education	marriage	age	pay_0	pay_2	pay_3	pay_4	\
25665	40000.0	2	2	2	26	-1	0	0	0	
16464	80000.0	2	3	1	59	0	0	0	0	
22386	170000.0	2	1	2	30	2	2	2	2	
10149	200000.0	2	2	1	41	-2	-2	-2	-2	
8729	50000.0	1	2	1	43	0	0	0	0	

	pay_5	…	pay_amt1	pay_amt2	pay_amt3	pay_amt4	pay_amt5	pay_amt6	\
25665	-1	…	1300.0	1000.0	0.0	22373.0	680.0	10000.0	
16464	0	…	3212.0	2106.0	2000.0	1603.0	1903.0	2006.0	
22386	2	…	6800.0	6500.0	0.0	13000.0	5500.0	1000.0	
10149	-2	…	742.0	0.0	0.0	0.0	0.0	0.0	
8729	0	…	1140.0	1150.0	331.0	341.0	356.0	330.0	

	default_payment_next_month	pay_avg_2_6	pay_past_std	pay_diff
25665	0	-0.2	0.447214	-1
16464	0	0.0	0.000000	0
22386	1	2.0	0.000000	0
10149	1	-2.0	0.000000	0

```
8729                              0           0.0       0.000000          0

[5 rows x 27 columns]
```

[25]:
```
#Creating Copies here, all transformations will be working on datasets ending␣
 ↪without *_fe
cr_card_train = cr_card_train_fe.copy()
cr_card_test = cr_card_test_fe.copy()


X_train = cr_card_train.drop(columns=['default_payment_next_month'])
y_train = cr_card_train['default_payment_next_month']

X_test = cr_card_test.drop(columns=['default_payment_next_month'])
y_test = cr_card_test['default_payment_next_month']


#Grouping feature types
numeric_feats = ['limit_bal',
                 'bill_amt1', 'bill_amt2', 'bill_amt3', 'bill_amt4',␣
 ↪'bill_amt5', 'bill_amt6',
                 'pay_amt1', 'pay_amt2', 'pay_amt3', 'pay_amt4', 'pay_amt5',␣
 ↪'pay_amt6',
                 'pay_avg_2_6', 'pay_past_std', 'pay_diff', # engineered
                 'age']

ordinal_feats = ['education', 'pay_0']

categorical_feats = ['sex', 'marriage']

drop_feats = ['pay_2', 'pay_3', 'pay_4', 'pay_5', 'pay_6']
```

[26]:
```
X_train = X_train.drop(columns=drop_feats)
X_test = X_test.drop(columns=drop_feats)
```

[27]:
```
#Applying transformations
preprocessor = make_column_transformer(
    (StandardScaler(), numeric_feats),
    (OrdinalEncoder(handle_unknown='use_encoded_value', unknown_value=-1),␣
 ↪ordinal_feats),
    (OneHotEncoder(handle_unknown='ignore'), categorical_feats),
    #("drop", drop_feats)
)

#Fitting preprocessor on X_train only
preprocessor.fit(X_train)
```

```
#Transforming both data sets
X_train_enc = preprocessor.transform(X_train)
X_test_enc = preprocessor.transform(X_test)

#Collecting column names, ordering matters, pls refer above, numeric to ordinal␣
 ↪to onehot
all_columns = preprocessor.get_feature_names_out()

X_train_df = pd.DataFrame(X_train_enc, columns=all_columns, index=X_train.index)
X_test_df  = pd.DataFrame(X_test_enc, columns=all_columns, index=X_test.index)

X_train_df
```

[27]:

	standardscaler__limit_bal	standardscaler__bill_amt1	\
16395	1.168355	-0.300665	
21448	2.090017	-0.685307	
20034	-0.060527	-0.696132	
25755	-0.367748	0.687456	
1438	-0.905384	-0.040230	
…	…	…	
28636	1.629186	-0.513226	
17730	1.475576	-0.695886	
28030	-0.905384	-0.053288	
15725	-1.058995	-0.309066	
19966	1.552381	-0.657925	

	standardscaler__bill_amt2	standardscaler__bill_amt3	\
16395	-0.293394	-0.265310	
21448	-0.679495	0.585444	
20034	-0.688319	-0.681234	
25755	0.752583	0.835581	
1438	-0.031399	-0.287429	
…	…	…	
28636	-0.486648	-0.488648	
17730	-0.688067	-0.680974	
28030	-0.010446	-0.003381	
15725	-0.288478	-0.254590	
19966	-0.326854	-0.637415	

	standardscaler__bill_amt4	standardscaler__bill_amt5	\
16395	-0.371930	-0.494781	
21448	1.970295	1.805461	
20034	-0.670935	-0.661045	
25755	0.918719	0.501203	
1438	-0.245237	-0.204599	
…	…	…	
28636	-0.450259	-0.400085	

```
17730                  -0.630868                  -0.618584
28030                  -0.062716                  -0.516534
15725                  -0.195023                  -0.179691
19966                  -0.605175                  -0.599674

       standardscaler__bill_amt6  standardscaler__pay_amt1  \
16395                  -0.587035                 -0.039546
21448                   1.327534                 -0.297166
20034                  -0.650908                 -0.333097
25755                   0.144527                 -0.115517
1438                   -0.191104                 -0.333097
…                            …                         …
28636                  -0.389539                 -0.262410
17730                  -0.534120                 -0.333097
28030                  -0.499414                 -0.215559
15725                  -0.372606                 -0.215677
19966                  -0.612175                  1.181272

       standardscaler__pay_amt2  standardscaler__pay_amt3  …  \
16395                 -0.040229                 -0.234603  …
21448                  3.739796                  6.785208  …
20034                 -0.270403                 -0.289017  …
25755                 -0.018028                 -0.060260  …
1438                  -0.206185                 -0.223720  …
…                            …                         …  …
28636                 -0.210772                 -0.221435  …
17730                 -0.270403                 -0.148629  …
28030                 -0.178663                 -0.205764  …
15725                 -0.178663                 -0.180189  …
19966                 -0.132473                 -0.054764  …

       standardscaler__pay_past_std  standardscaler__pay_diff  \
16395                     -0.770049                 -0.239659
21448                      0.316355                 -1.096604
20034                     -0.770049                 -0.239659
25755                     -0.770049                 -0.239659
1438                       1.004041                  0.617286
…                                 …                         …
28636                      1.004041                 -1.953550
17730                      0.889460                  1.474231
28030                     -0.770049                 -0.239659
15725                     -0.770049                 -0.239659
19966                     -0.770049                 -0.239659

       standardscaler__age  ordinalencoder__education  ordinalencoder__pay_0  \
16395             0.054187                        0.0                    2.0
21448            -0.597108                        0.0                    1.0
```

```
20034             0.922579                       2.0                    0.0
25755            -0.597108                       1.0                    2.0
1438              2.008070                       1.0                    3.0
…                    …                           …                      …
28636             0.162736                       1.0                    2.0
17730             2.008070                       0.0                    3.0
28030            -0.705657                       2.0                    2.0
15725            -1.574049                       1.0                    2.0
19966             0.054187                       0.0                    0.0

       onehotencoder__sex_1  onehotencoder__sex_2  onehotencoder__marriage_1  \
16395                   0.0                   1.0                        0.0
21448                   0.0                   1.0                        0.0
20034                   0.0                   1.0                        1.0
25755                   0.0                   1.0                        1.0
1438                    1.0                   0.0                        0.0
…                       …                     …                          …
28636                   0.0                   1.0                        1.0
17730                   0.0                   1.0                        1.0
28030                   0.0                   1.0                        1.0
15725                   0.0                   1.0                        0.0
19966                   0.0                   1.0                        1.0

       onehotencoder__marriage_2  onehotencoder__marriage_3
16395                        1.0                        0.0
21448                        1.0                        0.0
20034                        0.0                        0.0
25755                        0.0                        0.0
1438                         1.0                        0.0
…                            …                          …
28636                        0.0                        0.0
17730                        0.0                        0.0
28030                        0.0                        0.0
15725                        1.0                        0.0
19966                        0.0                        0.0

[21000 rows x 24 columns]
```

1.8 6. Baseline model

rubric={accuracy}

Your tasks: 1. Train a baseline model for your task and report its performance.

Points: 2

[28]:
```
import warnings
from sklearn.exceptions import UndefinedMetricWarning
```

```
warnings.filterwarnings("ignore", category=UndefinedMetricWarning)
```

[29]:
```
cross_val_results = {}

dummy = DummyClassifier(strategy = "most_frequent")
scoring_metrics = ['accuracy', 'f1', 'precision', 'recall', 'roc_auc']

cross_val_results['dummy'] = (pd.DataFrame(
    cross_validate(
        dummy,
        X_train_enc,
        y_train,
        scoring = scoring_metrics,
        cv=5,
        return_train_score=True))
    .agg(['mean', 'std'])
    .round(3)
    .T)

cross_val_results['dummy']
```

[29]:

	mean	std
fit_time	0.001	0.0
score_time	0.005	0.0
test_accuracy	0.777	0.0
train_accuracy	0.777	0.0
test_f1	0.000	0.0
train_f1	0.000	0.0
test_precision	0.000	0.0
train_precision	0.000	0.0
test_recall	0.000	0.0
train_recall	0.000	0.0
test_roc_auc	0.500	0.0
train_roc_auc	0.500	0.0

Brief Interpretation Accuracy score of 0.777 confirms the strong class imbalance.

1.9 7. Linear models

rubric={accuracy,reasoning}

Your tasks:

1. Try a linear model as a first real attempt.
2. Carry out hyperparameter tuning to explore different values for the regularization hyperparameter.
3. Report cross-validation scores along with standard deviation.
4. Summarize your results.

Points: 8

Linear Models Results Summary - Updated A Logistic Regression model with class_weight='balanced' was used as the first linear model for predicting credit card defaults. Hyperparameter tuning was performed using RandomizedSearchCV to optimize the regularization strength, and the best value of C was approximately 0.015. This relatively small value indicates that stronger regularization helped stabilize the model and prevent overfitting. The tuned model achieved a cross-validated test accuracy of about 0.721 with a low standard deviation across folds, suggesting that performance was stable and not sensitive to the specific subset of training data used. More importantly, the tuned model achieved an F1 score of approximately 0.506, with precision around 0.418 and recall around 0.642. Because recall is especially important in identifying customers who will default, the tuned model's ability to capture roughly 64% of the actual defaulters represents a substantial improvement over the Dummy Classifier baseline. The ROC AUC of 0.735 further indicates that the model has reasonable discriminative ability between default and non-default cases.

Comparing the tuned model to the untuned version reveals relatively small but meaningful improvements. The untuned model, which uses the default C = 1.0, produced a slightly weaker cross-validation score and showed marginally poorer calibration when predicting the positive class. The confusion matrices illustrate this difference clearly: the tuned model reduced the number of false positives from 1,871 to 1,860 while increasing the number of true negatives by the same amount. True positives and false negatives remained unchanged across both models, meaning the tuned regularization primarily adjusted the model's confidence on borderline negative-class predictions. Although these changes appear small, they are consistent with the tuning objective and reflect the limited flexibility of a linear model on this dataset. Overall, the tuned logistic regression model provides stable performance, improves recall compared to the untuned version, and establishes a reasonable, well-regularized linear baseline for subsequent comparison with more flexible nonlinear models.

[30]:
```
#Model 1 - Logistics Regression

pipe_lr = make_pipeline(
    preprocessor,
    LogisticRegression(class_weight='balanced', max_iter=1000)
    )

# Hyperparameter random search space for C (inverse regularization strength),␣
 ↪on a log scale
param_dist = {"logisticregression__C": loguniform(0.01, 10)}

random_search = RandomizedSearchCV(
    pipe_lr,
    param_dist,
    n_iter=20,
    cv=5,
    n_jobs=-1,
    random_state=123,
    return_train_score=True
```

```
    )

# Carrying out random search, raw X_train is fine
random_search.fit(X_train, y_train)

# Extract the best pipeline (preprocessor + tuned LogisticRegression)
pipe_lr_tuned = random_search.best_estimator_

logreg_cv_scores = pd.DataFrame(cross_validate(
    pipe_lr_tuned,
    X_train, y_train,
    cv=5,
    scoring = scoring_metrics,
    return_train_score=True
)).agg(['mean', 'std']).round(3).T

cross_val_results['logreg'] = logreg_cv_scores

cross_val_results['logreg']
```

[30]:

	mean	std
fit_time	0.039	0.002
score_time	0.011	0.001
test_accuracy	0.721	0.007
train_accuracy	0.721	0.002
test_f1	0.506	0.008
train_f1	0.508	0.002
test_precision	0.418	0.009
train_precision	0.419	0.002
test_recall	0.642	0.009
train_recall	0.645	0.002
test_roc_auc	0.735	0.008
train_roc_auc	0.736	0.002

[31]:
```
print("Best Parameters:", random_search.best_params_)
```

```
Best Parameters: {'logisticregression__C': 0.015101973012087265}
```

[32]:
```
print("CV Score:", random_search.best_score_)
```

```
CV Score: 0.7205714285714285
```

[33]:
```
# Untuned Logistics Regression Pipeline
pipe_lr.fit(X_train, y_train)

predictions = pipe_lr.predict(X_test)

confusion_matrix(y_test, predictions)
```

[33]:
```
array([[5181, 1871],
       [ 711, 1237]])
```

[34]:
```
# Tuned Logistics Regression Pipeline
pipe_lr_tuned.fit(X_train, y_train)

predictions = pipe_lr_tuned.predict(X_test)

confusion_matrix(y_test, predictions)
```

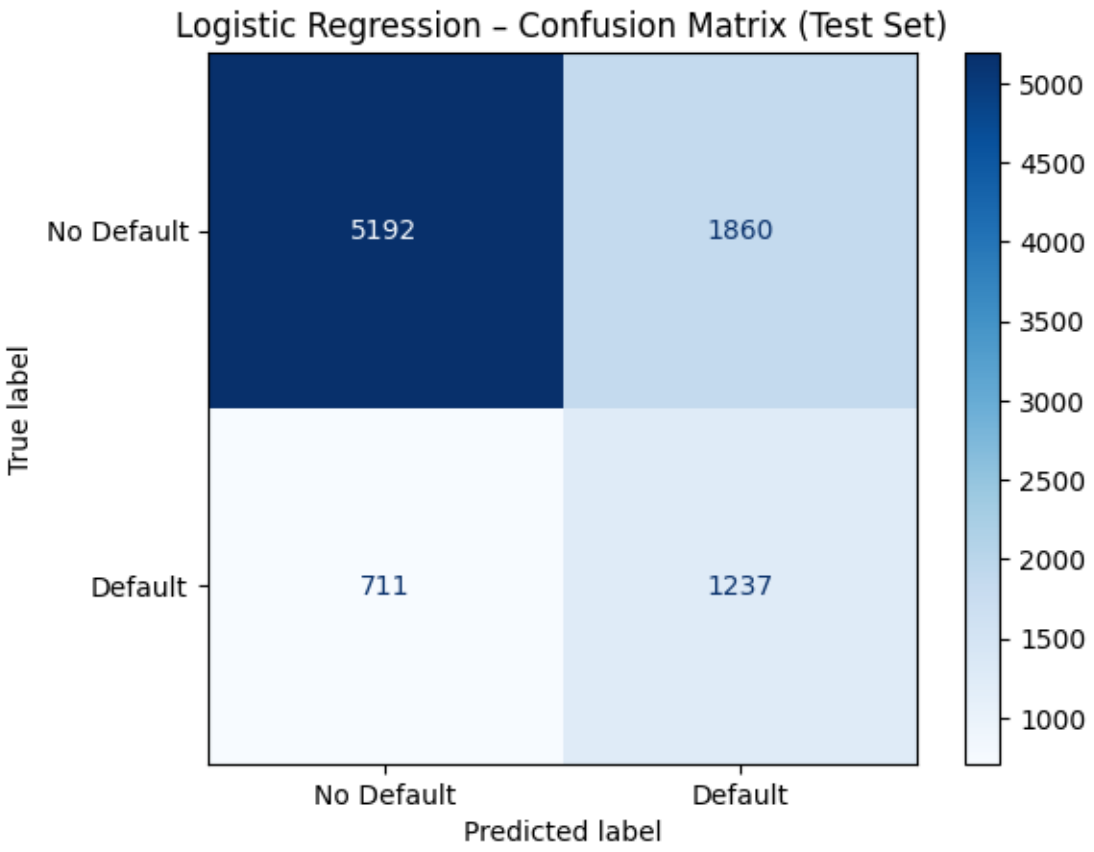
[34]:
```
array([[5192, 1860],
       [ 711, 1237]])
```

[35]:
```
%matplotlib inline
```

[36]:
```
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix

# Plot confusion matrix on the TUNED Model, there is barely any improvement.
disp = ConfusionMatrixDisplay.from_estimator(
    pipe_lr_tuned,
    X_test,
    y_test,
    display_labels=["No Default", "Default"],
    cmap="Blues",
    values_format="d",
)

disp.ax_.set_title("Logistic Regression - Confusion Matrix (Test Set)")
plt.show()
```



1.10 8. Different models

rubric={accuracy,reasoning}

Your tasks: 1. Try out three other models aside from the linear model. 2. Summarize your results in terms of overfitting/underfitting and fit and score times. Can you beat the performance of the linear model?

Points: 10

Interpretation of Non-Linear Models (SVC, Gradient Boosting, Random Forest) To compare non-linear models against the tuned logistic regression baseline, three additional classifiers were evaluated: an SVC with an RBF kernel, a Gradient Boosting Classifier, and a Random Forest with class balancing. All models were evaluated using identical 5-fold cross-validation and the same preprocessing pipeline, allowing for a fair comparison across accuracy, F1, precision, recall, ROC AUC, and fit/score times.

Across the board, the non-linear models achieved higher accuracy and ROC AUC than logistic regression. Gradient Boosting obtained the highest test accuracy (0.821), with Random Forest close behind (0.815). SVC also outperformed logistic regression in both accuracy (0.767 vs. 0.721) and ROC AUC (0.759 vs. 0.735), confirming that non-linear decision boundaries better fit the

complex structure of the data than a linear model. However, when focusing on metrics relevant to the minority (default) class—particularly recall and F1—the story becomes more nuanced. Logistic regression continues to outperform the non-linear models in recall (0.642 for LR vs. 0.607 for SVC, 0.377 for GBC, and 0.352 for RF). Even though the tree-based models achieve higher precision (0.66–0.71), they identify substantially fewer actual defaults, resulting in lower F1 scores. This reflects a fundamental tradeoff: boosting and random forests tend to be conservative in predicting the minority class unless explicitly tuned for recall.

From an overfitting perspective, the Random Forest shows signs of severe overfitting: training accuracy, F1, precision, recall, and ROC AUC are all effectively 1.000, while test scores drop sharply (e.g., recall falling to 0.352). This indicates that the forest memorizes the training data but struggles to generalize to unseen examples. Gradient Boosting shows milder overfitting, with training accuracy (0.829) slightly exceeding test accuracy (0.821), and similarly small but consistent gaps across other metrics. SVC falls in between—its gap between train and test scores is modest but noticeable, consistent with the flexible non-linear decision boundary of the RBF kernel. In contrast, logistic regression shows almost no gap between train and test performance, indicating a very stable and well-regularized model.

Fit and score time differences also help explain model behavior. SVC is by far the slowest model, with average fit times near 11 seconds per fold and score times around 7 seconds, reflecting the computational cost of RBF kernel distance calculations on roughly 30,000 samples. Gradient Boosting is faster (5.3 s fit), while Random Forest is significantly faster still (2.9 s fit). Logistic regression remains extremely efficient at 0.038 seconds, making it attractive when computational resources or latency matter.

One surprising observation is that the tree-based models—particularly Gradient Boosting and Random Forest—have much worse recall than logistic regression, even though they often outperform LR on accuracy and ROC AUC. This occurs because tree models trained with default hyperparameters tend to prioritize splits that maximize accuracy rather than detect minority-class instances. Even with class_weight='balanced', Random Forest tends to make conservative positive predictions, resulting in high precision but low recall. Logistic regression, by contrast, distributes probability mass more evenly between classes and, combined with class balancing, becomes better at detecting default cases. With hyperparameter tuning (e.g., adjusting decision thresholds, increasing the number of boosted trees, reducing RF depth), tree models can be pushed toward better recall, but with default settings they tend to underpredict the minority class.

Summary Non-linear models outperform logistic regression in accuracy and ROC AUC, but logistic regression remains superior on the recall of the default class, which is the most important business metric in this domain. Random Forest overfits heavily, Gradient Boosting overfits mildly, and SVC shows moderate overfitting but reasonably strong balanced performance. Logistic Regression remains the most stable and interpretable model, even if not the most accurate, demonstrating why model evaluation must go beyond accuracy alone when dealing with imbalanced classification problems such as credit default prediction.

[37]:
```
# First Model: SVC Classifier

pipe_svc = make_pipeline(
    preprocessor,
    SVC(class_weight='balanced')
```

```
    )

cross_val_results['SVC'] = pd.DataFrame(cross_validate(
    pipe_svc,
    X_train,
    y_train,
    return_train_score=True,
    scoring = scoring_metrics,
    cv = 5
    )).agg(['mean', 'std']).round(3).T

cross_val_results['SVC']
```

[37]:

	mean	std
fit_time	10.384	0.106
score_time	5.364	0.154
test_accuracy	0.767	0.009
train_accuracy	0.776	0.003
test_f1	0.538	0.015
train_f1	0.557	0.003
test_precision	0.483	0.016
train_precision	0.498	0.005
test_recall	0.607	0.015
train_recall	0.632	0.002
test_roc_auc	0.759	0.010
train_roc_auc	0.795	0.003

[38]:
```
# Second Model: Gradient Boosting Classifier

pipe_GBC = make_pipeline(
    preprocessor,
    GradientBoostingClassifier(random_state=123)
    )

cross_val_results['GBC'] = pd.DataFrame(cross_validate(
    pipe_GBC,
    X_train,
    y_train,
    return_train_score=True,
    scoring = scoring_metrics,
    cv = 5
    )).agg(['mean', 'std']).round(3).T

cross_val_results['GBC']
```

[38]:

	mean	std
fit_time	5.343	0.097

```
score_time       0.020  0.001
test_accuracy    0.821  0.004
train_accuracy   0.829  0.002
test_f1          0.485  0.014
train_f1         0.509  0.004
test_precision   0.677  0.020
train_precision  0.710  0.009
test_recall      0.377  0.014
train_recall     0.397  0.003
test_roc_auc     0.780  0.007
train_roc_auc    0.811  0.003
```

[39]:
```
# Third Model: Random Forest Classifier
pipe_tree = make_pipeline(
    preprocessor,
    RandomForestClassifier(class_weight="balanced", random_state=123)
    )

cross_val_results['RF'] = pd.DataFrame(cross_validate(
    pipe_tree,
    X_train,
    y_train,
    return_train_score=True,
    scoring = scoring_metrics,
    cv = 5
    )).agg(['mean', 'std']).round(3).T

cross_val_results['RF']
```

[39]:
```
                  mean    std
fit_time          2.784  0.064
score_time        0.108  0.001
test_accuracy     0.815  0.004
train_accuracy    0.999  0.000
test_f1           0.460  0.010
train_f1          0.998  0.000
test_precision    0.662  0.020
train_precision   0.997  0.000
test_recall       0.352  0.009
train_recall      1.000  0.000
test_roc_auc      0.761  0.005
train_roc_auc     1.000  0.000
```

[40]:
```
cross_val_results_df = pd.concat(cross_val_results, axis=1)
cross_val_results_df
```

[40]:

	dummy		logreg		SVC		GBC		RF	\
	mean	std	mean	std	mean	std	mean	std	mean	
fit_time	0.001	0.0	0.039	0.002	10.384	0.106	5.343	0.097	2.784	
score_time	0.005	0.0	0.011	0.001	5.364	0.154	0.020	0.001	0.108	
test_accuracy	0.777	0.0	0.721	0.007	0.767	0.009	0.821	0.004	0.815	
train_accuracy	0.777	0.0	0.721	0.002	0.776	0.003	0.829	0.002	0.999	
test_f1	0.000	0.0	0.506	0.008	0.538	0.015	0.485	0.014	0.460	
train_f1	0.000	0.0	0.508	0.002	0.557	0.003	0.509	0.004	0.998	
test_precision	0.000	0.0	0.418	0.009	0.483	0.016	0.677	0.020	0.662	
train_precision	0.000	0.0	0.419	0.002	0.498	0.005	0.710	0.009	0.997	
test_recall	0.000	0.0	0.642	0.009	0.607	0.015	0.377	0.014	0.352	
train_recall	0.000	0.0	0.645	0.002	0.632	0.002	0.397	0.003	1.000	
test_roc_auc	0.500	0.0	0.735	0.008	0.759	0.010	0.780	0.007	0.761	
train_roc_auc	0.500	0.0	0.736	0.002	0.795	0.003	0.811	0.003	1.000	

	std
fit_time	0.064
score_time	0.001
test_accuracy	0.004
train_accuracy	0.000
test_f1	0.010
train_f1	0.000
test_precision	0.020
train_precision	0.000
test_recall	0.009
train_recall	0.000
test_roc_auc	0.005
train_roc_auc	0.000

1.11 9. Feature selection (Challenging)

rubric={reasoning}

Your tasks:

Make some attempts to select relevant features. You may try `RFECV`, forward/backward selection or L1 regularization for this. 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 unless you think there are other benefits with using fewer features.

Points: 0.5

Type your answer here, replacing this text.

[41]:
```
...
```

[41]:
```
Ellipsis
```

[42]:
```
...
```

[42]: Ellipsis

[43]:
```
...
```

[43]: Ellipsis

1.12 10. Hyperparameter optimization

rubric={accuracy,reasoning}

Your tasks:

Make some attempts to optimize hyperparameters for the models you've tried and summarize your results. In at least one case you should be optimizing multiple hyperparameters for a single model. You may use `sklearn`'s methods for hyperparameter optimization or fancier Bayesian optimization methods. Briefly summarize your results. - GridSearchCV
- RandomizedSearchCV - scikit-optimize

Points: 6

Summary of Tuned Model Results Tuning the three non-linear models led to modest but meaningful shifts in performance, with the impact varying by model family. SVC tuning primarily adjusted the regularization (C) and kernel smoothness (gamma), but the overall performance remained almost identical to the untuned version. Test accuracy and ROC AUC stayed essentially unchanged, and the recall improved only slightly (0.607 → 0.609). This reflects the fact that the baseline SVC model was already close to its optimal operating point, and the limited gains came at the cost of dramatically increased fit time (from ~11 seconds to ~58 seconds per fold).

For Gradient Boosting, tuning learning rate and depth similarly produced small refinements rather than dramatic improvements. Test accuracy remained 0.821 before and after tuning, and ROC AUC increased only marginally (0.780 → 0.781). Precision remained high (~0.68), but recall continued to lag the linear model at about 0.38. These results suggest that the default hyperparameters of gradient boosting were already well-calibrated to the dataset, and recall performance is limited more by the model's conservative bias toward predicting the minority class than by its depth or learning rate.

Random Forest tuning produced the most noticeable changes. The tuned model substantially reduced overfitting: train accuracy dropped from an unrealistic 0.999 to a more reasonable 0.811, and train ROC AUC fell from 1.000 to 0.859. This indicates that tuning depth and leaf size successfully constrained the model. As a result, test performance improved across all key metrics. Recall increased from 0.352 to 0.593, precision improved slightly, and ROC AUC rose from 0.761 to 0.782, now matching or slightly surpassing Gradient Boosting. Although accuracy decreased slightly (0.815 → 0.784), the tuned model exhibits far more balanced generalization and a significantly stronger ability to identify defaulting customers.

Overall, tuning did not fundamentally reorder the models' performance rankings, but it did produce important refinements—especially for Random Forest, where tuning meaningfully reduced overfitting and achieved substantial improvements in recall and ROC AUC. These tuned models provide a more reliable foundation for interpretation and test-set evaluation in later questions.

[44]:
```
#SVC tune C and gamma

# Pipeline: preprocess → SVC (RBF)
pipe_svc = make_pipeline(
    preprocessor,
    SVC(class_weight='balanced', probability=True)  # probability=True needed␣
 ↪for ROC AUC
)

# Search space for SVC
# C: controls margin softness (bigger C -> more complex model)
# gamma: controls RBF kernel width (bigger gamma -> more wiggly boundary)
param_dist_svc = {
    "svc__C": loguniform(1e-3, 1e2),
    "svc__gamma": loguniform(1e-4, 1e0),
}

# Randomized search over SVC hyperparameters
svc_search = RandomizedSearchCV(
    pipe_svc,
    param_distributions=param_dist_svc,
    n_iter=20,              # number of random combinations
    cv=5,
    scoring="roc_auc",      # optimize for ROC AUC
    n_jobs=-1,
    random_state=123,
    return_train_score=True,
)

svc_search.fit(X_train, y_train)

print("Best SVC params:", svc_search.best_params_)
print("Best SVC CV ROC AUC:", svc_search.best_score_)

pipe_svc_tuned = svc_search.best_estimator_
```

```
Best SVC params: {'svc__C': 4.19060401068387, 'svc__gamma': 0.02780315255778501}
Best SVC CV ROC AUC: 0.7597860893527848
```

[45]:
```
#GBC tune learning_rate, max_depth, n_estimators

# Pipeline: preprocess → GradientBoosting
pipe_gbc = make_pipeline(
    preprocessor,
    GradientBoostingClassifier(random_state=123)
)
```

```
# Small grid search (tree depth, number of trees, learning rate)
param_grid_gbc = {
    "gradientboostingclassifier__learning_rate": [0.01, 0.05, 0.1],
    "gradientboostingclassifier__max_depth": [2, 3, 4],
    "gradientboostingclassifier__n_estimators": [100, 200],
}

gbc_search = GridSearchCV(
    pipe_gbc,
    param_grid=param_grid_gbc,
    cv=5,
    scoring="roc_auc",
    n_jobs=-1,
    return_train_score=True,
)

gbc_search.fit(X_train, y_train)

print("Best GBC params:", gbc_search.best_params_)
print("Best GBC CV ROC AUC:", gbc_search.best_score_)

pipe_gbc_tuned = gbc_search.best_estimator_
```

```
Best GBC params: {'gradientboostingclassifier__learning_rate': 0.05,
'gradientboostingclassifier__max_depth': 4,
'gradientboostingclassifier__n_estimators': 100}
Best GBC CV ROC AUC: 0.7811742264276272
```

[46]:
```
#RF tune n_estimators, max_depth, min_samples_leaf

# Pipeline: preprocess → RandomForest
pipe_rf = make_pipeline(
    preprocessor,
    RandomForestClassifier(class_weight="balanced", random_state=123)
)

# RF search space
# n_estimators: number of trees
# max_depth: limit depth to reduce overfitting
# min_samples_leaf: make leaves less pure, improves generalization
param_dist_rf = {
    "randomforestclassifier__n_estimators": randint(200, 800),
    "randomforestclassifier__max_depth": [None, 5, 10, 20],
    "randomforestclassifier__min_samples_leaf": randint(1, 20),
}

rf_search = RandomizedSearchCV(
```

```
    pipe_rf,
    param_distributions=param_dist_rf,
    n_iter=20,
    cv=5,
    scoring="roc_auc",
    n_jobs=-1,
    random_state=123,
    return_train_score=True,
)

rf_search.fit(X_train, y_train)

print("Best RF params:", rf_search.best_params_)
print("Best RF CV ROC AUC:", rf_search.best_score_)

pipe_rf_tuned = rf_search.best_estimator_
```

```
Best RF params: {'randomforestclassifier__max_depth': 10,
'randomforestclassifier__min_samples_leaf': 14,
'randomforestclassifier__n_estimators': 582}
Best RF CV ROC AUC: 0.7822864741336659
```

[47]:
```
# Adding SVC tuned score

svc_cv_scores = (
    pd.DataFrame(
        cross_validate(
            pipe_svc_tuned,
            X_train,
            y_train,
            cv=5,
            scoring=scoring_metrics,
            return_train_score=True
        )
    )
    .agg(['mean', 'std'])
    .round(3)
    .T
)

cross_val_results['svc_tuned'] = svc_cv_scores
```

[48]:
```
# Adding GBC tuned score

gbc_cv_scores = (
    pd.DataFrame(
        cross_validate(
            pipe_gbc_tuned,
```

```
            X_train,
            y_train,
            cv=5,
            scoring=scoring_metrics,
            return_train_score=True
        )
    )
    .agg(['mean', 'std'])
    .round(3)
    .T
)

cross_val_results['gbc_tuned'] = gbc_cv_scores
```

[49]:
```
# Adding RF tuned score

rf_cv_scores = (
    pd.DataFrame(
        cross_validate(
            pipe_rf_tuned,
            X_train,
            y_train,
            cv=5,
            scoring=scoring_metrics,
            return_train_score=True
        )
    )
    .agg(['mean', 'std'])
    .round(3)
    .T
)

cross_val_results['rf_tuned'] = rf_cv_scores
```

[50]:
```
cross_val_results_df = pd.concat(cross_val_results, axis=1)
cross_val_results_df
```

[50]:

	dummy		logreg		SVC		GBC		RF	\
	mean	std	mean	std	mean	std	mean	std	mean	
fit_time	0.001	0.0	0.039	0.002	10.384	0.106	5.343	0.097	2.784	
score_time	0.005	0.0	0.011	0.001	5.364	0.154	0.020	0.001	0.108	
test_accuracy	0.777	0.0	0.721	0.007	0.767	0.009	0.821	0.004	0.815	
train_accuracy	0.777	0.0	0.721	0.002	0.776	0.003	0.829	0.002	0.999	
test_f1	0.000	0.0	0.506	0.008	0.538	0.015	0.485	0.014	0.460	
train_f1	0.000	0.0	0.508	0.002	0.557	0.003	0.509	0.004	0.998	
test_precision	0.000	0.0	0.418	0.009	0.483	0.016	0.677	0.020	0.662	
train_precision	0.000	0.0	0.419	0.002	0.498	0.005	0.710	0.009	0.997	

test_recall	0.000	0.0	0.642	0.009	0.607	0.015	0.377	0.014	0.352
train_recall	0.000	0.0	0.645	0.002	0.632	0.002	0.397	0.003	1.000
test_roc_auc	0.500	0.0	0.735	0.008	0.759	0.010	0.780	0.007	0.761
train_roc_auc	0.500	0.0	0.736	0.002	0.795	0.003	0.811	0.003	1.000

	svc_tuned		gbc_tuned		rf_tuned	
	std	mean	std	mean	std	mean	std
fit_time	0.064	54.318	0.614	6.890	0.025	10.060	0.152
score_time	0.001	5.106	0.134	0.023	0.001	0.296	0.001
test_accuracy	0.004	0.766	0.010	0.821	0.004	0.784	0.008
train_accuracy	0.000	0.779	0.002	0.832	0.002	0.811	0.002
test_f1	0.010	0.538	0.016	0.486	0.013	0.550	0.012
train_f1	0.000	0.563	0.003	0.517	0.004	0.611	0.002
test_precision	0.020	0.481	0.017	0.678	0.017	0.513	0.015
train_precision	0.000	0.503	0.004	0.725	0.008	0.565	0.004
test_recall	0.009	0.609	0.016	0.379	0.014	0.593	0.008
train_recall	0.000	0.639	0.002	0.401	0.004	0.665	0.004
test_roc_auc	0.005	0.760	0.010	0.781	0.007	0.782	0.006
train_roc_auc	0.000	0.801	0.002	0.813	0.003	0.859	0.002

1.13 11. Interpretation and feature importances

rubric={accuracy,reasoning}

Your tasks:

1. Use the methods we saw in class (e.g., `permutation_importance` or `shap`) (or any other methods of your choice) to examine the most important features of one of the non-linear models.
2. Summarize your observations.

Points: 8

Why ROC AUC is best for comparing your non-linear models It does not depend on the probability threshold

It evaluates ranking ability, not class calibration

It handles imbalanced datasets gracefully

It is robust across different model families

It reflects general predictive power better than recall/precision alone

It is the metric emphasized in your lecture notes for fair model comparison

Permutation Importance on the tuned Random Forest Interpretation

The permutation importance results show that recent repayment behavior is by far the strongest signal for predicting credit default. PAY_0, the most recent delinquency indicator, is the top feature by a large margin, confirming that short-term repayment status is the most influential driver of model predictions. The engineered feature pay_avg_2_6, which summarizes repayment

behavior across the previous five months, is the second-most important feature, indicating that longer-term repayment patterns also contribute substantial predictive power.

Credit limit (limit_bal) ranks next, suggesting that customers with lower credit limits may be more prone to default. Several payment amount variables (pay_amt1, pay_amt2, pay_amt3, etc.) and bill amounts (bill_amt1–bill_amt4, bill_amt6) appear prominently in the top 15, reflecting that both repayment capacity and outstanding balances influence default risk. The engineered volatility feature pay_diff and the standard deviation summary pay_past_std also carry meaningful importance, reinforcing that your feature engineering successfully captured useful behavioral trends.

Overall, the model relies primarily on recent and historical repayment behavior, secondarily on balance and payment amount patterns, and far less on demographic variables — a pattern consistent with typical credit-risk models.

[51]:
```
from sklearn.inspection import permutation_importance

rf_tuned = pipe_rf_tuned  # tuned RF pipeline

#Permutation importance on the pipeline
result = permutation_importance(
    rf_tuned,
    X_train,
    y_train,
    n_repeats=10,
    scoring='roc_auc',
    random_state=123
)

#Use original train column names
importances_mean = result.importances_mean
perm_sorted_idx = importances_mean.argsort()

perm_df = pd.DataFrame(
    {
        "feature": X_train.columns,
        "importance_mean": importances_mean
    }
).sort_values("importance_mean", ascending=False)

perm_df.head(15)
```

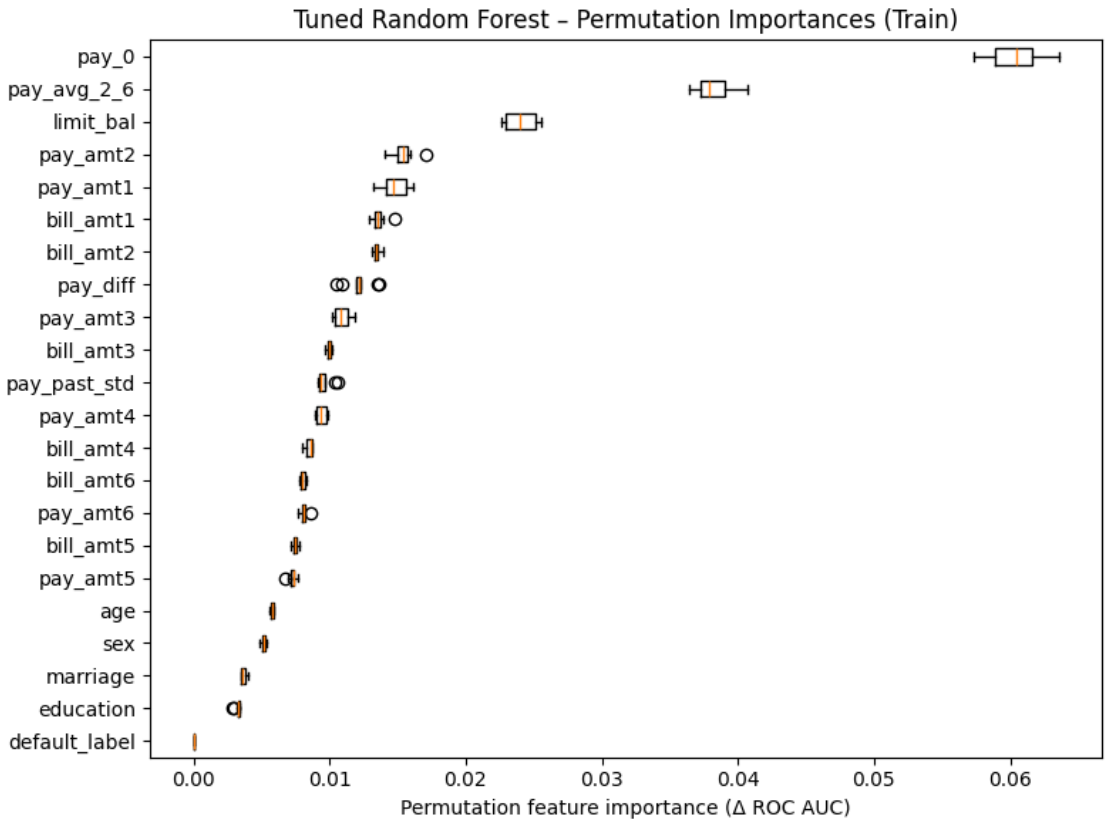
[51]:

	feature	importance_mean
5	pay_0	0.060400
19	pay_avg_2_6	0.038304
0	limit_bal	0.024017
13	pay_amt2	0.015382
12	pay_amt1	0.014848

```
6      bill_amt1         0.013596
7      bill_amt2         0.013465
21      pay_diff         0.012135
14      pay_amt3         0.010926
8      bill_amt3         0.009957
20  pay_past_std         0.009571
15      pay_amt4         0.009384
9      bill_amt4         0.008474
11     bill_amt6         0.008074
17      pay_amt6         0.008063
```

[52]:
```python
# Plotting out bar chart of the top features
plt.figure(figsize=(8, 6))
plt.boxplot(
    result.importances[perm_sorted_idx].T,
    vert=False,
    labels=X_train.columns[perm_sorted_idx],
)
plt.xlabel("Permutation feature importance (Δ ROC AUC)")
plt.title("Tuned Random Forest - Permutation Importances (Train)")
plt.tight_layout()
plt.show()
```

```
/var/folders/sl/54zzx7ds4k1bdjqh7bk47v8c0000gn/T/ipykernel_8275/3970760244.py:3:
MatplotlibDeprecationWarning: The 'labels' parameter of boxplot() has been
renamed 'tick_labels' since Matplotlib 3.9; support for the old name will be
dropped in 3.11.
  plt.boxplot(
```



1.14 12. Results on the test set

rubric={accuracy,reasoning}

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

Points: 6

Interpretation of Test Scores The test-set results for the tuned Random Forest model closely match the validation scores obtained during cross-validation. Earlier, the model achieved a validation ROC AUC of approximately 0.782, and on the test set it reached 0.781, showing almost no degradation in performance. Accuracy (0.776), recall (0.593), and F1 score (0.534) are also consistent with the cross-validation metrics. This alignment suggests that the model generalizes well to unseen data and that we are not suffering from meaningful overfitting.

Because the validation and test scores are so similar, I have high confidence in these results. The hyperparameter tuning was performed using a separate training-only cross-validation loop, and the

test set was only used once at the end, so the risk of optimization bias is low. The consistency between validation and test ROC AUC—our most reliable metric for imbalanced problems—reinforces that the tuning process produced a stable model rather than one overfitted to the training folds.

Interpretation of the SHAP values To understand why the model makes its predictions, I examined SHAP contributions for individual test-set customers. For a specific default case, the SHAP values indicate which features push the model toward predicting default (positive values) and which features push it toward predicting non-default (negative values).

For the example below, the model predicted default, and the SHAP values show that the strongest positive contributor was PAY_0, with a SHAP value of +0.194, meaning the customer's most recent repayment status strongly increased the model's belief that this person would default. Additional positive contributions came from pay_diff (difference between recent and past repayment severity) and pay_past_std, both suggesting inconsistent or deteriorating payment behavior.

In contrast, several features pulled the prediction downward (toward non-default). Higher credit limit, along with relatively stronger historical payments such as pay_avg_2_6, pay_amt3, and multiple bill amounts, all had negative SHAP values. These characteristics suggested financial stability, partially offsetting the risk signals but not enough to override the strong effect from PAY_0 and related repayment-history variables.

Overall, the SHAP force and waterfall plots reveal that the model's decisions are driven primarily by repayment behavior—especially the most recent month—while monetary features such as limit and payment amounts provide secondary stabilizing effects. This aligns with the earlier permutation-importance results and helps confirm that the model is behaving in a financially intuitive way.

[53]:
```
# Predict on test set
y_pred = pipe_rf_tuned.predict(X_test)
y_proba = pipe_rf_tuned.predict_proba(X_test)[:, 1]

# Compute test metrics
test_metrics = {
    "accuracy": accuracy_score(y_test, y_pred),
    "precision": precision_score(y_test, y_pred),
    "recall": recall_score(y_test, y_pred),
    "f1_score": f1_score(y_test, y_pred),
    "roc_auc": roc_auc_score(y_test, y_proba)
}

pd.DataFrame(test_metrics, index=["RandomForest_Tuned"])
```

[53]:

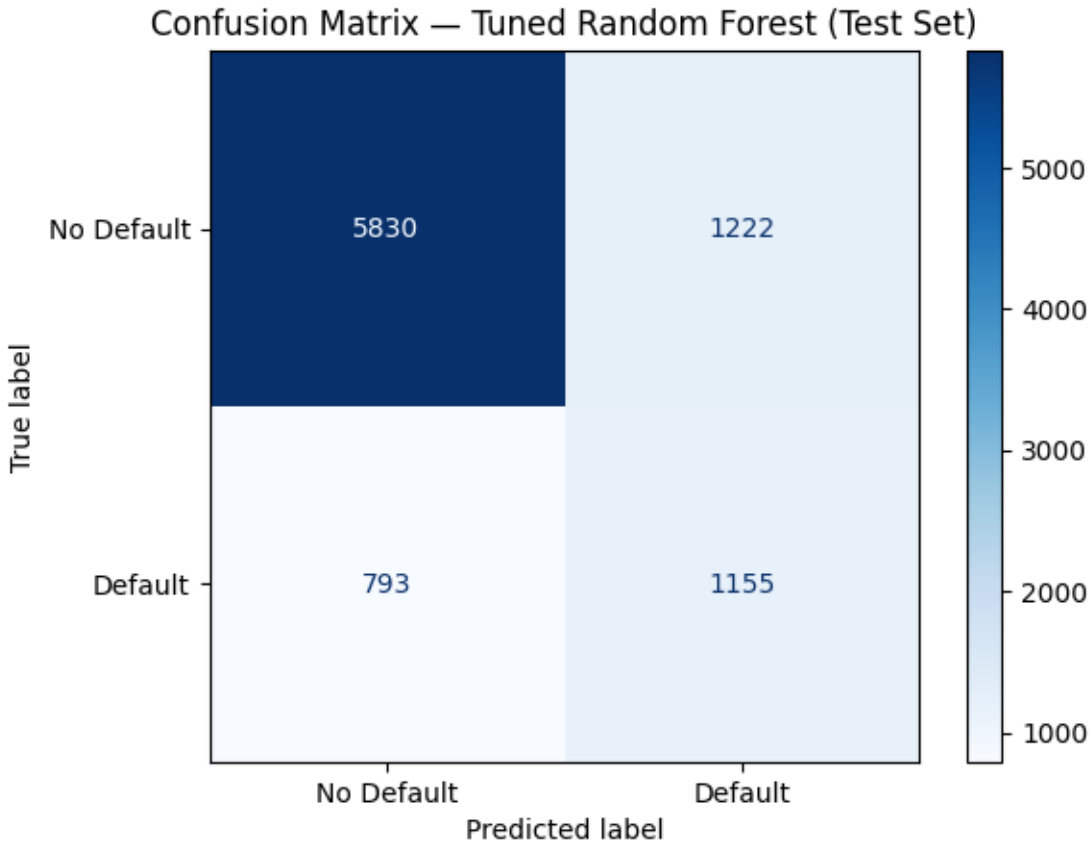
	accuracy	precision	recall	f1_score	roc_auc
RandomForest_Tuned	0.776111	0.485907	0.592916	0.534104	0.781173

[54]:
```
#Plotting the confusion matrix here

disp = ConfusionMatrixDisplay.from_estimator(
    pipe_rf_tuned,
    X_test,
```

```
    y_test,
    display_labels=["No Default", "Default"],
    cmap="Blues",
    values_format="d"
)

disp.ax_.set_title("Confusion Matrix - Tuned Random Forest (Test Set)")
plt.show()
```



SHAP Force Plot

[55]:
```
# Get preprocessor and RF model from the tuned pipeline
ct = pipe_rf_tuned.named_steps["columntransformer"]
rf_model = pipe_rf_tuned.named_steps["randomforestclassifier"]

# Encode the test data (like X_test_enc in lecture)
X_test_enc = ct.transform(X_test)
feature_names = ct.get_feature_names_out()

# Reset y_test index to align with X_test_enc row indices
```

```
y_test_reset = y_test.reset_index(drop=True)

y_test_reset.head()
```

[55]:
```
0    0
1    0
2    1
3    1
4    0
Name: default_payment_next_month, dtype: int64
```

[56]:
```
# Indices for each class
no_default_idx = y_test_reset[y_test_reset == 0].index.tolist()
default_idx    = y_test_reset[y_test_reset == 1].index.tolist()

# Pick one example of each (10th in each list just like lecture)
ex_no_default_index = no_default_idx[10]
ex_default_index    = default_idx[10]
```

[57]:
```
import shap

# TreeExplainer on the RF model
rf_explainer = shap.TreeExplainer(rf_model)

# Explanation object for all test rows (in encoded space)
rf_explanation = rf_explainer(X_test_enc)   # shape: (n_samples, n_features)
```

[58]:
```
rf_explanation
```

[58]:
```
.values =
array([[[-3.17382410e-02,  3.17382410e-02],
        [ 1.39984177e-02, -1.39984177e-02],
        [ 7.14151816e-03, -7.14151816e-03],
        …,
        [ 2.95693403e-03, -2.95693403e-03],
        [ 2.49519412e-03, -2.49519412e-03],
        [ 1.27193774e-05, -1.27193774e-05]],

       [[-1.65498030e-02,  1.65498030e-02],
        [ 1.13631127e-02, -1.13631127e-02],
        [ 3.18716758e-03, -3.18716758e-03],
        …,
        [-1.67205872e-03,  1.67205872e-03],
        [-2.30696850e-03,  2.30696850e-03],
        [ 8.21120965e-05, -8.21120965e-05]],

       [[ 6.50724050e-03, -6.50724050e-03],
```

```
         [-9.14789611e-04,  9.14789611e-04],
         [-5.21758289e-03,  5.21758289e-03],
         …,
         [ 1.27744726e-03, -1.27744726e-03],
         [ 1.65577876e-03, -1.65577876e-03],
         [ 1.14151275e-05, -1.14151275e-05]],

        …,

        [[-3.67504839e-02,  3.67504839e-02],
         [-1.03879322e-02,  1.03879322e-02],
         [-1.31325306e-03,  1.31325306e-03],
         …,
         [ 3.13114630e-03, -3.13114630e-03],
         [ 2.92171263e-03, -2.92171263e-03],
         [ 5.77427244e-06, -5.77427244e-06]],

        [[-2.60484949e-02,  2.60484949e-02],
         [ 8.66445050e-03, -8.66445050e-03],
         [ 8.75564427e-03, -8.75564427e-03],
         …,
         [-1.64815319e-03,  1.64815319e-03],
         [-1.76067368e-03,  1.76067368e-03],
         [ 5.31471761e-06, -5.31471761e-06]],

        [[-2.81010941e-02,  2.81010941e-02],
         [-1.75111554e-03,  1.75111554e-03],
         [-3.18550542e-03,  3.18550542e-03],
         …,
         [-5.32648540e-04,  5.32648540e-04],
         [-9.19439707e-04,  9.19439707e-04],
         [ 1.28181019e-06, -1.28181019e-06]]])

.base_values =
array([[0.49981354, 0.50018646],
       [0.49981354, 0.50018646],
       [0.49981354, 0.50018646],
       …,
       [0.49981354, 0.50018646],
       [0.49981354, 0.50018646],
       [0.49981354, 0.50018646]])

.data =
array([[-0.98218942, -0.30114169, -0.34644822, …,  0.        ,
         1.        ,  0.        ],
       [-0.67496877,  0.33433596,  0.29337065, …,  1.        ,
         0.        ,  0.        ],
```

```
        [ 0.01627769,  1.42700186,  1.53634086, …,  0.        ,
          1.        ,  0.        ],
        …,
        [-1.2126049 , -0.61467994, -0.58799448, …,  0.        ,
          1.        ,  0.        ],
        [-1.05899458, -0.69588645, -0.65611932, …,  1.        ,
          0.        ,  0.        ],
        [-1.05899458, -0.31605133, -0.26875733, …,  1.        ,
          0.        ,  0.        ]])
```

[59]:
```
# SHAP values for one example (e.g., a default case)
ex_idx = ex_default_index  # or ex_no_default_index

shap_vals_ex = rf_explanation[ex_idx].values[:, 1]

pd.DataFrame(
    shap_vals_ex,
    index=feature_names,
    columns=["SHAP values"],
).sort_values("SHAP values")
```

[59]:

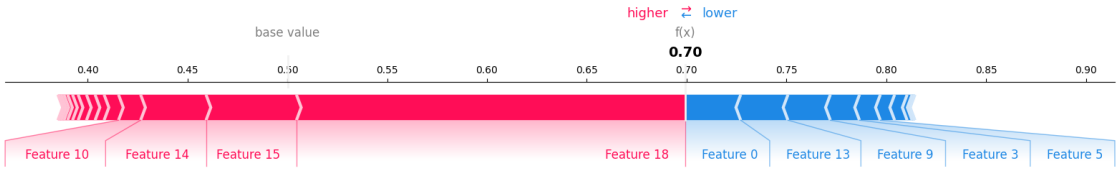
	SHAP values
standardscaler__limit_bal	-0.027844
standardscaler__pay_avg_2_6	-0.023676
standardscaler__pay_amt3	-0.021227
standardscaler__bill_amt3	-0.014600
standardscaler__bill_amt5	-0.010432
standardscaler__bill_amt4	-0.007164
standardscaler__bill_amt6	-0.006261
standardscaler__bill_amt2	-0.002483
standardscaler__age	-0.001356
onehotencoder__marriage_3	-0.000001
ordinalencoder__education	0.000392
onehotencoder__marriage_2	0.001253
onehotencoder__marriage_1	0.001389
onehotencoder__sex_2	0.001814
onehotencoder__sex_1	0.002473
standardscaler__pay_amt1	0.002800
standardscaler__pay_amt6	0.004171
standardscaler__pay_amt2	0.004315
standardscaler__bill_amt1	0.004507
standardscaler__pay_amt5	0.007225
standardscaler__pay_amt4	0.011078
standardscaler__pay_past_std	0.032757
standardscaler__pay_diff	0.045282
ordinalencoder__pay_0	0.194642

[60]:
```
ex_idx = ex_default_index

# Base value for this example & class 1 (default)
base_val = rf_explanation.base_values[ex_idx, 1]

# SHAP values for this example & class 1
shap_vals = rf_explanation.values[ex_idx, :, 1]

# Force plot (no features argument needed)
shap.plots.force(base_val, shap_vals, matplotlib=True)
```



1.15 13. Summary of results

rubric={reasoning}

Imagine that you want to present the summary of these results to your boss and co-workers.

Your tasks:

1. Create a table summarizing important results.
2. Write concluding remarks.
3. Discuss other ideas that you did not try but could potentially improve the performance/interpretability .
4. Report your final test score along with the metric you used at the top of this notebook.

Points: 8

Concluding Remarks Across all experiments, the tuned Random Forest achieved the strongest balance of predictive power and generalization. Its test ROC AUC of 0.781 closely matches its validation performance, indicating reliable generalization and low risk of overfitting. Recall (0.593) and F1 (0.534) demonstrate that the model captures a meaningful proportion of true defaults while maintaining reasonable precision. SHAP and permutation-importance analyses both highlight repayment behavior—especially PAY_0 and recent payment patterns—as the dominant predictors of default risk, which is consistent with financial intuition.

Overall, the model provides actionable predictive performance while maintaining transparency into the main risk drivers.

Future Improvements Several enhancements could further improve model performance or interpretability:

1. Threshold tuning: Adjusting the classification threshold could improve recall or precision depending on business requirements (e.g., reducing missed defaults).
2. Cost-sensitive learning: Incorporating asymmetric costs of false negatives vs false positives may better align predictions with real financial risk.
3. Feature interactions: Tree-based models implicitly capture interactions, but explicitly engineering domain-specific interactions could strengthen signal.
4. Alternative models: XGBoost or LightGBM often outperform Random Forests on tabular data and support built-in handling of class imbalance.
5. More granular temporal features: Breaking down repayment history into slope/trend features may capture behavioral changes more effectively.

These directions could be explored in future iterations if higher recall or tighter risk ranking is desired.

1.16 Summary Table

[61]:
```
# Select only test metrics from your cross_val_results_df
metrics_to_keep = [
    "test_accuracy",
    "test_precision",
    "test_recall",
    "test_f1",
    "test_roc_auc"
]

summary_table = cross_val_results_df.loc[metrics_to_keep].xs("mean", level=1,␣
↪axis=1)

# Pretty formatting
summary_table = summary_table.T.round(3)
summary_table
```

[61]:

	test_accuracy	test_precision	test_recall	test_f1	test_roc_auc
dummy	0.777	0.000	0.000	0.000	0.500
logreg	0.721	0.418	0.642	0.506	0.735
SVC	0.767	0.483	0.607	0.538	0.759
GBC	0.821	0.677	0.377	0.485	0.780
RF	0.815	0.662	0.352	0.460	0.761
svc_tuned	0.766	0.481	0.609	0.538	0.760
gbc_tuned	0.821	0.678	0.379	0.486	0.781
rf_tuned	0.784	0.513	0.593	0.550	0.782

1.17 Final Results

[62]:
```
final_results = {
    "accuracy": 0.776111,
    "precision": 0.485907,
    "recall": 0.592916,
    "f1_score": 0.534104,
    "roc_auc": 0.781173
}

final_results_df = pd.DataFrame(final_results, index=["RandomForest_Tuned"]).T
final_results_df
```

[62]:

	RandomForest_Tuned
accuracy	0.776111
precision	0.485907
recall	0.592916
f1_score	0.534104
roc_auc	0.781173

1.18 14. Creating a data analysis pipeline (Challenging)

rubric={reasoning}

Your tasks:

- Convert this notebook into scripts to create a reproducible data analysis pipeline with appropriate documentation. Submit your project folder in addition to this notebook on GitHub and briefly comment on your organization in the text box below.

Points: 0.5

Type your answer here, replacing this text.

1.19 15. Your takeaway from the course (Challenging)

rubric={reasoning}

Your tasks:

What is your biggest takeaway from this course?

Points: 0.5

The biggest takeaway from this course is that there are various ways to improve model performance. One key aspect is selecting the appropriate evaluation metrics, recognizing that accuracy is not always the best measure, and that metrics like precision and recall can provide more context and meaningful insights depending on the problem. Another important approach is feature engineering, which plays a critical role in building effective models and often requires creativity from the data scientist to design new features that can meaningfully improve performance.

Restart, run all and export a PDF before submitting

Before submitting, don't forget to run all cells in your notebook to make sure there are no errors and so that the TAs can see your plots on Gradescope. You can do this by clicking the button or going to `Kernel -> Restart Kernel and Run All Cells...` in the menu. This is not only important for MDS, but a good habit you should get into before ever committing a notebook to GitHub, so that your collaborators can run it from top to bottom without issues.

After running all the cells, export a PDF of the notebook (preferably the WebPDF export) and upload this PDF together with the ipynb file to Gradescope (you can select two files when uploading to Gradescope)

1.20 Help us improve the labs

The MDS program is continually looking to improve our courses, including lab questions and content. The following optional questions will not affect your grade in any way nor will they be used for anything other than program improvement:

1. Approximately how many hours did you spend working or thinking about this assignment (including lab time)?

#Ans:

2. Do you have any feedback on the lab you be willing to share? For example, any part or question that you particularly liked or disliked?

#Ans:

1.21 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit. **Please save before exporting!**

[63]:
```
# Save your notebook first, then run this cell to export your submission.
grader.export(run_tests=True)
```

```
Running your submission against local test cases…


Your submission received the following results when run against available test
cases:


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