# FIN 417: QUANTITATIVE RISK MANAGEMENT PROJECT II

Credit Risk and Statistical Learning

**Philipp Mayer** **Rafael Barroso**

**SCIPER: 329758** **SCIPER: 406607**

Ingenierie Mathématique Mathématiques

École Polytechnique Fédérale de Lausanne

December 16, 2025

**Abstract**

This project investigates the quantitative modeling of credit risk in a retail loan portfolio using synthetic borrower data. Each applicant is characterized by age, income, and employment status, from which repayment probabilities are generated under two underlying data-generating mechanisms. Logistic regression and support vector machine classifiers are trained to estimate default risk and are evaluated using cross-entropy loss, ROC curves, and area-under-the-curve metrics. Building on these predictive models, several lending strategies are analyzed, including unconditional lending and selective lending based on estimated repayment probabilities. Portfolio-level profit and loss distributions are simulated under a true default model, allowing for the estimation of expected returns, Value-at-Risk, and Expected Shortfall. The results highlight the interaction between predictive accuracy, interest rate structure, and downside risk, illustrating the challenges of achieving profitability under conservative lending policies.

# Contents

# 1 Feature generation

## 1.1 Preliminaries

We begin by providing a brief overview of the methodology used to generate the data that will serve as the basis for our numerical experiments. This synthetic dataset represents the set of loan applicants considered in the analysis and is constructed in a controlled manner so that it reflects key characteristics relevant to credit risk modeling. The generated data will then be used to evaluate and compare different credit lending strategies.

We consider a training sample size of $m = 20,000$ and a test sample size of $n = 10,000$. For each borrower $i = 1, \dots, m+n$, we simulate a random feature vector

$$x^i = (x_1^i, x_2^i, x_3^i) \in \mathbb{R}^3,$$

where the three components are defined as follows:

- $x_1^i \sim \text{Unif}[18, 80]$ represents the borrower's age;
- $x_2^i \sim \text{Unif}[1, 15]$ represents the monthly income, measured in thousands of CHF;
- $x_3^i \sim \text{Ber}(0.1)$ is a binary indicator of employment status, where $x_3^i = 1$ corresponds to a self-employed borrower and $x_3^i = 0$ to a salaried borrower.[a]

For each borrower $i$, we assume that the three components of $x^i$ are mutually independent. Moreover, the feature vectors $\{x^i\}_{i=1}^{m+n}$ are assumed to be independent and identically distributed across borrowers.

[a] That is, $\mathbb{P}(x_3^i = 1) = 0.1$ and $\mathbb{P}(x_3^i = 0) = 0.9$.

After obtaining this synthetic data, we then proceed in computing *empirical means* and *standard deviation* of each feature vector ($\{x^i\}_{i=1}^{m+n}$) in order to summarize their empirical distribution. For each coordinate $x_j$, $j = 1, 2, 3$, we focus on the first $m$ observations, which constitute the training sample.

The empirical mean of feature $j$ is defined as

$$\hat{\mu}_j = \frac{1}{m} \sum_{i=1}^{m} x_j^i,$$

and provides an estimate of the average value of the corresponding borrower characteristic in the training population.
Similarly, the empirical variance of feature $j$ is given by

$$\hat{\sigma}_j^2 = \frac{1}{m-1} \sum_{i=1}^{m} \left(x_j^i - \hat{\mu}_j\right)^2,$$

with $\hat{\sigma}_j$ denoting the associated empirical standard deviation.

## 1.2 Empirical Analysis and Discussion

The resulting quantities of the empirical statistics may be found below on Table 1. On another note, we'd like to give a brief discussion on additional creditor characteristics.

In practice, credit scoring models often rely on a bigger set of borrower characteristics in order to capture

| Variable | Empirical mean | Empirical standard deviation |
|---|---|---|
| Age | 48.4547 | 17.9461 |
| Income | 7.9979 | 4.0281 |
| Employment status | 0.1010 | 0.3013 |

Table 1: Empirical means and standard deviations of the feature components computed from the training sample.

different dimensions of 'credit-worthiness' beyond basic demographic and income information. In addition to the variables considered in this project, other realistic ones that could be incorporated into a credit scoring framework include the following:

- **Debt-to-income ratio**, which measures the borrower's existing financial obligations relative to their income and provides a direct indicator of repayment capacity;
- **Past defaults or delinquencies**, reflecting previous difficulties in meeting financial commitments and typically serving as a strong predictor of future default risk;
- **Savings amount**, capturing the level of low volatility readily accessible assets in the possession to the borrower as a buffer against adverse income or expenditure shocks;
- **Length and structure of credit history**, which summarizes past interactions with credit products and repayment behavior, although the relevance of this variable can be limited in the Swiss context (due to the absence of a centralized, publicly accessible credit history system).

## 2 Default model and data generation

We begin by specifying the data-generating mechanisms that describe the borrower repayment behavior.

Let $\xi^1, \ldots, \xi^{m+n} \overset{\text{i.i.d.}}{\sim} \text{Unif}(0,1)$ and define the *sigmoid* function

$$\psi(z) = \frac{1}{1+e^{-z}}.$$

Using this function, we introduce two repayment probability models $p_s : \mathbb{R}^3 \to (0,1)$, for $s \in \{1,2\}$, given by

- $$p_1(x) = \psi\big(13.3 - 0.33x_1 + 3.5x_2 - 3x_3\big),$$
- $$p_2(x) = \psi\Big(5 - 10\left[\mathbb{I}_{(-\infty,25)} + \mathbb{I}_{(75,\infty)}\right] x_1 + 1.1x_2 - x_3\Big).$$

The two functions differ in structure: while $p_1$ depends linearly on all features, $p_2$ introduces a non-linear effect in age through indicator functions that penalize very young and very old borrowers.

Based on these repayment probabilities, we construct two datasets $\{(x^i, y_s^i)\}_{i=1}^{m+n}$, for $s = 1, 2$, by defining the binary repayment indicator

$$y_s^i := \mathbb{I}_{\{\xi^i \le p_s(x^i)\}}.$$

Here, $y_s^i = 1$ indicates that borrower $i$ repays the loan, and $y_s^i = 0$ corresponds to a default, under model $s$. These two datasets serve as the basis for the subsequent training and evaluation of statistical learning models.

### 2.1 Logistic regression and cross-entropy evaluation

We estimate repayment probabilities using logistic regression, which assumes the conditional model

$$\mathbb{P}(y = 1 \mid x) = \psi(\beta_0 + \beta^\top x),$$

where $\psi$ denotes the previously defined sigmoid function. The parameter vector $(\beta_0, \beta)$ is estimated on the training data by minimizing the empirical *cross-entropy loss* (see appendix A.1 for a formal definition on this evaluation function).

We report the resulting cross-entropy losses on both the training and test sets. For dataset $s = 1$, the training loss is 0.0328, while the test loss is 0.0318. For dataset $s = 2$, the corresponding losses are 0.4852 on the training set and 0.4799 on the test set.

We suspect that the observable difference in magnitude between the losses for $s = 1$ and $s = 2$ reflects the alignment between the model specification and the previously defined data-generating process. While $p_1$ is linear in the features and therefore well suited to logistic regression, $p_2$ adds non-linear effects that are not captured by the model, leading to higher cross-entropy losses.

### 2.2 Kernel-based classification via Support Vector Machines

We now consider a Support Vector Machine (SVM) classifier applied to standardized feature vectors.

Given training data $\{(x^i, y_s^i)\}_{i=1}^m$, the SVM estimates a decision function of the form

$$f(x) = \sum_{i=1}^{m} \alpha_i \, k(x, x^i) + b,$$

where $k$ is a Gaussian kernel

$$k(x, x') = \exp\left(-\frac{1}{10}\|x - x'\|^2\right),$$

and the coefficients $\alpha_i$ are obtained by minimizing a *regularized hinge loss* (see appendix A.1 for more details). To enable probabilistic evaluation, class probabilities are attained/recovered via Platt scaling, giving an estimate $\hat{p}_s^{\text{svm}}(x)$.

We evaluate the SVM models using cross-entropy loss on both the training and test sets. For dataset $s = 1$, the training loss is 0.0402, while the test loss is 0.0386. For dataset $s = 2$, the corresponding losses are 0.2672 on the training set and 0.2501 on the test set.

Compared to logistic regression, the SVM achieves slightly higher cross-entropy loss for dataset $s = 1$, but significantly improves performance for dataset $s = 2$. This reflects the ability of the kernel-based model to capture non-linear structures (regarding feature relationships) in the repayment probability that are not explicitly modeled by a linear classifier.

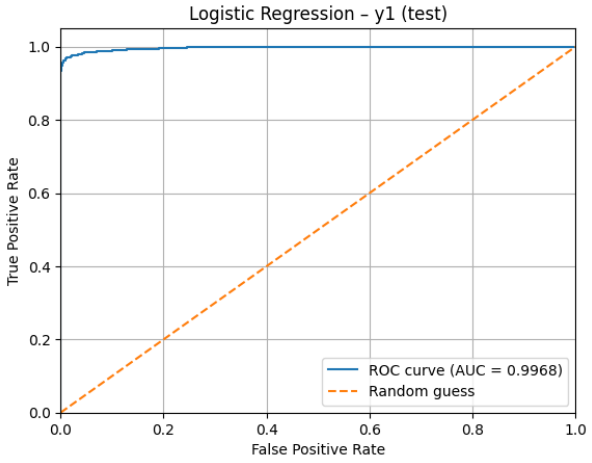
## 2.3 Model evaluation via ROC curves and AUC metric

We evaluate the predictive performance of the logistic regression and SVM models using Receiver Operating Characteristic (ROC) curves on the *test* set, and summarize each curve through its Area Under the Curve (AUC) (i.e. typical Riemann sum under the curve). A model with AUC close to 1 achieves strong separation between repaid loans ($y = 1$) and defaults ($y = 0$) across a wide range of decision thresholds, where in the other hand an AUC close to 0.5 corresponds to near-random classification.

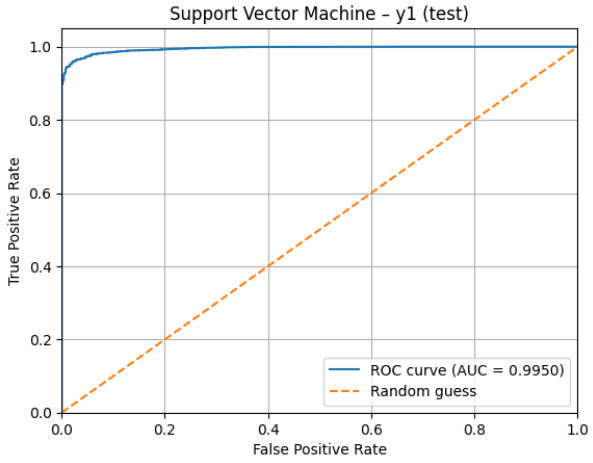
Figure 1 collects the ROC curves for both models and both datasets. For dataset $s = 1$, both curves lie very close to the top-left corner, which imply an excellent discriminative power. The logistic regression model attains an AUC of 0.9968, while the SVM achieves an AUC of 0.9950. The difference is small (about 0.18% relative), which may hint at the fact that both methods perform essentially 'optimally' in this setting.

For dataset $s = 2$, the ROC curves are visibly less aligned near the top-left corner, indicating a more challenging classification task. In this case, the logistic regression curve is substantially closer to the diagonal compared to the SVM curve, and the AUC values reflect this gap: logistic regression achieves an AUC of 0.8403, whereas the SVM reaches 0.9596. This corresponds to an improvement of roughly 12.1% in AUC when using the SVM instead of logistic regression on dataset $s = 2$.

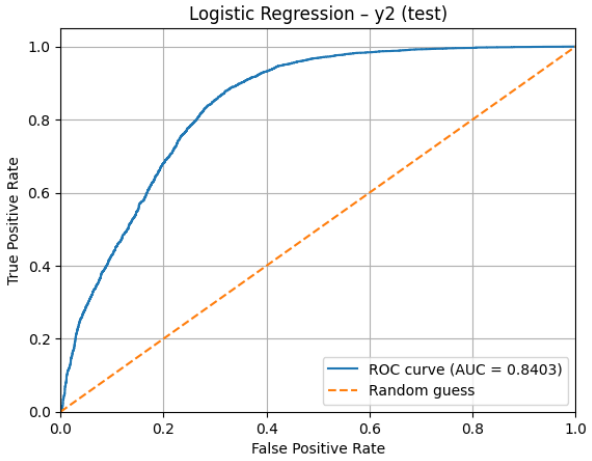
Overall, Figure 1 shows that while logistic regression and SVM perform similarly on dataset $s = 1$, the SVM provides a clear improvement on dataset $s = 2$ in terms of ROC shape and AUC, indicating better separation of repayment and default outcomes in the second setting.



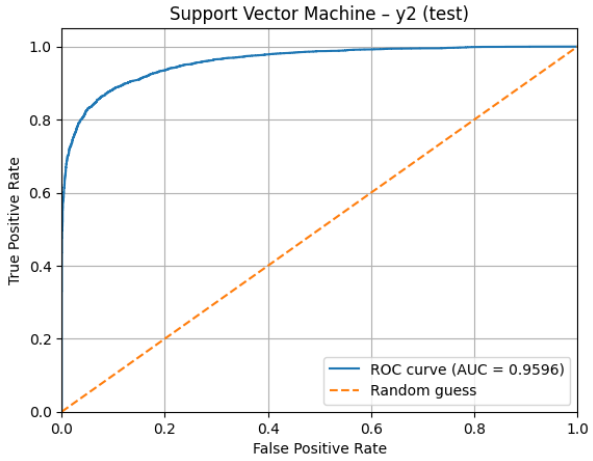

(a) Logistic regression, dataset $s = 1$ (AUC $= 0.9968$).




(b) SVM, dataset $s = 1$ (AUC $= 0.9950$).




(c) Logistic regression, dataset $s = 2$ (AUC $= 0.8403$).




(d) SVM, dataset $s = 2$ (AUC $= 0.9596$).

Figure 1: ROC curves on the test set for logistic regression and SVM, for both datasets $s = 1$ and $s = 2$. The dashed diagonal in each plot corresponds to random guessing (AUC $= 0.5$).

# 3 Lending strategies

For this last part, we only focus on dataset $s = 2$, which we treat as the true data-generating mechanism which underlies the borrower repayment behavior.

We assume that each loan has a fixed principal[a] of 1000 CHF and that repayment is all-or-nothing. We consider the following lending policies, applied to the test set borrowers $i = m + 1, \ldots, m + n$:

1. lend to all applicants at an interest rate of 5.5%;
2. lend selectively at an interest rate of 1%, but only to applicants satisfying $\hat{p}_2^{\log}(x^i) \geq 95\%$;
3. lend selectively at an interest rate of 1%, but only to applicants satisfying $\hat{p}_2^{\mathrm{SVM}}(x^i) \geq 95\%$.

[a] Original sum of money borrowed (like a loan) or invested.

Here, $\hat{p}_2^{\log}(x^i)$ and $\hat{p}_2^{\mathrm{SVM}}(x^i)$ denote the estimated repayment probabilities for borrower $i$ obtained from the logistic regression and SVM models trained on dataset $s = 2$, respectively. To assess the risk and profitability of each policy, we simulate $50,000$ repayment scenarios. For each borrower $i$ and scenario $k$, we draw $\xi^{i,k} \overset{\text{i.i.d.}}{\sim} \mathrm{Unif}(0, 1)$ and define the repayment indicator

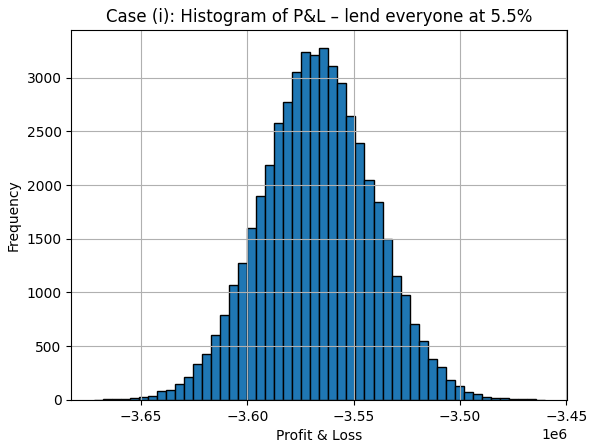
$$D^{i,k} := \mathbb{I}_{\{\xi^{i,k} \leq p_2(x^{m+i})\}},$$

where $p_2$ is the true repayment probability under dataset $s = 2$. Adding profits and losses across borrowers gives an empirical distribution of portfolio profit and loss (P&L) for each lending strategy.

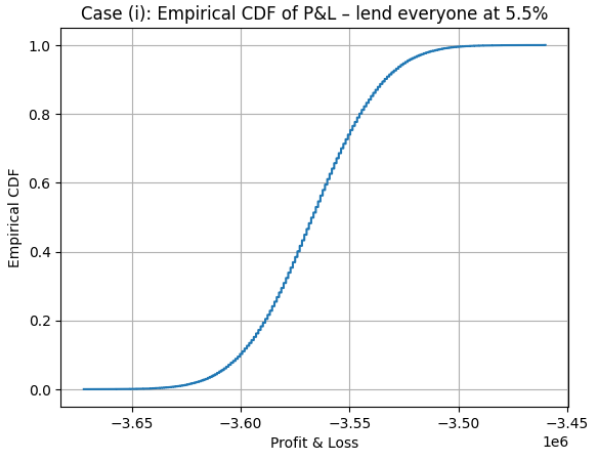
## 3.1 Lend to everyone with 5.5% interest

Figure 2 shows the empirical distribution of portfolio P&L under the policy of lending to all applicants at an interest rate of 5.5%. The histogram in Figure 2a shows a concentrated distribution centered far in the negative region, which shows that losses occur in the majority of simulated scenarios. This reflects the fact that, while the interest rate is relatively high, defaults among riskier borrowers generate large losses that dominate interest income.

The corresponding empirical CDF in Figure 2b confirms this observation: almost the entire distribution lies below zero, and the probability of achieving a positive portfolio return is effectively none. Although this strategy involves no model risk or selection error, it exposes the lender to heavy downside risk due to the unfiltered default exposure.




(a) Histogram of portfolio P&L under the policy of lending to all applicants at 5.5% interest.




(b) Empirical CDF of portfolio P&L under the policy of lending to all applicants at 5.5% interest.

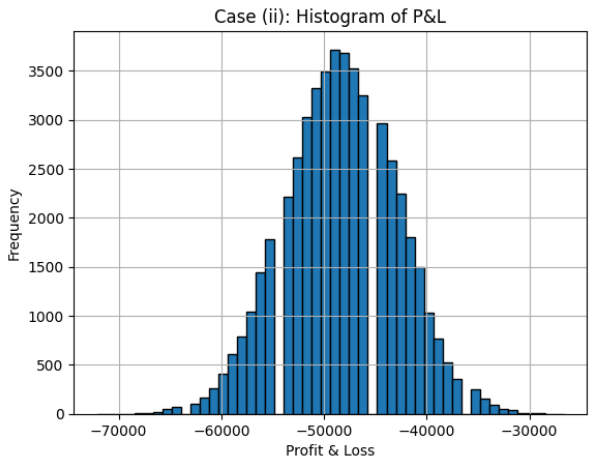
Figure 2: Case 1: Lend to everyone at 5.5% interest.

The $\text{VaR}_{95\%}$ is 3609865 CHF and $\text{ES}_{95\%}$ is 3620303 CHF.

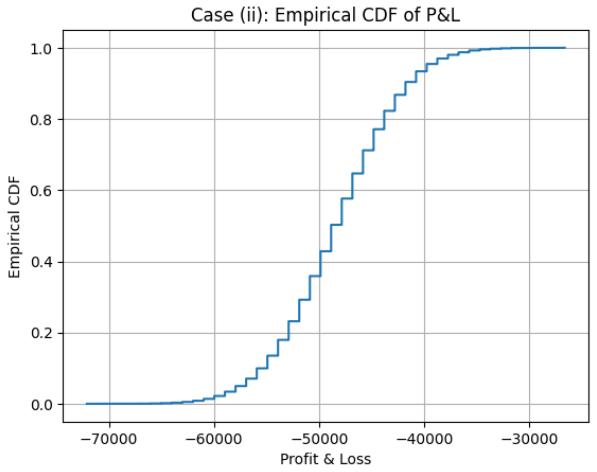
## 3.2 Lend selectively at 1% interest based on logistic classification

Figure 3 presents the P&L distribution obtained by lending selectively at 1% interest using the logistic regression model. Compared to the unconditional lending strategy, the histogram in Figure 3a shows a noticeable shift of the distribution towards less negative values, reflecting the exclusion of applicants considered high-risk by the model.

But, the distribution is still entirely concentrated in the negative region, and the empirical CDF in Figure 3b indicates that large losses still occur with high probability. This behavior demonstrates the asymmetric risk–return profile of low-interest lending: while the model reduces default frequency, the small gains from successful repayments are insufficient to cancel out losses from the remaining defaults. As a result, the expected portfolio P&L remains negative despite selective lending.




(a) Histogram of portfolio P&L under selective lending at 1% interest based on the logistic regression model.




(b) Empirical CDF of portfolio P&L under selective lending at 1% interest based on the logistic regression model.

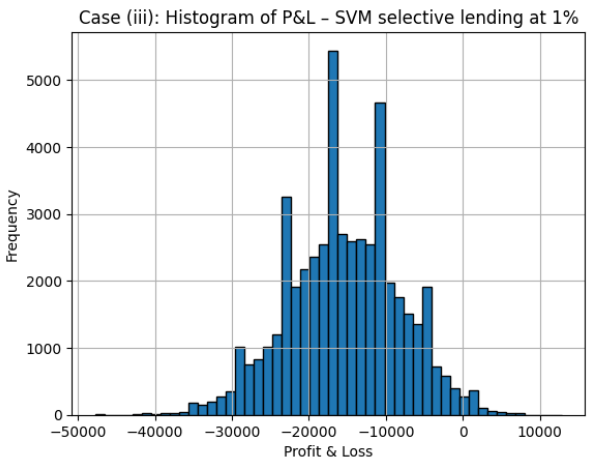
Figure 3: Case 2: Lend selectively at 1% interest, but only to applicants with $\hat{p}_2^{\log}(x^i) \geq 95\%$

The $\text{VaR}_{95\%}$ is 56940 CHF and $\text{ES}_{95\%}$ is 58933 CHF.

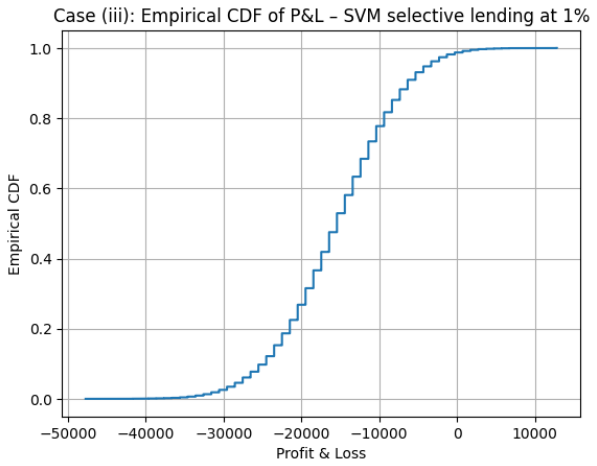
### 3.3 Lend selectively at 1% interest based on SVM classification

Figure 4 shows the P&L distribution obtained when selective lending is performed using the SVM classification model. The histogram in Figure 4a is more shifted towards higher values relative to the logistic case, showing improved risk screening. This is consistent with the 'better' classification performance of the SVM observed in the ROC and AUC analysis for dataset $s = 2$.

Needless to say, the empirical CDF in Figure 4b reveals that the distribution still places noticeable probability mass on negative outcomes. While the SVM-based strategy reduces downside risk compared to the logistic approach, the low interest rate limits the achievable upside. So, even the most accurate classification model considered does not suffice to render the lending strategy profitable under the chosen economic assumptions.




(a) Histogram of portfolio P&L under selective lending at 1% interest based on the SVM classification model.




(b) Empirical CDF of portfolio P&L under selective lending at 1% interest based on the SVM classification model.

Figure 4: Case 3: Lend selectively at 1% interest, but only to applicants with $\hat{p}_2^{\text{svm}}(x^i) \geq 95\%$

The $\text{VaR}_{95\%}$ is 27550 CHF and $\text{ES}_{95\%}$ is 30319 CHF.

### 3.4 Conclusion

In conclusion, looking at all lending strategies, we observe significantly high Value at Risks and Expected Shortfall indicating non-profitable lending strategies. We believe this could be explained by multiple factors. First of all, all-or-nothing repayment wouldn't be a viable contract, nor underwritten by most insurers in the majority of cases. Second of all, we believe that the repayment probabilities computed from the synthetic dataset are not realistic (lending to unemployed people without taking personal fortune into account for instance). We believe that in practice, these factors would lead to stricter risk management rules and thus higher probability of making a profit per contract signed.

Moreover, analyzing the underlying distributions for the synthetic feature generation reveals a lack of consideration for the true population looking for a credit. Especially since most individual characteristics have cross dependencies that influence their realization, it could be useful to consider a more robust approach which is able to capture these relationships.

Finally, we consider the performance of the SVM model to hold certain validity upon the classification task. Although, if a more revised approach is given into the creditor's generation, a more robust machinery could required in order to capture these new complexities in a more proficient manner.

# A Appendix

## A.1 Loss functions

We briefly recall the definitions of the loss functions used for training and evaluating the classification models.

**Cross-entropy loss.** Given predicted probabilities $\hat{p}(x) = \mathbb{P}(y = 1 \mid x)$ and binary labels $y \in \{0, 1\}$, the cross-entropy loss is defined as

$$\ell_{\mathrm{CE}}(y, \hat{p}) = -\big[y \log(\hat{p}) + (1 - y) \log(1 - \hat{p})\big].$$

The empirical cross-entropy loss over a dataset $\{(x^i, y^i)\}_{i=1}^N$ is given by

$$\frac{1}{N} \sum_{i=1}^{N} \ell_{\mathrm{CE}}\big(y^i, \hat{p}(x^i)\big).$$

This loss is used throughout the report to evaluate probabilistic predictions from both logistic regression and SVM models.

**Hinge loss.** For labels $y \in \{-1, +1\}$ and a real-valued decision function $f(x)$, the hinge loss is defined as

$$\ell_{\mathrm{hinge}}(y, f(x)) = \max\{0,\, 1 - y f(x)\}.$$

In the SVM framework, model parameters are obtained by minimizing a regularized empirical hinge loss of the form

$$\frac{1}{m} \sum_{i=1}^{m} \ell_{\mathrm{hinge}}\big(y^i, f(x^i)\big) + \lambda \|f\|^2,$$

where $\lambda > 0$ is a regularization parameter. Although the SVM is trained using hinge loss, probabilistic outputs are obtained via Platt scaling, allowing model performance to be evaluated using cross-entropy loss.

## A.2 Python Code

This appendix contains the Python code used to generate the synthetic data, estimate the statistical learning models, and evaluate the different lending strategies discussed in the main text. The implementation follows a modular design and closely mirrors the methodology described in the report. Executing the provided scripts reproduces all tables and figures presented in the main sections.

```python
# %% [markdown]
# # Project 2: Credit Risk and Statistical Learning

# %% [markdown]
# **Names of all group members:**
# - Rafael Barroso (rafael.barroso@epfl.ch)
# - Philipp Mayer (philipp.mayer@epfl.ch)

# %%
## IMPORT NECESARY LIBRARIES
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
```

```python
from sklearn.metrics import roc_curve, auc


# %% [markdown]
# ## 1: **Feature generation**

# %%
def gen_features(m: int, n: int, random_seed: int = 0) -> pd.DataFrame:
    """
    Generate (m + n) feature vectors for loan applicants.

    Each applicant i has:
        x_i1: age                  ~ Uniform[18, 80]
        x_i2: monthly income (CHF) ~ Uniform[1, 15]
        x_i3: employment status    ~ Bernoulli(p = 0.1); so we label as
                                     1 = self-employed
                                     0 = salaried

    All three coordinates are assumed independent.

    Parameters
    ----------
    m : int
        Number of training samples.
    n : int
        Number of test samples.
    random_seed : int or None, optional

    Returns
    -------
    features_df : pandas.DataFrame
        DataFrame with (m + n) rows and 3 columns:
            - "age"
            - "income"
            - "employment"
    """

    total_samples = m + n
    rng = np.random.default_rng(seed=random_seed) # Use numpy's random generator

    ## Simulate features
    ages = rng.integers(18, 80, size=total_samples)
    incomes = rng.uniform(low=1.0, high=15.0, size=total_samples)
    employment_status = rng.binomial(1, 0.1, size=total_samples)

    # Create DataFrame
    features_df = pd.DataFrame({
        "age": ages,
        "income": incomes,
        "employment": employment_status
    })

    return features_df

def compute_empirical_stats(features: pd.DataFrame, m: int) -> tuple[pd.Series, pd.Series]:
    """
    Compute empirical mean and variance of each feature using the training set.

    Parameters
    ----------
    features : pandas.DataFrame
        DataFrame containing features for all applicants.
    m : int
        Number of training samples.

    Returns
    -------
    means : pandas.Series
        Empirical means of each feature.
    variances : pandas.Series
```

```
        Empirical variances of each feature.
    """

    training_features = features.iloc[:m]

    means = training_features.mean()
    stds = training_features.std()

    return means, stds

# %%
m, n = 20000, 10000
features = gen_features(m, n, random_seed=0)
means, stds = compute_empirical_stats(features, m)
print(features.head())
print("Empirical Means:\n", means)
print("Empirical Standard Deviations:\n", stds)

# %%
(features['employment'] == 0).sum()

# %% [markdown]
# dayum that's a lot of unemployed people lol

# %% [markdown]
# Some other realistic variables that might be used for credit scoring are:
# - Debt to income ratio
# - Past defaults/delinquencies
# - Savings ammount
# - Existing credit history (Although for Switzerland this is not usefull)

# %% [markdown]
# ## 2: **Default model and data generation**

# %%
def sigmoid(z: np.ndarray) -> np.ndarray:
    """
    Standard logistic (sigmoid) function:
        sigma(z) = 1 / (1 + exp(-z))
    """
    return 1.0 / (1.0 + np.exp(-z))

def p1_probability(features: pd.DataFrame) -> np.ndarray:
    """
    Compute repayment probabilities p1(x) for each row in `features`.

    Model 1:
        p1(x) = sigmoid(13.3 - 0.33 * age + 3.5 * income - 3 * employment)

    Here:
        - age = features["age"]
        - income = features["income"]
        - employment = features["employment"] (0 or 1)

    Returns
    -------
    probs : np.ndarray
        Array of repayment probabilities p1(x_i) with shape (n_samples,).
    """
    age = features["age"].to_numpy()
    income = features["income"].to_numpy()
    emp = features["employment"].to_numpy()

    # Linear predictor for model 1
    z = 13.3 - 0.33 * age + 3.5 * income - 3.0 * emp

    # Apply sigmoid to obtain probabilities in (0,1)
    probs = sigmoid(z)
    return probs
```

```python
def p2_probability(features: pd.DataFrame) -> np.ndarray:
    """
    Compute repayment probabilities p2(x) for each row in `features`.

    Model 2 (piecewise in age via indicator functions):

        Let:
            young = 1 if age <= 25, else 0
            old   = 1 if age >= 75, else 0

        Then:
            p2(x) = sigmoid( -5 - 10 * young - 10 * old + 1.1 * income - 1.0 * employment )

    Returns
    -------
    probs : np.ndarray
        Array of repayment probabilities p2(x_i) with shape (n_samples,).
    """
    age = features["age"].to_numpy()
    income = features["income"].to_numpy()
    emp = features["employment"].to_numpy()

    # Indicator variables for age
    # young = 1 if age in (18, 25], old = 1 if age in [75, 80]
    young = (age <= 25.0).astype(float)
    old = (age >= 75.0).astype(float)

    # Linear predictor for model 2
    z = -5.0 - 10.0 * young - 10.0 * old + 1.1 * income - 1.0 * emp

    probs = sigmoid(z)
    return probs

def generate_repayment_data(features: pd.DataFrame, random_seed: int = 0) -> pd.DataFrame:
    """
    Given a matrix of features X (as a DataFrame), generate repayment
    outcomes y^1 and y^2 for both models using a common Uniform(0,1)
    random variable eprilon_i for each applicant i.

    Parameters
    ----------
    features : pd.DataFrame
        Must contain at least the columns:
            "age", "income", "employment".
        It should have (m + n) rows.
    random_seed : int or None, optional

    Returns
    -------
    data : pd.DataFrame
        Copy of `features` with 4 new columns:
            - "p1_true": p1(x_i)
            - "p2_true": p2(x_i)
            - "y1": repayment indicator under model 1
            - "y2": repayment indicator under model 2
    """
    n_samples = len(features)

    # Set up RNG
    rng = np.random.default_rng(seed=random_seed)

    # 1) Compute true repayment probabilities under each model
    p1 = p1_probability(features)  # shape (n_samples,)
    p2 = p2_probability(features)  # shape (n_samples,)

    # 2) Draw a single Uniform(0,1) value _i for each applicant i
    eps = rng.uniform(low=0.0, high=1.0, size=n_samples)

    # 3) Generate repayment indicators using the same _i for both models
    y1 = (eps <= p1).astype(int)  # 1 = repay, 0 = default
```

```python
    y2 = (eps <= p2).astype(int)

    # 4) Build output DataFrame (do not modify input in-place)
    data = features.copy()
    data["p1_true"] = p1
    data["p2_true"] = p2
    data["y1"] = y1
    data["y2"] = y2

    return data

# %%
data = generate_repayment_data(features, random_seed=0)
print(data.head(-10))

# %% [markdown]
# ### (A): **Fit Logistic Regression model**

# %%
## HELPER FUNCTION TO SPLIT TRAIN/TEST SETS
def tts_split(
    data: pd.DataFrame,
    target_col: str,
    m: int,
    feature_cols: list[str] = None,
):
    """
    Split the full DataFrame into training and test sets.

    Parameters
    ----------
    data : pd.DataFrame
        Full dataset with (m + n) rows.
    target_col : str
        Name of the target column to predict ("y1" or "y2").
    m : int
        Number of training samples (first m rows are training).
    feature_cols : list of str or None, optional
        Names of feature columns. If None, we default to:
            ["age", "income", "employment"].

    Returns
    -------
    X_train : np.ndarray of shape (m, d)
    y_train : np.ndarray of shape (m,)
    X_test  : np.ndarray of shape (n, d)
    y_test  : np.ndarray of shape (n,)
    """
    # If no specific feature list is given, use the standard three
    if feature_cols is None:
        feature_cols = ["age", "income", "employment"]

    # Extract feature matrix and target vector as NumPy arrays
    X = data[feature_cols].to_numpy()
    y = data[target_col].to_numpy()

    # First m rows: training set
    X_train = X[:m, :]
    y_train = y[:m]

    # Remaining rows: test set
    X_test = X[m:, :]
    y_test = y[m:]

    return X_train, y_train, X_test, y_test

# %% [markdown]
# can also use `train_test_split` prebuilt sklearn function but anyways....

# %%
```

```python
def fit_logistic_regression(X_train: np.ndarray, y_train: np.ndarray) -> LogisticRegression:
    """
    Fit a simple logistic regression model using scikit-learn.

    We do NOT do any fancy preprocessing here:
    - no feature scaling,
    - default L2 regularization,
    - just more iterations to ensure convergence.

    Parameters
    ----------
    X_train : np.ndarray
        Training feature matrix of shape (m, d).
    y_train : np.ndarray
        Training labels (0/1) of shape (m,).

    Returns
    -------
    model : LogisticRegression
        Fitted logistic regression model.
    """
    model = LogisticRegression(
        solver="lbfgs",  # standard solver
        max_iter=1000,   # more iterations just to be safe
    )
    model.fit(X_train, y_train)
    return model

def cross_entropy_loss(y_true: np.ndarray, p_hat: np.ndarray) -> float:
    """
    Compute the (average) cross-entropy loss for binary labels.

    Formula:
        L = - (1/N) * sum_{i=1}^N [ y_i * log(p_i) + (1 - y_i) * log(1 - p_i) ]

    where:
        - y_i are the true labels in {0,1}
        - p_i are the predicted probabilities P(y=1 | x_i)

    Parameters
    ----------
    y_true : np.ndarray
        Vector of true labels in {0,1}.
    p_hat : np.ndarray
        Vector of predicted probabilities in [0,1].

    Returns
    -------
    loss : float
        Cross-entropy loss (smaller is better, if you know how to use it hehe).
    """
    # Small epsilon to avoid log(0)
    eps = 1e-15

    # Clip predicted probabilities to stay away from 0 and 1
    p_hat_clipped = np.clip(p_hat, eps, 1.0 - eps)

    # Compute each term y_i * log(p_i) + (1 - y_i) * log(1 - p_i)
    term1 = y_true * np.log(p_hat_clipped)
    term2 = (1 - y_true) * np.log(1.0 - p_hat_clipped)

    # Average cross-entropy
    loss = -np.mean(term1 + term2)
    return loss

def evaluate_logistic_model(model: LogisticRegression, X_train: np.ndarray, y_train:
↪  np.ndarray, X_test: np.ndarray, y_test: np.ndarray):
    """
    Compute predicted probabilities and cross-entropy losses
    for both TRAINING and TEST sets.
```

```
368
369         Parameters
370         ----------
371         model : LogisticRegression
372             Fitted logistic regression.
373         X_train, y_train : np.ndarray
374             Training data.
375         X_test, y_test : np.ndarray
376             Test data.
377
378         Returns
379         -------
380         results : dict
381             Dictionary with:
382                 - "p_train": predicted probabilities on train set
383                 - "p_test" : predicted probabilities on test set
384                 - "ce_train": cross-entropy on train set
385                 - "ce_test" : cross-entropy on test set
386         """
387         # predict_proba returns a matrix with 2 columns:
388         #   [:,0] = P(y=0 | x), [:,1] = P(y=1 | x)
389         p_train = model.predict_proba(X_train)[:, 1]
390         p_test = model.predict_proba(X_test)[:, 1]
391
392         # Compute cross-entropy losses
393         ce_train = cross_entropy_loss(y_train, p_train)
394         ce_test = cross_entropy_loss(y_test, p_test)
395
396         results = {
397             "p_train": p_train,
398             "p_test": p_test,
399             "ce_train": ce_train,
400             "ce_test": ce_test,
401         }
402         return results
403
404     # %%
405     X1_train_log1, y1_train_log1, X1_test_log1, y1_test_log1 = tts_split(data, target_col="y1",
        ↪  m=m)
406     log_model1 = fit_logistic_regression(X1_train_log1, y1_train_log1)
407     log_results1 = evaluate_logistic_model(log_model1, X1_train_log1, y1_train_log1, X1_test_log1,
        ↪  y1_test_log1)
408     print("Logistic Regression Model 1 Cross-Entropy Loss:")
409     print("  Train:", log_results1["ce_train"])
410     print("  Test :", log_results1["ce_test"])
411
412     # %%
413     X1_train_log2, y1_train_log2, X1_test_log2, y1_test_log2 = tts_split(data, target_col="y2",
        ↪  m=m)
414     log_model2 = fit_logistic_regression(X1_train_log2, y1_train_log2)
415     log_results2 = evaluate_logistic_model(log_model2, X1_train_log2, y1_train_log2, X1_test_log2,
        ↪  y1_test_log2)
416     print("Logistic Regression Model 2 Cross-Entropy Loss:")
417     print("  Train:", log_results2["ce_train"])
418     print("  Test :", log_results2["ce_test"])
419
420     # %% [markdown]
421     # ### (B): **Fit Support Vector Machine model**
422
423     # %%
424     def standardize_train_test(X_train: np.ndarray, X_test: np.ndarray):
425         """
426         Standardize features:
427
428             For each feature j:
429                 mean_j = average of column j on the TRAINING set
430                 std_j  = standard deviation of column j on the TRAINING set
431
432             Then:
433                 X_std[:, j] = (X[:, j] - mean_j) / std_j
```

```
434
435         IMPORTANT:
436         ---------
437         - We fit the scaler on the TRAINING data only.
438         - We then apply the same transformation to the TEST data.
439
440         Parameters
441         ----------
442         X_train : np.ndarray
443             Training feature matrix of shape (m, d).
444         X_test : np.ndarray
445             Test feature matrix of shape (n, d).
446
447         Returns
448         -------
449         X_train_std : np.ndarray
450             Standardized training features.
451         X_test_std : np.ndarray
452             Standardized test features.
453         scaler : StandardScaler
454             Fitted scaler (in case you want to reuse it).
455         """
456         scaler = StandardScaler()
457         X_train_std = scaler.fit_transform(X_train)
458         X_test_std = scaler.transform(X_test)
459
460         return X_train_std, X_test_std, scaler
461
462     def fit_svm_classifier(X_train_std: np.ndarray, y_train: np.ndarray, C: float = 1.0, kernel:
    ↪  str = "rbf", gamma: str = "scale") -> SVC:
463         """
464         Fit a Support Vector Machine (SVM) classifier with probability outputs.
465
466         We keep the setup simple:
467             - RBF kernel (non-linear)
468             - Default C and gamma (you can change them later if needed)
469             - probability=True so we can call predict_proba and compute cross-entropy
470
471         Parameters
472         ----------
473         X_train_std : np.ndarray
474             Standardized training features of shape (m, d).
475         y_train : np.ndarray
476             Binary training labels (0/1) of shape (m,).
477         C : float, optional
478             Regularization parameter. Default = 1.0.
479         kernel : str, optional
480             Kernel type. Default = "rbf".
481         gamma : str, optional
482             Kernel coefficient. "scale" is scikit-learn's default.
483
484         Returns
485         -------
486         model : SVC
487             Fitted SVM classifier.
488         """
489         model = SVC(
490             C=C,
491             kernel=kernel,
492             gamma=gamma,
493             probability=True,  # needed to get probabilities
494         )
495         model.fit(X_train_std, y_train)
496         return model
497
498     def evaluate_svm_model(model: SVC, X_train_std: np.ndarray, y_train: np.ndarray, X_test_std:
    ↪  np.ndarray, y_test: np.ndarray):
499         """
500         Compute predicted probabilities and cross-entropy losses
501         for both TRAINING and TEST sets using the already defined
```

```
    `cross_entropy_loss` function.

    Parameters
    ----------
    model : SVC
        Fitted SVM classifier.
    X_train_std, y_train : np.ndarray
        Standardized training data and labels.
    X_test_std, y_test : np.ndarray
        Standardized test data and labels.

    Returns
    -------
    results : dict
        Dictionary with:
            - "p_train": predicted probabilities on train set
            - "p_test" : predicted probabilities on test set
            - "ce_train": cross-entropy on train set
            - "ce_test" : cross-entropy on test set
    """
    # As with logistic regression, predict_proba returns:
    #   [:,0] = P(y=0 | x), [:,1] = P(y=1 | x)
    p_train = model.predict_proba(X_train_std)[:, 1]
    p_test = model.predict_proba(X_test_std)[:, 1]

    ce_train = cross_entropy_loss(y_train, p_train)
    ce_test = cross_entropy_loss(y_test, p_test)

    results = {
        "p_train": p_train,
        "p_test": p_test,
        "ce_train": ce_train,
        "ce_test": ce_test,
    }
    return results

# %%
X2_train_svm1, y2_train_svm1, X2_test_svm1, y2_test_svm1 = tts_split(data, target_col="y1",
↪  m=m)
X_train, X_test, scaler = standardize_train_test(X2_train_svm1, X2_test_svm1)
svm_model1 = fit_svm_classifier(X2_train_svm1, y2_train_svm1)
results_svm1 = evaluate_svm_model(svm_model1, X2_train_svm1, y2_train_svm1, X2_test_svm1,
↪  y2_test_svm1)
print("SVM Model 1 Cross-Entropy Loss:")
print("  Train:", results_svm1["ce_train"])
print("  Test :", results_svm1["ce_test"])

# %%
X2_train_svm2, y2_train_svm2, X2_test_svm2, y2_test_svm2 = tts_split(data, target_col="y2",
↪  m=m)
X_train, X_test, scaler = standardize_train_test(X2_train_svm2, X2_test_svm2)
svm_model2 = fit_svm_classifier(X2_train_svm2, y2_train_svm2)
results_svm2 = evaluate_svm_model(svm_model2, X2_train_svm2, y2_train_svm2, X2_test_svm2,
↪  y2_test_svm2)
print("SVM Model 2 Cross-Entropy Loss:")
print("  Train:", results_svm2["ce_train"])
print("  Test :", results_svm2["ce_test"])

# %%
## HELPER FUNCTIONS FOR ROC AUC ANALYSIS
def compute_roc_auc(y_true: np.ndarray, p_hat: np.ndarray):
    """
    Compute ROC curve and AUC given true labels and predicted probabilities.

    Parameters
    ----------
    y_true : np.ndarray
        Binary true labels (0/1), shape (n_samples,)
    p_hat : np.ndarray
        Predicted probabilities P(y=1|x), shape (n_samples,) (i.e. prob of y=1 given x)
```

```
568
569         Returns
570         -------
571         fpr : np.ndarray
572             False positive rate values for different thresholds.
573         tpr : np.ndarray
574             True positive rate values for different thresholds.
575         roc_auc : float
576             Area under the ROC curve.
577         """
578         # roc_curve returns (FPR, TPR, thresholds)
579         fpr, tpr, _ = roc_curve(y_true, p_hat) # we want only true possitive rate and false
            ↪  possitive rate, no need thresholds
580         roc_auc = auc(fpr, tpr)
581         return fpr, tpr, roc_auc
582
583     def plot_roc_curve(fpr, tpr, roc_auc: float, title: str):
584         """
585         Plot ROC curve with diagonal reference line and AUC in the legend.
586
587         Parameters
588         ----------
589         fpr : np.ndarray
590             False positive rate values.
591         tpr : np.ndarray
592             True positive rate values.
593         roc_auc : float
594             Area under the ROC curve.
595         title (trivial) : str
596         """
597         plt.figure()
598         # ROC curve
599         plt.plot(fpr, tpr, label=f"ROC curve (AUC = {roc_auc:.4f})")
600         # Diagonal baseline (naive or random classifier)
601         plt.plot([0, 1], [0, 1], linestyle="--", label="Random guess")
602
603         plt.xlim([0.0, 1.0])
604         plt.ylim([0.0, 1.05])
605         plt.xlabel("False Positive Rate")
606         plt.ylabel("True Positive Rate")
607         plt.title(title)
608         plt.legend(loc="lower right")
609         plt.grid(True)
610         plt.show()
611
612     # %%
613     fpr_log_y1, tpr_log_y1, auc_log_y1 = compute_roc_auc(y1_test_log1, log_results1["p_test"])
614     plot_roc_curve(fpr_log_y1, tpr_log_y1, auc_log_y1,"Logistic Regression - y1 (test)")
615
616     # %% [markdown]
617     # This is a NICE curve. Usually we want it to be as L shaped as possible which would translate
        ↪  to having a perfect true possitive rate. In other words, we barely missed any possitive
        ↪  classifications.
618
619     # %%
620     fpr_log_y2, tpr_log_y2, auc_log_y2 = compute_roc_auc(y1_test_log2, log_results2["p_test"])
621     plot_roc_curve(fpr_log_y2, tpr_log_y2, auc_log_y2,"Logistic Regression - y2 (test)")
622
623     # %% [markdown]
624     # Less 'nicer'.
625
626     # %%
627     fpr_svm_y1, tpr_svm_y1, auc_svm_y1 = compute_roc_auc(y2_test_svm1, results_svm1["p_test"])
628     plot_roc_curve(fpr_svm_y1, tpr_svm_y1, auc_svm_y1,"Support Vector Machine - y1 (test)")
629
630     # %% [markdown]
631     # Compared to the logarithmic regression model, we may start our assumptions by stating that
        ↪  since we get a slightly 'bigger' area under the curve (which we have yet to discuss). In
        ↪  other words, we have a better *rate* at classifying correct possitives using log model.
632     #
```

```python
# Needless to say, this is also a pretty good classification.

# %%
fpr_svm_y2, tpr_svm_y2, auc_svm_y2 = compute_roc_auc(y2_test_svm2, results_svm2["p_test"])
plot_roc_curve(fpr_svm_y2, tpr_svm_y2, auc_svm_y2,"Support Vector Machine - y2 (test)")

# %% [markdown]
# Here we actually see the efficience of the svm model. Although for the first probabilistic test the logarithmic regression yields superior metrics, here the svm model clearly comes out on top. As a preliminary result, we might state that if we take both probabilistic approaches into consideration, the svm model performs better on average (taking into account both probs.)

# %% [markdown]
# ## 3: **Lending Strategies**

# %%
## HELPER FUNCTIONS FOR P&L ANALYSIS
def compute_pl_expectation(pl_samples: np.ndarray) -> float:
    """
    Compute the EXPECTED profit & loss (P&L).

    Here, P&L is a random variable:
        - positive values  -> profit
        - negative values  -> loss

    The expectation is simply:
        E[P&L] is approzimately = (1/N) * sum_i pl_i

    Parameters
    ----------
    pl_samples : np.ndarray
        1D array of simulated P&L values.

    Returns
    -------
    expected_pl : float
        Sample mean of the P&L.
    """
    expected_pl = np.mean(pl_samples)
    return expected_pl

def plot_pl_histogram(pl_samples: np.ndarray,
                      title: str = "Histogram of Profit & Loss",
                      bins: int = 50):
    """
    Plot a histogram of simulated P&L values.

    Parameters
    ----------
    pl_samples : np.ndarray
        1D array of simulated P&L values.
    title : str, optional
        Title for the plot.
    bins : int, optional
        Number of bins in the histogram.

    Notes
    -----
    - The vertical axis shows how many simulations fall into each
      P&L interval.
    """
    plt.figure()
    plt.hist(pl_samples, bins=bins, edgecolor="black")
    plt.xlabel("Profit & Loss")
    plt.ylabel("Frequency")
    plt.title(title)
    plt.grid(True)
    plt.show()

```

```python
def estimate_var_es(pl_samples: np.ndarray,
                    alpha: float = 0.95) -> tuple[float, float]:
    """
    Estimate Value-at-Risk (VaR) and Expected Shortfall (ES)
    at confidence level `alpha` based on simulated P&L.

    Convention used here:
    ---------------------
    - P&L > 0  : profit
    - P&L < 0  : loss

    We define LOSS = -P&L.
      -> large positive LOSS corresponds to large negative P&L.

    For losses L:

        VaR_alpha = quantile_alpha(L)
        ES_alpha  = average of losses that are WORSE than VaR_alpha, i.e.

            ES_alpha = E[L | L >= VaR_alpha]

    Parameters
    ----------
    pl_samples : np.ndarray
        1D array of simulated P&L values.
    alpha : float, optional
        Confidence level (e.g. 0.95 for 95% VaR and ES).

    Returns
    -------
    var_alpha : float
        Estimated VaR at level `alpha` (in loss units).
    es_alpha : float
        Estimated ES at level `alpha` (in loss units).
    """
    # Convert P&L to losses: L = -P&L
    losses = -pl_samples

    # 1) VaR is the alpha-quantile of the loss distribution
    var_alpha = np.quantile(losses, alpha)

    # 2) ES is the average loss in the tail beyond VaR
    tail_losses = losses[losses >= var_alpha]
    es_alpha = tail_losses.mean() if tail_losses.size > 0 else var_alpha

    return var_alpha, es_alpha

def plot_pl_distribution(pl_samples: np.ndarray,
                         title: str = "Empirical CDF of Profit & Loss"):
    """
    Plot the empirical distribution function (CDF) of P&L.

    Steps:
    ------
    1) Sort the P&L samples.
    2) For each sorted value x_(i), define F(x_(i)) = i / N,
       where N is the number of samples.
    3) Plot x_(i) vs F(x_(i)).

    Parameters
    ----------
    pl_samples : np.ndarray
        1D array of simulated P&L values.
    title : str, optional
        Title for the plot.

    Notes
    -----
    - The CDF shows, for any value x, the proportion of scenarios
      with P&L <= x.
```

```
    """
    # Sort P&L values
    pl_sorted = np.sort(pl_samples)

    # Empirical CDF values from 1/N to N/N
    n = pl_sorted.size
    cdf = np.arange(1, n + 1) / n

    plt.figure()
    plt.step(pl_sorted, cdf, where="post")
    plt.xlabel("Profit & Loss")
    plt.ylabel("Empirical CDF")
    plt.title(title)
    plt.grid(True)
    plt.show()

def simulate_repayment_scenarios_p2(
    data: pd.DataFrame,
    m: int,
    num_scenarios: int = 50000,
    random_seed: int | None = 123,
) -> np.ndarray:
    """
    Simulate repayment scenarios for the TEST set using the true p2(x).

    We only use the TEST set:
        i = m, ..., m+n-1

    For each test borrower i and each scenario k, we draw:
        D_{i,k} = 1{ U_{i,k} <= p2(x_i) },
    where U_{i,k} ~ Unif(0,1) are independent.

    Parameters
    ----------
    data : pd.DataFrame
        Full dataset with at least the column "p2_true".
    m : int
        Number of TRAINING observations (first m rows). The rest are TEST.
    num_scenarios : int, optional
        Number of scenarios K to simulate).
    random_seed : int or None, optional
        Seed for reproducibility.

    Returns
    -------
    D : np.ndarray
        2D array of shape (n_test, num_scenarios) with entries in {0,1},
        where n_test = len(data) - m.
        D[i,k] = 1 means borrower i (in test set) repays in scenario k.
    """
    rng = np.random.default_rng(seed=random_seed)

    # Extract true p2(x) for test borrowers
    p2_all = data["p2_true"].to_numpy()   # shape (m + n,)
    p2_test = p2_all[m:]                  # shape (n,)

    n_test = p2_test.size

    # Draw U_{i,k} ~ Unif(0,1) for all i,k
    # Shape: (n_test, num_scenarios)
    U = rng.uniform(low=0.0, high=1.0, size=(n_test, num_scenarios))

    # Repayment indicator: 1 if U <= p2(x), else 0
    D = (U <= p2_test[:, np.newaxis]).astype(int)

    return D

# %%
D_test = simulate_repayment_scenarios_p2(data, m=m, num_scenarios=50000, random_seed=123)
```

```python
# %%
# Case (i)
def compute_pl_lend_everyone(
    D: np.ndarray,
    loan_amount: float = 1000.0,
    interest_rate: float = 0.055,
) -> np.ndarray:
    """
    Compute scenario P&L for policy (i): lend to EVERY test borrower
    at a fixed interest rate.

    We work scenario by scenario. For each borrower i and scenario k:

        If D_{i,k} = 1 (repayment):
            - We get back principal + interest: loan_amount * (1 + r)
            - We had to put up the principal: loan_amount
            -> Profit on this loan = loan_amount * r

        If D_{i,k} = 0 (default):
            - We lose the principal: - loan_amount
            -> Profit on this loan = - loan_amount

    So, for each loan:
        P&L_{i,k} = loan_amount * [ r * D_{i,k} - 1 * (1 - D_{i,k}) ]

    Total P&L in scenario k = sum over all test borrowers i.

    Parameters
    ----------
    D : np.ndarray
        Repayment indicator matrix of shape (n_test, num_scenarios),
        with entries 0 or 1.
    loan_amount : float, optional
        Principal per loan (CHF). Default: 1000.
    interest_rate : float, optional
        Interest rate

    Returns
    -------
    pl_scenarios : np.ndarray
        1D array of length num_scenarios, where each entry is the
        TOTAL P&L across all test loans in that scenario.
    """
    r = interest_rate

    # Compute P&L per loan and scenario:
    # shape (n_test, num_scenarios)
    pl_per_loan = loan_amount * (r * D - (1 - D))

    # Sum across loans (axis=0) to get total P&L per scenario
    pl_scenarios = pl_per_loan.sum(axis=0)

    return pl_scenarios

# %%
pl_case_i = compute_pl_lend_everyone(
    D_test,
    loan_amount=1000.0,
    interest_rate=0.055,
)

exp_pl_i = compute_pl_expectation(pl_case_i)
var_95_i, es_95_i = estimate_var_es(pl_case_i, alpha=0.95)

print("E[P&L] =", exp_pl_i)
print("VaR_95 (loss units) =", var_95_i)
print("ES_95  (loss units) =", es_95_i)

plot_pl_histogram(pl_case_i, "Case (i): Histogram of P&L - lend everyone at 5.5%", bins=50)
plot_pl_distribution(pl_case_i, "Case (i): Empirical CDF of P&L - lend everyone at 5.5%")
```

```python

# %%
# case (ii)
def compute_lending_indicator_logistic_y2(
    data: pd.DataFrame,
    m: int,
    log_model_y2,
    feature_cols: list[str] | None = None,
    threshold: float = 0.95,
) -> np.ndarray:
    """
    Compute a 0/1 lending indicator for EACH TEST applicant based on
    the logistic regression model fitted for y2.

    We do:

        1) Extract TEST features X_test (rows m,...,m+n-1).
        2) Compute predicted probabilities:
               p_hat = P(y2 = 1 | x, logistic model).
        3) Define lending rule:
               lend_i = 1 if p_hat_i >= threshold, else 0.

    Parameters
    ----------
    data : pd.DataFrame
        Full dataset with at least the feature columns.
    m : int
        Number of training observations (first m rows). The rest are TEST.
    log_model_y2 :
        Fitted LogisticRegression model for target y2.
        This is the model you obtained earlier with fit_logistic_regression.
    feature_cols : list of str or None, optional
        Names of feature columns. If None, defaults to:
            ["age", "income", "employment"]
    threshold : float, optional
        Lending threshold on predicted probability (default: 0.95).

    Returns
    -------
    lend_indicator : np.ndarray
        1D array of length n_test = len(data) - m, with entries in {0,1}.
        lend_indicator[i] = 1 means "lend to test borrower i".
    """
    if feature_cols is None:
        feature_cols = ["age", "income", "employment"]

    # Extract feature matrix X for all borrowers
    X_all = data[feature_cols].to_numpy()

    # Split into train and test manually (we only need test here)
    X_test = X_all[m:, :]  # rows m,...,m+n-1

    # Predicted probabilities on TEST set:
    # predict_proba returns [:,0] = P(y=0), [:,1] = P(y=1)
    p_hat_test = log_model_y2.predict_proba(X_test)[:, 1]

    # Lending decision: lend if predicted prob >= threshold
    lend_indicator = (p_hat_test >= threshold).astype(int)

    return lend_indicator

def compute_pl_lend_selective(
    D: np.ndarray,
    lend_indicator: np.ndarray,
    loan_amount: float = 1000.0,
    interest_rate: float = 0.01,
) -> np.ndarray:
    """
    Compute scenario P&L for a selective lending policy.

```

```
    Inputs:
    -------
    - D: 2D array of shape (n_test, num_scenarios)
         D[i,k] = 1 if test borrower i repays in scenario k
                = 0 if test borrower i defaults in scenario k

    - lend_indicator: 1D array of shape (n_test,)
         lend_indicator[i] = 1 if we decide to lend to borrower i
                           = 0 otherwise

    P&L per loan and scenario:

        If lend_indicator[i] = 1:

            If D[i,k] = 1 (repayment):
                P&L_{i,k} = + loan_amount * interest_rate

            If D[i,k] = 0 (default):
                P&L_{i,k} = - loan_amount

        If lend_indicator[i] = 0:
            P&L_{i,k} = 0

    We compute this vectorized for all i,k.

    Parameters
    ----------
    D : np.ndarray
        Repayment indicator matrix (n_test, num_scenarios), entries 0/1.
    lend_indicator : np.ndarray
        Lending indicator vector (n_test,), entries 0/1.
    loan_amount : float, optional
        Principal per loan, default 1000 CHF.
    interest_rate : float, optional
        Interest rate

    Returns
    -------
    pl_scenarios : np.ndarray
        1D array of length num_scenarios, total P&L per scenario.
    """
    r = interest_rate

    # Compute P&L per loan & scenario as if we lent to everyone:
    # shape (n_test, num_scenarios)
    pl_per_loan_full = loan_amount * (r * D - (1 - D))

    # Apply the lending indicator: if lend_indicator[i] = 0,
    # then the entire row i becomes 0.
    # We do this by multiplying each row i by lend_indicator[i].
    # Expand lend_indicator to shape (n_test, 1) so broadcasting works.
    lend_matrix = lend_indicator[:, np.newaxis]  # shape (n_test, 1)
    pl_per_loan = pl_per_loan_full * lend_matrix

    # Total P&L in each scenario: sum across all test borrowers (axis=0)
    pl_scenarios = pl_per_loan.sum(axis=0)

    return pl_scenarios

# %%
lend_indicator_log2 = compute_lending_indicator_logistic_y2(
    data, m=m, log_model_y2=log_model2, threshold=0.95
)
pl_case_ii = compute_pl_lend_selective(
    D_test,
    lend_indicator=lend_indicator_log2,
    loan_amount=1000.0,
    interest_rate=0.01,
)
```

```python
exp_pl_ii = compute_pl_expectation(pl_case_ii)
var_95_ii, es_95_ii = estimate_var_es(pl_case_ii, alpha=0.95)

print("Case (ii): E[P&L] =", exp_pl_ii)
print("Case (ii): VaR_95 =", var_95_ii)
print("Case (ii): ES_95  =", es_95_ii)

plot_pl_histogram(pl_case_ii, "Case (ii): Histogram of P&L", bins=50)
plot_pl_distribution(pl_case_ii, "Case (ii): Empirical CDF of P&L")

# %%
# case (iii)
def compute_lending_indicator_svm_y2(
    data: pd.DataFrame,
    m: int,
    svm_model_y2: SVC,
    scaler_y2: StandardScaler,
    feature_cols: list[str] | None = None,
    threshold: float = 0.95,
) -> np.ndarray:
    """
    Compute a 0/1 lending indicator for EACH TEST applicant based on
    the SVM model fitted for y2.

    Steps:
    ------
    1) Extract feature matrix for all borrowers.
    2) Take the TEST rows (indices m,...,m+n-1).
    3) Standardize them using the SAME scaler used to train the SVM.
    4) Compute predicted probabilities:
           p_hat_svm2(x_i) = P(y2 = 1 | x_i, SVM model)
    5) Lending rule on TEST set:
           lend_i = 1 if p_hat_svm2(x_i) >= threshold, else 0.

    Parameters
    ----------
    data : pd.DataFrame
        Full dataset with at least the feature columns.
    m : int
        Number of TRAINING observations (first m rows). The rest are TEST.
    svm_model_y2 : SVC
        Fitted SVM classifier for target y2 with probability=True.
    scaler_y2 : StandardScaler
        Fitted scaler used to standardize the features for svm_model_y2.
    feature_cols : list of str or None, optional
        Names of feature columns.
        If None, defaults to ["age", "income", "employment"].
    threshold : float, optional
        Lending threshold on predicted probability (default 0.95).

    Returns
    -------
    lend_indicator : np.ndarray
        1D array of length n_test = len(data) - m, with entries in {0,1}.
        lend_indicator[i] = 1 means "lend to test borrower i".
    """
    if feature_cols is None:
        feature_cols = ["age", "income", "employment"]

    # All features
    X_all = data[feature_cols].to_numpy()

    # TEST features: rows m,...,m+n-1
    X_test = X_all[m:, :]

    # Standardize using the SAME scaler as in training
    X_test_std = scaler_y2.transform(X_test)

    # Probabilities on TEST set
    # predict_proba returns [:,0] = P(y=0), [:,1] = P(y=1)
```

```
    p_hat_test = svm_model_y2.predict_proba(X_test_std)[:, 1]

    # Lending decision: lend if predicted prob >= threshold
    lend_indicator = (p_hat_test >= threshold).astype(int)

    return lend_indicator

# %%
lend_indicator_svm2 = compute_lending_indicator_svm_y2(
    data=data,
    m=m,
    svm_model_y2=svm_model2,
    scaler_y2=scaler,        # the StandardScaler used for SVM training
    feature_cols=["age", "income", "employment"],
    threshold=0.95,             # lend only if predicted prob >= 95%
)

pl_case_iii = compute_pl_lend_selective(
    D=D_test,
    lend_indicator=lend_indicator_svm2,
    loan_amount=1000.0,
    interest_rate=0.01,         # 1% interest
)

exp_pl_iii = compute_pl_expectation(pl_case_iii)
print(f"Case (iii) - E[P&L]   {exp_pl_iii:.2f} CHF")

plot_pl_histogram(
    pl_case_iii,
    title="Case (iii): Histogram of P&L - SVM selective lending at 1%",
    bins=50,
)

plot_pl_distribution(
    pl_case_iii,
    title="Case (iii): Empirical CDF of P&L - SVM selective lending at 1%",
)

var_95_iii, es_95_iii = estimate_var_es(pl_case_iii, alpha=0.95)
print(f"Case (iii) - 95% VaR (loss units): {var_95_iii:.2f} CHF")
print(f"Case (iii) - 95% ES  (loss units): {es_95_iii:.2f} CHF")
```