Deep Reinforcement Learning Application in Credit Scoring

Eduardo Perez Denadai

MSc in Artificial Intelligence
The University of Bath
2023

Deep Reinforcement Learning Application in Credit Scoring

Submitted by: Eduardo Perez Denadai

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Abstract

In the financial services sector, the integration of artificial intelligence (AI) into credit scoring systems marks a significant shift from conventional methods. Essential for financial institutions, credit scoring critically assesses individuals' creditworthiness, influencing decisions for both lenders and borrowers. Credit scoring encompasses a predictive phase, where the probability of missed payments is estimated, and a decision phase, where these probabilities are turned into decisions based on optimizing specific optimization objectives. The predictive phase often uses supervised learning to solve a binary classification problem using logistic regression but recent trends have shifted towards more sophisticated, data-driven machine learning (ML) techniques for enhanced prediction accuracy. The decision phase typically involves optimizing performance metrics, like the f1-score, or occasionally, minimizing misclassification costs in the confusion matrix.

This dissertation explores the utilization of deep reinforcement learning (DRL) for credit risk management within the mortgage lending sector, a different learning paradigm that stands in contrast to traditional supervised learning methods and is less explored based on the literature research. The primary focus is on developing and comparing different DRL agents in an environment where their performance is evaluated based on their ability to identify default signals in mortgage clients and learn the current portfolio default rate. The research begins with framing the problem as a binary classification task, where the agents are trained to distinguish between clients who will likely pay or default. As the study progresses, the complexity of the problem is increased by incorporating elements like delayed rewards and multi-class actions.

The implications of this research suggests that DRL could be a beneficial tool for credit risk evaluation, potentially leading to more dynamic or precise risk management methods in lending. While the study is exploratory in nature, it shows that under certain reward structures and hyper-parameter tuning DRL can be a viable modeling option to be used in financial risk management and financial decision-making for industry professionals seeking innovative risk assessment strategies.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank god, my family and my son and all those who had the patience to bear with me while I pursued this dream.

Chapter 1

Introduction

1.1 Motivation

The field of finance has witnessed a paradigm shift with the introduction of machine learning (ML) techniques, particularly in the realms of trading, portfolio optimization, and risk management. Among these techniques, Reinforcement Learning (RL) stands out due to its ability to learn optimal strategies through interaction with a dynamic environment. Recent explorations into the application of RL in finance have yielded promising results, yet in in specialized areas such as default prediction in credit scoring, is still in early stages of exploration and application.

Credit scoring, a pivotal component in the financial sector for risk estimation, has traditionally relied on statistical and supervised learning methods to assess credit risk. However, the advent of more sophisticated computational capabilities has paved the way for advanced methodologies in this domain. This thesis posits Deep Reinforcement Learning (DRL) as a feasible approach to credit scoring. Contrary to conventional models that primarily focus on computing the probability of default followed by a threshold-based classification, DRL agents propose an end-to-end framework for decision-making. This framework directly maps input data to predictions of default events. This approach offers a multifaceted advantage: it simplifies the modeling process, effectively manages complex reward structures, and is flexible enough to incorporate aspects such as interpretability, fairness, and adherence to regulatory standards within the scope of credit risk assessment.

A significant challenge in credit scoring is the abstract nature of financial data and the requirement for models to adapt to its dynamic landscape. The proposed research aims to construct a DRL-based environment adept at discerning between high and low-risk credit applications. This environment will be designed to account for scenarios characterized by delayed rewards and the necessity for decisions involving multiple actions. The focus will be on developing a DRL model that not only performs in accuracy for credit risk evaluation but also showcases flexibility and adaptability to evolving financial data trends.

1.2 Research Question

The central aim of this thesis is to investigate the capacity of model-free deep reinforcement learning (DRL) agents to classify loans as either good or bad. This exploration involves experimenting with various representations of the environment to determine how these agents

can effectively discern between loan classes. The research questions that drive this inquiry include:

- **Evaluating whether model-free DRL agents can successfully learn to classify good and bad clients, considering the complex patterns inherent in financial data.**
- **Determining the generalization capabilities of model-free DRL agents to new, unseen populations, which enables the model's applicability across different demographic segments and economic conditions.**

These questions are fundamental for advancing the application of DRL in financial risk assessment, potentially leading to more nuanced and adaptable credit scoring models.

Chapter 2

Literature and Technology Review

2.1 Credit Scoring

2.1.1 Credit Scoring Fundamentals

Credit scoring is the term used to describe methods used for classifying credit applications into 'good' and 'bad' risk classes (Hand and Henley, 1997). It is an essential component in financial decision-making, has evolved considerably since its inception around six decades ago. Initially based in the qualitative 5C's approach (i.e. eliciting data about applicant's character, capital, collateral, capacity, and condition) during the 1950's (Thomas, 2000).

Credit scoring has transitioned to more quantitative methods, driven by the necessity to efficiently process an increasing volume of loan applications (Dastile, Celik and Potsane, 2020) and more regulated market conditions in capital provisioning (IFRS 9 Financial Instruments, 2009). This shift was crucial in lending, aiming for effective and impartial assessment of loan applications.

The transition to statistical and automated procedures, primarily logistic regression models focused on default risk, marked a pivotal change in credit scoring in the mid-20th century (Dastile, Celik and Potsane, 2020; Baesens et al., 2003). However, these models often fell short in addressing factors like loan profitability, leading to the development of integrated models that assess both default risk and potential financial returns (H. Greene, 2008).

Similarly, default prediction has evolved since the 1950's from traditional statistical techniques, such as Fisher's linear discriminant analysis (LDA) (Fisher, 1936) and logistic regression (LR), to advanced machine learning methods, from support vector machines (SVM) (Cortes and Vapnik, 1995) to artificial neural networks (ANN) (Jensen, 1992). This early progression reflects the trend towards more sophisticated, accurate predictive methods in financial risk assessment leveraging large, diverse and heterogeneous data we see today (Devi and Radhika, 2018).

2.1.2 Credit Scoring in Industry

Credit scoring is a pivotal factor in commercial lending, as it directly influences capital liquidity. It serves as a key component in loan loss provisioning, which is mandated by regulatory standards such as Basel Accord III and the International Financial Reporting Standard (i.e. IFRS9)

published by the International Accounting Standards Board (IFRS 9 Financial Instruments, 2009). These regulations require that financial institutions maintain adequate capital reserves in accordance with the credit risks they undertake. The capital provisioned is meant to ensure that banks can absorb potential losses and continue to provide essential services, such as issuing credit and maintaining liquidity, without excessively restricting their economic activities (Begenau, 2020). Within this guidelines, the principal quantity of interest is the probability of default of every operation (Thomas, Crook and Edelman, 2017).

The expected credit loss $\mathbb{E}(\text{Loss}_i)$ provisioning, as outlined by international standards such as Basel III accord IFRS9 Standard, is predicated using three general of risk factors: Probability of Default (PD_i), Loss Given Default (LGD_i), and Exposure at Default (EAD_i) for every i^{th} client. These factors are quantitatively captured in the capital provisioning formula:

$$\mathbb{E}(\text{Loss}_i) = EAD_i \times LGD_i \times \mathbf{PD}_i \tag{2.1}$$

Within this formula, the Probability of Default (PD) quantifies the likelihood of a borrower failing to meet their loan repayments, which is modeled by means of Equation 2.2. The Loss Given Default (LGD) gauges the expected loss as a proportion of the exposure, should default occur. Meanwhile, the Exposure at Default (EAD) denotes the total amount of credit exposed at the point of default. Credit scoring models are used to build the PD term, while LGD and EAD assessments are integral to the broader credit risk evaluation framework. Improvements in the methods to estimate PD promise to substantially improve the accuracy of credit loss forecasting, thus impacting the determination of necessary capital reserves and, consequently, influencing the robustness and efficiency of the banking sector's credit offerings.

In practice, the development of credit scoring models within financial institutions unfolds in a complex environment, encompassing many departments, and processes that must align with a multitude of business, regulatory, and operational considerations. In general, the development begins with the meticulous gathering of relevant data, tailored to the specific populations, products, and timelines. This initial phase is constrained and shaped by business requirements, which filter and modify the data to adhere to regulatory standards and pertinent business use case characteristics. The model development process requires a precise definition of the target variable, which is crucial for capturing instances of credit mispayment. This definition must be carefully checked to ensure the integrity of the target variable, safeguarding against any potential leakage from the feature space that could compromise the model's accuracy (Siddiqi, 2012)

The subsequent steps involve methodically splitting datasets to ensure the model's generalization, followed by feature selection, where business and regulatory constraints heavily dictate the process, often limiting the extent of feature engineering due to demands for interpretability, quality, or even fairness. Once features are engineered, often within the bounds of these constraints, the model progresses through a series of candidate model training, selection, validation through backtesting and hold-out sets, and verification stages. This involves not just the application of statistical methods but also the incorporation of business insights and compliance with regulatory frameworks. The final stages of the process include an approval scheme, which can range from straightforward to highly bureaucratic, and ultimately, the model is subject to continuous performance monitoring to ensure it remains effective over time and reflective of the dynamic financial landscape (see figure 4.2. Each of these steps is critical in the lifecycle of a credit scoring model, contributing to its robustness, compliance, and ability

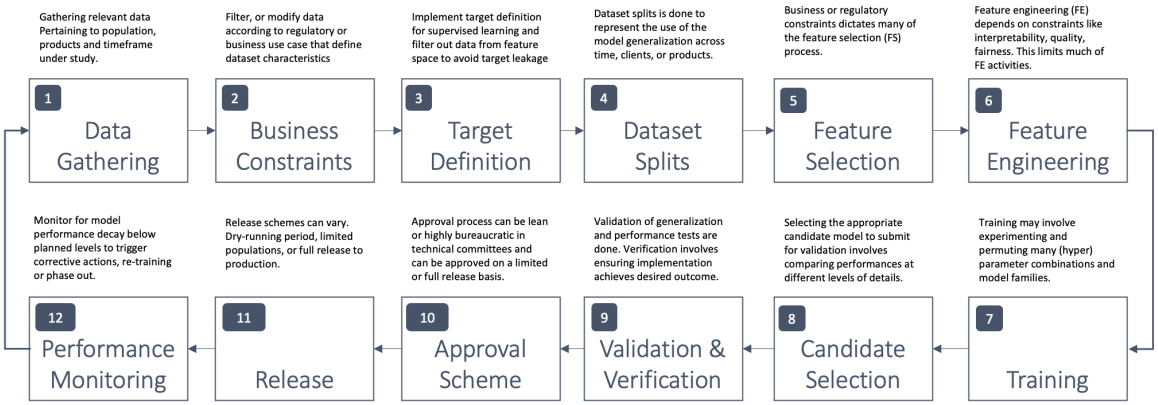


Figure 2.1: Example of Credit Scoring Model Development Process

to predict credit risk in a commercially viable manner.

2.2 Machine Learning Methods

2.2.1 Supervised Learning Approach

Credit scoring models in finance are often structured as supervised learning classifiers. The task begins by defining a mispayment event, which refers to the failure to meet a contractual monthly installment on a loan. The default signal is then constructed based on the accumulation of such mispayment events over a contiguous period of N days, typically 90 days. This establishes the foundation for the positive class, which corresponds to the occurrence of a default event (Siddiqi, 2012).

The simplest formulation of the problem proceeds in two stages. Initially, the problem is modeled as a logistic regression where the probability of default given a feature vector $\mathbf{x}$ is represented as:

$$P(\text{y="default"}|\mathbf{x}) = f(\mathbf{x};\theta) \tag{2.2}$$

$$P(y = \text{"default"}|\mathbf{x};\boldsymbol{\theta}) = \frac{1}{1+e^{-\boldsymbol{\theta}^\top \mathbf{x}}} \tag{2.3}$$

In this equation, $P(y = \text{"default"}|\mathbf{x};\boldsymbol{\theta})$ specifies the probability that a client defaults on their obligation, with $\mathbf{x} = [x_1, x_2, \dots, x_n]^\top$ denoting the vector of client features. The logistic regression parameters are given by $\boldsymbol{\theta} = [\theta_0, \theta_1, \dots, \theta_n]^\top$, where θ_0 is the bias term and $\theta_1, \dots, \theta_n$ are the feature weights.

Subsequently, in the second stage, the predicted probabilities are converted into binary classification outcomes via a decision function:

$$\hat{y} = \mathbb{1}_{\{\hat{p}\geq\delta\}}(p(y = \text{"default"}|\mathbf{x};\boldsymbol{\theta})) \tag{2.4}$$

The indicator function $\mathbb{1}_{\{\hat{p}\geq\delta\}}$ is defined as:

$$\mathbb{1}_{\{\hat{p}\geq\delta\}}(x) := \begin{cases} 1 & \text{if } x \geq \delta, \\ 0 & \text{otherwise,} \end{cases} \tag{2.5}$$

where x is the predicted probability of default and δ is the decision threshold, or cutoff. The selection of δ is informed by the need to balance the class distribution in the training data, minimize classification errors, or other more complex criteria designed to optimize other performance metrics, and more importantly, generalization to unseen data.

The choice of employing a more expressive model for Equation 2.2 — be it Ensembles, Boosting, Bagging, Nearest Neighbor, Support Vector Machines, Decision Trees, or more sophisticated Artificial Neural Networks (ANN) and Deep Learning (DL) approaches— is contingent upon heuristics, software or hardware constraints, business or regulatory guidelines, and the benefits of the modeling approach. Each of these methodologies has its own set of advantages and disadvantages Adam et al. (2019); Dastile, Celik and Potsane (2020).

2.2.2 Reinforcement Learning (RL)

The integration of Reinforcement Learning (RL) within the financial sector has led to significant advancements in various applications, including trading, portfolio optimization, and risk management. This review focuses on the impact of RL in these areas, noting the potential yet to be unlocked in specific sub-domains such as default prediction within credit scoring.

Reinforcement Learning is an important paradigm in learning theory with active research in artificial intelligence and machine learning, focusing on decision-making processes of agents within an environment. It stipulates that an agent learns to act by trial and error guided by reward signal given by the environment. It has been applied across diverse fields such as natural and social sciences, and engineering (Sutton and Barto, 1998, 2018; Bertsekas and Tsitsiklis, 1996; Bertsekas, 2012; Szepesvári, 2010; Powell, 2011). The integration of RL with neural networks, particularly deep learning, has led to significant advancements in the field, marking the renaissance of reinforcement learning (Sutton and Barto, 2018; Bertsekas and Tsitsiklis, 1996; Schmidhuber, 2015; Krakovsky, 2016).

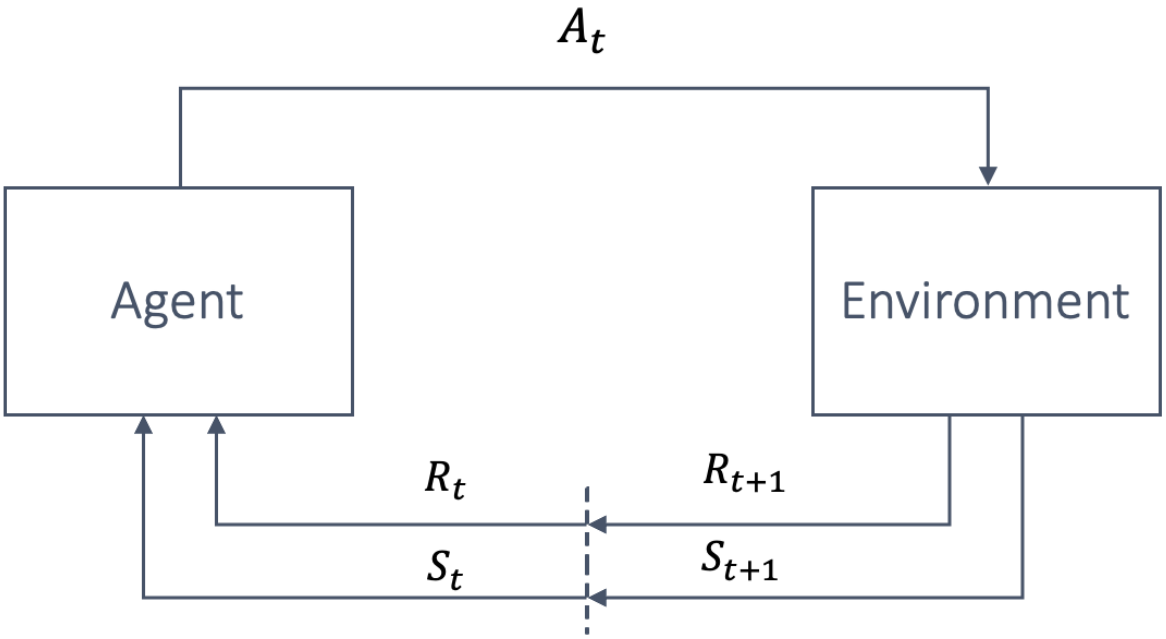


Figure 2.2: General agent-environment interaction in a Markov Decision Process (Sutton and Barto, 1998)

Deep Reinforcement Learning (DRL) has achieved remarkable success in various industrial applications. In the realm of autonomous vehicles, companies are leveraging DRL to enhance self-driving algorithms, leading to more efficient and safer navigation (Kiran et al., 2021). Another notable application is in robotics, where DRL has been used by Boston Dynamics for complex motion planning and control in their robots (Han et al., 2023). In finance, fintechs and large financial investment firms have applied DRL to optimize trading strategies, minimize portfolio risk and fraud detection (Bouchti et al., 2017; Hu and Lin, 2019; Zhang, Zohren and Roberts, 2019). Additionally, DRL has made significant contributions to healthcare, particularly in personalized medicine and predictive analytics, with breakthroughs in protein folding prediction (Jumper et al., 2021). These successes illustrate the versatile and transformative impact of DRL across multiple industries.

Reinforcement Learning (RL) is mathematically conceptualized within the framework of a Markov Decision Process (MDP). This framework is particularly suited for RL due to its efficacy in modeling learning processes that are based on interactions aimed at achieving specific goals. In this context, the learning entity is commonly referred to as the *agent*. The *agent* interacts with, and learns from, an external system known as the *environment*. This interaction is central to the RL paradigm, where the *agent* dynamically makes decisions based on the state of the *environment* to maximize certain predefined objectives (see figure 2.2.

A Markov Decision Process (MDP) is fundamental to understanding Reinforcement Learning (RL). An MDP is defined by a set of states S, an action space A, a reward function $r : S \times A \to \mathbb{R}$, and a transition probability function $P : S \times A \times S \to [0, 1]$. Here, $r(s, a)$ denotes the reward for taking action $a \in A$ in state $s \in S$, and $P(s'|s, a)$ represents the probability of transitioning to state s' after action a is taken in state s (Sutton and Barto, 1998).

- A *state* $s \in S$ represents the current situation or configuration of the environment.
- An *action* $a \in A$ is a choice made by the agent in response to the current state.
- A *reward* $r(s, a)$ is the immediate feedback received after taking action a in state s.
- A *policy* $\pi : S \to A$ is the strategy the agent employs to determine its actions. it maps states to probability of taking each possible action.
- *Value Functions*: In a value-based model-free approach, the agent learns a value function that estimates the expected return (cumulative discounted reward) from a given state or state-action pair. This can be represented as $V(s)$ for state-value functions or $Q(s, a)$ for action-value functions, where s represents the state and a represents the action.

The goal in reinforcement learning is to find an optimal policy π^* that maximizes the cumulative discounted reward G_t. The value function $V^\pi(s)$ under a policy π and the action-value function $Q^\pi(s, a)$ are useful functions used to estimate how good it is for an agent to be in a given state and are defined as follows:

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k, \tag{2.6}$$

$$V^\pi(s) \doteq \mathbb{E}[G_t | S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right] \tag{2.7}$$

$$Q^{\pi}(s,a) \doteq \mathbb{E}[G_t|S_t = s, A_t = a] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right], \tag{2.8}$$

where γ is the discount factor.

Solving a task, therefore, means finding a policy that achieves the most reward in the long run. Value functions are useful in this pursuit as they induce a partial ordering over the policies such that $\pi \geq \pi'$ if and only if $V_{\pi}(s) \geq V_{\pi}(s')$. An optimal policy π^* is one that maximizes $Q^{\pi}(s,a)$ for all $s \in S$ and $a \in A$. It can be expressed in terms of $V^{\pi}(s)$ as:

$$V^{\pi^*}(s) = \max_{a \in A} Q^{\pi^*}(s,a). \tag{2.9}$$

Value functions in Reinforcement Learning (RL) possess a fundamental recursive property, as articulated by the *Bellman optimality equation*. Under this equation, the value of a state under an optimal policy is equated to the expected return of the best action taken in that state. This is expressed as:

$$V^{\pi^*}(s) = \max_{a \in A} \mathbb{E}\left[R_{t+1} + \gamma V^{\pi^*}(S_{t+1})|S_t = s, A_t = a\right], \tag{2.10}$$

where $V^{\pi^*}(s)$ is the value of state s under the optimal policy π^*, R_{t+1} is the reward at the next time step, γ is the discount factor, and S_{t+1} is the state at the next time step. Similarly, the action-value function under the optimal policy can be defined recursively as:

$$Q^{\pi^*}(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a' \in A} Q^{\pi^*}(S_{t+1}, a')|S_t = s, A_t = a\right], \tag{2.11}$$

where $Q^{\pi^*}(s,a)$ is the value of taking action a in state s under the optimal policy. These recursive formulations are central to the understanding of RL, providing a framework through which agents learn and make decisions to achieve optimal outcomes.

2.2.3 Classification of RL Algorithms

2.2.4 Model-Based vs. Model-Free RL

In the domain of Reinforcement Learning (RL), algorithms are typically categorized into two primary families: model-based and model-free. Each of these families adopts a distinct approach towards learning and decision-making, catering to different aspects of environment interaction and policy development.

Model-Based RL

Model-based RL algorithms operate by constructing an internal representation of the environment, known as the model. This model is an approximation of the Markov Decision Process (MDP), a fundamental concept in RL which provides a mathematical framework for modeling decision-making situations where outcomes are partly random and partly under the control of a decision-maker.

In model-based RL, the agent estimates the transition probability matrix $P(s_{t+1}|s_t, a_t)$ and the reward function $R(s_t, a_t)$ and uses them to make decisions. The key advantage is that having a model allows the agent to plan by thinking ahead over simulated model of the environment, observing the consequences of actions before taking them.

Model-Free RL

Conversely, model-free RL algorithms do not attempt to build a model of P and R. Instead, they directly learn either the value function or the policy from the interactions with the environment. In model-free approaches, learning and decision-making are based solely on the agent's experiences, without the need for an explicit model of the environment. This type of Rl algorithms can be value-based methods or policy-based methods.

1. **Value-Based Methods**: These methods focus on learning the value function. A quintessential example is Q-learning, where the Q-value function $Q(s, a)$ represents the expected utility of taking action a in state s, and then following the optimal policy thereafter. The Q-values are updated using the Bellman equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right)$$

   where α is the learning rate, R_{t+1} is the reward received after taking action a_t, and γ is the discount factor.

2. **Policy-Based Methods**: These methods directly learn the policy function $\pi(a|s)$ that maps states to actions. An example is the policy gradient method, where the policy is typically represented by a parametric probability distribution that is adjusted in the direction of higher expected return. The policy gradient theorem provides a foundation for this learning, represented by:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_\pi \left[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s, a) \right]$$

   where $\nabla_\theta J(\pi_\theta)$ is the gradient of the policy's performance, and $Q^\pi(s, a)$ is the action-value function under policy π.

The choice between model-based and model-free RL approaches depends on various factors including the complexity of the environment, the availability of computational resources, and the specific requirements of the task at hand. Model-based methods are advantageous in environments where accurate models are available or can be learned with reasonable effort, as they can be more sample efficient. Model-free methods, on the other hand, are often more straightforward and can be more effective in complex environments where modeling the dynamics is impractical or infeasible. They are generally preferred when the primary goal is to learn an effective policy without necessarily understanding the underlying environmental model.

2.2.5 Actor-Only Approach

The agent learns a direct policy from states to actions, suitable for environments with continuous action spaces, faster convergence, and higher transparency. Some example algorithms in this category are:

- **Policy Gradient Methods**: These methods optimize the policy function directly, often using gradient ascent techniques (Sutton et al., 1999).

- **Deterministic Policy Gradient (DPG)**: DPG is particularly suitable for high-dimensional, continuous action spaces. It represents a deterministic policy that directly selects actions without the need for a probability distribution (Silver et al., 2014).
- **Deep Deterministic Policy Gradient (DDPG)**: A deep learning-based extension of DPG for high-dimensional, continuous action spaces (Lillicrap et al., 2019).
- **Proximal Policy Optimization (PPO)**: PPO simplifies and improves upon TRPO. It uses a clipped objective to maintain stable policy updates, thereby ensuring that the new policy does not deviate too far from the old one. This approach has gained popularity due to its ease of implementation and effective performance across a range of environments (Schulman et al., 2017a).
- **Trust Region Policy Optimization (TRPO)**: TRPO aims to take the largest possible improvement step on a policy while remaining within a specific trust region, thus ensuring stable and reliable policy updates. It's designed to prevent drastic policy updates that might lead to performance degradation, making it a key algorithm in the actor-only family (Schulman et al., 2015).

2.2.6 Critic-Only Approach

his approach learns a value function to evaluate the outcomes of different actions. It is highly flexible and applicable to a wide range of problems, including complex reward schemes. Some example algorithms in this category are:

- **Value Iteration Algorithms**: These algorithms continuously update the state value to the best expected return (Bellman and Bellman, 1957; Bertsekas, 2007).
- **Q-Learning and Deep Q-Networks (DQN)**: Model-free methods where the agent learns the value of actions in specific states, with DQN integrating deep learning for high-dimensional spaces (Watkins and Dayan, 1992; Mnih et al., 2013).

2.2.7 Actor-Critic Approach

Combines the benefits of both the actor-only and critic-only approaches. It uses two agents – an actor that determines actions and a critic that evaluates these actions. Some example algorithms in this category are:

- **Advantage Actor-Critic (A2C) and Asynchronous Advantage Actor-Critic (A3C)**: These algorithms estimate the advantage of actions in each state, enhancing policy focus on beneficial actions (Mnih et al., 2016).
- **Soft Actor-Critic (SAC)**: Designed for continuous action spaces, SAC uses policy entropy to encourage exploration (Haarnoja et al., 2018).

2.3 Reinforcement Learning in Finance

2.3.1 RL in Trading Strategies

Reinforcement Learning (RL) has emerged as an important approach in the realm of algorithmic trading, reflecting a paradigm shift from traditional rule-based systems to adaptive, data-driven

strategies (Dixon, Halperin and Bilokon, 2018). In the dynamic and often unpredictable financial markets, RL agents excel by continuously interacting with market data, thereby adapting and optimizing trading strategies over time (Zhang, Zohren and Roberts, 2019; Guan and Liu, 2021). This adaptability is particularly important given the volatile nature of financial markets, where trading algorithms are heavily reliant on their capacity to discern and respond to subtle market changes (Moody and Saffell, 1998).

The autonomous learning feature of RL, grounded in trial-and-error, empowers these agents to uncover patterns in price movements, often unseen to traditional analytical techniques. This has significant implications for high-frequency trading, portfolio management, and risk assessment. Additionally, the integration of RL in financial applications extends to the realm of credit risk analysis, where the ability to accurately classify loans or predict default rates can substantially impact profitability and risk mitigation (Kolm and Ritter, 2019).

2.3.2 RL in Portfolio Optimization

Portfolio optimization is a critical aspect of financial management and involves dynamically allocating funds across various financial products. Traditional methods in portfolio management often fall short in addressing the complex, ever-changing nature of financial markets. In contrast, Reinforcement Learning (RL) has offered a dynamic and adaptive approach to asset allocation. By continuously learning from market conditions, RL algorithms are capable of adjusting portfolios in real-time, effectively balancing risk and return (Jiang, Xu and Liang, 2017; Hu and Lin, 2019).

The application of RL in portfolio management has shown promise in navigating the multifaceted landscape of financial markets. These algorithms leverage historical and real-time data to make informed decisions, optimizing asset allocation to maximize returns while mitigating risks. Furthermore, the integration of deep learning techniques has enhanced the capability of RL models to process complex, high-dimensional data, providing more robust investment strategies (Costa, 2023; Lu, 2023).RL models offer a sophisticated framework for understanding and reacting to market dynamics. They represent a significant shift from traditional, static models towards a more agile, data-driven approach in financial decision-making (Jang and Seong, 2023).

In conclusion, RL stands as a serious framework for financial portfolio management. Its ability to learn and adapt in an environment characterized by uncertainty and rapid changes positions it as a pivotal tool in modern finance, departing from traditional methodologies and paving the way for innovative investment strategies.

2.3.3 RL in Financial Risk Management

Risk management is pivotal in the finance sector, where Reinforcement Learning (RL) has found numerous applications.The dynamic nature of RL allows for continuous adaptation to market conditions, crucial in managing financial risks effectively (Song et al., 2020; Handhika, Sabri and Murni, 2021). This adaptability becomes particularly valuable in complex scenarios such as loan underwriting, financial distress prediction, and fraud detection, where the capabilities of Deep Learning (DL) enhance the precision of risk management processes (Bouchti et al., 2017).

Its versatility enables RL algorithms to handle a wide range of risk-related tasks, offering more

accurate predictions. For instance, in loan underwriting, RL can optimize decision-making by learning from historical data, thus reducing the likelihood of financial defaults (Jiang, Xu and Liang, 2017). In the field of fraud detection, RL algorithms are capable of identifying anomalous patterns, aiding in the early detection and prevention of fraudulent activities (Lu, 2023).

While RL has been applied in other areas of risk management finance, its use in credit scoring, specifically in default prediction, is less explored. RL's potential to dynamically adjust to new information and learn from complex patterns makes it an ideal candidate for improving the accuracy of default predictions in credit scoring systems.

Chapter 3

Methodology

3.1 Dataset

3.1.1 Sample

Indexes		Target	Features										
date_code	client_id	default_120_12m	mispay_days	mispay_d_90d_12m	total_balance	income	age	mortgage	mortgage_qty	maturity_months	monthly_payment	subsidy	interest_rate
11	261	1	155	211	29,425	855	49	50,000	1	435	274	0	6
9	504	0	0	3	42,416	1,227	54	66,300	1	240	640	0	8
14	361	1	421	635	39,607	871	42	57,285	1	381	365	0	6
5	392	1	574	939	49,603	742	27	52,920	1	360	193	1	5
4	256	0	0	29	42,739	892	36	53,900	1	360	230	1	5
6	460	0	0	5	223,355	1,180	37	228,740	1	398	1,321	0	6
6	381	0	24	52	41,173	777	36	46,579	1	360	210	1	5
14	364	1	89	203	39,924	888	45	61,250	1	360	349	0	6
⋮													
14	437	1	57	166	58,310	773	25	62,230	1	365	250	1	5
2	152	0	25	53	75,883	850	34	86,786	1	410	418	0	4

Figure 3.1: Dataset sample.

3.1.2 Provenance

The dataset utilized for this study is a synthetic construct, produced by employing a Tabular Variational Autoencoder (TVAE) to model genuine mortgage loan data, subsequently stripped of any personally identifiable information (Xu et al., 2019). This synthetic dataset was procured from a financial institution in the Latin American region and encompasses records spanning from March 2020 to December 2022, yielding a historical breadth of 34 months.

There are four distinct informational domains—client demographics, loan specifics, payment transactions, and target variable (see table 3.1). The dataset covers 14 variables of different information domains, 603 individual clients holding mortgage loans without censoring, amounting to a total of 12,663 data entries. This compilation of synthetic data ensures privacy while providing a realistic experimental analysis, for ease of visualization and referencing in plots the original variable names were changed to their corresponding aliases.

Table 3.1: Dataset feature definitions showing all information domains and Aliases used in the thesis.

Feature Domain	Feature Name	Definition	Alias
Transactional Data	mispay_days	days without paying loan.	x_1
	date_code	numerical date	x_2
	mispay_d_90d_12m	mispayments of the 90 days.	x_3
	total_balance	Total balance of debt.	x_4
Client Data	client_id	Deidentified client id.	x_5
	income	Declared income of the client.	x_6
	age	Client age in years.	x_7
Loan Data	mortgage	Total mortgage loan amount	x_8
	mortgage_qty	Count of number of loans	x_9
	maturity_months	Mean of total maturity of loans	x_10
	monthly_payment	Monthly payment amount of loan.	x_11
	subsidy	Whether loan has subsidy or not.	x_12
	interest_rate	Interest rate of the loan.	x_13
Target	default_120_12m	120 contiguous mispayments.	x_14

3.1.3 Description

The dataset used represents a typical example of *profile data* in a *hierarchical* or *repeated measures* structure (Johnson, 2020). Each row in the dataset corresponds to a specific measurement of a client's loan status, with columns detailing aspects of their loan payment history over time. This longitudinal aspect allows for the analysis of trends and patterns in credit behavior over extended periods, which is crucial for understanding the dynamics of credit risk and financial health of clients (figure 3.2).

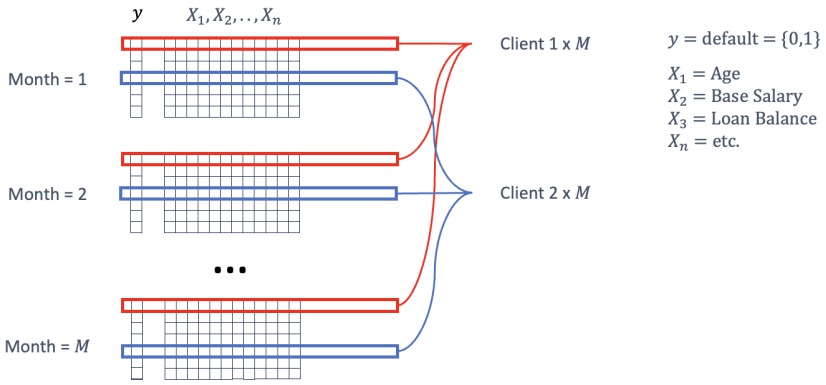


Figure 3.2: Dataset Architecture. This depicts a repeated measures dataset, showcasing individual client loan profiles. Within each month, clients are treated as independent entities, while their data exhibits temporal correlation, reflecting continuity and trends in their loan profiles over time.

This dataset shows the evolution of each client's credit status, affecting their creditworthiness. However, the challenge lies in appropriately summarizing and extracting predictive information from this hierarchical data structure. Techniques like aggregation of temporal data, transformation of time-series variables, and modeling of intra-client correlations are essential to leverage the full potential of this dataset.

The raw dataset contains a very 34 months worth of data from the March, 2020 until December 2022 (figure 3.3). The default rate can be observed to have suffered a large increase after June, 2021. The dataset shows heteroskedastic behavior with a large increase in volatility after June, 2021.

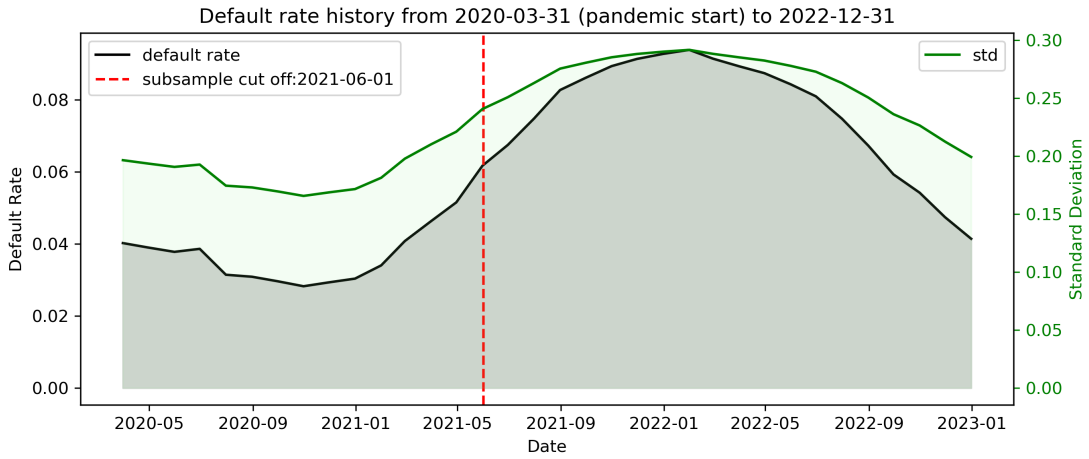


Figure 3.3: Default rate historical data. The raw dataset has a constant sample size of clients over the whole timeline. Pandemic effect can be observed affecting the payment behavior.

An important quantity is the *default rate* which used in traditional supervised learning approach to modeling the probabilities of default shown in equation 2.2) as the target for learning. Hover, the real effect of interest is default rate, so to compute it we use the formula:

$$\text{Default Rate}_m = \frac{1}{N_m} \sum_{i=1}^{N_m} y_{i,m} \tag{3.1}$$

where:

- Default Rate$_m$ is the default rate for month m.
- N_m is the total number of loans for month m.
- $\sum$ denotes the summation over all loans.
- $y_{i,m}$ is the default variable for the i-th loan in month m, which takes a value of 1 if the loan has defaulted and 0 otherwise.

The historical analysis of total default rates, as depicted in Figure 3.3, captures the substantial impact of the COVID-19 pandemic on credit loan defaults. A marked increased in default rates is evident at the onset of the pandemic, with rates surging from 2.8% to 9.3%, reflecting the adverse economic uncertainty and the ensuing financial hardships faced by loan holders during this period. This trend is corroborated by literature indicating that banking sector interventions

during the pandemic provided temporary relief to borrowers, mitigating the immediate rise in defaults. However, the real effects the reliefs resulted in a delayed increase in post-pandemic default rates, stabilizing after two years (Byun, Game et al., 2021; Nigmonov and Shams, 2021).

3.1.4 Statistics

Target Imbalance

The dataset defines the default variable y as both a measure of the default rate and an index of target variable class imbalance over time, as shown in Equation 3.1. Consequently, the data exhibits a significant class imbalance.

In machine learning, such imbalances can skew models towards the over-represented class, compromising predictions for the under-represented, yet often more important, class—such as defaults in financial models. Standard metrics like accuracy are inadequate for these datasets; hence, more nuanced metrics are advisable. Strategies to correct imbalance include resampling, cost-sensitive learning, or anomaly detection (Ramyachitra and Manikandan, 2014; Kotsiantis, Kanellopoulos and Pintelas, 2005).

Moreover, considering the computational constraints, undersampling of the majority class was perfomed to address class imbalancem leading to a more balanced and stable default rate, as depicted in Figure 3.4. The cut off date for out-of-time testing set is defined June, 2021.

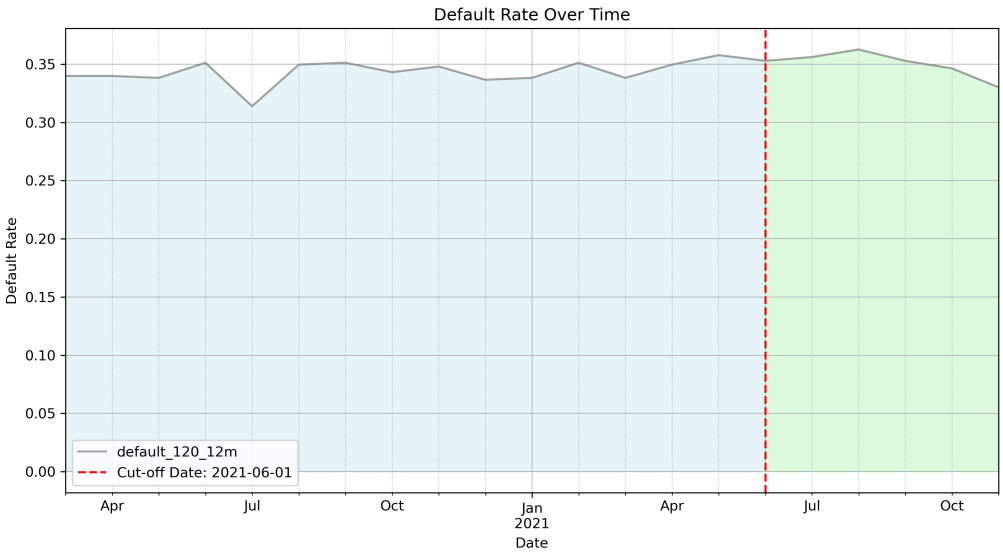


Figure 3.4: Undersampled showing new default rate historical data showcasing a more balanced target class in the variable but default.

General Statistics

When dealing with longitudinal tabular datasets, especially in the context of time series like credit characteristics over time, selecting an appropriate splitting methodology is crucial. Traditional random splitting methods might not be ideal due to the temporal nature of the data. Instead, an out-of-time (OOT)generalization approach, where the dataset is split based

on time, offers a more realistic assessment of the model's performance on unseen future data. This method ensures that the training set consists of observations from an earlier period, while the test set comprises data from a later period, mimicking real-world scenarios where models must generalize to future events. Such an approach is essential for reinforcement learning environments where the agent's performance is tested on its ability to make predictions for future time periods (Catania et al., 2022).

Table 3.2: Training set descriptive statistics

Variable Name	Count	Mean	Std	Min	25%	50%	75%	Max
mispay_days	9840	61.31	148.61	0	0	0	26	1067
date_code	9840	7.5	4.61	0	3.75	7.5	11.25	15
mispay_d_90d_12m	9840	139.92	225.61	0	3	41	158	1927
total_balance	9840	68816.8	92212.32	2550.93	30842.11	46055.29	70185.56	1256219.66
client_id	9840	307	177.54	0	153	307	461	614
income	9840	899.27	274.5	711.77	784.52	840.32	926.14	5042.41
age	9840	43	11.16	24	34	41	51	75
mortgage	9840	84076.58	108971.7	11205.95	40000	57025	79800	1275000
mortgage_qty	9840	1.05	0.22	1	1	1	1	2
maturity_months	9840	359.07	52	120	360	360	375	720
monthly_payment	9840	577.87	7587.56	71.42	190.97	268.62	420.22	530384.24
subsidy	9840	0.38	0.49	0	0	0	1	1
interest_rate	9840	5.35	0.95	3	5	5	5.75	9.5
default_120_12m	9840	0.34	0.47	0	0	0	1	1

Table 3.3: Test set descriptive statistics

Variable Name	count	mean	std	min	25%	50%	75%	max
mispay_days	3075	70.6	180.48	0	0	0	31	1913
date_code	3075	18	1.41	16	17	18	19	20
mispay_d_90d_12m	3075	142.12	240.25	0	3	37	177	1927
total_balance	3075	68081.13	90784.76	1936.65	30120.67	45983.57	69967.49	1229677.27
client_id	3075	307	177.56	0	153	307	461	614
income	3075	899.27	274.54	711.77	784.52	840.32	926.14	5042.41
age	3075	43.56	11.15	25	35	42	52	75
mortgage	3075	84256.1	109512.54	11205.95	40000	57000	79800	1275000
mortgage_qty	3075	1.05	0.22	1	1	1	1	2
maturity_months	3075	367.17	62.66	120	360	360	385	720
monthly_payment	3075	470.69	709.31	50.39	186.33	268	422	10528.25
subsidy	3075	0.38	0.49	0	0	0	1	1
interest_rate	3075	5.35	0.95	3	5	5	5.75	9.5
default_120_12m	3075	0.35	0.48	0	0	0	1	1

The descriptive statistics of the training and test sets for a machine learning model on credit characteristics of clients reveal insights critical for modeling. The training set, comprising 9,840 observations, displays a wide range of values across variables like mispayment days, total balance, and income. Notably, the mean mispayment days (61.31) with a high standard deviation (148.61) indicate significant variability, which is essential for training robust models. The test set, with 3,075 observations, mirrors these characteristics, ensuring consistency in model evaluation.

Key metrics like the average age and income across both sets are similar, suggesting a representative split. However, the maximum values for certain variables, such as monthly

payment, exhibit extreme ranges, highlighting potential outliers or edge cases. The presence of categorical variables like subsidy and default status with binary values will aid in classification tasks.

Overall, the datasets provide a enough comprehensive view of clients' credit behavior over time, suggesting that the model trained on such data could generalize well when predicting future trends or assessing credit risk. The similarity in distribution between the training and test sets is crucial for out-of-time generalization.

Other useful visual analysis tools like correlation plots are shown in the appendix A section.

Chapter 4

Implementation

4.1 Environment Formulation

In reinforcement learning (RL), three core elements are pivotal to the training and performance of an agent: the design of the environment, the construction of appropriate reward mechanisms, and the formulation of state representation. The environment establishes the context in which the agent operates, thereby influencing the intricacies and dynamics inherent to the learning task. Effective reward structuring is important in steering the agent towards achieving the desired behaviors, encapsulating the objectives of the learning process. Moreover, state representation is critical in enabling the agent to perceive and interpret its surroundings efficiently, thus impacting its capacity for learning and decision-making.

The sequential characteristic of the dataset, as depicted in Figure 3.2, necessitates a sampling approach that maintains the temporal continuity of the data. To this end, we implement a stratified sampling methodology, which ensures a thorough evaluation of all records within a given month prior to proceeding to the subsequent one. This technique is crucial in preserving the chronological order of events, thereby facilitating the agent's ability to recognize and assimilate temporal patterns indicative of defaults. This approach aligns with the established methodologies in time-series analysis within the domain of machine learning, which is critical for making inferences from historical data to future predictions (Brownlee, 2018).

In the context of model-free RL, the nature of the environment exerts a significant influence on the learning process. *Episodic environments* refer to those in which the agent's experiences are segmented into distinct episodes, each comprising a series of states, actions, and rewards, culminating in a terminal state. This episodic framework fosters learning from discrete experiences and is aptly suited for tasks characterized by definitive beginnings and conclusions. Conversely, *non-episodic environments* are marked by continuous interactions that persist indefinitely, making them essential for scenarios necessitating continuous decision-making, or control, without explicit endpoints (Sutton and Barto, 2018).

In this thesis, the aim for the RL agent is to correctly forecast default events for clients across the entire temporal span of the dataset. This objective entails the agent's examination of individual observations and the execution of binary predictions: assigning a '1' to denote a client's default status or a '0' for non-default. Accordingly, we conceptualize the environment as episodic, with each month representing an episode, and where rewards do not carry over to subsequent episodes.

4.1.1 Reward Representation

For classification tasks in a reinforcement learning context, the reward R_t is calculated based on the agent's performance, assessed through a confusion matrix M and a corresponding cost matrix C. M is represented as $M = \begin{bmatrix} TP & FP \\ FN & TN \end{bmatrix}$, with standard definitions for TP, TN, FP, and FN. C is defined as $C = \begin{bmatrix} C_{TP} & -C_{FP} \\ -C_{FN} & C_{TN} \end{bmatrix}$, assigning specific costs to each outcome. The Hadamard product of these matrices is given by:

$$M \circ C = \begin{bmatrix} -TP \cdot C_{TP} & -FP \cdot C_{FP} \\ -FN \cdot C_{FN} & TN \cdot C_{TN} \end{bmatrix} \tag{4.1}$$

Summing all elements of $M \circ C$ gives the reward R_t:

$$R_t = \sum_i \sum_j (M \circ C)_{ij} = (TP \cdot C_{TP}) - (FP \cdot C_{FP}) - (FN \cdot C_{FN}) + (TN \cdot C_{TN}) \tag{4.2}$$

Defining X as the set of possible outcomes ($X = \{TP, FP, FN, TN\}$), the instantaneous reward R_t for an action taken by the agent at time t can be represented compactly as:

$$R_t = \sum_{x \in X} C_x \cdot \mathbb{I}_{\{x\}} \tag{4.3}$$

Moreover, to encourage the reinforcement learning agent to take into account the deviation from the actual default rates computed at the end of each month, a penalty term was defined. This penalty term was defined as the running average of the last N samples of the *mean absolute error* (MAE) for a given month. However, to minimize MAE we need to define the penalty term as $-$MAE. It can be mathematically represented as:

$$\text{Penalty}_{m,t} = -\text{MAE}_{m,t} = -\frac{1}{N} \sum_{i=t-N+1}^{t} |y_i - \hat{y}_i| \tag{4.4}$$

Here, y_i and $\hat{y}_i$ are the true and predicted values, respectively, for the i-th sample in month m. The sum runs from the $(t - N + 1)$-th to the t-th sample, capturing the average absolute error over the most recent N samples within that month.

Finally, the total reward given by the environment at each observation is defined as the composition of an immediate reward term R_t and error term MAE_t:

$$\text{Total Reward}_t = R_t + MAE_t \tag{4.5}$$

4.1.2 Observation Space

In reinforcement learning, effective state representation is crucial, particularly in the complex domain of financial data. Financial data exhibit high-dimensional, noisy, and non-stationary

characteristics, making state representation challenging yet vital for capturing the underlying dynamics. An appropriate state representation enables an RL agent to discern relevant patterns and trends from financial data, facilitating effective decision-making and strategy formulation (Deng et al., 2016).

Static state space representation

In this thesis, different feature variables representing transactional, client and product information were used to build the state (see figure 3.1). An observation is comprised of the pair set of features and corresponding target $(features, target)_{i,m}$ for the i-th client in m-th month.

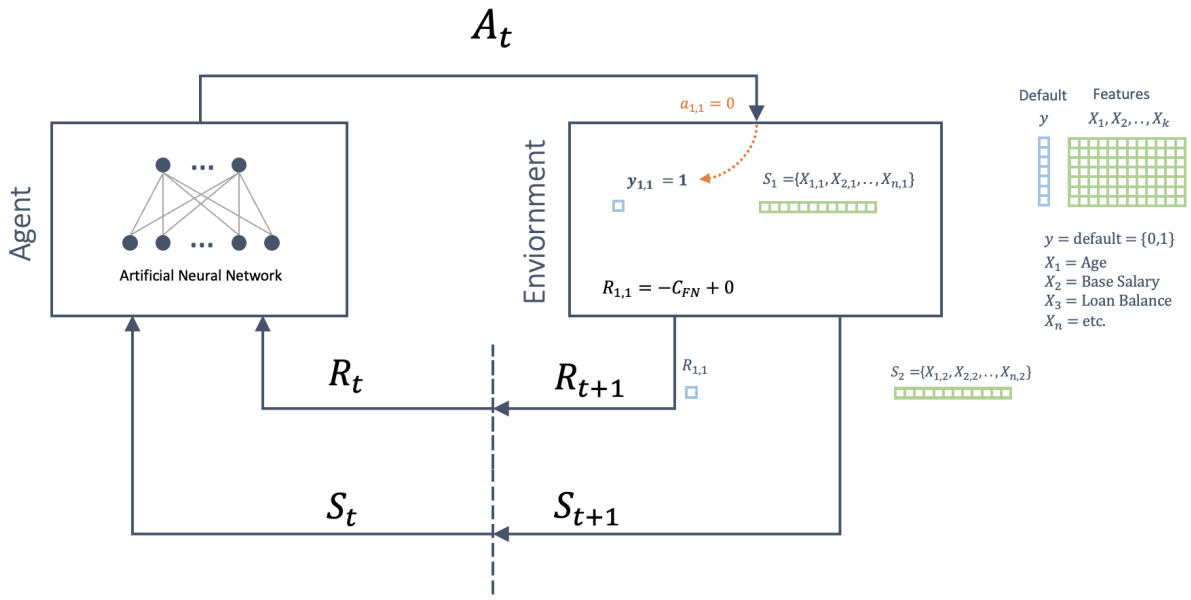


Figure 4.1: Illustration of an environment step with action $a_{1,1} = 0$. The environment calculates the total reward by combining the immediate reward based on the cost matrix ($R_t = -C_{FN}$) and the penalty for the Running Mean of the last N actions (MAE), which is 0 for the first action, resulting in a Total Reward of $-C_{FN}$.

Dynamic state space representation

Incorporating rewards and the history of past actions into the state representation of reinforcement learning environments presents substantial benefits. It endows agents with the ability to view the environment dynamics and the consequences of their actions. By integrating past rewards, agents could gain insight into the effectiveness of previous decisions, enabling them to adapt and optimize their strategies over time. This approach could be particularly advantageous in volatile financial environments, where the significance of historical actions and rewards plays a critical role in predicting future market trends and making informed decisions. Applications, of this type of representation in financial applications have been applied to trading and pricing of credit assets, highlighting how incorporating past rewards can lead to more informed and effective trading strategies (Nousi, Passalis and Tefas, 2023; Alfonso-Sánchez et al., 2024).

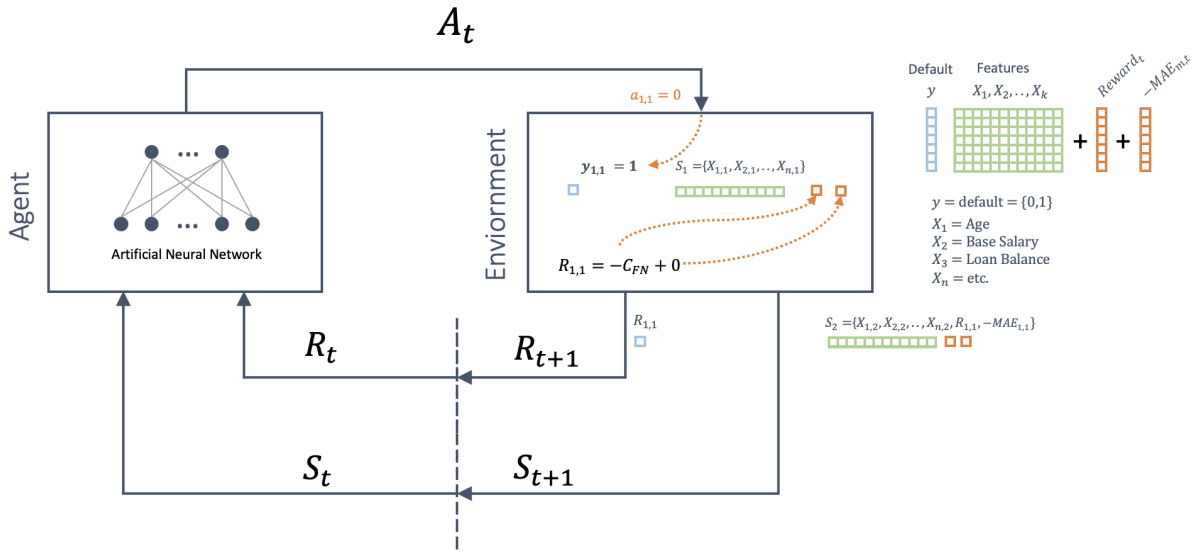


Figure 4.2: Illustration of a dynamic environment step with action $a_{1,1} = 0$. The environment calculates the total reward by combining the immediate reward based on the cost matrix ($R_t = -C_{FN}$) and the penalty for the Running Mean of the last N actions (MAE), which is 0 for the first action, resulting in a Total Reward of $-C_{FN}$. This values become part of this state $S_t = f(X, R_t, -MAE_t)$.

4.1.3 *RiskManagementEnv* Class

Gymnasium

`Gymnasium` stands out as a widely recognized library, presenting a standardized application programming interface (API) specifically designed for crafting environments in the realm of single-agent reinforcement learning. It comes equipped with a range of implementations for wrappers, catering to a variety of popular environments including CartPole, Pendulum, Mountain Car, Mujoco, and Atari, among others. In addition to these ready-to-use environments, Gymnasium also offers a standardized framework for developing custom environments. This dual capability renders Gymnasium an exceptionally versatile tool, adept at both leveraging established environments and devising new ones tailored to the unique requirements of specific reinforcement learning scenarios. A key benefit of Gymnasium is its compatibility with numerous benchmark agent implementations, making it an integral component in the reinforcement learning toolkit (Towers et al., 2023).

The general approach to code a custom environment involves extending the `Gymnasium.Env` base class as follows:

```
import gymnasium as gym
from gymnasium import spaces

class RiskManagementEnv(gym.Env):
    """
    Custom Environment that follows gym interface.
    """
    metadata = {'render.modes': ['human']}
```

```
    def __init__(self):
        super(CustomEnvironment, self).__init__()
        # Define action and observation space
        self.action_space = spaces.Discrete(..)
        self.observation_space = spaces.Box(low=..., high=...)

    def step(self, action):
        # Execute one time step within the environment
        ...
        return observation, reward, done, info

    def reset(self):
        # Reset the state of the environment to an initial state
        ...
        return observation  # reward, done, info can't be included

    def render(self, mode='human', close=False):
        # Render the environment to the screen
        ...

    def close(self):
        # Close the environment
        ...
```

the required methods are:

- The `__init__` method initializes the environment, including the action space and observation space.
- The `step` method is used to update the environment's state in response to an action.
- The `reset` method resets the environment to its initial state.
- The `render` method is for rendering the environment's state to the screen.
- The `close` method performs any necessary cleanup.

However, the most important ones are the `step` where most of the logic of the environment is programmed and the `reset` method where the logic to reset environment variables happen before advancing to the next episodes. Five different variations testing different environment definitions were developed:

- `RiskManagementEnv`: Environment with static state representation where the an episode is 15 month periods (i.e., the environment gets reset after 15 months).
- `RiskManagementEnvMonthlyEpisodes`: Environment with static state representation where the an episode is defined as a single month (i.e., the environment gets reset after every month).
- `RiskManagementEnvMultiTarget`: Environment with static state representation where the an episode is defined as a single month (i.e., the environment gets reset after every month).

- `RiskManagementEnvDynaState`: Environment with dynamic state representation that adds running mean of the instantaneous reward and running mean of the penalty term into the state vector given by $S_t = f(X, R_t, -MAE_t)$.
-

The full implementation of this environments are provided in the appendix code section.

4.2 Agent Formulation

4.2.1 Stable Baselines 3

Stable-Baselines3 (SB3) is a reinforcement learning (RL) framework for designing agents, offering a comprehensive suite of benchmarked RL agent implementations. It a provides user-friendly interface, optimized code, improved performance, and a focus on PyTorch for deep learning models (Raffin et al., 2019). Moreover, there are other contributing initiatives that provide more complex agents implementations such as `SB3 contrib` from the same authors.

Central to SB3's appeal is its integration with the Gymnasium library through the provision of vectorized environments, known as `VecEnv`. This wrapper transforms the training landscape for RL agents by enabling simultaneous training across multiple, independent environments. Traditional RL training processes involve a single environment per step. However, `VecEnv` empowers agents to interact with 'N' number of environments within the same timeframe. This multi-environment approach results in actions, observations, rewards, and episode termination signals (dones) being processed as vectors, each corresponding to the dimensions of the environments being handled. This vectorization significantly accelerates the training process, allowing for more efficient utilization of computational resources and faster convergence of models.

Deciding whether to implement custom agents from scratch was a critical decision for the project, considering the constraints of time and hardware resources. While creating custom agents from scratch allows for more detailed modifications, it also carries the risk of introducing errors and potentially error prone implementations. Utilizing SB3 the library, provided access to well-established algorithmic code. This choice ensured a focus on testing without concerns over optimal data structure selection, bespoke design patterns, or compatibility issues with various dependencies.

There are several reinforcement learning algorithms implemented in Stable Baselines 3:

- A2C (Advantage Actor-Critic) - A synchronous, deterministic variant of Asynchronous Advantage Actor Critic (A3C), known for its efficiency and simplicity in implementation.
- DDPG (Deep Deterministic Policy Gradient) - An algorithm which combines Q-learning with a policy gradient method, using deep neural networks to learn policies in high-dimensional, continuous action spaces.
- DQN (Deep Q-Networks) - An algorithm that introduced deep learning to reinforcement learning, making it possible to achieve human-level performance in complex environments like Atari games.

- HER (Hindsight Experience Replay) - An approach to handle sparse rewards in robotic environments by replaying past experiences with different goals.
- PPO (Proximal Policy Optimization) - An algorithm that improves the stability and robustness of policy gradient methods through a novel objective function.
- SAC (Soft Actor-Critic) - A state-of-the-art algorithm that aims for maximum entropy reinforcement learning, optimal for continuous action spaces.
- TD3 (Twin Delayed Deep Deterministic Policy Gradients) - An algorithm that addresses the overestimation of Q-values in DDPG by introducing twin Q-networks and delayed policy updates.

and its contribution extension:

- ARS (Augmented Random Search) - A direct policy search method that is simple yet effective, particularly in environments with continuous action spaces.
- Maskable PPO - An extension of PPO that allows for masking certain actions in the action space, useful in environments with variable action spaces.
- Recurrent PPO - A variant of PPO that incorporates recurrent neural networks, suitable for environments requiring memory or handling partial observability.
- QR-DQN (Quantile Regression DQN) - An extension of DQN that uses quantile regression to approximate the full distribution of the return instead of just the mean.
- TQC (Truncated Quantile Critics) - A novel algorithm that improves upon SAC by truncating the critic's value distribution, leading to more stable learning.
- TRPO (Trust Region Policy Optimization) - An algorithm that makes large updates to policies while ensuring the divergence between new and old policies remains within a trust region, promoting stable improvement.

However, SB3 Implementation of the DQN (i.e., Vanilla DQN) Agent is based only in the original DQN without any modern improvements and advanced experienced replay buffers. Vanilla DQN, while groundbreaking in reinforcement learning, faces several challenges. Key issues include the tendency to overestimate Q-values, sample inefficiency, and stability problems due to correlated samples and non-stationary targets. Subsequent enhancements like Double DQN, Dueling DQN, and Prioritized Experience Replay have been developed to address these limitations, significantly advancing DQN architectures (Mnih et al., 2015; Schaul et al., 2016).

Given this limitations, this thesis implements Double DQN and Dueling DQN improvements extending SB3's Vanilla DQN.

4.2.2 Dueling DQN

Dueling DQN, introduced in the paper "Dueling Network Architectures for Deep Reinforcement Learning" by Wang et al. (2016)., is an enhancement in the field of Deep Q-Networks (DQN). This architecture differentiates itself by implementing a novel approach to the Q-function. In a conventional DQN, the Q-function estimates the value of taking an action in a given state. However, Dueling DQN refines this concept by decomposing the Q-function into two distinct streams: one for estimating the state value function $V(s)$ and another for assessing the advantage function $A(s, a)$.

The main innovation lies in the equation for the advantage term, which is:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a') \right)$$

Here, $Q(s,a)$ is the action-value function, $V(s)$ is the value function, and $A(s,a)$ is the advantage function. The term $\frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a')$ represents the average advantage of all actions in state s. This decomposition allows the network to learn which states are valuable without having to learn the effect of each action for each state, thereby simplifying the learning process and improving the efficiency of the algorithm.

The paper by Wang et al. has been a cornerstone in advancing deep reinforcement learning, providing a framework that enables more effective training of RL agents, especially in environments with a high-dimensional action space. By explicitly modeling the state value and the advantages of actions, Dueling DQN offers a more robust approach to approximating the optimal policy compared to traditional DQN architectures.

4.2.3 Double DQN

Double DQN, introduced in the paper "Deep Reinforcement Learning with Double Q-learning" by Van Hasselt, Guez and Silver (2016), addresses a critical issue in the original DQN algorithm—overestimation of Q-values. Double DQN modifies the Q-learning update by decoupling the selection and evaluation of the action in the max operation. The main equation in Double DQN is given by:

$$Q(s,a) = R(s,a) + \gamma Q(s', \underset{a'}{\operatorname{argmax}}\, Q(s', a'; \theta); \theta')$$

Here, $Q(s,a)$ represents the action-value function, $R(s,a)$ is the reward, γ is the discount factor, s' is the next state, θ are the weights of the current Q-network, and θ' are the weights of the target Q-network. This approach mitigates the overestimation bias, leading to more stable and reliable learning.

4.2.4 Policy Networks

Custom feature extractors in Stable Baselines 3 offer a tool for tailoring reinforcement learning models to specific environments. By leveraging customized neural network architectures, these feature extractors allow for more expressive processing of environmental observations. This capability is particularly valuable in complex scenarios where default feature extractors may not capture all necessary aspects of the environment.Custom feature extractors enable a more flexible approach to learning from environmental data.

Various types of policy networks where developed following the stable baselines 3 library by extending the `BaseFeaturesExtractor` base class as follows:

```
from gymnasium.spaces import Space
from stable_baselines3.common.torch_layers import BaseFeaturesExtractor
```

```
class CustomPolicyNetwork(BaseFeaturesExtractor):

    def __init__(self, observation_space, **params):
        """
        Parameter initialization
        """
        super(CustomPolicyNetwork, self).__init__( **params)

        # example of a fully connected network
        self.net = nn.Sequential(
            nn.Linear(observation_space.shape[0], 100),
            nn.BatchNorm1d(100),
            nn.RReLU(lower=0.1, upper=0.5),
            nn.Dropout(p=0.5),
            ...
        )

    def forward(self, observations: Tensor) -> Tensor:
        """
        Forward pass definition
        """
        return self.net(observations)
```

Where,

- `GRUNetwork`:
  - Utilizes a Gated Recurrent Unit (GRU) for processing sequential data.
  - Suitable for environments where temporal dependencies are crucial.
  - Includes pre-processing and post-processing layers for additional feature extraction and transformation.
- `GRUNetworkBidirectional`:
  - Features a bidirectional GRU, capturing dependencies from both past and future contexts.
  - Enhances the agent's ability to understand sequential patterns in both directions.
  - Particularly useful in environments where future context is as important as past context.
- `LSTMNetwork`:
  - Implements a Long Short-Term Memory (LSTM) network.
  - Ideal for environments with long-term dependencies.
  - LSTM helps in remembering information over extended time periods.
- `FCNetwork`:
  - A Fully Connected (FC) network structure.

  - Suitable for environments where inputs can be considered independently of each other.
  - Simpler structure, focusing on individual feature importance.

- `ConvNetwork`:
  - Incorporates convolutional layers, ideal for spatially structured data.
  - Useful in image-based or grid-based environments.
  - Can capture localized patterns within the observation space.

4.2.5 Performance Measure

The primary aim of this thesis is to explore the efficacy of model-free reinforcement learning algorithms in discriminating between "good (0)" and "bad (1)" loan classifications. Key performance indicators for evaluating classification prowess include F1-score, precision, recall, and accuracy. Furthermore, given that the collective outcome of binary predictions monthly inherently constitutes the default rate as defined in 3.1, the critical metrics for appraising the agent's proficiency are the R-squared (R2) value and the Mean Absolute Error (MAE), which quantify the deviations of the predicted default rate from the actual observed rate.

Chapter 5

Results

This thesis evaluates six agents across two key policy optimization categories: value-based methods (DQN, Double DQN, Dueling DQN) and policy gradient methods (A2C, PPO, TRPO), using Stable Baselines 3. The Double DQN and Dueling DQN are custom extensions of the library's standard DQN.

Due to early developmental constraints in time and computational resources, the scope of deep learning training was limited, especially considering the extensive hyper-parameterization inherent in Reinforcement Learning (RL) algorithms. The complexity of testing various permutations of environments, policy networks, and vectorized environments necessitated a focused approach. This section thus presents the most effective combination of all different components in RL modeling, with supplementary results included in the appendix.

Additionally, the intricate nature of RL algorithms guided the initial use of heuristics for key aspects like environment and policy network design. A promising heuristic for reward structure was selected based on its effectiveness early on to be used for the rest of all experiments. `GRUNetwork`, with its fewer parameters compared to LSTM, was chosen for the policy network, and `RiskManagementEnvMonthlyEpisodes` was utilized as the primary environment to show the main results.

5.1 Effectiveness of Reward Structures and State Representations

This section delves into identifying the most effective reward structures and state representations for the classification task. Since these elements are pivotal in guiding the learning and decision-making processes of the agents, the discussion focuses on their role in enhancing or impeding the agent's performance and the overall classification accuracy.

5.1.1 Reward Structure

5.1.2 Immediate Reward R_t

Asymmetric reward structures in reinforcement learning can significantly influence agent performance, particularly in scenarios with imbalanced datasets. By calibrating rewards to emphasize certain outcomes, agents can be steered towards more nuanced behaviors that

may be underrepresented in the training data. This strategy is crucial in contexts where the cost of false negatives substantially outweighs that of false positives, or vice versa. For example, in credit risk assessment, failing to identify a bad loan could be more detrimental than incorrectly classifying a good loan as risky. Thus, asymmetric rewards facilitate a focus on critical decision-making aspects, enhancing the model's predictive accuracy and generalization (Louie, 2022; Gershman, 2015).

As explained in subsection 4.1.1, The immediate the cost matrix was defined as

$$C = \begin{bmatrix} C_{TP} & -C_{FP} \\ -C_{FN} & C_{TN} \end{bmatrix} = \begin{bmatrix} 2 & -1 \\ -2 & 1 \end{bmatrix} \tag{5.1}$$

to be asymmetric, giving more reward weight to the correct classification of "bad" loans, or True Positives (TP), and more penalty to misclassified "good" events, or False Negatives (FN).

During testing, a static cost matrix C, as per Equation 5.1, proved ineffective for monthly default rate prediction and learning the data distribution. This necessitated a dynamic strategy for the cost matrix to adapt monthly, enhancing the reward structure in response to the variable nature of credit data. This thesis employs a heuristic that modifies the cost matrix based on the monthly distribution of good and bad clients, thereby tailoring the agent's rewards and penalties.

$$C = \begin{bmatrix} C_{TP} + w_1 & -C_{FP} - w_0 \\ -C_{FN} - w_0 & C_{TN} + w_1 \end{bmatrix} = \begin{bmatrix} 2 + w_1 & -1 - w_0 \\ -2 - w_0 & 1 + w_1 \end{bmatrix} \tag{5.2}$$

Where, the class weight w_i in the target label y (default)for each class i is computed as:

$$w_i = \frac{N}{n \times \text{count}(y_i)} \tag{5.3}$$

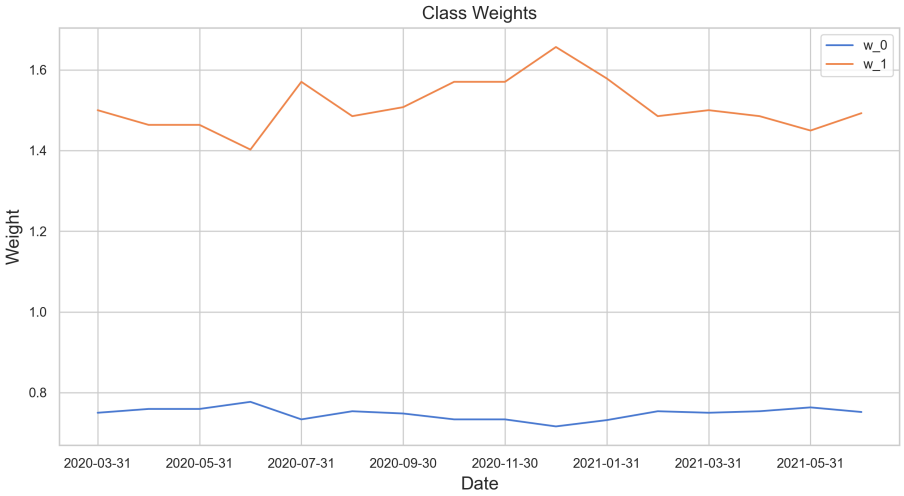
where:

- N is the total number of samples.
- n is the number of classes. $n = |\{0, 1\}| = 2$.
- $\text{count}(y_i)$ is the number of samples of the class i in y.

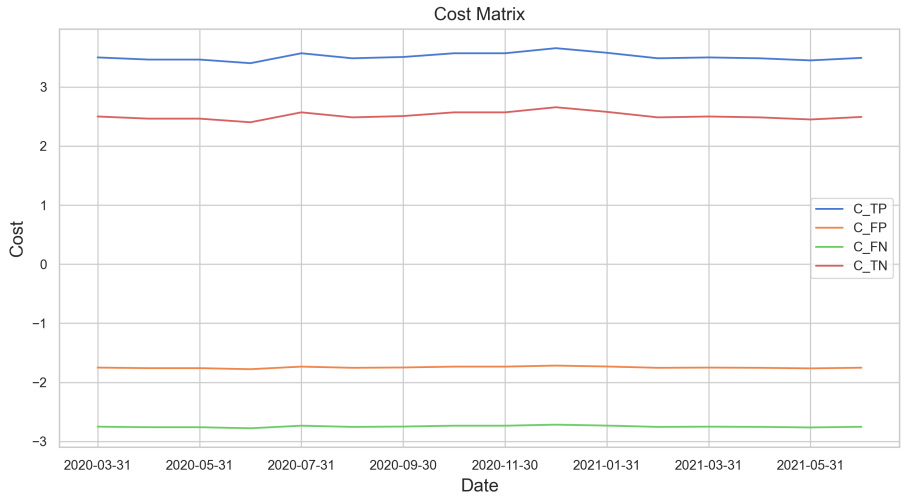
The vector of class weights is then given by:

$$\mathbf{w} = [w_0 w_1, \dots, w_n] = [w_0, w_1] \tag{5.4}$$

The weight schedule shows a more dynamic behavior across the dataset timeline leading to a more pronounced asymmetry in the cost matrix as shown in figure 5.1 below.



(a) Class weights schedule used in training.



(b) Cost matrix schedule used in training.

Figure 5.1: Schedules used in training for (a) class weights and (b) cost matrix.

All results here shown are used with this heuristic reward scheme as it showed higher performance after testing static reward representation with different asymmetric costs.

5.1.3 Penalty Term

The original formulation using running mean absolute error (MAE) was observed to have little impact in the reward due to the scale mismatch between the instantaneous reward and penalty. Moreover, making use of other performance metrics besides MAE was shown to be helpful to aid the agent in learning the to classify the events so f1-score and recall as defined below

$$\text{F1-score} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{5.5}$$

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}} \tag{5.6}$$

Where precision is defined as the number of true positive results divided by the number of all positive results, including those not identified correctly, and recall (also known as sensitivity) is the number of true positive results divided by the number of positives that should have been identified. The F1-score is the harmonic mean of precision and recall, providing a balance between the two metrics.

5.1.4 Reward Scaling

In initial experiments, a notable observation was the occurrence of large penalty term errors in the early stages. These errors, when compounded, led to significant errors, impeding the agent's ability to learn meaningful patterns and achieve good performance. To address this challenge, reward scaling of the penalty was implemented as a critical component of the total reward signal. This technique adjusted the magnitude of penalties to be on par with the immediate rewards, ensuring that the learning process was not overwhelmed by excessive negative feedback in the face of initial errors.

The literature supports the effectiveness of reward scaling in enhancing RL algorithms. By carefully calibrating the reward scale, algorithms can strike a balance between the need for exploration and the necessity of exploitation, a duality crucial in complex and uncertain environments (Cabi et al., 2020; Schaul et al., 2021). Optimal reward scaling aids in achieving a stable and efficient learning trajectory, avoiding the pitfalls of overly dominant or insignificant reward signals (Ostrovski et al., 2023). Furthermore, in contexts where rewards are sparse or subtle, such as in financial applications, reward scaling is instrumental in sensitizing the agent to these nuanced signals, aligning the learning process more closely with the desired objectives (Leike et al., 2018; Wu et al., 2023).

The penalty term was scaled between the minimum and maximum values of the immediate reward signal given by the formula:

$$Penalty_{\text{scaled}} = \frac{Penalty - R_{\text{min}}}{R_{\text{max}} - R_{\text{min}}} \tag{5.7}$$

where R_{min} is the minimum value of the immediate reward at that timestep, and R_{max} is the maximum value.

5.1.5 Total Reward

The total reward signal for the environment with the best result from initial testing was given by the heuristic formula:

$$\text{Total Reward}_t = R_t + Penalty_{\text{scaled},t} \tag{5.8}$$

5.2 Evaluating Model-Free DRL Agents for Client Classification

This segment evaluates the capability of model-free Deep Reinforcement Learning (DRL) agents in classifying clients effectively. The analysis explores how well these agents discern between good and bad clients, providing insights into the viability of DRL in financial decision-making contexts.

5.2.1 Agent Performances

The core of the experiment was to design an environment and rank different performances across agents. `RiskManagementEnvMonthlyEpisodes` was a chosen to be the main testing environment due to initial heuristics showing promising results (see section 4.1.2).

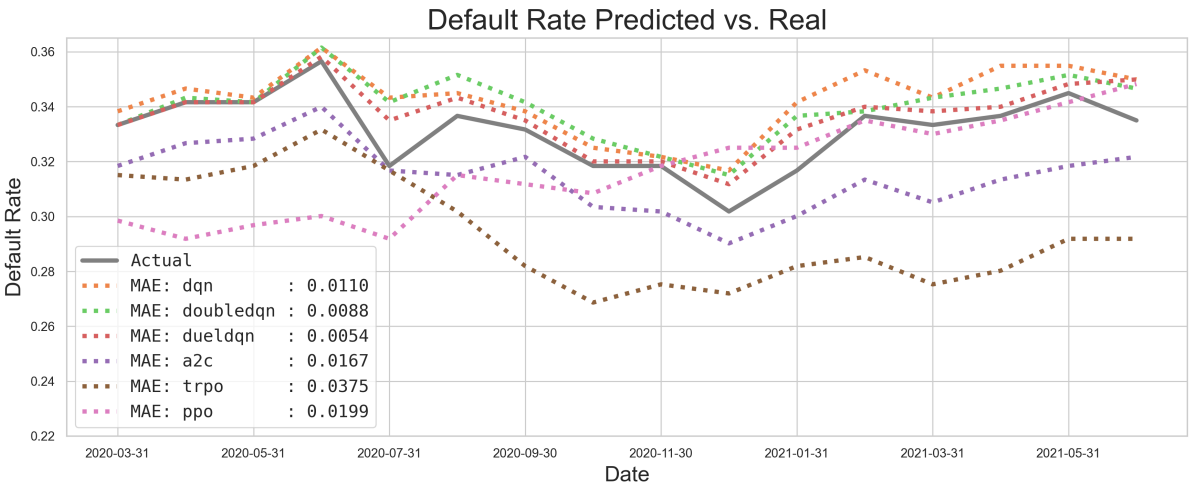


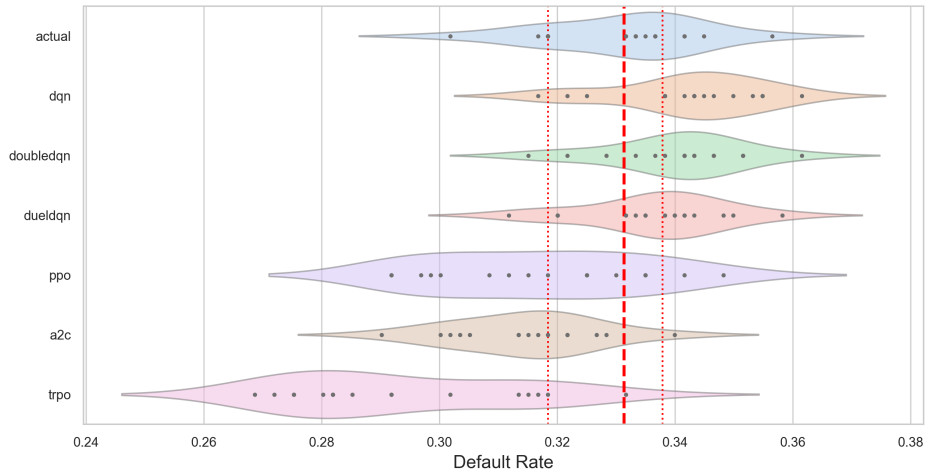
Figure 5.2: Predicted vs actual default rate.

The results in figure 5.2 show the results for all agents of both families of DRL algorithms (i.e., value based and policy gradient). From the results, we can observe that value-based policy optimization performed in general much better than gradient-policy methods.

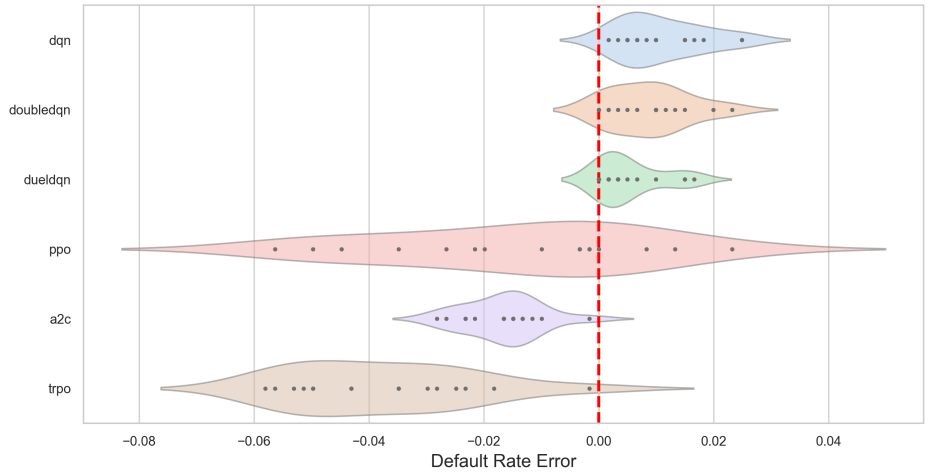
Table 5.1: Model Performance Metrics on`RiskManagementEnvMonthlyEpisodes`

	Model	R2	MAE
1	dueldqn	0.654698	0.005390
2	doubledqn	0.284155	0.008810
3	dqn	-0.003597	0.010987
4	a2c	-0.880987	0.016687
5	ppo	-3.165836	0.019900
6	trpo	-8.648262	0.037520

In the observed results from table 5.1, value-based policy optimization methods (DQN, Double DQN, Dueling DQN) demonstrated superior performance compared to policy gradient methods (A2C, PPO, TRPO) in a tabular RL environment characterized by loan information features. To explore potential reasons behind this finding, insights from general reinforcement learning literature can provide useful hypotheses.



(a) Default rate distributions.



(b) Default rate error distributions.

Figure 5.3: Comparative analysis of default rates (a) and their errors (b)

Observing the results, it's evident that value-based policy optimization algorithms yield distributions more aligned with the actual distribution. A comparison of errors, defined as error $= y_{\text{true}} - y_{\text{pred}}$, in predicting default rates shows that value-based methods exhibit significantly lower variance in errors compared to policy gradient methods (see figure 5.3).

Value-based methods, such as DQN and its variants, Dueling DQN and Double DQN, are known for their efficiency in discrete action spaces and environments with clear and immediate reward signals (Mnih et al., 2015; Van Hasselt, Guez and Silver, 2016). They excel in learning optimal policies by focusing on estimating value functions, which can be advantageous in

settings with more predictable state transitions. This can be particularly relevant in tabular RL environments, where the state space is defined by distinct features such as loan information.

5.2.2 Policies

On the other hand, policy gradient methods, including A2C, PPO, and TRPO, are often favored in environments with continuous action spaces and where the reward structure is less explicit or the state transitions are more variable (Schulman et al., 2017b; Wu et al., 2017). This methods directly optimize the policy and are typically more suitable for high-dimensional or complex state representations. However, their application in environments with more stable and predictable dynamics, such as those present in the small feature size dataset used, may not leverage their strengths to the fullest extent.

In figure 5.4 and 5.5, value-based methods exhibit a higher magnitude of policy loss compared to policy gradient methods in this type of environment, which interestingly correlates with improved performance. This phenomenon could be attributed to several factors in the experimental setup. Firstly, the dataset features were not scaled, which, combined with the inherent batch normalization in the policy network architecture, might have influenced the results. Additionally, further hyper-parameter tuning and extended training duration could potentially alter these outcomes.

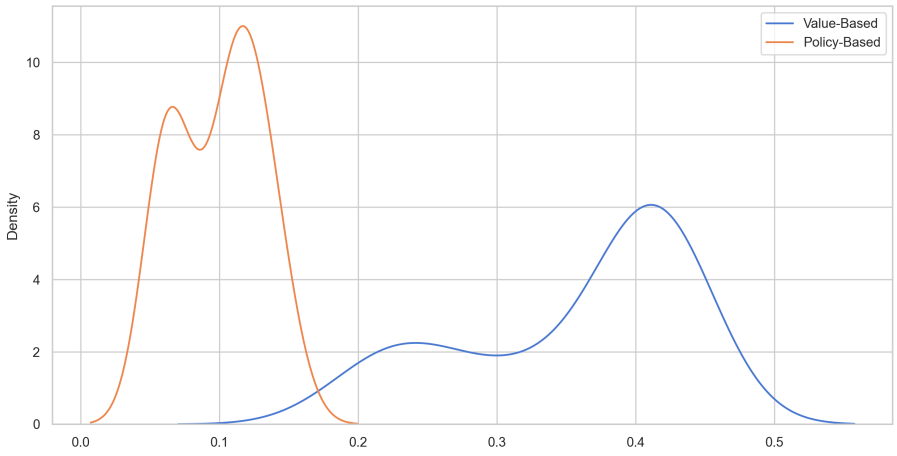
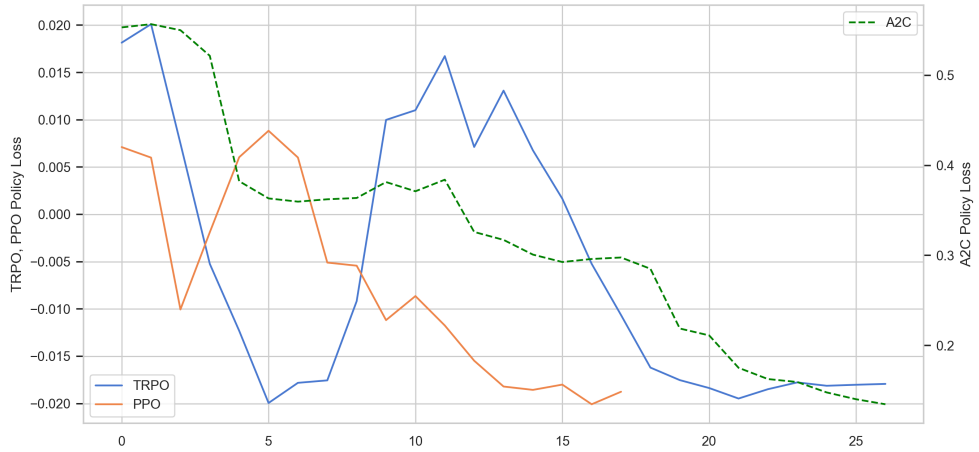


Figure 5.4: Distribution of policy losses between value-based and policy-gradient algorithms where policy-gradient algorithm show higher variance.

(a) Training loss for value-based family of algorithms.

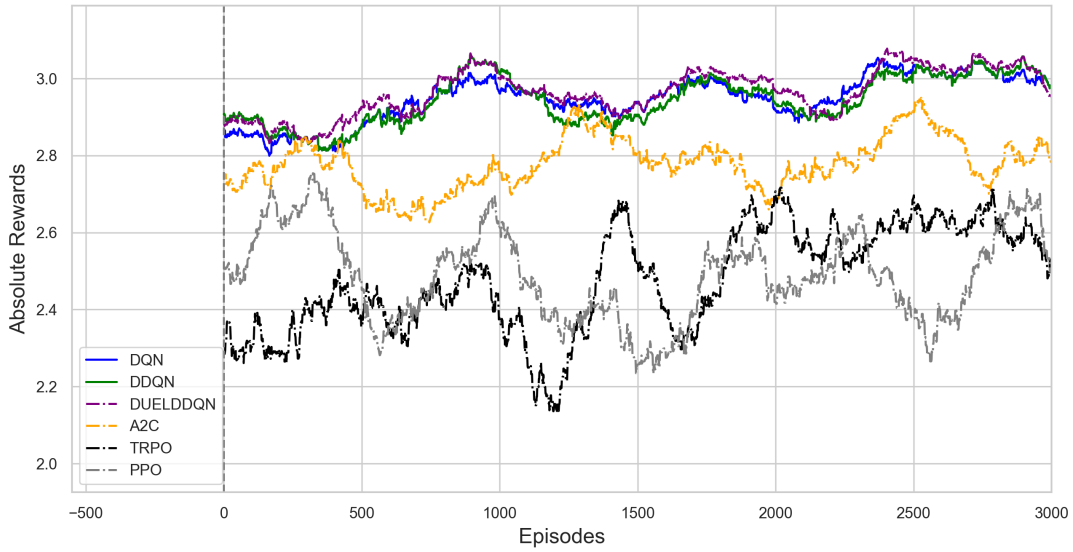


(b) Policy loss for policy gradient family of algorithms (note the scale of secondary axis.)

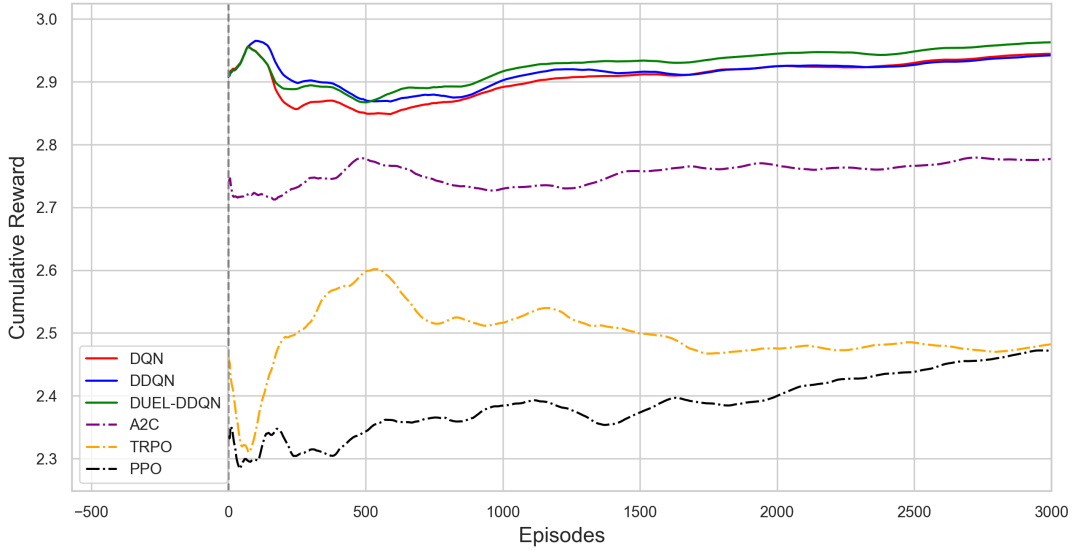
Figure 5.5: Comparative analysis of training and policy losses in RL Algorithm Families

5.2.3 Rewards

The performance of RL algorithms is significantly influenced by factors such as the complexity and dimensionality of state representation, as well as the clarity of reward signals. In environments with well-defined state transitions and reward signals, value-based methods often have a performance advantage. This is corroborated by experimental results showing consistently higher rewards for Dueling DQN, Double DQN, and DQN, compared to A2C, TRPO, and PPO, as illustrated in figure 5.6.



(a) Smoothed episode rewards.



(b) Smoothed Cumulative episode rewards.

Figure 5.6: Comparative Analysis of Default Rates (a) and their Errors (b)

5.3 Generalization Capabilities of Model-Free DRL Agents

Generalization to out-of-time, unseen populations is essential for the practical application of DRL models. This part of the analysis examines the agents' ability to adapt to different demographic segments and varying economic conditions, assessing their robustness and applicability in diverse financial scenarios.

5.3.1 Agent Performances

The unexpected shift in performance on the test set, with PPO outperforming the value-based algorithms, contrasts with the training results (see figure 5.7). This discrepancy warrants a

further examination of the training procedures and algorithmic interactions with the dataset.

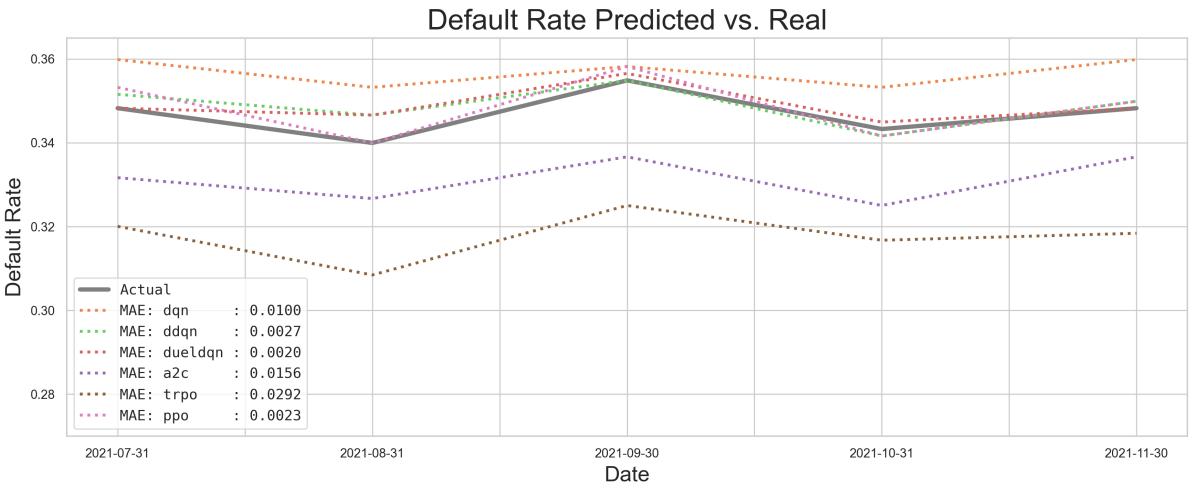


Figure 5.7: Predicted vs actual default rates on testing set.

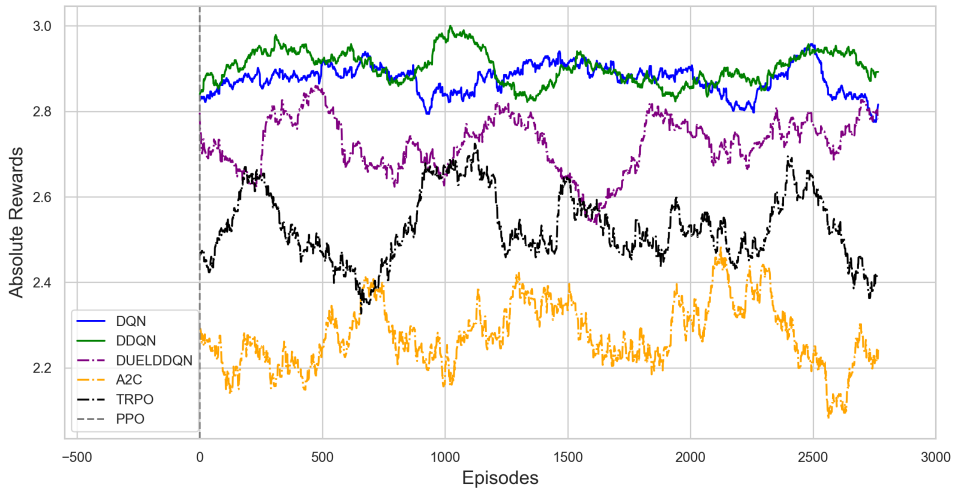
Table 5.2: Model Performance Metrics

Model	R2	MAE	RMSE
PPO	0.679487	0.002322	0.002872
DuelDQN	0.615385	0.001990	0.003147
DDQN	0.529915	0.002653	0.003479
DQN	-3.316239	0.009950	0.010541
A2C	-8.722222	0.015589	0.015820
TRPO	-32.205128	0.029187	0.029236

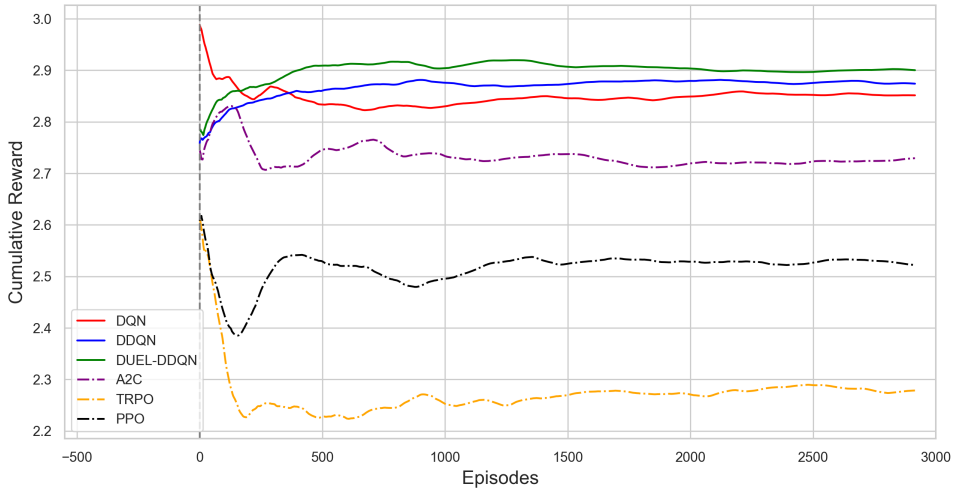
PPO's success in the test phase, as noted by Schulman et al. (2017b), can be ascribed to its conservative policy update strategy, marked by a clipping mechanism that ensures modest deviations from the previous policy. This approach may have granted PPO a more robust generalization capability, an essential trait for dealing with unforeseen data distributions in the test set. Conversely, the DQN variants, known for their aggressive policy updates Mnih et al. (2015), might have overfitted to the training data, impairing their performance in new scenarios.

The divergence in performance also points to potential limitations in the testing set. It suggests that the DQN family's proficiency in maximizing immediate rewards may not have translated well to the unexplored patterns of the test set, indicating a possible gap in the training data's representativeness. This aligns with Goodfellow, Bengio and Courville (2016)'s assertion that comprehensive and diverse training is crucial for effective model generalization, especially in dynamic domains like financial markets.

These results prompt a reevaluation of the training methodology and the need for a more nuanced approach to data selection and algorithm training. It highlights the significance of balancing exploration and exploitation in RL algorithms and calls for a careful assessment of data distributions during training to develop robust models capable of adapting to new conditions.



(a) Smoothed episode rewards.



(b) Smoothed Cumulative episode rewards.

Figure 5.8: Comparative Analysis of Default Rates (a) and their Errors (b)

In summary, while PPO's performance on the test set is noteworthy, it should be interpreted with caution. The findings suggest a need for a more thorough investigation into the training process and a reconsideration of the algorithms' capabilities in relation to the dataset used.

Chapter 6

Conclusions and Future Work

6.1 Project Achievements and Constraints

This thesis explored the applicability of model-free Deep Reinforcement Learning (DRL) in credit scoring. While the project achieved its primary goal of demonstrating DRL's potential in classifying mortgage clients, the results should be interpreted with caution. The study faced computational limitations, which restricted the scope of hyperparameter tuning and exploration of more sophisticated DRL models. These constraints suggest the need for further investigation to fully ascertain the capabilities of DRL in this context.

6.2 Assessment and Implications

The study's findings, though preliminary, indicate a promising direction for using DRL in financial risk assessment. The performance discrepancy among DRL algorithms, particularly the effectiveness of value-based methods in structured environments, provides a starting point for understanding the suitability of different DRL approaches in finance. However, the results should be viewed as initial insights rather than conclusive evidence, given the limitations in computational resources and model exploration.

6.3 Reflection on Methodology

The choice of dynamic reward structure and state representation played a significant role in the observed performances. However, the limited scope of testing and the initial heuristic approach suggest that more comprehensive methodological strategies might yield different insights. In retrospect, investing in more powerful computational resources and extending training duration could potentially enhance the robustness and generalizability of the findings.

6.4 Directions for Future Research

The project highlights several areas for future exploration: 1. **Advanced DRL Architectures**: Investigating more complex DRL models could uncover deeper insights into their applicability and effectiveness in financial risk assessment. 2. **Extensive Hyperparameter Optimization**: More thorough hyperparameter tuning might lead to improved model performance and better

understanding of the agents' learning dynamics. 3. **Wider Application Scope**: Testing the models across different financial datasets and environments would provide a more comprehensive view of their adaptability and robustness.

6.5 Concluding Remarks

The dissertation serves as an exploratory step into the application of DRL in credit risk management. The findings, while encouraging, should be approached with a degree of skepticism, considering the constraints and the need for more extensive testing and validation. The potential of DRL in this domain is evident, but it necessitates further research, better computational resources, and more extensive training to fully harness its capabilities in financial risk assessment.

Bibliography

Adam, S.P., Alexandropoulos, S.A.N., Pardalos, P.M. and Vrahatis, M.N., 2019. No Free Lunch Theorem: A Review [Online]. In: I.C. Demetriou and P.M. Pardalos, eds. *Approximation and Optimization : Algorithms, Complexity and Applications*. Cham: Springer International Publishing, Springer Optimization and Its Applications, pp.57–82. Available from: https://doi.org/10.1007/978-3-030-12767-1_5 [Accessed 2024-01-12].

Alfonso-Sánchez, S., Solano, J., Correa-Bahnsen, A., Sendova, K.P. and Bravo, C., 2024. Optimizing credit limit adjustments under adversarial goals using reinforcement learning. *European journal of operational research* [Online]. Available from: https://doi.org/10.1016/j.ejor.2023.12.025 [Accessed 2024-01-08].

Baesens, B., Van Gestel, T., Viaene, S., Stepanova, M., Suykens, J. and Vanthienen, J., 2003. Benchmarking state-of-the-art classification algorithms for credit scoring. *Journal of the operational research society* [Online], 54(6), pp.627–635. Available from: https://doi.org/10.1057/palgrave.jors.2601545 [Accessed 2024-01-10].

Begenau, J., 2020. Capital requirements, risk choice, and liquidity provision in a business-cycle model. *Journal of financial economics* [Online], 136(2), pp.355–378. Available from: https://doi.org/10.1016/j.jfineco.2019.10.004 [Accessed 2024-01-12].

Bellman, R. and Bellman, R.E., 1957. *Dynamic Programming*. Princeton University Press. Google-Books-ID: rZW4ugAACAAJ.

Bertsekas, D.P., 2007. *Dynamic programming and optimal control, vol. ii*. 3rd ed. Athena Scientific.

Bertsekas, D.P., 2012. *Dynamic programming and optimal control (3rd edition)*. Athena Scientific.

Bertsekas, D.P. and Tsitsiklis, J.N., 1996. *Neuro-dynamic programming*. Athena Scientific.

Bouchti, A.E., Chakroun, A., Abbar, H. and Okar, C., 2017. Fraud detection in banking using deep reinforcement learning [Online]. *2017 Seventh International Conference on Innovative Computing Technology (INTECH)*. Luton: IEEE, pp.58–63. Available from: https://doi.org/10.1109/INTECH.2017.8102446 [Accessed 2024-01-08].

Brownlee, J., 2018. *Deep learning for time series forecasting: Predict the future with mlps, cnns and lstms in python*. Machine Learning Mastery.

Byun, S., Game, A. et al., 2021. The pandemic's impact on credit risk: Averted or delayed? *Federal reserve*. https://www.federalreserve.gov.

Cabi, S. et al., 2020. Scaling data-driven robotics with reward sketching and batch reinforcement learning. *arxiv: Robotics* [Online]. Available from: `https://arxiv.org/abs/1909.12200`.

Catania, C., Guerra, J.J., Romero, J.M., Caffaratti, G.D. and Marchetta, M.G., 2022. Beyond random split for assessing statistical model performance. *arxiv preprint arxiv:2209.03346*.

Cortes, C. and Vapnik, V., 1995. Support-vector networks. *Machine learning* [Online], 20(3), pp.273–297. Available from: `https://doi.org/10.1007/BF00994018` [Accessed 2024-01-11].

Costa, A.H.R., 2023. Poe: A general portfolio optimization environment for finrl [Online]. [Online]. Available from: `https://doi.org/10.5753/bwaif.2023.231144`.

Dastile, X., Celik, T. and Potsane, M., 2020. Statistical and machine learning models in credit scoring: A systematic literature survey. *Applied soft computing* [Online], 91, p.106263. Available from: `https://doi.org/10.1016/j.asoc.2020.106263` [Accessed 2024-01-10].

Deng, Y., Bao, F., Kong, Y., Ren, Z. and Dai, Q., 2016. Deep direct reinforcement learning for financial signal representation and trading. *Ieee transactions on neural networks and learning systems*, 28(3), pp.653–664.

Devi, S.S. and Radhika, Y., 2018. A Survey on Machine Learning and Statistical Techniques in Bankruptcy Prediction. *International journal of machine learning and computing* [Online], 8(2), pp.133–139. Available from: `https://doi.org/10.18178/ijmlc.2018.8.2.676` [Accessed 2024-01-10].

Dixon, M., Halperin, I. and Bilokon, P., 2018. Applications of reinforcement learning [Online]. [Online]. Available from: `https://doi.org/10.1007/978-3-030-41068-1_10`.

Fisher, R.A., 1936. The Use of Multiple Measurements in Taxonomic Problems. *Annals of eugenics* [Online], 7(2), pp.179–188. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1469-1809.1936.tb02137.x. Available from: `https://doi.org/10.1111/j.1469-1809.1936.tb02137.x` [Accessed 2024-01-11].

Gershman, S.J., 2015. Do learning rates adapt to the distribution of rewards. *Psychonomic bulletin & review* [Online], 22(1), pp.132–137. Available from: `https://doi.org/10.3758/s13423-014-0790-3`.

Goodfellow, I., Bengio, Y. and Courville, A., 2016. *Deep learning* [Online]. MIT Press. Available from: `http://www.deeplearningbook.org`.

Guan, M. and Liu, X.Y., 2021. Explainable Deep Reinforcement Learning for Portfolio Management: An Empirical Approach [Online]. ArXiv:2111.03995 [cs, q-fin]. Available from: `http://arxiv.org/abs/2111.03995` [Accessed 2024-01-08].

H. Greene, W., 2008. A statistical model for credit scoring [Online]. In: S. Jones and D.A. Hensher, eds. *Advances in Credit Risk Modelling and Corporate Bankruptcy Prediction*. Cambridge University Press, pp.14–43. 1st ed. Available from: `https://doi.org/10.1017/CBO9780511754197.002` [Accessed 2024-01-10].

Haarnoja, T., Zhou, A., Abbeel, P. and Levine, S., 2018. Soft actor-critic: Off-policy maximum

entropy deep reinforcement learning with a stochastic actor. *Corr* [Online], abs/1801.01290. 1801.01290, Available from: http://arxiv.org/abs/1801.01290.

Han, D., Mulyana, B., Stankovic, V. and Cheng, S., 2023. A Survey on Deep Reinforcement Learning Algorithms for Robotic Manipulation. *Sensors* [Online], 23(7), p.3762. Number: 7 Publisher: Multidisciplinary Digital Publishing Institute. Available from: https://doi.org/10.3390/s23073762 [Accessed 2024-01-14].

Hand, D.J. and Henley, W.E., 1997. Statistical Classification Methods in Consumer Credit Scoring: A Review. *Journal of the royal statistical society series a: Statistics in society* [Online], 160(3), pp.523–541. Available from: https://doi.org/10.1111/j.1467-985X.1997.00078.x [Accessed 2024-01-10].

Handhika, T., Sabri, A. and Murni, M., 2021. Reinforcement Learning on the Credit Risk-Based Pricing [Online]. *2021 2nd International Conference on Computational Methods in Science & Technology (ICCMST)*. Mohali, India: IEEE, pp.233–236. Available from: https://doi.org/10.1109/ICCMST54943.2021.00056 [Accessed 2024-01-08].

Hu, Y.J. and Lin, S.J., 2019. Deep Reinforcement Learning for Optimizing Finance Portfolio Management [Online]. *2019 Amity International Conference on Artificial Intelligence (AICAI)*. Dubai, United Arab Emirates: IEEE, pp.14–20. Available from: https://doi.org/10.1109/AICAI.2019.8701368 [Accessed 2024-01-08].

IFRS 9 Financial Instruments, 2009. [Online]. Available from: https://www.ifrs.org.

Jang, J.K. and Seong, N., 2023. Model based reinforcement learning with non-gaussian environment dynamics and its application to portfolio optimization. *Expert systems with applications* [Online]. Available from: https://doi.org/10.1016/j.eswa.2023.119556.

Jensen, H.L., 1992. Using Neural Networks for Credit Scoring. *Managerial finance* [Online], 18(6), pp.15–26. Available from: https://doi.org/10.1108/eb013696 [Accessed 2024-01-10].

Jiang, Z., Xu, D. and Liang, J., 2017. A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem [Online]. ArXiv:1706.10059 [cs, q-fin]. Available from: https://doi.org/10.48550/arXiv.1706.10059 [Accessed 2023-03-02].

Johnson, M.K.a.K., 2020. *Feature Engineering and Selection: A Practical Approach for Predictive Models* [Online]. Available from: http://www.feat.engineering/ [Accessed 2024-01-17].

Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., Tunyasuvunakool, K., Bates, R., Žídek, A., Potapenko, A., Bridgland, A., Meyer, C., Kohl, S.A.A., Ballard, A.J., Cowie, A., Romera-Paredes, B., Nikolov, S., Jain, R., Adler, J., Back, T., Petersen, S., Reiman, D., Clancy, E., Zielinski, M., Steinegger, M., Pacholska, M., Berghammer, T., Bodenstein, S., Silver, D., Vinyals, O., Senior, A.W., Kavukcuoglu, K., Kohli, P. and Hassabis, D., 2021. Highly accurate protein structure prediction with AlphaFold. *Nature* [Online], 596(7873), pp.583–589. Number: 7873 Publisher: Nature Publishing Group. Available from: https://doi.org/10.1038/s41586-021-03819-2 [Accessed 2024-01-14].

Kiran, B.R., Sobh, I., Talpaert, V., Mannion, P., Sallab, A.A.A., Yogamani, S. and Pérez, P., 2021. Deep Reinforcement Learning for Autonomous Driving: A Survey [Online].

ArXiv:2002.00444 [cs]. Available from: http://arxiv.org/abs/2002.00444 [Accessed 2024-01-14].

Kolm, P.N. and Ritter, G., 2019. Modern perspectives on reinforcement learning in finance. *Ssrn electronic journal* [Online]. Available from: https://doi.org/10.2139/ssrn.3449401.

Kotsiantis, S., Kanellopoulos, D. and Pintelas, P., 2005. Handling imbalanced datasets: A review. *Gests international transactions on computer science and engineering*, 30, pp.25–36.

Krakovsky, M., 2016. The renaissance of reinforcement learning. *Communications of the acm*, 59(8), pp.14–16.

Leike, J. et al., 2018. Scalable agent alignment via reward modeling: a research direction. *arxiv: Learning* [Online]. Available from: https://arxiv.org/abs/1811.07871.

Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. and Wierstra, D., 2019. Continuous control with deep reinforcement learning [Online]. ArXiv:1509.02971 [cs, stat]. Available from: https://doi.org/10.48550/arXiv.1509.02971 [Accessed 2024-01-15].

Louie, K., 2022. Asymmetric and adaptive reward coding via normalized reinforcement learning. *Plos computational biology* [Online], 18(7), p.e1010350. Available from: https://doi.org/10.1371/journal.pcbi.1010350.

Lu, C.I., 2023. Evaluation of deep reinforcement learning algorithms for portfolio optimisation. 2307.07694.

Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T.P., Harley, T., Silver, D. and Kavukcuoglu, K., 2016. Asynchronous methods for deep reinforcement learning. *Corr* [Online], abs/1602.01783. 1602.01783, Available from: http://arxiv.org/abs/1602.01783.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing Atari with Deep Reinforcement Learning [Online]. ArXiv:1312.5602 [cs]. Available from: https://doi.org/10.48550/arXiv.1312.5602 [Accessed 2024-01-15].

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. et al., 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540), pp.529–533.

Moody, J. and Saffell, M., 1998. Reinforcement learning for trading. *Neural information processing systems* [Online]. Available from: https://papers.nips.cc/paper/1551-reinforcement-learning-for-trading.pdf.

Nigmonov, A. and Shams, S., 2021. COVID-19 pandemic risk and probability of loan default: evidence from marketplace lending market. *Financial innovation* [Online], 7(1), p.83. Available from: https://doi.org/10.1186/s40854-021-00300-x [Accessed 2024-01-18].

Nousi, P., Passalis, N. and Tefas, A., 2023. A sharpe ratio based reward scheme in deep reinforcement learning for financial trading. *Artificial intelligence applications and innovations*. Springer, pp.17–28.

Ostrovski, G. et al., 2023. Scaling laws for reward model overoptimization. *arxiv preprint* [Online]. Available from: https://arxiv.org/abs/2210.10760.

Powell, W.B., 2011. *Approximate dynamic programming: Solving the curses of dimensionality*. John Wiley & Sons.

Raffin, A., Hill, A., Ernestus, M., Gleave, A., Kanervisto, A. and Dormann, N., 2019. Stable baselines3. `https://github.com/DLR-RM/stable-baselines3`.

Ramyachitra, D.D. and Manikandan, P., 2014. enIMBALANCED DATASET CLASSIFICATION AND SOLUTIONS: A REVIEW. *International journal of computing and business research*, 5(4).

Schaul, T., Quan, J., Antonoglou, I. and Silver, D., 2016. Prioritized experience replay. *International conference on learning representations (iclr)*.

Schaul, T. et al., 2021. Return-based scaling: Yet another normalisation trick for deep rl. *arxiv: Learning* [Online]. Available from: `https://arxiv.org/abs/2105.05347`.

Schmidhuber, J., 2015. Deep learning in neural networks: An overview. *Neural networks*, 61, pp.85–117.

Schulman, J., Levine, S., Moritz, P., Jordan, M.I. and Abbeel, P., 2015. Trust region policy optimization. *Corr* [Online], abs/1502.05477. `1502.05477`, Available from: `http://arxiv.org/abs/1502.05477`.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2017a. Proximal policy optimization algorithms. *Corr* [Online], abs/1707.06347. `1707.06347`, Available from: `http://arxiv.org/abs/1707.06347`.

Schulman, J. et al., 2017b. Proximal policy optimization algorithms. *arxiv preprint* [Online]. Available from: `https://arxiv.org/abs/1707.06347`.

Siddiqi, N., 2012. *Credit Risk Scorecards: Developing and Implementing Intelligent Credit Scoring*. John Wiley & Sons.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D. and Riedmiller, M., 2014. Deterministic Policy Gradient Algorithms.

Song, M., Wang, J., Zhang, T., Zhang, G., Zhang, R. and Su, S., 2020. Effective Automated Feature Derivation via Reinforcement Learning for Microcredit Default Prediction [Online]. *2020 International Joint Conference on Neural Networks (IJCNN)*. Glasgow, United Kingdom: IEEE, pp.1–8. Available from: `https://doi.org/10.1109/IJCNN48605.2020.9207410` [Accessed 2024-01-08].

Sutton, R.S. and Barto, A.G., 1998. *Reinforcement learning: An introduction*. MIT Press.

Sutton, R.S. and Barto, A.G., 2018. *Reinforcement learning: An introduction (2nd edition)*. MIT Press.

Sutton, R.S., McAllester, D., Singh, S. and Mansour, Y., 1999. Policy Gradient Methods for Reinforcement Learning with Function Approximation [Online]. *Advances in Neural Information Processing Systems*. MIT Press, vol. 12. Available from: `https://proceedings.neurips.cc/paper/1999/hash/464d828b85b0bed98e80ade0a5c43b0f-Abstract.html` [Accessed 2024-01-15].

Szepesvári, C., 2010. Algorithms for reinforcement learning. *Synthesis lectures on artificial intelligence and machine learning*, 4(1), pp.1–103.

Thomas, L., Crook, J. and Edelman, D., 2017. *Credit Scoring and Its Applications, Second Edition*. SIAM. Google-Books-ID: 7UQzDwAAQBAJ.

Thomas, L.C., 2000. A survey of credit and behavioural scoring: forecasting financial risk of lending to consumers. *International journal of forecasting* [Online], 16(2), pp.149–172. Available from: https://doi.org/10.1016/S0169-2070(00)00034-0 [Accessed 2024-01-10].

Towers, M., Terry, J.K., Kwiatkowski, A., Balis, J.U., Cola, G.d., Deleu, T., Goulão, M., Kallinteris, A., KG, A., Krimmel, M., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J.J., Shen, A.T.J. and Younis, O.G., 2023. Gymnasium [Online]. Available from: https://doi.org/10.5281/zenodo.8127026 [Accessed 2023-07-08].

Van Hasselt, H., Guez, A. and Silver, D., 2016. Deep reinforcement learning with double q-learning. *Aaai*. vol. 16, pp.2094–2100.

Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M. and De Freitas, N., 2016. Dueling network architectures for deep reinforcement learning. *arxiv preprint arxiv:1511.06581*.

Watkins, C.J.C.H. and Dayan, P., 1992. Q-learning. *Machine learning* [Online], 8(3), pp.279–292. Available from: https://doi.org/10.1007/BF00992698 [Accessed 2024-01-15].

Wu, Y., Mansimov, E., Liao, S., Grosse, R. and Ba, J., 2017. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. 1708.05144.

Wu, Y.H. et al., 2023. Ans: Adaptive network scaling for deep rectifier reinforcement learning models. *arxiv: Learning* [Online]. Available from: https://arxiv.org/abs/1809.02112.

Xu, L., Skoularidou, M., Cuesta-Infante, A. and Veeramachaneni, K., 2019. enModeling Tabular data using Conditional GAN.

Zhang, Z., Zohren, S. and Roberts, S., 2019. Deep Reinforcement Learning for Trading [Online]. ArXiv:1911.10107 [cs, q-fin]. Available from: http://arxiv.org/abs/1911.10107 [Accessed 2024-01-08].

Appendix A

Dataset Correlations

A.1 Monthly Correlations

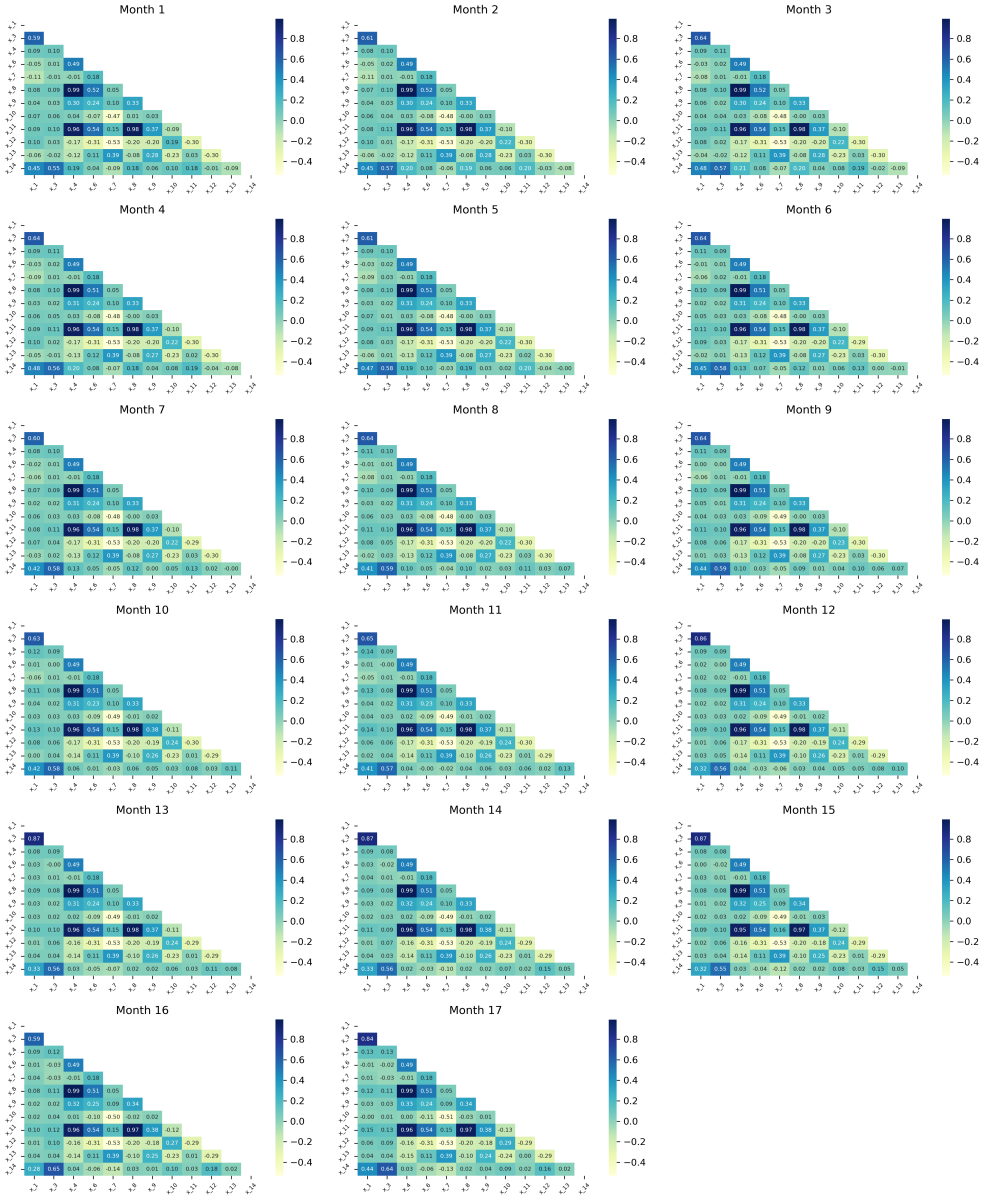


Figure A.1: Monthly correlations between variables.

A.2 Correlation histories between features

Figure A.2: Correlation among variables over time. Alternative view, of figure 3.1. The time series nature of the dataset requires viewing the correlations over time (Aliases defined in table 3.1.

A.3 Pair Distribution plots between features

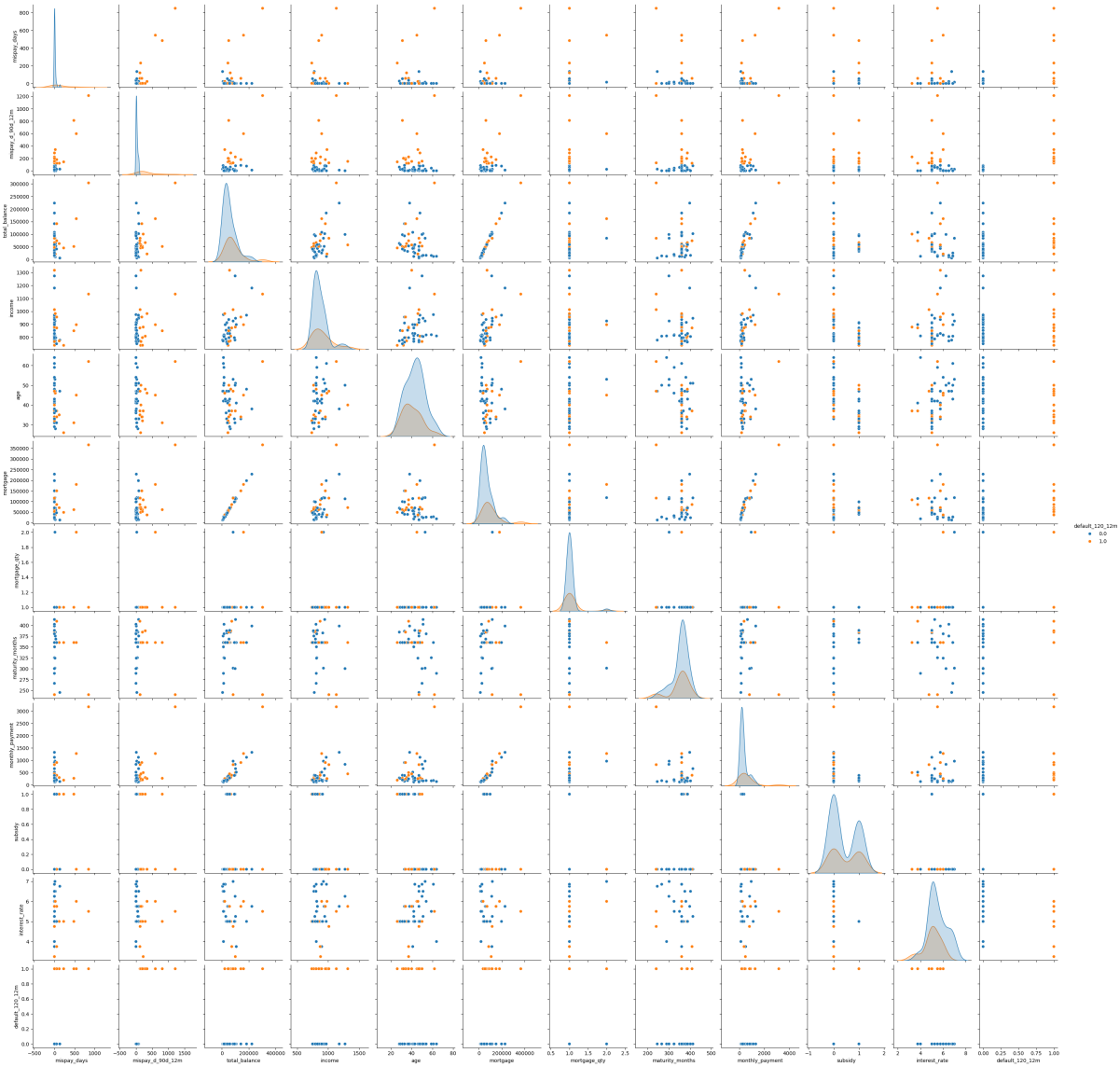


Figure A.3: Pair distributions plot for all variables showing broad variance among all variables.

Appendix B

User Documentation

B.1 Github Repository

All developed code assets is provided in Appendix section however, for proper access to the files and folder structures required to run jupyter notebooks and the associated custom developed packages please visit the following GitHub repository for more information:

https://github.com/edunuke/Bath_MScAI.git.

This repository contains all code and dataset related to the thesis for MSc. in AI.

Appendix C

Raw Results Output

C.1 Multi Class Environment

C.1.1 Multiclass Classification Performance Metrics

Due to time limitations this section could not be included inside the thesis discussions but showed promising results.

Table C.1: Dueling DQN

	precision	recall	f1-score	support
0	0.84	0.88	0.86	1547
1	0.84	0.78	0.80	1991
2	0.81	0.84	0.82	1530
accuracy			0.83	5068
macro avg	0.83	0.83	0.83	5068
avg	0.83	0.83	0.83	5068

Table C.2: Gradient Boosting

	precision	recall	f1-score	support
0	0.97	1	0.98	1547
1	0.99	0.96	0.98	1991
2	0.98	0.99	0.99	1530
accuracy			0.98	5068
macro avg	0.98	0.98	0.98	5068
macro avg	0.98	0.98	0.98	5068

C.1.2 Default Rate Predicted vs Actual

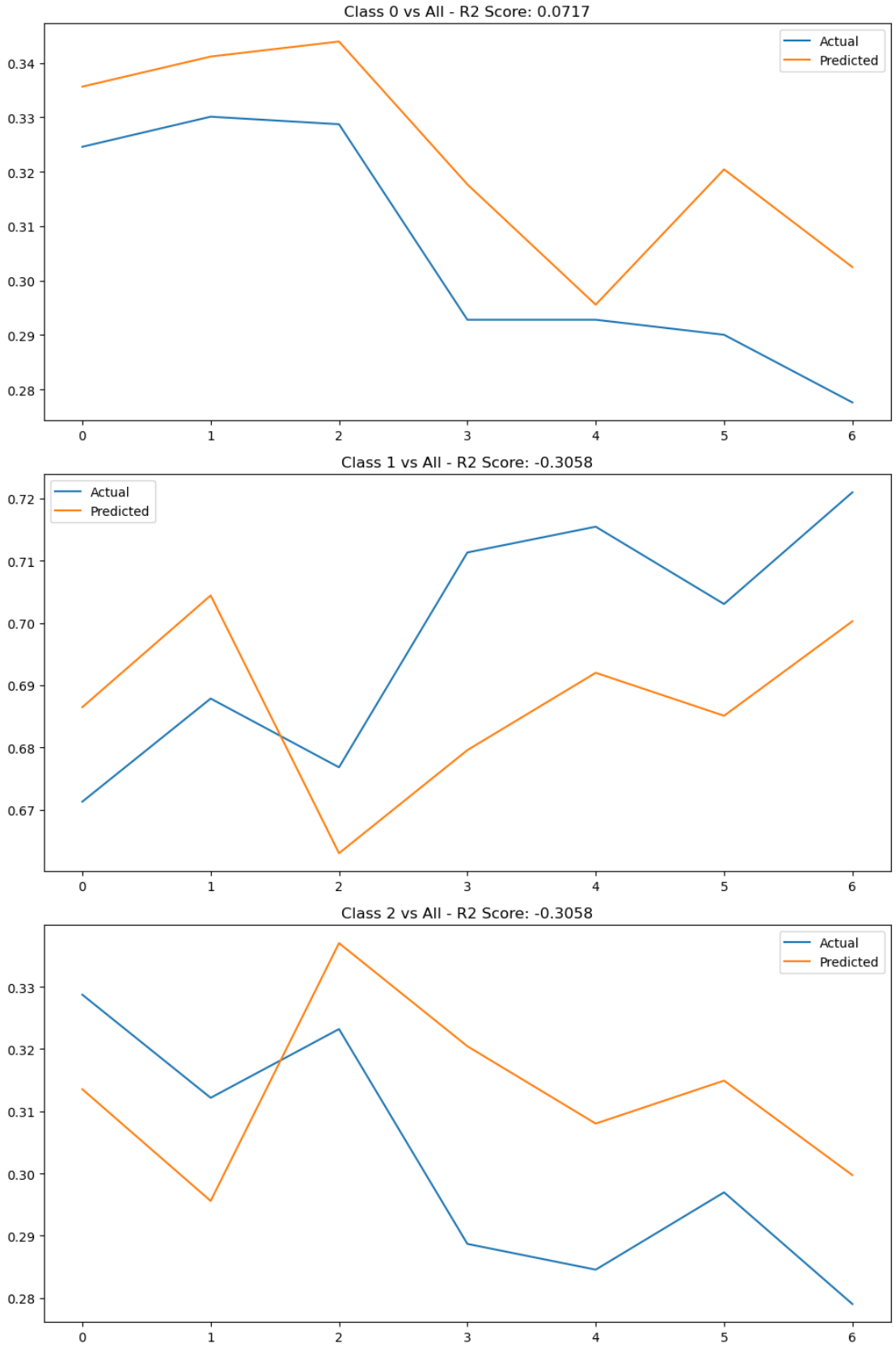


Figure C.1: Monthly correlations between variables.

C.2 Default Rate Outputs

Table C.3: Predicted and actual default rates on the training set.

Date	Actual	DQN	DoubleDQN	DuelDQN	A2C	TRPO	PPO
2020-03-31	0.333333	0.338308	0.333333	0.333333	0.318408	0.315091	0.298507
2020-04-30	0.341625	0.346600	0.343284	0.341625	0.326700	0.313433	0.291874
2020-05-31	0.341625	0.343284	0.341625	0.341625	0.328358	0.318408	0.296849
2020-06-30	0.356551	0.361526	0.361526	0.358209	0.339967	0.331675	0.300166
2020-07-31	0.318408	0.343284	0.341625	0.334992	0.316750	0.316750	0.291874
2020-08-31	0.336650	0.344942	0.351575	0.343284	0.315091	0.301824	0.315091
2020-09-30	0.331675	0.338308	0.341625	0.334992	0.321725	0.281924	0.311774
2020-10-31	0.318408	0.325041	0.328358	0.320066	0.303483	0.268657	0.308458
2020-11-30	0.318408	0.321725	0.321725	0.320066	0.301824	0.275290	0.318408
2020-12-31	0.301824	0.316750	0.315091	0.311774	0.290216	0.271973	0.325041
2021-01-31	0.316750	0.341625	0.336650	0.331675	0.300166	0.281924	0.325041
2021-02-28	0.336650	0.353234	0.338308	0.339967	0.313433	0.285240	0.334992
2021-03-31	0.333333	0.343284	0.343284	0.338308	0.305141	0.275290	0.330017
2021-04-30	0.336650	0.354892	0.346600	0.339967	0.313433	0.280265	0.334992
2021-05-31	0.344942	0.354892	0.351575	0.348259	0.318408	0.291874	0.341625
2021-06-30	0.334992	0.349917	0.346600	0.349917	0.321725	0.291874	0.348259

Table C.4: Predicted and actual default rates on the testing set.

Date	Actual	DQN	DDQN	DuelDQN	A2C	TRPO	PPO
2021-07-31	0.348259	0.359867	0.351575	0.348259	0.331675	0.320066	0.353234
2021-08-31	0.339967	0.353234	0.346600	0.346600	0.326700	0.308458	0.339967
2021-09-30	0.354892	0.358209	0.354892	0.356551	0.336650	0.325041	0.358209
2021-10-31	0.343284	0.353234	0.341625	0.344942	0.325041	0.316750	0.341625
2021-11-30	0.348259	0.359867	0.349917	0.348259	0.336650	0.318408	0.349917

C.3 Train Set Performance Metrics

Table C.5: DQN

	precision	recall	f1-score	support
0	1	0.98	0.99	6451
1	0.97	1	0.98	3197
accuracy			0.99	9648
macro avg	0.98	0.99	0.99	9648
weighted avg	0.99	0.99	0.99	9648

Table C.6: PPO

	precision	recall	f1-score	support
0	0.92	0.94	0.93	6451
1	0.87	0.83	0.85	3197
accuracy			0.9	9648
macro avg	0.89	0.88	0.89	9648
weighted avg	0.9	0.9	0.9	9648

Table C.7: A2C

	precision	recall	f1-score	support
0	0.96	0.98	0.97	6451
1	0.97	0.92	0.94	3197
accuracy			0.96	9648
macro avg	0.96	0.95	0.96	9648
weighted avg	0.96	0.96	0.96	9648

Table C.8: TRPO

	precision	recall	f1-score	support
0	0.9	0.95	0.92	6451
1	0.88	0.78	0.83	3197
accuracy			0.89	9648
macro avg	0.89	0.86	0.87	9648
weighted avg	0.89	0.89	0.89	9648

Table C.9: Double DQN

	precision	recall	f1-score	support
0	1	0.99	0.99	6451
1	0.97	1	0.99	3197
accuracy			0.99	9648
macro avg	0.99	0.99	0.99	9648
weighted avg	0.99	0.99	0.99	9648

Table C.10: Duel DQN

	precision	recall	f1-score	support
0	1	0.99	1	6451
1	0.98	1	0.99	3197
accuracy			0.99	9648
macro avg	0.99	0.99	0.99	9648
weighted avg	0.99	0.99	0.99	9648

Table C.11: Gradient Boosting (Baseline

	precision	recall	f1-score	support
0	1	1	1	6451
1	1	1	1	3197
accuracy			1	9648
macro avg	1	1	1	9648
weighted avg	1	1	1	9648

C.4 Testing Set Performance Metrics

Table C.12: DQN

	precision	recall	f1-score	support
0	1	0.98	0.99	1969
1	0.96	0.99	0.98	1046
accuracy			0.98	3015
macro avg	0.98	0.99	0.98	3015
weighted avg	0.98	0.98	0.98	3015

Table C.13: PPO

	precision	recall	f1-score	support
0	0.94	0.94	0.94	1969
1	0.89	0.89	0.89	1046
accuracy			0.92	3015
macro avg	0.91	0.92	0.91	3015
weighted avg	0.92	0.92	0.92	3015

Table C.14: A2C

	precision	recall	f1-score	support
0	0.96	0.99	0.97	1969
1	0.97	0.93	0.95	1046
accuracy			0.97	3015
macro avg	0.97	0.96	0.96	3015
weighted avg	0.97	0.97	0.97	3015

Table C.15: TRPO

	precision	recall	f1-score	support
0	0.89	0.93	0.91	1969
1	0.86	0.79	0.83	1046
accuracy			0.88	3015
macro avg	0.88	0.86	0.87	3015
weighted avg	0.88	0.88	0.88	3015

Table C.16: Double DQN

	precision	recall	f1-score	support
0	0.99	0.99	0.99	1969
1	0.98	0.99	0.99	1046
accuracy			0.99	3015
macro avg	0.99	0.99	0.99	3015
weighted avg	0.99	0.99	0.99	3015

Table C.17: Dueling DQN

	precision	recall	f1-score	support
0	1	0.99	1	1969
1	0.99	0.99	0.99	1046
accuracy			0.99	3015
macro avg	0.99	0.99	0.99	3015
weighted avg	0.99	0.99	0.99	3015

Table C.18: Gradient Boostin (Baseline)

	precision	recall	f1-score	support
0	1	1	1	1969
1	1	1	1	1046
accuracy			1	3015
macro avg	1	1	1	3015
weighted avg	1	1	1	3015

Appendix D

Code

D.1 File: README.md

This file contains a conda virtual environment configuration file to create the virtual environment to run the code.

```
# Introduction

This folder contains all code and dataset related to the the thesis for MSc. in AI.

The folder structure is as follows:

1. data:
   1. dataset_1: Synthetic dataset trained on TVAE from real data (main artifact for the thesis)
      1. processed: processed dataset after running model/1_dataprep.ipynb
   2. dataset_2: Secondary dataset from https://www.kaggle.com/datasets/parisrohan/credit-score-classifica
      1. raw: raw dataset as downloaded

2. envs: virtual environment yaml file to run a replicated environment for running all code

3. imgs: images used during the thesis write up.


4. model: all code sits inside this directory
   1. logs: folder holds all logs from all agents experiment runs
   2. output: contains all serialized trained models
   3. utils:
      1. common.py : common code used across notebooks
      2. constants.py : constants to replicate sampled population
      3. doubledqn.py : personal implementation of double dqn extending stable baselines 3
      4. duelingdqn.py: personal implementation of dueling dqn extending stable baselines 3
      5. networks.py : personal implementation of different neural network arquitectures (feature extractors) for stable baselines 3
      6. riskenv.py : personal implementation of different environment for testing
   4. 1_dataprep.ipynb : all preprocessing of the raw data happens here.
   5. 2_modeling.ipynb : all modeling code for testing the agents and environment for dataset_1 (binary class)
   6. 3_modeling.ipynb : all modeling code for testing the agents and environment for dataset_2 (multi class Environment dataset)


** All code is commented for better legibility **
```

D.2 File: environment.yml

This file contains a conda virtual environment configuration file to create the virtual environment to run the code.

```
name: risk_env
channels:
  - conda-forge
  - pytorch
  - defaults
dependencies:
  - python=3.10
  - ipykernel
  - matplotlib
  - pandas
  - numpy
  - seaborn
  - pytorch
  - torchvision
  - torchaudio
  - scikit-learn
  - imbalanced-learn
  - swig
  - pip==21
  - pip:
    - ucimlrepo
    - setuptools==65.5.0
    - wheel==0.38.0
    - stable-baselines3
    - sb3-contrib
    - gym[extra]
```

D.3 File: riskenv.py

This file contains all of the environment developed as described in this thesis.

```
"""
Author: Eduardo Perez Denadai
Date: 2022-10-25
Description: Risk Management Environment
"""
import numpy as np
import pandas as pd
from collections import deque
import gymnasium as gym
from gymnasium import spaces
from sklearn.utils import compute_class_weight
from sklearn.metrics import (f1_score,
                             recall_score,
                             accuracy_score)


class RiskManagementEnv(gym.Env):
    """
    Initializes the Gymnasium environment.

    Parameters:
            df (DataFrame): The input DataFrame containing the features and default column.
            debug (bool): A flag indicating whether to print debug information.
            scaled_features (bool): A flag indicating whether the features should be scaled.
            accepts_discrete_action (bool): A flag indicating whether the agent accepts discrete actions.
            features_col (list): A list of feature column names.
            default_col (str): The name of the default column.
            obs_dim (int): The dimension of the observation space.
            client_dim (int): The dimension of the client space.
            action_dim (int): The dimension of the action space.
            reward_delay_steps (int, optional): The number of steps to delay the reward. Defaults to 1.
            seed (int, optional): The random seed. Defaults to 123.
            model_name (str, optional): The name of the model. Defaults to "".

    Returns:
        None
    """
    def __init__(self, df, debug, scaled_features, accepts_discrete_action,
                 features_col, default_col, obs_dim, client_dim, action_dim,
                 rng, eward_delay_steps=1, seed=123, model_name=""):

        super().__init__()
        self.debug = debug
        self.rng = rng
        self.df = df[features_col + [default_col]].copy().astype(np.float32)
        self.client_list = self.df.loc[0].index.tolist()
        self.features = features_col
        self.default_col = default_col
        self.client_dim = client_dim
        self.obs_dim = obs_dim
        self.action_dim = action_dim
        self.reward_delay_steps = 1
        self.action_outcome_history = deque(maxlen=self.reward_delay_steps)
        self.accepts_discrete_action = accepts_discrete_action
        self.state = None
        self.month = 0
        self.max_month = self.df.index.get_level_values(0).max()
        self.current_client_id = self.rng.choice(self.client_list)
        self.max_client_id = self.df.index.get_level_values(1).max()
        self.default_rate = 0.0
        self.model_name = model_name
        self.reward = 0
        self.action_hist = []
        self.penalty = 0
        self.count = 0
        self.step_counter = 0
        self.N_SAMPLES = 50
        self.sampled_clients = []
        self.action_space = self._define_action_space(self.accepts_discrete_action)
        self.observation_space = self._define_observation_space(scaled_features, features_col, df)

    def _define_action_space(self, accepts_discrete_action):
        """
```

```
        Define the action space for the agent.

        Parameters:
            accepts_discrete_action (bool): Whether the agent accepts discrete actions.

        Returns:
            spaces.Space: The defined action space.
        """
        if accepts_discrete_action:
            return spaces.Discrete(self.action_dim)
        else:
            return spaces.Box(low=0, high=1, shape=(self.action_dim - 1,), dtype="int")

    def _define_observation_space(self, scaled_features, features_col, df):
        """
        Generates the observation space for the agent based on the given scaled features, features column, and dataframe.

        Parameters:
            scaled_features (bool): A boolean value indicating whether the features should be scaled.
            features_col (str): The name of the column containing the features in the dataframe.
            df (pandas.DataFrame): The dataframe containing the features.

        Returns:
            gym.spaces.Box: The observation space for the agent.
        """
        if scaled_features:
            return spaces.Box(low=-1, high=1, shape=(self.obs_dim,), dtype=np.float32)
        else:
            low = df[features_col].min().values
            high = df[features_col].max().values
            return spaces.Box(low=low, high=high, shape=(self.obs_dim,), dtype=np.float32)

    def reset(self, seed=None, options=None):
        """
        Reset the environment to its initial state.

        Parameters:
```

```
        - seed (int or None): The random seed to use for initializing the environment. If None, a random seed will be used.
        - options (dict or None): Additional options for resetting the environment. If None, default options will be used.

        Returns:
        - state (numpy.ndarray): The initial state of the environment.
        - info (dict): Additional information about the reset.
        """
        self.reward = 0
        self.step_counter = 0
        self.action_outcome_history.clear()
        self.month = 0
        self.current_client_id = self.rng.choice(self.client_list)
        self.state = self.df.loc[self.month, self.current_client_id][self.features].values
        return self.state, {}

    def _calculate_delayed_reward(self,):
        """
        Calculate the delayed reward based on the action-outcome history.

        Returns:
            reward (int): The calculated delayed reward.

        Notes:
            - The delayed reward is calculated based on the action-outcome history.
            - The reward is determined based on the values of 'acts' (actions) and 'outs' (outcomes).
        """
        reward = 0
        acts =[]
        outs = []
        for action, outcome in self.action_outcome_history:
            acts.append(action)
            outs.append(outcome)

        # cost matrix
        if (acts[0] == 0) & (outs[-1] == 0): #TN
            reward = 1+self.class_weight[1]
        if (acts[0] == 1) & (outs[-1] == 1): # TP
            reward = 2+self.class_weight[1]
        if (acts[0] == 0) &  (outs[-1] == 1): # FN
```

```
            reward = -2-self.class_weight[0]
        if (acts[0] == 1) & (outs[-1] == 0): #FP
            reward = -1-self.class_weight[0]
        return reward

    def _scale_reward(self, reward):
        """
        Scales a given reward value between 0 and 1 based on the
    minimum and maximum reward values.

        Parameters:
            reward (float): The reward value to be scaled.

        Returns:
            float: The scaled reward value between 0 and 1.
        """
        max_reward = 2+self.class_weight[1]
        min_reward = -2-self.class_weight[0]
        return (reward - min_reward) / (max_reward - min_reward)

    def _calculate_penalty(self):
        """
        Calculate the penalty for the current month based on the
    default rate and the action history.

        Returns:
            None
        """
        actual_default_rate =
    self.df.loc[self.month][self.default_col].sum() /
    self.N_SAMPLES
        self.default_rate = sum(self.action_hist) / self.N_SAMPLES

        if len(self.action_hist) <
    self.df.loc[self.month].shape[0]:
            self.action_hist.extend([0] *
    (self.df.loc[self.month].shape[0] - len(self.action_hist)))

        self.penalty = -(1/2) * abs(actual_default_rate -
    self.default_rate)

        actual_defaults =
    self.df.loc[self.month][self.default_col].values
        self.penalty += f1_score(actual_defaults, self.action_hist)
        self.penalty += recall_score(actual_defaults,
    self.action_hist)

        self.action_hist = []

    def step(self, action):
        """
        Updates the environment state based on the given action
    and returns the
        updated state, reward, done flag, and additional
    information.

        Parameters:
            action (any): The action taken by the agent.

        Returns:
            state (numpy.ndarray): The updated state of the
    environment.
            reward (float): The reward obtained after taking the
    action.
            done (bool): A flag indicating if the episode is done.
            False (bool): A flag indicating if the episode has
    terminated with success.
            {} (dict): Additional information about the step.
        """
        done = False
        self.sampled_clients.append(self.current_client_id)
        self.current_default =
    self.df.loc[self.month][self.default_col].mean()
        self.class_weight = compute_class_weight('balanced',
    classes=list(set(self.df.loc[self.month][self.default_col])),

    y=self.df.loc[self.month][self.default_col])

        if not self.accepts_discrete_action:
            action = int(action.round())

        self.action_hist.append(action)
        default = self.df.loc[self.month,
    self.current_client_id][self.default_col]
        self.action_outcome_history.append((action, default))

        if len(self.sampled_clients) >= self.max_client_id:
            self.action_hist = []
            self.month += 1
            self.action_outcome_history.clear()
            self.sampled_clients = []

        if self.debug and (self.count % 100 == 0):
```

```
            print(f"default: {self.current_default:>5.3f} - penalty: {self.penalty:>5.2f} "
                  f"reward: {self.reward:>5.2f} - client: {self.current_client_id:>6} "
                  f"month: {self.month:>3} - done: {done}")

        if self.month >= self.max_month:
            done = True
            return self.state, self.reward, done, False, {}

        self.step_counter += 1
        if self.step_counter >= self.N_SAMPLES:
            self._calculate_penalty()
            self.step_counter = 0

        if len(self.action_outcome_history) == self.reward_delay_steps:
            self.reward = self._calculate_delayed_reward() + self._scale_reward(self.penalty)
            self.action_outcome_history.clear()
        else:
            self.reward += 0

        self.count += 1
        while True:
            next_client = self.rng.choice(self.client_list)
            if next_client not in self.sampled_clients:
                self.current_client_id = next_client
                break

        self.state = self.df.loc[self.month, self.current_client_id][self.features].values
        return self.state, self.reward, done, False, {}

    def render(self, mode='human', close=False):
        """
        Render the environment.

        Parameters:
            mode (str): The rendering mode. Defaults to 'human'.
            close (bool): Whether to close the environment after rendering. Defaults to False.

        Returns:
            None
        """
        pass


##################################################################
    Adds penalty and reward history to state
    ##################################################################

class RiskManagementEnvDynaState(gym.Env):
    """
    Initializes the Gymnasium environment.

    Parameters:
            df (DataFrame): The input DataFrame containing the features and default column.
            debug (bool): A flag indicating whether to print debug information.
            scaled_features (bool): A flag indicating whether the features should be scaled.
            accepts_discrete_action (bool): A flag indicating whether the agent accepts discrete actions.
            features_col (list): A list of feature column names.
            default_col (str): The name of the default column.
            obs_dim (int): The dimension of the observation space.
            client_dim (int): The dimension of the client space.
            action_dim (int): The dimension of the action space.
            reward_delay_steps (int, optional): The number of steps to delay the reward. Defaults to 1.
            seed (int, optional): The random seed. Defaults to 123.
            model_name (str, optional): The name of the model. Defaults to "".

    Returns:
        None
     """
    def __init__(self, df, debug, scaled_features, accepts_discrete_action,
                 features_col, default_col, obs_dim, client_dim, action_dim,
                 rng, eward_delay_steps=1, seed=123, model_name=""):

        super().__init__(df, debug, scaled_features, accepts_discrete_action,
                 features_col, default_col, obs_dim, client_dim, action_dim,
                 rng, eward_delay_step, seed, model_name)
```

```
    def _define_observation_space(self, scaled_features,
features_col, df):
        """
        Define the observation space for the environment.

        Parameters:
            scaled_features (bool): Flag indicating whether the
features should be scaled.
            features_col (str): Column name for the features in
the dataframe.
            df (pandas.DataFrame): The input dataframe containing
the features.

        Returns:
            gym.Space: The defined observation space.
        """
        if scaled_features:
            return spaces.Box(low=-1, high=1, shape=(self.obs_dim
+ 2,), dtype=np.float32)  # +2 for default_rate and penalty
        else:
            low = np.append(df[features_col].min().values, [0,
-np.inf])  # Extend the low with -inf for new features
            high = np.append(df[features_col].max().values, [0,
np.inf])  # Extend the high with inf for new features
            return spaces.Box(low=low, high=high,
shape=(self.obs_dim + 2,), dtype=np.float32)  # +2 for
default_rate and penalty

    def reset(self, seed=None, options=None):
        """
        Reset the environment to its initial state.

        Parameters:
        - seed (int or None): The random seed to use for
initializing the environment. If None, a random seed will be
used.
        - options (dict or None): Additional options for resetting
the environment. If None, default options will be used.

        Returns:
        - state (numpy.ndarray): The initial state of the
environment.
        - info (dict): Additional information about the reset.
        """
        self.reward = 0
        self.step_counter = 0
        self.action_outcome_history.clear()
        self.month = 0
        self.current_client_id = self.rng.choice(self.client_list)
        self.state = self.df.loc[self.month,
self.current_client_id][self.features].values
        self.state = np.append(self.state, [self.default_rate,
self.penalty])
        self.state = self.state.astype(np.float32)
        return self.state, {}


    def step(self, action):
        """
        Updates the environment state based on the given action
and returns the
        updated state, reward, done flag, and additional
information.

        Parameters:
            action (any): The action taken by the agent.

        Returns:
            state (numpy.ndarray): The updated state of the
environment.
            reward (float): The reward obtained after taking the
action.
            done (bool): A flag indicating if the episode is done.
            False (bool): A flag indicating if the episode has
terminated with success.
            {} (dict): Additional information about the step.
        """
        done = False
        self.sampled_clients.append(self.current_client_id)
        self.current_default =
self.df.loc[self.month][self.default_col].mean()
        self.class_weight = compute_class_weight('balanced',
classes=list(set(self.df.loc[self.month][self.default_col])),

y=self.df.loc[self.month][self.default_col])

        if not self.accepts_discrete_action:
            action = int(action.round())

        self.action_hist.append(action)
        default = self.df.loc[self.month,
self.current_client_id][self.default_col]
        self.action_outcome_history.append((action, default))
```

```
        if len(self.sampled_clients) >= self.max_client_id:
            self.action_hist = []
            self.month += 1
            self.action_outcome_history.clear()
            self.sampled_clients = []

        if self.debug and (self.count % 100 == 0):
            print(f"default: {self.current_default:>5.3f} - penalty: {self.penalty:>5.2f} "
                  f"reward: {self.reward:>5.2f} - client: {self.current_client_id:>6} "
                  f"month: {self.month:>3} - done: {done}")

        if self.month >= self.max_month:
            done = True
            return self.state, self.reward, done, False, {}

        self.step_counter += 1
        if self.step_counter >= self.N_SAMPLES:
            self._calculate_penalty()
            self.step_counter = 0

        if len(self.action_outcome_history) == self.reward_delay_steps:
            self.reward =  self._calculate_delayed_reward() + self._scale_reward(self.penalty)
            self.action_outcome_history.clear()
        else:
            self.reward += 0

        self.count += 1
        while True:
            next_client = self.rng.choice(self.client_list)
            if next_client not in self.sampled_clients:
                self.current_client_id = next_client
                break

        self.state = self.df.loc[self.month, self.current_client_id][self.features].values
        self.state = np.append(self.state, [self.default_rate, self.penalty])
        self.state = self.state.astype(np.float32)

        return self.state, self.reward, done, False, {}

################################################################
    Episode ends every month Env
    ################################################################


class RiskManagementEnvMonthlyEpisodes(RiskManagementEnv):
    def __init__(self, df, debug, scaled_features, accepts_discrete_action,
                 features_col, default_col, obs_dim, client_dim, action_dim,
                 rng, reward_delay_steps=1, seed=123, model_name=""):
        """
        Initializes an instance of the class RiskManagementEnvMonthlyEpisodes
        it inherits from the class RiskManagementEnv and modifies the _reset() and _step() methods.

        Args:
            df (pandas.DataFrame): The input DataFrame.
            debug (bool): A flag indicating whether to enable debug mode.
            scaled_features (bool): A flag indicating whether to use scaled features.
            accepts_discrete_action (bool): A flag indicating whether the model accepts discrete actions.
            features_col (str): The name of the column containing the features.
            default_col (str): The name of the column containing the default values.
            obs_dim (int): The dimension of the observation space.
            client_dim (int): The dimension of the client space.
            action_dim (int): The dimension of the action space.
            rng (np.random.RandomState): The random number generator object.
            reward_delay_steps (int, optional): The number of steps to delay the reward. Defaults to 1.
            seed (int, optional): The seed for the random number generator. Defaults to 123.
            model_name (str, optional): The name of the model. Defaults to "".

        Returns:
            None
        """

        super().__init__(df, debug, scaled_features, accepts_discrete_action,
```

```
                        features_col, default_col, obs_dim, client_dim, action_dim,
                        rng, reward_delay_steps, seed, model_name)

    def reset(self, seed=None, options=None):
        self.reward = 0
        self.step_counter = 0
        self.action_outcome_history.clear()
        self.current_client_id = self.rng.choice(self.client_list)
        self.state = self.df.loc[self.month, self.current_client_id][self.features].values
        return self.state, {}


    def step(self, action):
        done = False
        self.sampled_clients.append(self.current_client_id)
        self.current_default = self.df.loc[self.month][self.default_col].mean()
        self.class_weight = compute_class_weight('balanced', classes=list(set(self.df.loc[self.month][self.default_col])),

    y=self.df.loc[self.month][self.default_col])

        if not self.accepts_discrete_action:
            action = int(action.round())

        self.action_hist.append(action)
        default = self.df.loc[self.month, self.current_client_id][self.default_col]
        self.action_outcome_history.append((action, default))

        if len(self.sampled_clients) >= self.max_client_id:
            self.action_hist = []
            self.month += 1
            self.action_outcome_history.clear()
            self.sampled_clients = []
            done = True
            print(f"default: {self.current_default:>5.3f} - penalty: {self.penalty:>5.2f} "
                  f"reward: {self.reward:>5.2f} - client: {self.current_client_id:>6} "
                  f"month: {self.month:>3} - done: {done}")
            return self.state, self.reward, done, False, {}

        if self.month >= self.max_month:
            self.month = 0
            done = True


        self.step_counter += 1
        if self.step_counter >= self.N_SAMPLES:
            self._calculate_penalty()
            self.step_counter = 0

        if len(self.action_outcome_history) == self.reward_delay_steps:
            self.reward = self._calculate_delayed_reward() + self._scale_reward(self.penalty)
            self.action_outcome_history.clear()
        else:
            self.reward += 0

        self.count += 1
        while True:
            next_client = self.rng.choice(self.client_list)
            if next_client not in self.sampled_clients:
                self.current_client_id = next_client
                break

        self.state = self.df.loc[self.month, self.current_client_id][self.features].values
        return self.state, self.reward, done, False, {}



#############################################################
    MultiTarget Env
   #############################################################
class RiskManagementEnvMultiTarget(gym.Env):
    def __init__(self, df, debug, scaled_features, accepts_discrete_action,
                 features_col, default_col, obs_dim, client_dim, action_dim,
                 seed=123, model_name=""):
        super().__init__()
        self.debug = debug
        self.df = df[features_col + [default_col]].copy().astype(np.float32)
        self.seed = seed
        self.rng = np.random.default_rng(seed)
        self.features = features_col
        self.default_col = default_col
        self.client_dim = client_dim
```

```
        self.obs_dim = obs_dim
        self.action_dim = action_dim
        self.reward_delay_steps = 1
        self.action_outcome_history =
    deque(maxlen=self.reward_delay_steps)
        self.accepts_discrete_action = accepts_discrete_action
        self.state = None
        self.month = 0
        self.max_month = df.index.get_level_values(0).max()
        self.current_client_id = self.rng.integers(self.client_dim)
        self.model_name = model_name
        self.reward = 0
        self.action_hist = []
        self.penalty = 0
        self.count = 0
        self.step_counter = 0
        self.N_SAMPLES = 25
        self.current_defaults = np.zeros(self.action_dim)
        self.action_space =
    self._define_action_space(accepts_discrete_action)
        self.observation_space =
    self._define_observation_space(scaled_features, features_col,
    df)

    def _define_action_space(self, accepts_discrete_action):
        if accepts_discrete_action:
            return spaces.Discrete(self.action_dim)
        else:
            # Adjust this if continuous actions are needed for
    multiple labels
            return spaces.Discrete(self.action_dim)

    def _define_observation_space(self, scaled_features,
    features_col, df):
        if scaled_features:
            return spaces.Box(low=-1, high=1,
    shape=(self.obs_dim,), dtype=np.float32)
        else:
            low = df[features_col].min().values
            high = df[features_col].max().values
            return spaces.Box(low=low, high=high,
    shape=(self.obs_dim,), dtype=np.float32)

    def reset(self, seed=None, options=None):
        self.reward = 0
        self.step_counter = 0
        self.action_outcome_history.clear()
        self.month = 0
        self.current_client_id = self.rng.integers(self.client_dim)
        self.state = self.df.loc[self.month,
    self.current_client_id][self.features].values
        return self.state, {}

    def _calculate_delayed_reward(self):
        if self.action_outcome_history:
            acts, outs = zip(*self.action_outcome_history)
            unique_classes = np.unique(outs)
            class_weights = compute_class_weight('balanced',
    classes=unique_classes, y=np.array(outs))
            rewards = dict(zip(unique_classes, class_weights))
            return rewards.get(acts[0], -1) # simple reward scheme
        else:
            return -1

    def _scale_reward(self, reward):
        return np.clip(reward, -1, 1)

    def _calculate_penalty(self):
        # Assuming each action corresponds to a class prediction,
    calculate penalties accordingly
        actual_default_rates = [self.df[self.df[self.default_col]
    == i].shape[0] / self.client_dim for i in
    range(self.action_dim)]
        penalties = [-abs(rate - self.current_defaults[i]) / (rate
    if rate > 0 else 1) for i, rate in
    enumerate(actual_default_rates)]
        self.penalty = sum(penalties) / len(penalties)
        self.action_hist.clear()

    def _update_current_defaults(self):
        class_counts = np.bincount(self.action_hist,
    minlength=self.action_dim)
        self.current_defaults = class_counts / class_counts.sum()

    def step(self, action):
        done = False
        self.current_defaults = self.df[self.df[self.default_col]
    == action].shape[0] / self.client_dim
        self.action_hist.append(action)
        self._update_current_defaults()
        self.class_weight = compute_class_weight('balanced',
    classes=list(set(self.df.loc[self.month][self.default_col])),

    y=self.df.loc[self.month][self.default_col])
```

```
        default = self.df.loc[self.month, self.current_client_id][self.default_col]
        self.action_outcome_history.append((action, default))

        if self.debug & self.count % 100 == 0:
            print(f"reward: {self.reward :>3.5f} - Client: {self.current_client_id} -"
                  f"Action: {action} - Default: {default} - client: {self.month:>3}")

        self.step_counter += 1
        if self.step_counter >= self.N_SAMPLES:
            self._calculate_penalty()
            self.step_counter = 0

        if len(self.action_outcome_history) == self.reward_delay_steps:
            self.reward = self._calculate_delayed_reward()/2 # + self.penalty
            self.action_outcome_history.clear()

        self.month = (self.month + 1) % (self.max_month + 1)
        self.current_client_id = self.rng.integers(self.client_dim)
        self.state = self.df.loc[self.month, self.current_client_id][self.features].values

        if self.month == 0:
            done = True

        self.count += 1
        return self.state, self.reward, done, False, {}

    def render(self, mode='human', close=False):
        # Rendering logic here
        pass


################################################################
    BOOTSTRAPPED ENV
    ################################################################

class RiskManagementEnvBootstrapped(gym.Env):
    def __init__(self,
                 df: pd.DataFrame,
                 debug: bool,
                 scaled_features: bool,
                 accepts_discrete_action: bool,
                 features_col: list,
                 default_col: str,
                 obs_dim: int,
                 client_dim: int,
                 action_dim: int,
                 seed: int = 123,
                 model_name: str = ""):
        super().__init__()

        # Initialize your environment here
        self.debug = debug
        self.df = df[features_col+[default_col]].copy().astype(np.float32)
        self.seed = seed
        self.features = features_col
        self.default_col = default_col
        self.client_dim = client_dim
        self.obs_dim = obs_dim
        self.action_dim = action_dim
        self.reward_delay_steps = 1
        self.action_outcome_history = deque(maxlen=self.reward_delay_steps)

        # define spaces
        self.accepts_discrete_action = accepts_discrete_action

        if self.accepts_discrete_action:
            self.action_space = spaces.Discrete(self.action_dim)
        else:
            self.action_space = spaces.Box(low=0, high=1, shape=(self.action_dim-1,), dtype="int")

        if scaled_features:
            self.observation_space = spaces.Box(low=-1,
                                                high=1,
                                                shape=(self.obs_dim,),
                                                dtype=np.float32)
        else:
            low = self.df[self.features].min().values
            high = self.df[self.features].max().values
            self.observation_space = spaces.Box(low=low,
                                                high=high,

                                                shape=(self.obs_dim,), dtype=np.float32)

        self.state = None
```

```
        self.month = 0
        self.max_month = self.df.index.get_level_values(0).max()
        self.current_client_id = 0
        self.max_client_id =
    self.df.index.get_level_values(1).max()
        self.default_rate = 0.0
        self.rng = np.random.default_rng(self.seed)
        self.model_name = model_name
        self.reward = 0
        self.action_hist = []#np.zeros(int(self.max_client_id)+1)
        self.penalty = 0
        self.count = 0
        self.step_counter = 0
        self.N_SAMPLES = 100
        self.current_default_history = deque(maxlen=10)  # Adjust
    length as needed
        self.running_mean_default = 0.0

    def reset(self, seed=None, options=None):
        self.reward = 0
        self.penalty = 0
        self.step_counter = 0
        self.action_outcome_history.clear()
        self.running_mean_default = 0.0
        self.month = 0
        self.current_client_id = 0
        self.state = self.df.loc[self.month,
    self.current_client_id][self.features].values
        return self.state, {}


    def _calculate_delayed_reward(self):
        acts =[]
        outs = []
        for action, outcome in self.action_outcome_history:
            acts.append(action)
            outs.append(outcome)
        return (self.class_weight[1]+self.current_default) if
    (acts[0]-outs[-1]) == 0 else
    (-self.class_weight[0]+self.current_default)

    def _scale_reward(self, reward):
        # Define the maximum and minimum possible rewards
        max_reward = self.current_default  # Max of absolute
    reward values
        min_reward = -self.current_default  # Symmetric around zero
        scaled_reward = (reward - min_reward) / (max_reward -
    min_reward)

        return scaled_reward

    def _calculate_penalty(self):
        actual_default_rate =
    self.df.loc[self.month][self.default_col].sum() /
    self.N_SAMPLES
        self.default_rate = sum(self.action_hist) / self.N_SAMPLES

        if len(self.action_hist) <
    self.df.loc[self.month].shape[0]:
            self.action_hist.extend([0] *
    (self.df.loc[self.month].shape[0] - len(self.action_hist)))

        self.penalty = -1 * abs(actual_default_rate -
    self.default_rate)

        actual_defaults =
    self.df.loc[self.month][self.default_col].values
        self.penalty += f1_score(actual_defaults, self.action_hist)
        self.penalty += recall_score(actual_defaults,
    self.action_hist)
        self.penalty += accuracy_score(actual_defaults,
    self.action_hist)
        # self.penalty /= 10
        self.action_hist = []

    def _update_running_mean_default(self):
            if self.current_default_history:
                self.running_mean_default =
    np.mean(self.current_default_history)

    def _calculate_current_default(self):
        # Implement the logic to calculate current_default based
    on self.action_hist
        if len(self.action_hist) >= self.N_SAMPLES:
            self.current_default =
    np.mean(self.action_hist[-self.N_SAMPLES:])


    def step(self, action):
        done = False
        self.current_default =
    self.df.loc[self.month][self.default_col].mean()
        self.current_default_history.append(self.current_default)
```

```
        self._update_running_mean_default()
        self.class_weight = compute_class_weight('balanced',
classes=list(set(self.df.loc[self.month][self.default_col])),
y=self.df.loc[self.month][self.default_col])

        # Convert action if necessary
        if not self.accepts_discrete_action:
            action = action.round()[0]

        self.action_hist.append(action)
        if len(self.action_hist) == self.N_SAMPLES:
            self._calculate_current_default()
            self.action_hist.clear()  # Reset the history after
updating


        self._calculate_current_default()
        default = self.df.loc[self.month,
self.current_client_id][self.default_col]
        self.action_outcome_history.append((action, default))

        # Update the step counter and calculate the penalty if
needed
        self.step_counter += 1
        if self.step_counter >= self.N_SAMPLES:
            self._calculate_penalty()
            self.step_counter = 0

        # Check and give the delayed reward
        if len(self.action_outcome_history) ==
self.reward_delay_steps:
            self.reward =  self._calculate_delayed_reward() +
self.penalty/10
            self.action_outcome_history.clear()
        else:
            self.reward += 0  # No reward given until the delay
period is over


        # Condition to move to the next month (SHOULD BE RUNNT)
        if (abs(self.current_default - self.running_mean_default)
< 1e-6) & (self.current_client_id == self.max_client_id):
            self.month += 1
            self.action_hist.clear()
            self.action_outcome_history.clear()
            self.running_mean_default = 0.0
            self.current_client_id = 0


        if self.debug and (self.count % 1 == 0):
            print(f"default: {self.current_default:>5.6f} mean:
{self.running_mean_default:>5.6f} penalty: {self.penalty
:>5.2f} reward: {self.reward :>5.2f} - client:
{self.current_client_id :>6} - month: {self.month} -")#
flush=True, end="\r")

        # Check if the last month is reached
        if self.month >= self.max_month:
            done = True
            return self.state, self.reward, done, False, {}

        # Get the next state
        self.count += 1
        self.current_client_id += 1
        self.state = self.df.loc[self.month,
self.current_client_id][self.features].values
        return self.state, self.reward, done, False, {}

    def render(self, mode='human', close=False):
        pass
```

D.4 File: networks.py

This file contains all the policy networks developed as described in this thesis.

```
"""
Author: Eduardo Perez Denadai
Date: 2022-10-25
Reference:
1. https://stable-baselines3.readthedocs.io/en/v2.1.0/guide/custom_p
"""

import torch as th
import torch.nn as nn
from stable_baselines3.common.torch_layers import BaseFeaturesExtractor
from torch import Tensor
from gymnasium.spaces import Space


class GRUNetwork(BaseFeaturesExtractor):
    """
    GRU-based feature extractor for reinforcement learning agents.

    Args:
        observation_space (Space): The observation space of the environment.
        first_layer (int, optional): The size of the first layer. Defaults to 300.
        output_layer (int, optional): The size of the output feature dimension. Defaults to 200.
    """

    def __init__(self,
                 observation_space: Space,
                 first_layer: int = 300,
                 first_layer_dropout: float = 0.2,
                 output_layer: int = 100,
                 output_layer_dropout: float = 0.2
                 ):

        super(GRUNetwork, self).__init__(observation_space, output_layer)

        self.pre_gru = nn.Sequential(
            nn.Linear(observation_space.shape[0], first_layer, bias=False),
            nn.BatchNorm1d(first_layer),
            nn.RReLU(lower=0.1, upper=0.5),
            nn.Dropout(p=first_layer_dropout)
        )

        self.gru = nn.GRU(first_layer,
                          output_layer,
                          batch_first=True,
                          bias=False)

        self.post_gru = nn.Sequential(
            nn.Linear(output_layer, output_layer),
            nn.BatchNorm1d(output_layer),
            nn.RReLU(lower=0.1, upper=0.5),
            nn.Dropout(p=output_layer_dropout)
        )

    def forward(self, observations: Tensor) -> Tensor:
        """
        Forward pass through the GRU-based feature extractor.

        Args:
            observations (Tensor): Input observations.

        Returns:
            Tensor: Extracted features.
        """
        x = self.pre_gru(observations)
        x = x.unsqueeze(1)
        x, _ = self.gru(x)
        x = x.squeeze(1)
        x = self.post_gru(x)
        return x


class GRUNetworkBidirectional(BaseFeaturesExtractor):
    def __init__(self,
                 observation_space: Space,
                 first_layer: int = 300,
                 first_layer_dropout: float = 0.2,
                 gru_hidden_layer: int = 50,  # Half of output_layer size for bidirectional
                 output_layer: int = 100,
                 output_layer_dropout: float = 0.2
                 ):
```

```
        super(GRUNetworkBidirectional, self).__init__(observation_space, output_layer)

        self.pre_gru = nn.Sequential(
            nn.Linear(observation_space.shape[0], first_layer, bias=False),
            nn.BatchNorm1d(first_layer),
            nn.RReLU(lower=0.1, upper=0.5),
            nn.Dropout(p=first_layer_dropout)
        )

        # Note that the number of features for each direction in the GRU is half the output_layer size
        self.gru = nn.GRU(first_layer,
                          gru_hidden_layer,  # Half of output_layer size for bidirectional
                          batch_first=True,
                          bidirectional=True,  # Make the GRU bidirectional
                          bias=False)

        # The output will be twice the gru_hidden_layer because it's bidirectional
        self.post_gru = nn.Sequential(
            nn.Linear(2 * gru_hidden_layer, output_layer),  # Adjust the input size
            nn.BatchNorm1d(output_layer),
            nn.RReLU(lower=0.1, upper=0.5),
            nn.Dropout(p=output_layer_dropout)
        )

    def forward(self, observations: Tensor) -> Tensor:
        x = self.pre_gru(observations)
        x = x.unsqueeze(1)
        x, _ = self.gru(x)
        x = x.squeeze(1)
        x = self.post_gru(x)
        return x



class LSTMNetwork(BaseFeaturesExtractor):
    """
    LSTM-based feature extractor for reinforcement learning agents.

    Args:
        observation_space (Space): The observation space of the environment.
        first_layer (int, optional): The size of the first layer. Defaults to 300.
        output_layer (int, optional): The size of the output feature dimension. Defaults to 200.
    """

    def __init__(self,
                 observation_space: Space,
                 first_layer: int = 300,
                 first_layer_dropout: float = 0.2,
                 output_layer: int = 100,
                 output_layer_dropout: float = 0.2
                 ):

        super(LSTMNetwork, self).__init__(observation_space, output_layer)

        self.pre_lstm = nn.Sequential(
            nn.Linear(observation_space.shape[0], first_layer, bias=False),
            nn.BatchNorm1d(first_layer),
            nn.RReLU(lower=0.1, upper=0.5),
            nn.Dropout(p=first_layer_dropout)
        )

        self.lstm = nn.LSTM(first_layer,
                            output_layer,
                            batch_first=True,
                            bias=False)

        self.post_lstm = nn.Sequential(
            nn.Linear(output_layer, output_layer),
            nn.BatchNorm1d(output_layer),
            nn.RReLU(lower=0.1, upper=0.5),
            nn.Dropout(p=output_layer_dropout)
        )

    def forward(self, observations: Tensor) -> Tensor:
        """
        Forward pass through the LSTM-based feature extractor.

        Args:
            observations (Tensor): Input observations.

        Returns:
```

```
            Tensor: Extracted features.
        """
        x = self.pre_lstm(observations)
        x = x.unsqueeze(1)
        x, _ = self.lstm(x)
        x = x.squeeze(1)
        x = self.post_lstm(x)
        return x


class FCNetwork(BaseFeaturesExtractor):
    """
    Fully connected (FC) network-based feature extractor for reinforcement learning agents.

    Args:
        observation_space (Space): The observation space of the environment.
        first_layer (int, optional): The size of the first layer. Defaults to 300.
        output_layer (int, optional): The size of the output feature dimension. Defaults to 200.
    """

    def __init__(self,
                 observation_space: Space,
                 first_layer: int = 300,
                 first_layer_dropout: float = 0.2,
                 output_layer: int = 100,
                 output_layer_dropout: float = 0.1
                 ):

        super(FCNetwork, self).__init__(observation_space, output_layer)

        self.net = nn.Sequential(
            nn.Linear(observation_space.shape[0], first_layer),
            nn.BatchNorm1d(first_layer),
            nn.RReLU(lower=0.1, upper=0.5),
            nn.Dropout(p=first_layer_dropout),
            nn.Linear(first_layer, output_layer),
            nn.RReLU(lower=0.1, upper=0.5),
            nn.Dropout(p=output_layer_dropout),
        )

    def forward(self, observations: Tensor) -> Tensor:
        """
        Forward pass through the FC network-based feature extractor.

        Args:
            observations (Tensor): Input observations.

        Returns:
            Tensor: Extracted features.
        """
        return self.net(observations)


class ConvNetwork(BaseFeaturesExtractor):
    """
    Convolutional neural network-based feature extractor for reinforcement learning agents.

    Args:
        observation_space (Space): The observation space of the environment.
        first_layer (int, optional): The size of the first layer. Defaults to 300.
        output_layer (int, optional): The size of the output feature dimension. Defaults to 200.
    """

    def __init__(self,
                 observation_space: Space,
                 first_layer: int = 300,
                 first_layer_dropout: float = 0.2,
                 output_layer: int = 100,
                 output_layer_dropout: float = 0.1
                 ):

        super(ConvNetwork, self).__init__(observation_space, output_layer)

        self.pre_layers = nn.Sequential(
            nn.Linear(observation_space.shape[0], first_layer, bias=True),
            nn.BatchNorm1d(first_layer),
            nn.RReLU(lower=0.1, upper=0.5),
            nn.Dropout(p=first_layer_dropout)
        )

        self.conv = nn.Conv1d(in_channels=first_layer, out_channels=output_layer, kernel_size=3, stride=1, padding=1)
```

```
        self.post_layers = nn.Sequential(
            nn.Linear(output_layer, output_layer, bias=True),
            nn.BatchNorm1d(output_layer),
            nn.RReLU(lower=0.1, upper=0.5),
            nn.Dropout(p=output_layer_dropout)
        )

    def forward(self, observations: Tensor) -> Tensor:
        """
        Forward pass through the convolutional neural network-based feature extractor.

        Args:
            observations (Tensor): Input observations.

        Returns:
            Tensor: Extracted features.
        """
        x = self.pre_layers(observations)
        x = x.unsqueeze(2)
        x = self.conv(x)
        x = x.squeeze(2)
        x = self.post_layers(x)
        return x
```

D.5 File: doubledqn.py

This file contains the developed code for the double DQN agent as described in this thesis.

```
"""
Author: Eduardo Perez Denadai
Date: 2022-10-25
Reference:
1. https://stable-baselines3.readthedocs.io/en/v2.1.0/_modules/stable

This file contains a Double DQN implementation build on top of the stable-baselines3 DQN.

"""

import warnings
from typing import Any, ClassVar, Dict, List, Optional, Tuple, Type, TypeVar, Union

import numpy as np
import torch as th
from gymnasium import spaces
from torch.nn import functional as F

from stable_baselines3.common.buffers import ReplayBuffer
from stable_baselines3.common.off_policy_algorithm import OffPolicyAlgorithm
from stable_baselines3.common.policies import BasePolicy
from stable_baselines3.common.type_aliases import GymEnv, MaybeCallback, Schedule
from stable_baselines3.common.utils import get_linear_fn, get_parameters_by_name, polyak_update
from stable_baselines3.dqn.policies import CnnPolicy, DQNPolicy, MlpPolicy, MultiInputPolicy, QNetwork

SelfDQN = TypeVar("SelfDQN", bound="DQN")

class DoubleDQN(OffPolicyAlgorithm):
    policy_aliases: ClassVar[Dict[str, Type[BasePolicy]]] = {
        "MlpPolicy": MlpPolicy,
        "CnnPolicy": CnnPolicy,
        "MultiInputPolicy": MultiInputPolicy,
    }
    exploration_schedule: Schedule
    q_net: QNetwork
    q_net_target: QNetwork
    policy: DQNPolicy

    def __init__(
        self,
        policy: Union[str, Type[DQNPolicy]],
        env: Union[GymEnv, str],
        learning_rate: Union[float, Schedule] = 1e-4,
        buffer_size: int = 1_000_000,  # 1e6
        learning_starts: int = 50000,
        batch_size: int = 32,
        tau: float = 1.0,
        gamma: float = 0.99,
        train_freq: Union[int, Tuple[int, str]] = 4,
        gradient_steps: int = 1,
        replay_buffer_class: Optional[Type[ReplayBuffer]] = None,
        replay_buffer_kwargs: Optional[Dict[str, Any]] = None,
        optimize_memory_usage: bool = False,
        target_update_interval: int = 10000,
        exploration_fraction: float = 0.1,
        exploration_initial_eps: float = 1.0,
        exploration_final_eps: float = 0.05,
        max_grad_norm: float = 10,
        stats_window_size: int = 100,
        tensorboard_log: Optional[str] = None,
        policy_kwargs: Optional[Dict[str, Any]] = None,
        verbose: int = 0,
        seed: Optional[int] = None,
        device: Union[th.device, str] = "auto",
        _init_setup_model: bool = True,
    ) -> None:
        super().__init__(
            policy,
            env,
            learning_rate,
            buffer_size,
            learning_starts,
            batch_size,
            tau,
            gamma,
            train_freq,
            gradient_steps,
            action_noise=None,
            replay_buffer_class=replay_buffer_class,
            replay_buffer_kwargs=replay_buffer_kwargs,
            policy_kwargs=policy_kwargs,
```

```
            stats_window_size=stats_window_size,
            tensorboard_log=tensorboard_log,
            verbose=verbose,
            device=device,
            seed=seed,
            sde_support=False,
            optimize_memory_usage=optimize_memory_usage,
            supported_action_spaces=(spaces.Discrete,),
            support_multi_env=True,
        )

        self.exploration_initial_eps = exploration_initial_eps
        self.exploration_final_eps = exploration_final_eps
        self.exploration_fraction = exploration_fraction
        self.target_update_interval = target_update_interval
        self._n_calls = 0
        self.max_grad_norm = max_grad_norm

        self.exploration_rate = 0.0

        if _init_setup_model:
            self._setup_model()

    def _setup_model(self) -> None:
        super()._setup_model()
        self._create_aliases()
        self.batch_norm_stats = get_parameters_by_name(self.q_net, ["running_"])
        self.batch_norm_stats_target = get_parameters_by_name(self.q_net_target, ["running_"])
        self.exploration_schedule = get_linear_fn(
            self.exploration_initial_eps,
            self.exploration_final_eps,
            self.exploration_fraction,
        )

        if self.n_envs > 1:
            if self.n_envs > self.target_update_interval:
                warnings.warn(
                    "The number of environments used is greater than the target network "
                    f"update interval ({self.n_envs} > {self.target_update_interval}), "
                    "therefore the target network will be updated after each call to env.step() "
                    f"which corresponds to {self.n_envs} steps."
                )
```

```
    def _create_aliases(self) -> None:
        self.q_net = self.policy.q_net
        self.q_net_target = self.policy.q_net_target

    def _on_step(self) -> None:
        self._n_calls += 1
        if self._n_calls % max(self.target_update_interval // self.n_envs, 1) == 0:
            polyak_update(self.q_net.parameters(), self.q_net_target.parameters(), self.tau)
            polyak_update(self.batch_norm_stats, self.batch_norm_stats_target, 1.0)

        self.exploration_rate = self.exploration_schedule(self._current_progress_remaining)
        self.logger.record("rollout/exploration_rate", self.exploration_rate)

    def train(self, gradient_steps: int, batch_size: int = 100) -> None:
        self.policy.set_training_mode(True)
        self._update_learning_rate(self.policy.optimizer)

        losses = []
        for _ in range(gradient_steps):
            replay_data = self.replay_buffer.sample(batch_size, env=self._vec_normalize_env)

            with th.no_grad():
                next_actions = self.q_net(replay_data.next_observations).max(dim=1)[1].unsqueeze(-1)
                next_q_values = self.q_net_target(replay_data.next_observations).gather(1, next_actions)
                target_q_values = replay_data.rewards + (1 - replay_data.dones) * self.gamma * next_q_values

            current_q_values = self.q_net(replay_data.observations)
            current_q_values = th.gather(current_q_values, dim=1, index=replay_data.actions.long())

            loss = F.smooth_l1_loss(current_q_values, target_q_values)
            losses.append(loss.item())

            self.policy.optimizer.zero_grad()
```

```
            loss.backward()
            th.nn.utils.clip_grad_norm_(self.policy.parameters(),
    self.max_grad_norm)
            self.policy.optimizer.step()

        self._n_updates += gradient_steps
        self.logger.record("train/n_updates", self._n_updates,
    exclude="tensorboard")
        self.logger.record("train/loss", np.mean(losses))

    def predict(
        self,
        observation: Union[np.ndarray, Dict[str, np.ndarray]],
        state: Optional[Tuple[np.ndarray, ...]] = None,
        episode_start: Optional[np.ndarray] = None,
        deterministic: bool = False,
    ) -> Tuple[np.ndarray, Optional[Tuple[np.ndarray, ...]]]:
        if not deterministic and np.random.rand() <
    self.exploration_rate:
            if self.policy.is_vectorized_observation(observation):
                if isinstance(observation, dict):
                    n_batch =
    observation[next(iter(observation.keys()))].shape[0]
                else:
                    n_batch = observation.shape[0]
                action = np.array([self.action_space.sample() for
    _ in range(n_batch)])
            else:
                action = np.array(self.action_space.sample())
        else:
            action, state = self.policy.predict(observation,
    state, episode_start, deterministic)
        return action, state

    def learn(
        self: SelfDQN,
        total_timesteps: int,
        callback: MaybeCallback = None,
        log_interval: int = 4,
        tb_log_name: str = "DQN",
        reset_num_timesteps: bool = True,
        progress_bar: bool = False,
    ) -> SelfDQN:
        return super().learn(
            total_timesteps=total_timesteps,
            callback=callback,
            log_interval=log_interval,
            tb_log_name=tb_log_name,
            reset_num_timesteps=reset_num_timesteps,
            progress_bar=progress_bar,
        )

    def _excluded_save_params(self) -> List[str]:
        return [*super()._excluded_save_params(), "q_net",
    "q_net_target"]

    def _get_torch_save_params(self) -> Tuple[List[str],
    List[str]]:
        state_dicts = ["policy", "policy.optimizer"]

        return state_dicts, []
```

D.6 File: duelingdqn.py

This file contains the developed code for the Dueling DQN agent as described in this thesis.

```python
import numpy as np
import torch as th
import warnings
from typing import Any, Dict, List, Optional, Type, Union

from gymnasium import spaces
from stable_baselines3.common.buffers import ReplayBuffer
from stable_baselines3.common.off_policy_algorithm import OffPolicyAlgorithm
from stable_baselines3.dqn.policies import DQNPolicy, QNetwork, BasePolicy
from stable_baselines3.common.torch_layers import (
    BaseFeaturesExtractor, create_mlp)
from stable_baselines3.common.type_aliases import (
    GymEnv, MaybeCallback, Schedule)
from stable_baselines3.common.utils import (
    get_linear_fn, get_parameters_by_name, polyak_update)

class DuelingQNetwork(QNetwork):
    """
    Dueling Q-Network architecture.
    """

    def __init__(self,
                 observation_space: spaces.Space,
                 action_space: spaces.Discrete,
                 features_extractor: BaseFeaturesExtractor,
                 features_dim: int,
                 net_arch: Optional[List[int]] = None,
                 activation_fn: Type[th.nn.Module] = th.nn.ReLU,
                 normalize_images: bool = True):
        super(DuelingQNetwork, self).__init__(
            observation_space, action_space, features_extractor,
            features_dim, net_arch, activation_fn,
    normalize_images)

        # Define the advantage and value streams
        self.advantage = create_mlp(
            features_dim, action_space.n, net_arch, activation_fn)
        self.value = create_mlp(
            features_dim, 1, net_arch, activation_fn)

        # Replace the original q_net with the dueling architecture
        self.q_net = th.nn.ModuleList([
            th.nn.Sequential(*self.advantage),
            th.nn.Sequential(*self.value)])

    def forward(self, obs: th.Tensor) -> th.Tensor:
        features = self.extract_features(obs,
    self.features_extractor)

        # Compute advantage and value streams
        advantage = self.q_net[0](features)
        value = self.q_net[1](features)

        # Combine the streams
        return value + advantage - advantage.mean(dim=1,
    keepdim=True)

class DuelingDQNPolicy(DQNPolicy):
    """
    Policy class with a Dueling Q-Network for Dueling DQN.
    """

    def make_q_net(self) -> DuelingQNetwork:
        # Make sure we always have separate networks
        net_args = self._update_features_extractor(
            self.net_args, features_extractor=None)
        return DuelingQNetwork(**net_args).to(self.device)
```

D.7 File: common.py

This file contains common utility functions for plotting, aggregation, import data, and more.

```python
import os
import glob
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from stable_baselines3.common.logger import configure
from stable_baselines3.common.monitor import Monitor
from sklearn.metrics import (mean_squared_error, r2_score)



def get_csv_files(folder_path):
    """
    Get a list of all CSV files in the given folder.

    Args:
        folder_path (str): The path to the folder where the CSV files are located.

    Returns:
        list: A list of file paths to the CSV files.
    """
    csv_files = []
    for file_path in glob.glob(folder_path + '/**/*.csv', recursive=True):
        csv_files.append(file_path)
    return csv_files

def get_dataset(path):
    """
    Reads a dataset from a CSV file located at the given `path` and performs the following operations:

    - Sets the column `date_code` as the index of the DataFrame.
    - Encodes the `client_id` column by mapping each unique client ID to a corresponding integer index.
    - Converts all columns, except `date`, to the `float32` data type.

    Parameters:
        path (str): The file path of the CSV dataset.

    Returns:
        pandas.DataFrame: The dataset with the above transformations applied.
    """
    df = pd.read_csv(path, index_col="date_code")
    df["client_id"] = df["client_id"].map({client_id: i for i, client_id in enumerate(df["client_id"].unique())})
    df = df.astype({col: "float32" for col in df.columns if col != "date"})

    df.reset_index(drop=False, inplace=True)
    return df


def create_logger(log_path):
    """
    Creates a logger at the specified log path.

    Args:
        log_path (str): The path where the log files will be created.

    Returns:
        logger: The logger object that is created.
    """
    os.makedirs(log_path, exist_ok=True)
    logger = configure(log_path, ["csv"])
    return logger


def predict_from_env(model, env, n_steps):
    obs = env.reset()
    rew = []
    act = []
    for step in range(n_steps):
        action, _ = model.predict(obs, deterministic=True)
        obs, reward, done, info = env.step(action)
        rew.append(reward[0])
        act.append(action[0])
        if step % 100 == 0:
            print(f"action: {action} - reward : {reward} - done: {done}")
    return rew, act


def predict_from_frame(model, features):
```

```
    """
    Predict Actions in Regular binary environment
    """
    actions = []
    date_codes = []
    for step in features.iterrows():
        date, _ = step[0]
        obs = step[1]
        date_codes.append(date)
        action, _ = model.predict(obs, deterministic=True)
        actions.append(float(action))
    return actions, date_codes

def predict_from_frame_v2(model, features, default_rate=0.0, penalty=0.0):
    """
    Predict actions in multitarget enviornment
    """
    actions = []
    date_codes = []
    for step in features.iterrows():
        date, _ = step[0]
        obs = step[1]
        obs = np.append(obs, [default_rate, penalty])  # Append default_rate and penalty to each observation
        obs = obs.reshape(1, -1)  # Reshape to have a batch dimension
        date_codes.append(date)
        action, _ = model.predict(obs, deterministic=True)
        actions.append(float(action))
    return actions, date_codes


def running_mean_last_n_samples(data, n_samples):

    running_means = []

    for i in range(n_samples - 1, len(data)):
        last_n_samples = data[i - n_samples + 1 : i + 1]  # Extract the last n_samples samples
        mean = np.mean(last_n_samples)  # Calculate the mean
        running_means.append(mean)

    return np.array(running_means)


def running_mean_per_class(data, n_samples, n_classes):
    # Initialize an array to hold the running means for each class
    running_means = np.zeros((n_classes, len(data) - n_samples + 1))

    # Iterate over each class and calculate the running mean
    for class_index in range(n_classes):
        class_data = np.where(np.array(data) == class_index, 1, 0)
        for i in range(n_samples - 1, len(data)):
            last_n_samples = class_data[i - n_samples + 1: i + 1]
    # Extract the last n_samples for this class
            running_means[class_index, i - n_samples + 1] = np.mean(last_n_samples)

    return running_means



def plot_running_means_with_palette(running_means, title='Running Means', show_legend=True):
    """
    Plots running means for each class using a color palette suitable for any number of classes.
    This function now plots on the current active figure, allowing multiple calls to plot on the same graph.

    Args:
        running_means (numpy.ndarray): Array of running means where each row corresponds to a class.
        title (str): Title for the plot.
        show_legend (bool): Whether to show the legend or not. Defaults to True.
    """
    # Use a colormap to generate colors for plotting.
    colormap = plt.cm.viridis
    colors = [colormap(i) for i in np.linspace(0, 1, running_means.shape[0])]

    for i in range(running_means.shape[0]):
        cumsum = np.cumsum(running_means[i, :])
        cumavg = cumsum / np.arange(1, len(running_means[i, :]) + 1)
        plt.plot(cumavg, color=colors[i], linestyle="dashed", label=f"Class {i}" if show_legend else "_nolegend_")

    plt.title(title)
    plt.xlabel('Time Step')
    plt.ylabel('Cumulative Average')
```

```
    if show_legend:
        plt.legend()



def VecEnvMonitor(env_id, log_dir, env, data, **kwargs):
    """
    Monitor wrapper for vectorized gym environments.
    Writes to different log files per environment.

    Returns:
        Monitor: The initialized Monitor object.

    """
    def _init():
        environment = env(data, debug=True, scaled_features=True,
    accepts_discrete_action=True, **kwargs)
        log_file = os.path.join(log_dir, f"{env_id}.monitor.csv")
        return Monitor(environment, filename=log_file)
    return _init




def plot_default_rate_history(date_codes, actual, actions,
    dates_from_codes, title, save_path=None):
    """
    Generate a plot of the default rate history.

    Parameters:
        date_codes (list): A list of date codes.
        actual (list): A list of actual default rate values.
        actions (list): A list of predicted default rate values.
        dates_from_codes (dict): A dictionary mapping date codes
    to formatted dates.
        title (str): The title of the plot.

    Returns:
        None
    """
    result = pd.DataFrame({
        "dates": date_codes,
        "actual": actual,
        "pred": actions
    }).groupby("dates").mean()

    # Map index to the keys of date_code_mapping and format dates
    result.index = result.index.map(dates_from_codes)
    result.index = pd.to_datetime(result.index, format="%Y-%m-%d")

    plt.figure(figsize=(10, 5))
    plt.plot(result.actual, color="red", linestyle="-",
    label="actual")
    plt.plot(result.pred, color="gray", linestyle="--",
    label="predicted")

    # Set titles and labels
    plt.suptitle(title, fontsize=10)
    plt.title(f"RMSE : {np.sqrt(mean_squared_error(result.actual,
    result.pred)):1.5f} - R2 : {r2_score(result.actual,
    result.pred):1.5f}", fontsize=10)
    plt.ylabel("Reward")
    plt.xlabel("date")
    plt.legend()
    if save_path:
        plt.savefig(save_path)
    plt.show()

def plot_reward_history(reward, title,  window_size=10,
    save_path=None):
    """
    Plots the reward history for a given data set.

    Args:
        data (pandas.DataFrame): The data set containing the
    reward history.
        reward (numpy.ndarray): The array of rewards.
        title (str): The title of the plot.

    Returns:
        None
    """
    run_mean_rew = running_mean_last_n_samples(reward, window_size)

    plt.figure(figsize=(10, 5))
    plt.plot(run_mean_rew, linestyle="dashed", color="gray",
    label="reward")
    plt.plot((np.cumsum(run_mean_rew) / np.arange(1,
    len(run_mean_rew) + 1)),
             color="red", linestyle="-", label="running mean")
    plt.title(title, fontsize=10)
    plt.legend()
    if save_path:
        plt.savefig(save_path)
```

```
    plt.show()



def plot_actions_running_mean(actions, title, window_size=10, save_path=None):
    """
    Generate a running mean plot of actions.

    Parameters:
        data (pandas.DataFrame): The data containing the actions.
        actions (pandas.Series): The actions to plot.
        title (str): The title of the plot.

    Returns:
        None

    Raises:
        None
    """
    run_mean_act = running_mean_last_n_samples(actions, window_size)

    plt.figure(figsize=(10, 5))
    plt.plot(run_mean_act, label="Actions Running Mean", color="gray", linestyle="dashed")
    plt.plot(np.cumsum(run_mean_act) / np.arange(1, len(run_mean_act) + 1),
             color="red", linestyle="-", label="Cumulative Mean")
    plt.title(title, fontsize=10)
    plt.legend()
    if save_path:
        plt.savefig(save_path)
    plt.show()



def exponential_decay_schedule(initial_lr: float = 1e-1, decay_rate: float = 0.99, decay_steps: int = 1000):
    """
    Exponential decay learning rate schedule.
    :param initial_lr: (float) Initial learning rate.
    :param decay_rate: (float) Rate of decay.
    :param decay_steps: (int) Number of steps for decay.
    :return: (callable)
    """
    def func(progress_remaining: float):
        current_step = (1 - progress_remaining) * decay_steps
        return initial_lr * (decay_rate ** current_step)

    return func
```

D.8 File: constants.py

This file contains definitions kept for replication of the dataset used.

```
train_client_sublist = [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0,
    8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0,
    18.0, 19.0, 20.0, 21.0, 22.0, 23.0, 24.0, 25.0, 26.0, 27.0,
    28.0, 29.0, 30.0, 31.0, 32.0, 33.0, 34.0, 35.0, 36.0, 37.0,
    38.0, 39.0, 40.0, 41.0, 42.0, 43.0, 44.0, 45.0, 46.0, 47.0,
    48.0, 49.0, 50.0, 51.0, 52.0, 53.0, 54.0, 55.0, 56.0, 57.0,
    58.0, 59.0, 60.0, 61.0, 62.0, 63.0, 64.0, 65.0, 66.0, 67.0,
    68.0, 69.0, 70.0, 71.0, 72.0, 73.0, 74.0, 75.0, 76.0, 77.0,
    78.0, 79.0, 80.0, 81.0, 82.0, 83.0, 84.0, 85.0, 86.0, 87.0,
    88.0, 89.0, 90.0, 91.0, 92.0, 93.0, 94.0, 95.0, 96.0, 97.0,
    98.0, 99.0, 100.0, 101.0, 102.0, 103.0, 104.0, 105.0, 106.0,
    107.0, 108.0, 109.0, 110.0, 111.0, 112.0, 113.0, 114.0, 115.0,
    116.0, 117.0, 118.0, 119.0, 120.0, 121.0, 122.0, 123.0, 124.0,
    125.0, 126.0, 127.0, 128.0, 129.0, 130.0, 131.0, 132.0, 133.0,
    134.0, 135.0, 136.0, 137.0, 138.0, 139.0, 140.0, 141.0, 142.0,
    143.0, 144.0, 145.0, 146.0, 147.0, 148.0, 149.0, 150.0, 151.0,
    152.0, 153.0, 154.0, 155.0, 156.0, 157.0, 158.0, 159.0, 160.0,
    161.0, 162.0, 163.0, 164.0, 165.0, 166.0, 167.0, 168.0, 169.0,
    170.0, 171.0, 172.0, 173.0, 174.0, 175.0, 176.0, 177.0, 178.0,
    179.0, 180.0, 181.0, 182.0, 183.0, 184.0, 185.0, 186.0, 187.0,
    188.0, 189.0, 190.0, 191.0, 192.0, 193.0, 194.0, 195.0, 196.0,
    197.0, 198.0, 199.0, 200.0, 201.0, 202.0, 203.0, 204.0, 205.0,
    206.0, 207.0, 208.0, 209.0, 210.0, 211.0, 212.0, 213.0, 214.0,
    215.0, 216.0, 217.0, 218.0, 219.0, 220.0, 221.0, 222.0, 223.0,
    224.0, 225.0, 226.0, 227.0, 228.0, 229.0, 230.0, 231.0, 232.0,
    233.0, 234.0, 235.0, 236.0, 237.0, 238.0, 239.0, 240.0, 241.0,
    242.0, 243.0, 244.0, 245.0, 246.0, 247.0, 248.0, 249.0, 250.0,
    251.0, 252.0, 253.0, 254.0, 255.0, 256.0, 257.0, 258.0, 259.0,
    260.0, 261.0, 262.0, 263.0, 264.0, 265.0, 266.0, 267.0, 268.0,
    269.0, 270.0, 271.0, 272.0, 273.0, 274.0, 275.0, 276.0, 277.0,
    278.0, 279.0, 280.0, 281.0, 282.0, 283.0, 284.0, 285.0, 286.0,
    287.0, 288.0, 289.0, 290.0, 291.0, 292.0, 293.0, 294.0, 295.0,
    296.0, 297.0, 298.0, 299.0, 300.0, 301.0, 302.0, 303.0, 304.0,
    305.0, 306.0, 307.0, 308.0, 309.0, 310.0, 311.0, 312.0, 313.0,
    314.0, 315.0, 316.0, 317.0, 318.0, 319.0, 320.0, 321.0, 322.0,
    323.0, 324.0, 325.0, 326.0, 327.0, 328.0, 329.0, 330.0, 331.0,
    332.0, 333.0, 334.0, 335.0, 336.0, 337.0, 338.0, 339.0, 340.0,
    341.0, 342.0, 343.0, 344.0, 345.0, 346.0, 347.0, 348.0, 349.0,
    350.0, 351.0, 352.0, 353.0, 354.0, 355.0, 356.0, 357.0, 358.0,
    359.0, 360.0, 361.0, 362.0, 363.0, 364.0, 365.0, 366.0, 367.0,
    368.0, 369.0, 370.0, 371.0, 372.0, 373.0, 374.0, 375.0, 376.0,
    377.0, 378.0, 379.0, 380.0, 381.0, 382.0, 383.0, 384.0, 385.0,
    386.0, 387.0, 388.0, 389.0, 390.0, 391.0, 392.0, 393.0, 394.0,
    395.0, 396.0, 397.0, 398.0, 399.0, 400.0, 401.0, 402.0, 403.0,
    404.0, 405.0, 406.0, 407.0, 408.0, 409.0, 410.0, 411.0, 412.0,
    413.0, 414.0, 415.0, 416.0, 417.0, 418.0, 419.0, 420.0, 421.0,
    422.0, 423.0, 424.0, 425.0, 426.0, 427.0, 428.0, 429.0, 430.0,
    431.0, 432.0, 433.0, 434.0, 435.0, 436.0, 437.0, 438.0, 439.0,
    440.0, 441.0, 442.0, 443.0, 444.0, 445.0, 446.0, 447.0, 448.0,
    449.0, 450.0, 451.0, 452.0, 453.0, 454.0, 455.0, 456.0, 457.0,
    458.0, 459.0, 460.0, 461.0, 462.0, 463.0, 464.0, 465.0, 466.0,
    467.0, 468.0, 469.0, 470.0, 471.0, 472.0, 473.0, 474.0, 475.0,
    476.0, 477.0, 478.0, 479.0, 480.0, 481.0, 482.0, 483.0, 484.0,
    485.0, 486.0, 487.0, 488.0, 489.0, 490.0, 491.0, 492.0, 493.0,
    494.0, 495.0, 496.0, 497.0, 498.0, 499.0, 500.0, 501.0, 502.0,
    503.0, 504.0, 505.0, 506.0, 507.0, 508.0, 509.0, 510.0, 511.0,
    512.0, 513.0, 514.0, 515.0, 516.0, 517.0, 518.0, 519.0, 520.0,
    521.0, 522.0, 523.0, 524.0, 525.0, 526.0, 527.0, 528.0, 529.0,
    530.0, 531.0, 532.0, 533.0, 534.0, 535.0, 536.0, 537.0, 538.0,
    539.0, 540.0, 541.0, 542.0, 543.0, 544.0, 545.0, 546.0, 547.0,
    548.0, 549.0, 550.0, 551.0, 552.0, 553.0, 554.0, 555.0, 556.0,
    557.0, 558.0, 559.0, 560.0, 561.0, 562.0, 563.0, 564.0, 565.0,
    566.0, 567.0, 568.0, 569.0, 570.0, 571.0, 572.0, 573.0, 574.0,
    575.0, 576.0, 577.0, 578.0, 579.0, 580.0, 581.0, 582.0, 583.0,
    584.0, 585.0, 586.0, 587.0, 588.0, 589.0, 590.0, 591.0, 592.0,
    593.0, 594.0, 595.0, 596.0, 597.0, 598.0, 599.0, 600.0, 601.0,
    602.0]
test_client_sublist = [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0,
    8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0,
    18.0, 19.0, 20.0, 21.0, 22.0, 23.0, 24.0, 25.0, 26.0, 27.0,
    28.0, 29.0, 30.0, 31.0, 32.0, 33.0, 34.0, 35.0, 36.0, 37.0,
    38.0, 39.0, 40.0, 41.0, 42.0, 43.0, 44.0, 45.0, 46.0, 47.0,
    48.0, 49.0, 50.0, 51.0, 52.0, 53.0, 54.0, 55.0, 56.0, 57.0,
    58.0, 59.0, 60.0, 61.0, 62.0, 63.0, 64.0, 65.0, 66.0, 67.0,
    68.0, 69.0, 70.0, 71.0, 72.0, 73.0, 74.0, 75.0, 76.0, 77.0,
    78.0, 79.0, 80.0, 81.0, 82.0, 83.0, 84.0, 85.0, 86.0, 87.0,
    88.0, 89.0, 90.0, 91.0, 92.0, 93.0, 94.0, 95.0, 96.0, 97.0,
    98.0, 99.0, 100.0, 101.0, 102.0, 103.0, 104.0, 105.0, 106.0,
    107.0, 108.0, 109.0, 110.0, 111.0, 112.0, 113.0, 114.0, 115.0,
    116.0, 117.0, 118.0, 119.0, 120.0, 121.0, 122.0, 123.0, 124.0,
    125.0, 126.0, 127.0, 128.0, 129.0, 130.0, 131.0, 132.0, 133.0,
    134.0, 135.0, 136.0, 137.0, 138.0, 139.0, 140.0, 141.0, 142.0,
    143.0, 144.0, 145.0, 146.0, 147.0, 148.0, 149.0, 150.0, 151.0,
    152.0, 153.0, 154.0, 155.0, 156.0, 157.0, 158.0, 159.0, 160.0,
    161.0, 162.0, 163.0, 164.0, 165.0, 166.0, 167.0, 168.0, 169.0,
    170.0, 171.0, 172.0, 173.0, 174.0, 175.0, 176.0, 177.0, 178.0,
    179.0, 180.0, 181.0, 182.0, 183.0, 184.0, 185.0, 186.0, 187.0,
    188.0, 189.0, 190.0, 191.0, 192.0, 193.0, 194.0, 195.0, 196.0,
    197.0, 198.0, 199.0, 200.0, 201.0, 202.0, 203.0, 204.0, 205.0,
    206.0, 207.0, 208.0, 209.0, 210.0, 211.0, 212.0, 213.0, 214.0,
```

```
    215.0, 216.0, 217.0, 218.0, 219.0, 220.0, 221.0, 222.0, 223.0,
    224.0, 225.0, 226.0, 227.0, 228.0, 229.0, 230.0, 231.0, 232.0,
    233.0, 234.0, 235.0, 236.0, 237.0, 238.0, 239.0, 240.0, 241.0,
    242.0, 243.0, 244.0, 245.0, 246.0, 247.0, 248.0, 249.0, 250.0,
    251.0, 252.0, 253.0, 254.0, 255.0, 256.0, 257.0, 258.0, 259.0,
    260.0, 261.0, 262.0, 263.0, 264.0, 265.0, 266.0, 267.0, 268.0,
    269.0, 270.0, 271.0, 272.0, 273.0, 274.0, 275.0, 276.0, 277.0,
    278.0, 279.0, 280.0, 281.0, 282.0, 283.0, 284.0, 285.0, 286.0,
    287.0, 288.0, 289.0, 290.0, 291.0, 292.0, 293.0, 294.0, 295.0,
    296.0, 297.0, 298.0, 299.0, 300.0, 301.0, 302.0, 303.0, 304.0,
    305.0, 306.0, 307.0, 308.0, 309.0, 310.0, 311.0, 312.0, 313.0,
    314.0, 315.0, 316.0, 317.0, 318.0, 319.0, 320.0, 321.0, 322.0,
    323.0, 324.0, 325.0, 326.0, 327.0, 328.0, 329.0, 330.0, 331.0,
    332.0, 333.0, 334.0, 335.0, 336.0, 337.0, 338.0, 339.0, 340.0,
    341.0, 342.0, 343.0, 344.0, 345.0, 346.0, 347.0, 348.0, 349.0,
    350.0, 351.0, 352.0, 353.0, 354.0, 355.0, 356.0, 357.0, 358.0,
    359.0, 360.0, 361.0, 362.0, 363.0, 364.0, 365.0, 366.0, 367.0,
    368.0, 369.0, 370.0, 371.0, 372.0, 373.0, 374.0, 375.0, 376.0,
    377.0, 378.0, 379.0, 380.0, 381.0, 382.0, 383.0, 384.0, 385.0,
    386.0, 387.0, 388.0, 389.0, 390.0, 391.0, 392.0, 393.0, 394.0,
    395.0, 396.0, 397.0, 398.0, 399.0, 400.0, 401.0, 402.0, 403.0,
    404.0, 405.0, 406.0, 407.0, 408.0, 409.0, 410.0, 411.0, 412.0,
    413.0, 414.0, 415.0, 416.0, 417.0, 418.0, 419.0, 420.0, 421.0,
    422.0, 423.0, 424.0, 425.0, 426.0, 427.0, 428.0, 429.0, 430.0,
    431.0, 432.0, 433.0, 434.0, 435.0, 436.0, 437.0, 438.0, 439.0,
    440.0, 441.0, 442.0, 443.0, 444.0, 445.0, 446.0, 447.0, 448.0,
    449.0, 450.0, 451.0, 452.0, 453.0, 454.0, 455.0, 456.0, 457.0,
    458.0, 459.0, 460.0, 461.0, 462.0, 463.0, 464.0, 465.0, 466.0,
    467.0, 468.0, 469.0, 470.0, 471.0, 472.0, 473.0, 474.0, 475.0,
    476.0, 477.0, 478.0, 479.0, 480.0, 481.0, 482.0, 483.0, 484.0,
    485.0, 486.0, 487.0, 488.0, 489.0, 490.0, 491.0, 492.0, 493.0,
    494.0, 495.0, 496.0, 497.0, 498.0, 499.0, 500.0, 501.0, 502.0,
    503.0, 504.0, 505.0, 506.0, 507.0, 508.0, 509.0, 510.0, 511.0,
    512.0, 513.0, 514.0, 515.0, 516.0, 517.0, 518.0, 519.0, 520.0,
    521.0, 522.0, 523.0, 524.0, 525.0, 526.0, 527.0, 528.0, 529.0,
    530.0, 531.0, 532.0, 533.0, 534.0, 535.0, 536.0, 537.0, 538.0,
    539.0, 540.0, 541.0, 542.0, 543.0, 544.0, 545.0, 546.0, 547.0,
    548.0, 549.0, 550.0, 551.0, 552.0, 553.0, 554.0, 555.0, 556.0,
    557.0, 558.0, 559.0, 560.0, 561.0, 562.0, 563.0, 564.0, 565.0,
    566.0, 567.0, 568.0, 569.0, 570.0, 571.0, 572.0, 573.0, 574.0,
    575.0, 576.0, 577.0, 578.0, 579.0, 580.0, 581.0, 582.0, 583.0,
    584.0, 585.0, 586.0, 587.0, 588.0, 589.0, 590.0, 591.0, 592.0,
    593.0, 594.0, 595.0, 596.0, 597.0, 598.0, 599.0, 600.0, 601.0,
    602.0]
col_legibility_mapping = {'mispay_days': 'x_1', 'date_code':
    'x_2', 'mispay_d_90d_12m': 'x_3', 'total_balance': 'x_4',
    'client_id': 'x_5', 'income': 'x_6', 'age': 'x_7', 'mortgage':
    'x_8', 'mortgage_qty': 'x_9', 'maturity_months': 'x_10',
    'monthly_payment': 'x_11', 'subsidy': 'x_12', 'interest_rate':
    'x_13', 'default_120_12m': 'x_14'}
date_code_mapping = {'2020-03-31': 0, '2020-04-30': 1,
    '2020-05-31': 2, '2020-06-30': 3, '2020-07-31': 4,
    '2020-08-31': 5, '2020-09-30': 6, '2020-10-31': 7,
    '2020-11-30': 8, '2020-12-31': 9, '2021-01-31': 10,
    '2021-02-28': 11, '2021-03-31': 12, '2021-04-30': 13,
    '2021-05-31': 14, '2021-06-30': 15, '2021-07-31': 16,
    '2021-08-31': 17, '2021-09-30': 18, '2021-10-31': 19,
    '2021-11-30': 20, '2021-12-31': 21, '2022-01-31': 22,
    '2022-02-28': 23, '2022-03-31': 24, '2022-04-30': 25,
    '2022-05-31': 26, '2022-06-30': 27, '2022-07-31': 28,
    '2022-08-31': 29, '2022-09-30': 30, '2022-10-31': 31,
    '2022-11-30': 32, '2022-12-31': 33}
```

D.9 File: constants.py

This file contains definitions kept for replication of the dataset used.

```
train_client_sublist = [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0,
    8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0,
    18.0, 19.0, 20.0, 21.0, 22.0, 23.0, 24.0, 25.0, 26.0, 27.0,
    28.0, 29.0, 30.0, 31.0, 32.0, 33.0, 34.0, 35.0, 36.0, 37.0,
    38.0, 39.0, 40.0, 41.0, 42.0, 43.0, 44.0, 45.0, 46.0, 47.0,
    48.0, 49.0, 50.0, 51.0, 52.0, 53.0, 54.0, 55.0, 56.0, 57.0,
    58.0, 59.0, 60.0, 61.0, 62.0, 63.0, 64.0, 65.0, 66.0, 67.0,
    68.0, 69.0, 70.0, 71.0, 72.0, 73.0, 74.0, 75.0, 76.0, 77.0,
    78.0, 79.0, 80.0, 81.0, 82.0, 83.0, 84.0, 85.0, 86.0, 87.0,
    88.0, 89.0, 90.0, 91.0, 92.0, 93.0, 94.0, 95.0, 96.0, 97.0,
    98.0, 99.0, 100.0, 101.0, 102.0, 103.0, 104.0, 105.0, 106.0,
    107.0, 108.0, 109.0, 110.0, 111.0, 112.0, 113.0, 114.0, 115.0,
    116.0, 117.0, 118.0, 119.0, 120.0, 121.0, 122.0, 123.0, 124.0,
    125.0, 126.0, 127.0, 128.0, 129.0, 130.0, 131.0, 132.0, 133.0,
    134.0, 135.0, 136.0, 137.0, 138.0, 139.0, 140.0, 141.0, 142.0,
    143.0, 144.0, 145.0, 146.0, 147.0, 148.0, 149.0, 150.0, 151.0,
    152.0, 153.0, 154.0, 155.0, 156.0, 157.0, 158.0, 159.0, 160.0,
    161.0, 162.0, 163.0, 164.0, 165.0, 166.0, 167.0, 168.0, 169.0,
    170.0, 171.0, 172.0, 173.0, 174.0, 175.0, 176.0, 177.0, 178.0,
    179.0, 180.0, 181.0, 182.0, 183.0, 184.0, 185.0, 186.0, 187.0,
    188.0, 189.0, 190.0, 191.0, 192.0, 193.0, 194.0, 195.0, 196.0,
    197.0, 198.0, 199.0, 200.0, 201.0, 202.0, 203.0, 204.0, 205.0,
    206.0, 207.0, 208.0, 209.0, 210.0, 211.0, 212.0, 213.0, 214.0,
    215.0, 216.0, 217.0, 218.0, 219.0, 220.0, 221.0, 222.0, 223.0,
    224.0, 225.0, 226.0, 227.0, 228.0, 229.0, 230.0, 231.0, 232.0,
    233.0, 234.0, 235.0, 236.0, 237.0, 238.0, 239.0, 240.0, 241.0,
    242.0, 243.0, 244.0, 245.0, 246.0, 247.0, 248.0, 249.0, 250.0,
    251.0, 252.0, 253.0, 254.0, 255.0, 256.0, 257.0, 258.0, 259.0,
    260.0, 261.0, 262.0, 263.0, 264.0, 265.0, 266.0, 267.0, 268.0,
    269.0, 270.0, 271.0, 272.0, 273.0, 274.0, 275.0, 276.0, 277.0,
    278.0, 279.0, 280.0, 281.0, 282.0, 283.0, 284.0, 285.0, 286.0,
    287.0, 288.0, 289.0, 290.0, 291.0, 292.0, 293.0, 294.0, 295.0,
    296.0, 297.0, 298.0, 299.0, 300.0, 301.0, 302.0, 303.0, 304.0,
    305.0, 306.0, 307.0, 308.0, 309.0, 310.0, 311.0, 312.0, 313.0,
    314.0, 315.0, 316.0, 317.0, 318.0, 319.0, 320.0, 321.0, 322.0,
    323.0, 324.0, 325.0, 326.0, 327.0, 328.0, 329.0, 330.0, 331.0,
    332.0, 333.0, 334.0, 335.0, 336.0, 337.0, 338.0, 339.0, 340.0,
    341.0, 342.0, 343.0, 344.0, 345.0, 346.0, 347.0, 348.0, 349.0,
    350.0, 351.0, 352.0, 353.0, 354.0, 355.0, 356.0, 357.0, 358.0,
    359.0, 360.0, 361.0, 362.0, 363.0, 364.0, 365.0, 366.0, 367.0,
    368.0, 369.0, 370.0, 371.0, 372.0, 373.0, 374.0, 375.0, 376.0,
    377.0, 378.0, 379.0, 380.0, 381.0, 382.0, 383.0, 384.0, 385.0,
    386.0, 387.0, 388.0, 389.0, 390.0, 391.0, 392.0, 393.0, 394.0,
    395.0, 396.0, 397.0, 398.0, 399.0, 400.0, 401.0, 402.0, 403.0,
    404.0, 405.0, 406.0, 407.0, 408.0, 409.0, 410.0, 411.0, 412.0,
    413.0, 414.0, 415.0, 416.0, 417.0, 418.0, 419.0, 420.0, 421.0,
    422.0, 423.0, 424.0, 425.0, 426.0, 427.0, 428.0, 429.0, 430.0,
    431.0, 432.0, 433.0, 434.0, 435.0, 436.0, 437.0, 438.0, 439.0,
    440.0, 441.0, 442.0, 443.0, 444.0, 445.0, 446.0, 447.0, 448.0,
    449.0, 450.0, 451.0, 452.0, 453.0, 454.0, 455.0, 456.0, 457.0,
    458.0, 459.0, 460.0, 461.0, 462.0, 463.0, 464.0, 465.0, 466.0,
    467.0, 468.0, 469.0, 470.0, 471.0, 472.0, 473.0, 474.0, 475.0,
    476.0, 477.0, 478.0, 479.0, 480.0, 481.0, 482.0, 483.0, 484.0,
    485.0, 486.0, 487.0, 488.0, 489.0, 490.0, 491.0, 492.0, 493.0,
    494.0, 495.0, 496.0, 497.0, 498.0, 499.0, 500.0, 501.0, 502.0,
    503.0, 504.0, 505.0, 506.0, 507.0, 508.0, 509.0, 510.0, 511.0,
    512.0, 513.0, 514.0, 515.0, 516.0, 517.0, 518.0, 519.0, 520.0,
    521.0, 522.0, 523.0, 524.0, 525.0, 526.0, 527.0, 528.0, 529.0,
    530.0, 531.0, 532.0, 533.0, 534.0, 535.0, 536.0, 537.0, 538.0,
    539.0, 540.0, 541.0, 542.0, 543.0, 544.0, 545.0, 546.0, 547.0,
    548.0, 549.0, 550.0, 551.0, 552.0, 553.0, 554.0, 555.0, 556.0,
    557.0, 558.0, 559.0, 560.0, 561.0, 562.0, 563.0, 564.0, 565.0,
    566.0, 567.0, 568.0, 569.0, 570.0, 571.0, 572.0, 573.0, 574.0,
    575.0, 576.0, 577.0, 578.0, 579.0, 580.0, 581.0, 582.0, 583.0,
    584.0, 585.0, 586.0, 587.0, 588.0, 589.0, 590.0, 591.0, 592.0,
    593.0, 594.0, 595.0, 596.0, 597.0, 598.0, 599.0, 600.0, 601.0,
    602.0]
test_client_sublist = [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0,
    8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0,
    18.0, 19.0, 20.0, 21.0, 22.0, 23.0, 24.0, 25.0, 26.0, 27.0,
    28.0, 29.0, 30.0, 31.0, 32.0, 33.0, 34.0, 35.0, 36.0, 37.0,
    38.0, 39.0, 40.0, 41.0, 42.0, 43.0, 44.0, 45.0, 46.0, 47.0,
    48.0, 49.0, 50.0, 51.0, 52.0, 53.0, 54.0, 55.0, 56.0, 57.0,
    58.0, 59.0, 60.0, 61.0, 62.0, 63.0, 64.0, 65.0, 66.0, 67.0,
    68.0, 69.0, 70.0, 71.0, 72.0, 73.0, 74.0, 75.0, 76.0, 77.0,
    78.0, 79.0, 80.0, 81.0, 82.0, 83.0, 84.0, 85.0, 86.0, 87.0,
    88.0, 89.0, 90.0, 91.0, 92.0, 93.0, 94.0, 95.0, 96.0, 97.0,
    98.0, 99.0, 100.0, 101.0, 102.0, 103.0, 104.0, 105.0, 106.0,
    107.0, 108.0, 109.0, 110.0, 111.0, 112.0, 113.0, 114.0, 115.0,
    116.0, 117.0, 118.0, 119.0, 120.0, 121.0, 122.0, 123.0, 124.0,
    125.0, 126.0, 127.0, 128.0, 129.0, 130.0, 131.0, 132.0, 133.0,
    134.0, 135.0, 136.0, 137.0, 138.0, 139.0, 140.0, 141.0, 142.0,
    143.0, 144.0, 145.0, 146.0, 147.0, 148.0, 149.0, 150.0, 151.0,
    152.0, 153.0, 154.0, 155.0, 156.0, 157.0, 158.0, 159.0, 160.0,
    161.0, 162.0, 163.0, 164.0, 165.0, 166.0, 167.0, 168.0, 169.0,
    170.0, 171.0, 172.0, 173.0, 174.0, 175.0, 176.0, 177.0, 178.0,
    179.0, 180.0, 181.0, 182.0, 183.0, 184.0, 185.0, 186.0, 187.0,
    188.0, 189.0, 190.0, 191.0, 192.0, 193.0, 194.0, 195.0, 196.0,
    197.0, 198.0, 199.0, 200.0, 201.0, 202.0, 203.0, 204.0, 205.0,
    206.0, 207.0, 208.0, 209.0, 210.0, 211.0, 212.0, 213.0, 214.0,
```

```
    215.0, 216.0, 217.0, 218.0, 219.0, 220.0, 221.0, 222.0, 223.0,
    224.0, 225.0, 226.0, 227.0, 228.0, 229.0, 230.0, 231.0, 232.0,
    233.0, 234.0, 235.0, 236.0, 237.0, 238.0, 239.0, 240.0, 241.0,
    242.0, 243.0, 244.0, 245.0, 246.0, 247.0, 248.0, 249.0, 250.0,
    251.0, 252.0, 253.0, 254.0, 255.0, 256.0, 257.0, 258.0, 259.0,
    260.0, 261.0, 262.0, 263.0, 264.0, 265.0, 266.0, 267.0, 268.0,
    269.0, 270.0, 271.0, 272.0, 273.0, 274.0, 275.0, 276.0, 277.0,
    278.0, 279.0, 280.0, 281.0, 282.0, 283.0, 284.0, 285.0, 286.0,
    287.0, 288.0, 289.0, 290.0, 291.0, 292.0, 293.0, 294.0, 295.0,
    296.0, 297.0, 298.0, 299.0, 300.0, 301.0, 302.0, 303.0, 304.0,
    305.0, 306.0, 307.0, 308.0, 309.0, 310.0, 311.0, 312.0, 313.0,
    314.0, 315.0, 316.0, 317.0, 318.0, 319.0, 320.0, 321.0, 322.0,
    323.0, 324.0, 325.0, 326.0, 327.0, 328.0, 329.0, 330.0, 331.0,
    332.0, 333.0, 334.0, 335.0, 336.0, 337.0, 338.0, 339.0, 340.0,
    341.0, 342.0, 343.0, 344.0, 345.0, 346.0, 347.0, 348.0, 349.0,
    350.0, 351.0, 352.0, 353.0, 354.0, 355.0, 356.0, 357.0, 358.0,
    359.0, 360.0, 361.0, 362.0, 363.0, 364.0, 365.0, 366.0, 367.0,
    368.0, 369.0, 370.0, 371.0, 372.0, 373.0, 374.0, 375.0, 376.0,
    377.0, 378.0, 379.0, 380.0, 381.0, 382.0, 383.0, 384.0, 385.0,
    386.0, 387.0, 388.0, 389.0, 390.0, 391.0, 392.0, 393.0, 394.0,
    395.0, 396.0, 397.0, 398.0, 399.0, 400.0, 401.0, 402.0, 403.0,
    404.0, 405.0, 406.0, 407.0, 408.0, 409.0, 410.0, 411.0, 412.0,
    413.0, 414.0, 415.0, 416.0, 417.0, 418.0, 419.0, 420.0, 421.0,
    422.0, 423.0, 424.0, 425.0, 426.0, 427.0, 428.0, 429.0, 430.0,
    431.0, 432.0, 433.0, 434.0, 435.0, 436.0, 437.0, 438.0, 439.0,
    440.0, 441.0, 442.0, 443.0, 444.0, 445.0, 446.0, 447.0, 448.0,
    449.0, 450.0, 451.0, 452.0, 453.0, 454.0, 455.0, 456.0, 457.0,
    458.0, 459.0, 460.0, 461.0, 462.0, 463.0, 464.0, 465.0, 466.0,
    467.0, 468.0, 469.0, 470.0, 471.0, 472.0, 473.0, 474.0, 475.0,
    476.0, 477.0, 478.0, 479.0, 480.0, 481.0, 482.0, 483.0, 484.0,
    485.0, 486.0, 487.0, 488.0, 489.0, 490.0, 491.0, 492.0, 493.0,
    494.0, 495.0, 496.0, 497.0, 498.0, 499.0, 500.0, 501.0, 502.0,
    503.0, 504.0, 505.0, 506.0, 507.0, 508.0, 509.0, 510.0, 511.0,
    512.0, 513.0, 514.0, 515.0, 516.0, 517.0, 518.0, 519.0, 520.0,
    521.0, 522.0, 523.0, 524.0, 525.0, 526.0, 527.0, 528.0, 529.0,
    530.0, 531.0, 532.0, 533.0, 534.0, 535.0, 536.0, 537.0, 538.0,
    539.0, 540.0, 541.0, 542.0, 543.0, 544.0, 545.0, 546.0, 547.0,
    548.0, 549.0, 550.0, 551.0, 552.0, 553.0, 554.0, 555.0, 556.0,
    557.0, 558.0, 559.0, 560.0, 561.0, 562.0, 563.0, 564.0, 565.0,
    566.0, 567.0, 568.0, 569.0, 570.0, 571.0, 572.0, 573.0, 574.0,
    575.0, 576.0, 577.0, 578.0, 579.0, 580.0, 581.0, 582.0, 583.0,
    584.0, 585.0, 586.0, 587.0, 588.0, 589.0, 590.0, 591.0, 592.0,
    593.0, 594.0, 595.0, 596.0, 597.0, 598.0, 599.0, 600.0, 601.0,
    602.0]
col_legibility_mapping = {'mispay_days': 'x_1', 'date_code':
    'x_2', 'mispay_d_90d_12m': 'x_3', 'total_balance': 'x_4',
    'client_id': 'x_5', 'income': 'x_6', 'age': 'x_7', 'mortgage':
    'x_8', 'mortgage_qty': 'x_9', 'maturity_months': 'x_10',
    'monthly_payment': 'x_11', 'subsidy': 'x_12', 'interest_rate':
    'x_13', 'default_120_12m': 'x_14'}
date_code_mapping = {'2020-03-31': 0, '2020-04-30': 1,
    '2020-05-31': 2, '2020-06-30': 3, '2020-07-31': 4,
    '2020-08-31': 5, '2020-09-30': 6, '2020-10-31': 7,
    '2020-11-30': 8, '2020-12-31': 9, '2021-01-31': 10,
    '2021-02-28': 11, '2021-03-31': 12, '2021-04-30': 13,
    '2021-05-31': 14, '2021-06-30': 15, '2021-07-31': 16,
    '2021-08-31': 17, '2021-09-30': 18, '2021-10-31': 19,
    '2021-11-30': 20, '2021-12-31': 21, '2022-01-31': 22,
    '2022-02-28': 23, '2022-03-31': 24, '2022-04-30': 25,
    '2022-05-31': 26, '2022-06-30': 27, '2022-07-31': 28,
    '2022-08-31': 29, '2022-09-30': 30, '2022-10-31': 31,
    '2022-11-30': 32, '2022-12-31': 33}
```