POLITECNICO DI MILANO

School of Industrial and Information Engineering

Master of Science in Mathematical Engineering

FX trading with Reinforcement Learning: an application of Fitted Q Iteration (FQI)

Supervisor: Prof. Marcello Restelli

Co-supervisor: Dott. Lorenzo Bisi
Dott. Nico Montali
Dott. Luca Sabbioni

Candidate:
Reho Gianmarco, matricola 863850

Academic Year 2018-2019

a mia madre e mio padre

Ringraziamenti

Questa tesi é frutto di un progetto avviato a Gennaio 2019. É stata una grande opportunitá di crescita, proposta dal Professore Marcello Restelli, al quale porgo un doveroso ringraziamento. La sua professionalitá e le sue linee guida sono state per me fondamentali.
Ringrazio particolarmente Andrea Tirinzoni per avermi illustrato il codice da utilizzare nelle prime fasi, Nico Montali i cui contributi e disponibilitá sono stati essenziali, Lorenzo Bisi e Luca Sabbioni per avermi seguito costantemente, permettendomi di perfezionare al meglio il mio lavoro.
Il progetto fa parte di una collaborazione con la societá AGS (Advanced Global Soluion SpA), della quale ringrazio Gianluca De Cola, Nicola Marino e Cristiana Corno.

Ringrazio infine la mia famiglia e i miei amici, sui quali ho potuto sempre contare.

Contents

Ringraziamenti **5**

Sommario **x**

Abstract **xii**

1 Introduction **1**
1.1 Outline of the Thesis 3

2 Reinforcement Learning and FQI **4**
2.1 Reinforcement Learning (RL) 4
2.2 Markov Decision Process (MDP) 6
2.2.1 Bellman Optimality Equation 8
2.3 Temporal difference and Q-learning 11
2.4 Fitted Q-Iteration 12
2.5 Extra-Trees 15

3 Related Works **17**
3.1 Adaptive Reinforcement Learning 17
3.2 Recurrent Reinforcement Learning 18
3.3 Genetic algorithms and Reinforcement Learning 19
3.4 Support Vector Machine Stock Market Forecasting 21
3.5 Q-Learning and Sharpe Ratio Maximization 22
3.6 Multiagent Q-learning Framework 22

4 Problem Formulation **24**
4.1 Original Data 24
4.2 FQI Data 25
4.3 Features 26

4.4 Fees 27
4.5 Reward 28
4.5.1 Reference Price 29
4.5.2 Reward Cases 30

5 Analysis of the dataset **32**
5.1 Original Dataset 32
5.1.1 Stationarity 34
5.2 FQI Dataset 41
5.2.1 Regressor and Classification Analysis 41
5.2.2 Feature importance Analysis 55

6 Experimental Results **63**
6.1 Programming Language 63
6.2 FQI Results 63
6.2.1 Train and Validation 63
6.2.2 Validation Results with 1 year of Train 65
6.2.3 Validation Results with 2 year of Train 68
6.2.4 FQI Test Results 73
6.2.5 FQI vs Buy&Hold Results 76
6.2.6 FQI vs FFNN Results 79

7 Conclusions and Future Work **81**
7.1 Conclusions 81
7.2 Future Work 82

References **84**

List of Figures

2.1 Agent-Environment interaction in RL 5

3.1 Adaptive Reinforcement learning layers 18
3.2 Genetic Trading process 20
3.3 Optimization by Reinforcement Learning - The most predictive indicators are taken from the GA in-sample module and fed to RL engine 21
3.4 Structure of Multiagent Q-learning Framework 23

4.1 Actions - Positions Combination 25
4.2 Bear-Bull Candlesticks: typical representation of price movements in trading (in this case they represent prices in a minute) 29
4.3 Flat to Sell 30
4.4 Flat to Buy & Sell to Buy Combinations 31

5.1 2014 - EURUSD open price: we can see a general bearish (downward) trend 32
5.2 2014 - EURUSD Histogram: data (in a range between 1.2 and 1.4) are not distributed like a Gaussian; many data are concentrated between 1.3 and 1.4 (on the right) 35
5.3 2014 - EURUSD Differences: $O_{t+1} - O_t \ \forall t$ where O_t is the open price in t 39
5.4 Cross Validation Scheme 42
5.5 Accuracy Score - 2015-2016 45
5.6 Confusion Matrix - 0.0001% *min split* - 2015-2016 46
5.7 Confusion Matrix - 0.001% *min split* - 2015-2016 46
5.8 Confusion Matrix - 0.01% *min split* - 2015-2016 47
5.9 Confusion Matrix - 0.1% *min split* - 2015-2016 47
5.10 Confusion Matrix - 1% *min split* - 2015-2016 48

5.11 Confusion Matrix - 10% *min split* - 2015-2016 48
5.12 Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 0.0001% *min split* 49
5.13 Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 0.001% *min split* 49
5.14 Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 0.01% *min split* 50
5.15 Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 0.1% *min split* 50
5.16 Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 1% *min split* 51
5.17 Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 10% *min split* 51
5.18 zoom Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 0.0001% *min split* 52
5.19 zoom Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 0.001% *min split* 52
5.20 zoom Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 0.01% *min split* 53
5.21 zoom Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 0.1% *min split* 53
5.22 zoom Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 1% *min split* 53
5.23 zoom Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 10% *min split* 54
5.24 Feature importance prices & Feature importance differences - 3 ms - 2018 59
5.25 Feature importance other features & R^2 Score - 3 ms - 2018: in the R^2 plot the dotted line is the value of the R^2 when we use all the features in the fitting 59
5.26 Feature importance prices & Feature importance differences - 29 ms - 2018 60
5.27 Feature importance other features & R^2 Score - 29 ms - 2018: in the R^2 plot the dotted line is the value of the R^2 when we use all the features in the fitting 60
5.28 Feature importance prices & Feature importance differences - 286 ms - 2018 61

5.29 Feature importance other features & R^2 Score - 286 ms - 2018: in the R^2 plot the dotted line is the value of the R^2 when we use all the features in the fitting 61

6.1 Performance evaluation scheme - Train 1y 64
6.2 Performance evaluation scheme - Train 2y 64
6.3 Max of average daily reward - Train: 2015 - Validation: 2014 - We took, for each *min split*, the average daily reward and select the best one 65
6.4 Max of average daily reward - Train: 2016 - Validation: 2015 - We took, for each *min split*, the average daily reward and select the best one 66
6.5 Max of average daily reward - Train: 2017 - Validation: 2016 - We took, for each *min split*, the average daily reward and select the best one 66
6.6 Average daily reward - Train: 2015 - Validation: 2014 - minsplit = 2854 (0,1%) 67
6.7 Average daily reward - Train: 2016 - Validation: 2015 - minsplit = 2854 (0,1%) 67
6.8 Average daily reward - Train: 2017 - Validation: 2016 - minsplit = 2854 (0,1%) 67
6.9 Max of average daily reward - Train: 2015-2016 - Validation: 2014 - We took, for each *min split*, the average daily reward and select the best one 68
6.10 Max of average daily reward - Train: 2016-2017 - Validation: 2015 - We took, for each *min split*, the average daily reward and select the best one 69
6.11 Average daily reward - Train: 2015-2016 - Validation: 2014 - minsplit = 5763 (0,1%) 69
6.12 Average daily reward - Train: 2016-2017 - Validation: 2015 - minsplit = 5763 (0,1%) 69
6.13 Max of average daily reward - Train and Test: 2015 - different Delay and Action Frequency - We took, for each *min split*, the average daily reward and select the best one 71
6.14 Max of average daily reward - Train: 2015 - Validation: 2014 - different Delay and Action Frequency - We took, for each *min split*, the average daily reward and select the best one 71

6.15 Train: 2015 - Validation: 2014 - Test: 2016 - minsplit = 2854 (0,1%) - FQI iteration = 10 - From the Actions we can see how there are similar trends in certain time slots; for most days of the year, at the beginning of the day the agent tends to buy, then remains flat for a few minutes, then sells and returns to buy at the end of the day. This behavior is typical in the FX market and it is known as **intraday seasonality**. The cumulative reward seems to be stable and growing, with a bit of variance after the middle of the year. We have a final cumulative return of almost 10% (10000 \$ over an invested capital of 100000 €(≃ 100000 \$)). 73
6.16 Train: 2016 - Validation: 2015 - Test: 2017 - minsplit = 2854 (0,1%) - FQI iteration = 5 - We can observe similar actions at the beginning of the day, for a few minutes (almost always flat), then it is not possible to recognize a particular pattern common to almost every days (as happened in the preiovus case in Figure 6.15). The cumulative reward has a strong growth in the first month, then remain almost stable and constant for the following month and then strongly grows until the penultimate month of the year; in the last month there is some variance without growth. We have a final cumulative return of almost 11% (11000 \$ over an invested capital of 100000 €(≃ 100000 \$)). 74
6.17 Train: 2017 - Validation: 2016 - Test: 2018 - minsplit = 2854 (0,1%) - FQI iteration = 10 - We can see a more intraday seasonality behavior. In the cumulative reward there is a bit of variance in the first three months and then a strong growth, with a final cumulative return of almost 14% (14000 \$ over an invested capital of 100000 €(≃ 100000 \$)). 74
6.18 Train: 2015-2016 - Validation: 2014 - Test: 2017 - minsplit = 5763 (0,1%) - FQI iteration = 20 - We can see a clear intraday seasonality behavior in the actions which tend not to change as often as in the case with train over 1 year. We can see a stable growth in the cumulative reward with a final cumulative return of almost 21% (21000 \$ over an invested capital of 100000 €(≃ 100000 \$)), which compared with the previous results with train over 1 year (10%, 11% and 14% final cumulative return) shows that the train over 2 year allows to obtain better performances in terms of final cumulative return. 75

6.19 Train: 2016-2017 - Validation: 2015 - Test: 2018 - minsplit = 5763 (0,1%) - FQI iteration = 20 - We can see a intraday seasonality behavior in the actions with some noise compared with the previous case in Figure 6.18. Trend of the cumulative reward is growing and more linear than in the previous case in Figure 6.18, reaching a final cumulative return of almost 20% (20000 $ over an invested capital of 100000 €($\simeq$ 100000 $)). 75
6.20 FQI vs daily Buy&Hold - Test: 2016 (Train 1y) - FQI outperforms daily B&H. We have a cumulative return of almost 10% with FQI and almost 0% with daily B&H. 76
6.21 FQI vs daily Buy&Hold - Test: 2017 (Train 1y) - FQI slightly underperforms B&H, even if the final cumulative returns are very close (almost 11% for FQI and almost 13% for daily B&H). 77
6.22 FQI vs daily Buy&Hold - Test: 2018 (Train 1y) - FQI outperforms daily B&H, which performs negatively since the second month of the year. 77
6.23 FQI vs daily Buy&Hold - Test: 2017 (Train 2y) - FQI outperforms daily B&H by almost 7% of cumulative returns. 78
6.24 FQI vs daily Buy&Hold - Test: 2018 (Train 2y) - FQI strongly outperforms daily B&H, which performs negatively since the second month of the year. 78

List of Tables

4.1 sample of 2018 €/$ data 24
4.2 FQI Features 26

5.1 Mean and Variance Splitting 35
5.2 ADF Test - open price 38
5.3 ADF Test - open price differences ($O_{t+1} - O_t \ \forall t$ where O_t is the open price in t) 40
5.4 *min split* values 44

6.1 optimal (*min split*, FQI iterations) pairs 70
6.2 mean and std daily reward - FQI vs daily Buy&Hold 79
6.3 Results (end of the year cumulative returns) - FQI vs FFNN 80

List of Algorithms

1 Fitted Q-Iteration 14

2 IVS(D, V^o): Iterative Variable Selection 56

Sommario

Questa tesi fa parte di un progetto sviluppato in collaborazione con l'azienda AGS SpA (Advanced Global Solution), con l'obiettivo di applicare tecniche di Reinforcement Learning al trading su Foreign Exchange (Forex - FX). Lo scopo di questa tesi é quello di applicare una tecnica di Reinforcement Learning, una branca del Machine Learning, al trading su FX, in particolare sulla coppia euro/dollaro (€/$) e valutarne le performance (in backtest).
Il Machine Learning si occupa dello studio e dell'implementazione di algoritmi che siano capaci di apprendere informazioni direttamente dai dati e fare previsioni su di essi: tali algoritmi superano il classico approccio del seguire un insieme di istruzioni statiche. Abbiamo formulato l'attivitá di trading come un processo decisionale di Markov e abbiamo applicato un algoritmo di Reinforcement Learning, chiamato Fitted Q Iteration (FQI), dove la reward é rappresentata come il profitto generato dalla strategia di trading adottata. I risultati sono stati comparati con quelli di una strategia classica nei mercati finanziari (daily Buy&Hold).

Parole Chiave: Reinforcement Learning, Fitted Q Iteration, Forex

Abstract

This thesis is part of a project developed in collaboration with the AGS SpA (Advanced Global Solution) company, with the aim of applying Reinforcement Learning techniques to Foreign Exchange (Forex - FX) trading. The main goal of this thesis is to apply a Reinforcement Learning technique, one of the three fields of Machine Learning, to FX trading, in particular to the €/$ pair and evaluate the performance (in backtest).
Machine Learning deals with the study and implementation of algorithms that are capable of learning information directly from the data and making predictions on them: these algorithms go beyond the classic approach of following a set of static instructions.
We formulated the trading activity as a Markov decision process and we applied a Reinforcement Learning algorithm, called Fitted Q Iteration (FQI), where the reward is represented as the profit generated by the adopted trading strategy. Results have been compared with those of a classic trading strategy (daily Buy&Hold).

Keywords: Reinforcement Learning, Fitted Q Iteration, Forex

Chapter 1

Introduction

The Foreign exchange (also known as Forex or FX) market is a global marketplace for exchanging national currencies against one another.
The FX market is the largest financial market in the world with a daily volume of $ 6.6 trillion, in contrast with $ 84 billion for equities worldwide, according to the 2019 Triennal Central Bank Survey of FX and OTC derivatives markets [23]. Having such a large trading volume can bring many advantages to traders. A high volume means that traders can typically get their orders executed more easily and closer to the prices they want. Having more liquidity at each pricing point allows traders to enter and exit the market more easily.
Market participants use FX trading mainly for hedging and/or speculative reasons. Trading over the past 20 years has evolved exponentially, in terms of technologies and volumes. It is now quite easy to obtain data even with very high time frequencies (even less than a second). This, together with the increase in the computational power of the machines, has helped the development of new algorithms and the application of machine learning techniques in this field.
The applications of machine learning techniques to the trading problem have been extensively studied and tested in renowned financial environments such as the FX market and the stock markets. The goal of these applications was to construct automated systems able to outperform the profits generated by human traders, and they often showed promising results. In general, machine learning approaches to trading can be applied to a single price series as the unique trading target, or to try to manage all the shares in the market, solving the related portfolio-optimization problems: which assets do we have to buy or sell? When should we buy or sell them? How much of our budget do we have to invest in each operation ? Much work has been done on price prediction. The prediction problem is often treated with supervised learning, modeling the relationship between the input/output pair [1].

Unfortunately, those approaches focus on minimizing the prediction error and usually do not provide an explicit policy for translating predictions into a long-term investment strategy in the broader context of the entire market.
Reinforcement Learning (RL) techniques try to overcome this liability by focusing on learning an effective policy under which the agent collects the maximal average reward from the environment. However, the definition of a proper model is in this case fundamental to obtain meaningful results. In general, the main challenge to machine learning trading is posed by the difficulties in the financial environment summarization and representation. Financial data usually contains a large amount of noise and is difficult to decompose in a set of relevant features. The technical analysis indicators proposed by quantitative finance, like moving averages or relative strength indexes, tried to mitigate these uncertainties by offering ways to extrapolate relevant information from the data. Nonetheless, if they are simply used without exploiting the properties of the specific environment, they might perform poorly.
Among the different RL techniques, we used the Fitted Q Iteration (FQI), a value-based batch mode RL algorithm which yields an approximation of the Q-function corresponding to an infinite horizon optimal control problem with discounted rewards, by iteratively extending the optimization horizon (see [6]).
Value-based approaches are very effective in capturing patterns by exploiting Markov property through Bellman equation. The main obstacles to a value-based approach are state that is non Markovian and a high number of actions. The data itself is not Markovian, but it is possible to make it Markovian using a lagged data window. In this context, the type of data (prices) allows us to generate all possible combinations of portfolios and actions, also considering a small number of actions. Having therefore a batch with all possible scenarios available, using a value-based batch approach such as FQI becomes the most natural choice.
Before starting with the FQI algorithm, we made some analyses on the original dataset and the dataset constructed for the FQI algorithm. In particular, we started with a qualitative analysis of the original dataset and then we carried out a stationarity analysis. We then analyzed the FQI dataset, varying a parameter of the regressor used in the FQI algorithm, looking at both regression and classification metrics, with Cross-Validation, to take into account a possible overfitting that takes place when the algorithm stores specific characteristics of the dataset on which it is trained, without learning the general properties of the probabilistic model underlying the dataset.
To conclude the analysis of the FQI dataset, we carried out an analysis of the importance of each feature, following an IVS (Iterative Variable Selection) approach [2].
After applying the FQI algorithm, creating the models in training, we did an accurate series of tests on the data to evaluate the proposed system (in terms of generated profit), splitting the dataset to create different simulations distributed over several pe-

riods: training period, validation period, and testing period.
The performances have also been evaluated considering a possible delay in the estimation of the trading prices (5, 10 seconds) and a different time discretization in the process (changing the temporal discretization of the MDP), in which decision epochs occur at different time intervals (1, 5, and 10 minutes), in order to understand the impact of these different time windows.
We obtained positive results, compared to those obtained with a classic daily Buy&Hold strategy and those obtained using a different ML technique (Feed Forward Neural Networks - FFNN). Furthermore, we noticed the intraday seasonality behavior, an interesting and typical behavior in the FX market.

1.1 Outline of the Thesis

Here is a brief outline of the content of this thesis.

- in Chapter 2 we will introduce the main characteristics of Reinforcement Learning and the FQI algorithm.
- Chapter 3 will show some related works in RL applied also to FX trading.
- in Chapter 4 we will present the problem formulation, in particular, we will describe how we created the FQI dataset.
- in Chapter 5 we will describe the analysis (with results) of the original dataset (qualitative analysis) and FQI dataset (regression and classification analysis and feature importances analysis).
- in Chapter 6 we will present and discuss the experimental results with FQI and the comparison with daily Buy&Hold strategy and FFNN technique.
- in Chapter 7 we will draw conclusions on the work done, suggesting possible future improvements.

Chapter 2

Reinforcement Learning and FQI

We used Reinforcement Learning (RL) techniques as a bridge to connect the model definition of price trends to an actual speculative strategy. Defining the investment environment as a Markov Decision Process we took advantage of **Fitted Q-Iteration (FQI)** algorithm [6]. Moreover, since the FQI algorithm requires a function approximator, we used a forest of Extremely Randomized Trees (**Extra-Trees**) as regressor.
In this chapter we will present the main preliminaries about Reinforcement Learning and FQI. In particular we will describe the main characteristics and definitions in Reinforcement Learning (following [24]). Then we will introduce Q-learning, the FQI algorithm and finally we will show an overview of the Extra-Trees regressor.

2.1 Reinforcement Learning (RL)

Reinforcement learning (RL) is an area of Machine Learning aiming to determine an optimal control policy from interaction with a system or from observations gathered from a system.
For example, when an infant plays, he can directly experience from his interaction with the environment. From this interaction he builds a wealth of information about cause and effect, about consequences of actions and about what to do in order to achieve goals. This is the key idea behind RL: there is an environment which represents the outside world to the agent and an agent who takes actions, receives observations from the environment that consists of a reward for his action and information of his new state. The reward informs the agent of how good or bad was the taken action and the observation tells him what is the next state in the environment. The agent tries to figure out the best actions to take or the optimal way to behave in the environment in order to carry out his task in the best possible way.
As we can see in Figure 2.1, the agent and the environment interact with each other

over a sequence of discrete time steps, $t = 0, 1, 2, 3, ...$ At each time step t, the agent receives some representation of the environment's state, $S_t \in \mathcal{S}$ (the state space), and on that basis selects an action $A_t \in \mathcal{A}$ (the action space).
One time step later, the agent receives a numerical reward $R_{t+1} \in \mathcal{R}$ and observes the new environment's state S_{t+1}.

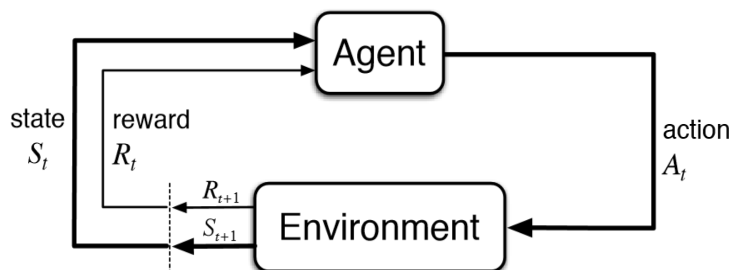


Figure 2.1: Agent-Environment interaction in RL

Unlike other ML approaches, in RL there is no supervisor, only a reward signal or a real number that tells the agent how good or bad was his action.
The success of its previous actions is evaluated to gradually refine which are the optimal actions to be taken in each situation, considering not only the immediate subsequent reward, but also the entire chain of potential future rewards openend by these actions. To do so, in the learning process the agent has to balance a trade-off between the **exploitation** of the actions defined, up to that moment, as the best available, and the **exploration** of suboptimal actions that could still lead to a greater cumulative reward in the future.
A reward R_t is a scalar feedback signal that indicates how well the agent is doing at time step t. The agent's job is to maximize the expected sum of discounted rewards.
How to represent the agent state ? In RL we use the Markovian states, which follow the so-called **Markov property**: in an informal way it states that the future is independent of the past given the present. In our case we will refer to the notion of **fully observable environments**, where the agent direcly observes the environment state and as a result, the observation emitted from the environment is the agent's new state as well as the environment's new state. This is a **Markov Decision Process (MDP)** and we will discuss this in the next section.

It is important to clarify what are a **policy** and a **value function** for an agent. A **policy** is a probability distribution over actions given states, i.e. the agent's behavior function or how the agent picks his actions given that it's in a certain state. It could be a deterministic policy or a stochastic policy,

$$\pi(a \mid s) = \mathbb{P}[A_t = a \mid S_t = s]$$

A **Value function** is a function that tells us how good is it to be in a particular state and how good is to take a particular action. It informs the agent of how much return to expect if it takes a particular action in a particular state.
It's a prediction of expected future returns used to evaluate goodness/badness of states, therefore enabling the agent to select between different actions,

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \mid S_t = s]$$

in the state s, at time step t, the value function informs the agent of the expected sum of future rewards on a given policy π; as to choose the right action that maximizes that expected sum of reward.
$\gamma \in [0,1]$ is a **discount factor** and it informs the agent of how much it should care about rewards now with respect to reward in the future. If $\gamma = 0$ the agent only cares about the first reward, if $\gamma = 1$ the agent cares about all future rewards.

This thesis is focused on **value based agent** and **batch mode** and **off-line** learning.
The **value based** agent will evaluate all the states in the state space and the policy will be kind of implicit.
In **batch mode** learning the learning agent is not directly interacting with the system but receives only a set of tuples and is asked to determine from this set a control policy which is as close as possible to an optimal policy.
In orer to solve RL problems we need data, which can be collected in two way: on-line or off-line. If the trajectories $(S_1, A_1, R_1, ...)$ are generated during the learning phase, we are in an online setting. If the agent is provided with a dataset, we are in an **off-line** setting, there is no control over how the data are generated. In this case, the dataset is usually in the form $D = \{(S_i, A_i, R_i, S_i')\}_{i=1}^n$ where $A_i \sim \pi_b(\cdot \mid S_i)$ and $\pi_b(\cdot \mid S_i)$ is called *behavioral* policy, the policy used to generate data, unknown to the agent. So in the off-line learning the agent will learn the optimal policy observing the actions chosen by another agent who follows a b*ehavioral* policy.

2.2 Markov Decision Process (MDP)

In order to explain the MDP it is important to show first the *Markov* property.
Remember what we said in the previous section, in an informal way the Markov property states that the future is independent of the past given the present. In more details, a stochastic process $S = (S_t : t \geq 0)$ has the *Markov* property, if and only if:

$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, ..., S_t]$$

so, the current state captures all relevant information from history.

For a Markov state s and a next state s', we can define the **state transition probability** function:

$$P_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$$

It is a probability distribution over next possible states, given the current state.

A **Markov process** is a tuple ($\mathcal{S}$, P) on state space $\mathcal{S}$ and transition function P.
The dynamics of the system can be defined by these two components.

A **Markov Reward process** si a tuple ($\mathcal{S}$, P, R, γ) where $\mathcal{S}$ is a finite state space, P is the state transition probability function and R is a reward function, with

$$R_s = \mathbb{E}[R_{t+1} \mid S_t = s]$$

R_s says how much immediate reward we expect to get from state s at the moment.

We can define the notion of the **return** G_t, which is the total discounted rewards from time step t

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where γ is the discount factor as we said in the previous section, with $\gamma \in [0,1]$.

The **state-value function** of a MRP is the expected return starting from state s

$$v(s) = \mathbb{E}[G_t \mid S_t = s]$$

Using **Bellman equation**, the value function is decomposed into two parts: an immediate reward R_{t+1} and a discounted value of the next state $\gamma v(S_{t+1})$.
We have:

$$\begin{aligned} v(s) =& \mathbb{E}[G_t \mid S_t = s] & (2.1) \\ =& \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \mid S_t = s] & (2.2) \\ =& \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s] & (2.3) \\ =& \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s] & (2.4) \end{aligned}$$

So for each state in the state space, the Bellman equation gives us the value of that state:

$$v(s) = R_s + \gamma \sum_{s' \in S} P_{ss'} v(s')$$

the value of the state s is the reward we get upon leaving that state, plus a discounted average over next possible next states, where the value of each possible next state is multiplied by the probability that we land in it.

A **Markov Decision Process (MDP)** is a Markov Reward process with decisions. In particular a MDP is a tuple ($\mathcal{S}$, $\mathcal{A}$, P, R, γ), where $\mathcal{S}$ is the state space, $\mathcal{A}$ is a finite set of actions, P is the state transition probability function

$$P^a_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$$

and R is the reward function

$$R^a_s = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$$

and γ∈[0,1] is the discount factor.
Thanks to the Markov property, from the definition of the state transition probability function we can observe that $\mathbb{P}[S_{t+1} \mid S_t, A_t] = \mathbb{P}[S_{t+1} \mid S_1, A_1, ..., S_t, A_t]$.
In the MDP policies depend on the current state, i.e. $A_t \sim \pi(\cdot \mid S_t)$, $\forall\ t > 0$.

Now we can define the state-value function and the action-value function of a MDP.
The **state-value function** $v_\pi(s)$ of a MDP is the expected return starting from state s, and following policy π:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right],\ for\ all\ s \in \mathcal{S} \tag{2.5}$$

and it tells us how good is it to be in state s if we are following policy π.
The **action-value function** (or **Q-function**) $Q_\pi(s, a)$ is the expected return starting from state s, taking action a and following policy π:

$$Q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right] \tag{2.6}$$

and it tells us how good is it to take a particular action from a particular state.
As we will see in the next section, if we know the Q-function, we can take the best actions by taking the maximum Q.

2.2.1 Bellman Optimality Equation

In this subsection we'll show how to get the optimal behavior in a MDP starting from the Bellman expectation equation.
Remembering Equation (2.4), we start in state s, following policy π and the value being

in that state is the immediate reward we get, added to the value of the next state, if we know we are going to follow the policy π from that state onwards.
In the similar way we can decompose the action-value function:

$$Q_\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma Q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] \tag{2.7}$$

Since we have multiple possible actions from one state s and the policy defines a probability distribution over those actions, we can write the **Bellman expectation equation**:

$$v_\pi(s) = \sum_{a \in A} \pi(a \mid s) Q_\pi(s,a) \tag{2.8}$$

After we took an action, we want to know, for each of the possible situations, what is the value of being in that situation following our policy onwards. So we can rewrite the Q-function as:

$$Q_\pi(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s') \tag{2.9}$$

From (2.8) and (2.9) we can rewrite the **Bellman expectation equation for** $v_\pi(s)$:

$$v_\pi(s) = \sum_{a \in A} \pi(a \mid s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s') \right) \tag{2.10}$$

From (2.9) and (2.10) we can write the **Bellman expectation equation for** $Q_\pi(s,a)$:

$$Q_\pi(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a' \mid s') Q_\pi(s', a') \tag{2.11}$$

Now that we have seen the Bellman expectation equations, we can move on with optimality, defining the optimal state-value function, the optimal action-value function, the optimal policy and showing how an optimal policy can be found.
The **optimal state-value function** $v_*(s)$ is the maximum value function over all policies:

$$v_*(s) \doteq \max_\pi v_\pi(s) \tag{2.12}$$

given the current state s.
The **optimal action-value function** $q_*(s,a)$ is the maximum action value function over all policies:

$$Q_*(s,a) \doteq \max_\pi Q_\pi(s,a) \tag{2.13}$$

given the current state s and action a.
The goal is to find an optimal policy. It is possible to define a partial ordering between policies by means of the value function:

$$\pi \geq \pi' \; if \; v_\pi(s) \geq v_{\pi'}(s), \; \forall s$$

one policy is better than another policy if the value function for that policy is greater than the value function of the other policy in all states.
For any MDP there exists an optimal policy π_* that is better than or equal to all other policies ($\pi_* \geq \pi$, $\forall\pi$) and it's possible to have more than one optimal policy.

A deterministic optimal policy can be found by maximizing over $Q_*(s,a)$, choosing the action that gives us the maximum $Q_*(s,a)$,

$$\pi_*(a \mid s) = \begin{cases} 1 & \text{if } a = \underset{a \in A}{\operatorname{argmax}}(Q_*(s,a)), \\ 0 & \text{otherwise.} \end{cases}$$

if we are in some state s, we choose with probability 1 the action a which maximizes $Q_*(s,a)$.
There is always a deterministic policy for any MDP and if we know $Q_*(s,a)$ we have the optimal policy.

The optimal value functions are recursively related by the Bellman optimality equation. **Bellman optimality equation for v_*:**

$$v_*(s) = \max_a Q_*(s,a) \tag{2.14}$$

instead of taking average like in Bellman expectation equation, we take the maximum of $Q_*(s,a)$. Following the same procedure as for the Bellman expectation equations, we will obtain:

$$Q_*(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s') \tag{2.15}$$

then the **Bellman optimality equation for $v_*(s)$:**

$$v_*(s) = \max_a R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s') \tag{2.16}$$

and the **Bellman optimality equation for $Q_*(s,a)$:**

$$Q_*(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_{a'} Q_*(s',a') \tag{2.17}$$

We can use algorithms of **Dynamic Programming (DP)** to compute optimal policies given a perfect model of the environment as a MDP. But classical DP algorithms are of limited utility in RL because of their assumption of a perfect model and because of their great computational expense. *Policy evalutaion* refers to the typically iterative computation of the value functions for a given policy. *Policy improvement* refers to the computation of an improved policy given the value function for that policy. Putting

these two computations together, we obtan *policy iteration* and *value iteration*, the two most popular DP methods. Either of these can be used to reliably compute optimal policies and value functions for finite MDPs given complete knowledge of the MDP.
In our case we have no complete knowledge of the MDP, in particular we don't know the transition probabilities. So, instead of using model-dependent RL algorithms as DP algorithms, we use model-free **Temporal Difference (TD)** Learning. TD algorithms can learn directly from raw experience without a model of the environment's dynamics.

2.3 Temporal difference and Q-learning

Temporal difference (TD) algorithms enable the agent to learn through every single action it takes. TD updates the knowledge of the agent on every timestep, following this rule:

$$New \leftarrow Old + \alpha(Target - Old) \tag{2.18}$$

where α is a learning-rate parameter ($0 < \alpha \leq 1$).
Q-learning is an off-policy TD algorithm that updates Q-values using the Q-value of the next state and the greedy action (by taking the max of Q over it).
How can an agent learn an optimal policy π_* for an arbitrary environment ? It is difficult to learn the function π_*: $\mathcal{S} \to \mathcal{A}$ directly. The only training information available to the learner is the sequence of immediate rewards $r(s_i, a_i)$ for i = 0,1,2,...
Learning the Q-function corresponds to learning the optimal policy. How can Q be learned ? The key problem is finding a reliable way to estimate training values for Q, given only a sequence of immediate rewards r spread out over time. This can be accomplished through iterative approximation.
The objective of the agent is to find the optimal policy for each state of the environment to maximize the long-run total reward. The Q-learning algorithm [25] uses optimal Q-values $Q_*(s, a)$ for states s and actions a. The optimal Q-values function satisfies Bellman's optimality Equation (2.17) in which we have the probability of a transition from state s to s' when action a is taken. Given the optimal Q-values $Q_*(s, a)$, it is possible to choose the best action:

$$a_* = \underset{a \in \mathcal{A}}{\text{argmax}}(Q_*(s, a))$$

The best advantage of using Q-learning is that there is no need to know the transition probabilities. The algorithm can find the $Q_*(s, a)$ in a recursive way.
From Equations (2.7) and (2.13) we can rewrite the Bellman optimality equation for $Q_*(s, a)$:

$$Q_*(s, a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q_*(S_{t+1}, a') \middle| S_t = s, A_t = a\right] \tag{2.19}$$

Therefore, TD Target in this case is

$$Target = R_{t+1} + \gamma \max_{a'} Q_*(S_{t+1}, a') \tag{2.20}$$

Before learning begins, Q is initialized to a possible arbitrary fixed value. Then, at each time t the agent takes an action A_t, observes a reward R_{t+1} and a new state S_{t+1} and Q is updated according to the following rule (following 2.18 and 2.20):

$$Q^{new}(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t) \right] \tag{2.21}$$

where α is the learning-rate and γ is the discount factor.
An episode of the algorithm ends when state S_{t+1} is a terminal state. If Equation (2.21) is repeatedly applied for each pair (s, a) and the learning rate α is gradually reduced toward 0 over time, then $Q(s, a)$ converges with probability 1 to $Q_*(s, a)$.

2.4 Fitted Q-Iteration

Inspired by the on-line Q-learning paradigm [25], [6] approached the batch mode learning problem by computing from the set of four-tuples an approximation of the so-called Q-function defined on the state-action space and by deriving from this latter function the control policy.
As we said in the previous sections, in a RL environment the experience gathered by the agent at the time instant t is represented by a set of four-values tuples (s_t, a_t, r_t, s_{t+1}), where s_t is the state in which the agent is at time t, a_t the action taken from that state, r_t the instantaneous reward collected from that action and s_{t+1} the next state reached, defining $\mathcal{S}$ and $\mathcal{A}$ respectively as the total state-space and action-space.
The so called Q-function $Q(s_t, a_t)$ defines the value of a state s_t and an action a_t at the time instant t, described as the expected utility of all the cumulative rewards collected starting from that state-action pair.
Approaches aimed to optimize the Q-function in an online way updating it as the agent explores the environment can be used in case of discrete state and action spaces. They scale poorly as the size of these spaces grows and are not suitable for continuous states and actions (see [6]).
Fitted Q-iteration is a batch mode learning algorithm that computes an approximation $\widehat{Q}^{(N)}(x, a)$ of the Q-function from the four-values tuples dataset obtained in the exploration of the environment. The algorithm was originally proposed in [5] and [6]. It needs a dataset $D = \{(s_i, a_i, r_i, s'_i)\}_{i=1}^n$; this is a set of n samples, representing the fact of having executed action a_i in state s_i, with the result of reaching a new state s'_i and getting a reward r_i. The Q-function approximation correspond to an infinite horizon optimal control problem with discounted rewards, solved by iteratively extending the

optimization horizon.
When the state and action spaces are finite and small enough, the Q-function can be represented in tabular form and its approximation as well as the control policy derivation are easy. When dealing with continuous or very large discrete state and action spaces, the Q-function cannot be represented anymore by a table with one entry for each state-action pair and in the context of RL an approximation of the Q-function all over the state-action space must be determined from finite and generally very sparse sets of four-tuples. To overcome this generalization problem, an interesting framework is the one used by ([22]) which applies the idea of fitted value iteration ([10]) to kernel-based reinforcement learning, and reformulates the Q-function determination problem as a sequence of kernel-based regression problems. This framework makes it possible to take full advantage in the context of reinforcement learning of the generalization capabilities of any regression algorithm and it represents the fitted Q iteration algorithm: it allows to fit (using a set of four-tuples) any (parametric or non-parametric) approximation architecture to the Q-function.
Following [5] at the first iteration the FQI algorithm produces an approximation of a Q_1-funcion corresponding to a 1-step optimization. Since the true Q_1-function is the conditional expectation of the instantaneous reward given the state-action pair, an approximation of it can be built by applying a batch mode regression algorithm to a training set whose inputs are the pairs (state, action) (s_t, a_t) and whose target output values are the instantaneous rewards r_t.
The Nth iteration derives (using a batch mode regression algorithm) an approximation of a Q_N-function corresponding to an N-step optimization horizon. The training set at this step is obtained by updating the output values of the training set of the previous step by using the value iteration based on the approximate Q_N-function returned at the previous step. To perform the learning of $\widehat{Q}^{(N)}(s, a)$ from the training set at the end of each iteration any regression algorithm could be use, e.g. linear regression, Gaussian Processes, or random forests. In Algorithm 1 we can see the pseudocode of the FQI algorithm.
Stopping conditions are required to decide for how many iterations N the process has to run. A commonly used way to stop the algorithm is to define a priori the number of iteratios to run, or setting a tolerance level on the sub-optimality of the approximated function and stop the process when the error is smalled than the tolerance. For a complete overview of the available stopping conditions and an analytic formulation of a sub-optimality error bound dependent on the number of iterations we remand to [6].

Algorithm 1 Fitted Q-Iteration

Initialization:

Batch of transitions D = $\{(s_t, a_t, r_t, s_{t+1})\}$

Training Set TS = $\emptyset$

Step N = 0

Initialize $\widehat{Q}^{(N)}$ as a function always equal to zero

on the state-action space $\mathcal{S} \times \mathcal{A}$

While (until stopping condition is reached):

N = N + 1

for each $(s_t, a_t, r_t, s_{t+1}) \in$ D **do**:

$$\hat{q}^i = r^i + \gamma \max_{a'} \widehat{Q}^{N-1}(s^i_{t+1}, a')$$

add $((s^i_t, a^i_t), \hat{q}^i)$ to TS

learn $\widehat{Q}^{(N)}(s, a)$ from TS

return $\widehat{Q}^{(N)}(s, a)$

2.5 Extra-Trees

Experiments in [6] showed that Extra-Trees is the supervised learning method able to extract at best information from sets of four-values tuples, making it the most suitable candidate to be used as regressor in FQI batch learning phase for a variety of applications.
Before describing the Extra-Tress we need to explain what are random forests and decision trees.

Random forests are an ensemble learning method for regression and classification; it constructs a multitude of decision trees at training time and outputting the class that is the mode of mean prediction (for regression) or the classes (for classification) of the individual trees. The term came from random decision forests that was first proposed by Tin Kam Ho in 1995 [14].
The basic element of a Random Forest is the **decision tree**. It is a flowchart-like structure made of nodes and branches. At each node, a split on the data is performed based on one of the input features, generating two or more branches as output. More splits are made in the upcoming nodes and increasing numbers of branches are generated to partition the original data. This continues until a node is generated where all or almost all of the data belong to the same class and further splits (or branches) are no longer possible. This process generates a tree-like structure. The first splitting node is called the *root* node. The end nodes are called *leaves*. The input of the decision tree is a vector of features and the outputs are a numerical value (for regression) or a label (for classification). Each internal node of the tree represents a test on an attribute, each branch represents the outcome of the test, leading to a new sub-tree and each leaf node represents the output.

Extremely Randomized Trees (**Extra-Trees**), is a tree-based ensemble methods designed to solve classification and regression problems.
These tree-based algorithms partition the input space into several regions, and can be used to determine constant predictions of input elements. To obtain the prediction, the region of the tree to which the input element belongs is checked, and the result is calculated as the average of the output values of the training set elements contained there.
Extra-Trees is an ensemble methods, since it builds a forest of different trees whose final prediction is obtained by averaging the predictions of each tree in the forest (see [18]).
Differently from others ensemble tree-based algorithms (see [19]), that use boot-strapping techniques to partition the training set in each tree construction, the Extra-Trees method [8] uses the **whole training set** to build each tree of the forest, but each time it splits its internal nodes choosing cut-points in a totally **random** way. The Extra-Trees splitting procedure has two parameters: K, which identifies the number of attributes of the

training set element randomly selected at each node and n_{min} (*min split*), which defines the minimum number of elements a node has to contain to be split. An additional parameter M defines the number of the generated trees in the forest.
At each node the generation algorithm selects K different attributes and for each one of them creates a possible split with a value randomly chosen between the attribute minimum and maximum value. To each of those splits is then assigned a split-score (based on different possible metrics) and then the best split is selected and associated with the node. This procedure is repeated for each node until the whole tree has been generated, i.e., all the nodes contain less then n_{min} samples from the training set.
One of the main advantages of Extra-Trees over classical regression tree methods is that it is able to strongly reduce the model variance thanks to the combined action of the explicit node randomization and the general forest ensemble averaging.

Chapter 3

Related Works

At the best of our knowledge, we are the first to try to apply FQI to Foreign Exchange (Forex - FX) trading.
We can find various ML approaches aimed to the solution of trading problems with applications to the FX market and the financial stock market.
Here we show some developments in ML algorithmic trading strategies.

3.1 Adaptive Reinforcement Learning

M.A.H. Dempstr and V.Leemans [3] introduced **Adaptive Reinforcement Learning (ARL)** as the basis for a fully automated trading system application. They proposed a novel approach for FX trading that combines Reinforcement Learning techniques with two additional control layers that govern the investment policy.
Recurrent Reinforcement Learning (RRL) is the algorithm at the core of the application: originally proposed by Moody and Saffell [20], RRL is based on the information on past exchange rates and outputs a position signal $F_t \in$ {-1,1}, where 1 indicates a long position to take on the exchange and -1 a short one.
The overall architecture of this method is presented in Figure 3.1. It is composed of a core algorithm (Layer 1) on top of which sits a risk management layer (Layer 2) balanced by five different parameters:

- ρ, learning rate of the RRL algorithm
- η, adaptation parameter of the RRL algorithm
- δ, transaction cost factor: this value can be set above the bid/ask spread to obtain a risk-aversion measure
- x, basis points of a stoploss threshold applied on top of the algorithm indications

- y, a threshold applied on a new RRL position signal instead of just applying a sign function: the position is taken only if the signal exceeds this value

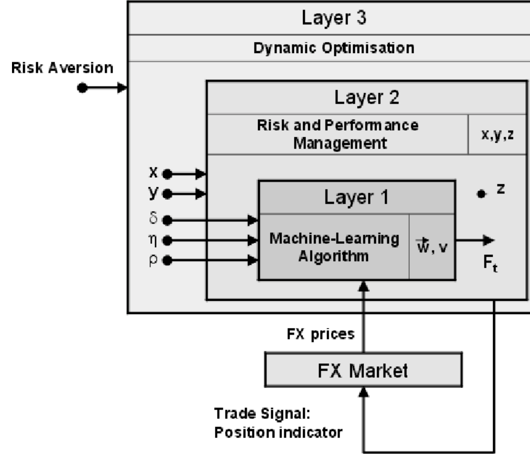


Figure 3.1: Adaptive Reinforcement learning layers

Those parameters are not set a priori but are optimized by a dynamic optimization layer over past periods of time of the training set (Layer 3). This optimization depends on a single parameter υ set by the user that manages the trade-off between strategy risk and desired return.

3.2 Recurrent Reinforcement Learning

C. Gold [9] investigated high frequency currency trading with neural networks trained via **Recurrent Reinforcement Learning (RRL)**.
Moody and Wu introduced Recurrent Reinforcement Learning for neural network trading systems in 1997 [21] and Moody and Saffell first published results for using such trading systems to trade in a currency market [20], as we shown in the previous section 3.1.
The goal of this study was to extend the results of [20] by giving detailed consideration to the impact of the fixed parameters of the trading system on performance, and by testing on a larger number of currency markets.
The goal of RRL is to update the weights in a recurrent neural network trader via gradient ascent in the performance function. As in [20], when RRL is used for a currency series with bid/ask prices the mid price is used to calculate returns and the bid/ask spread is accounted for as the transaction cost of trading. For a bid/ask price series the price returns input to the trader $r_t = p_t$ - p_{t-1} are calculated in terms of the mid-price

$$p_t^m = \frac{p_t^a + p_t^b}{2} \tag{3.1}$$

where p_t^a and p_t^b are respectively the bid and ask price at time t and an equivalend transaction cost rate is applied in

$$R_t = \mu(F_{t-1}r_t - \delta \mid F_t - F_{t-1} \mid) \tag{3.2}$$

to reflect the loss from position changes in bid/ask trading; μ is the number of shares traded and δ is the transaction cost rate per share traded. R_t is used to compute the profit at time T

$$P_T = \sum_{t=1}^{T} R_t \tag{3.3}$$

For trading returns the equivalent transaction cost is simply the spread divided by two $\delta(t) = \frac{p_t^a - p_t^b}{2}$. For all experiments trading and performance are computed in this way, but the final profit was calculated both by the approximate method described by using (3.1), (3.2) and (3.3) and also by the exact method of applying all trades at the exact bid and ask prices. he disagreement between the two methods for calculating profits was found to be insignificant.
The results of this work suggest that neural networks trained with Recurrent Reinforcement Learning can make effective traders in currency markets with a bid/ask spread. regardless of the price model used, the RRL method seems to suffer from a problem that is common to gradient ascent training of neural networks: there are a large number of fixed parameters that can only be tuned by trial and error.

3.3 Genetic algorithms and Reinforcement Learning

A. Hryshko and T.Downs describe the development of an advisory tool for FX traders that is based upon technical analysis and which maeks use of the machine learning techniques of **Genetic Algorithms (GA)** [13] and **Reinforcement Learning (RL)** [16]. The approach described in this paper is to design a system engine based on machine learning and embed it into a trafing system. This system draws upon available information to determine the optimum strategy for the trader. Unlike the human trader, it works on-line and around the clock all the time so its parameters are updated continuously over time to achieve the highest returns. This system makes its decisions and predicts the future market using a combination of different market models. It recognizes the state of the market by simultaneously examining signals from each indicator (rather than examining indicator signals one by one).
There are different ways of estimating the profitability of trading systems and in this paper the authors consider the Stirling Ratio, defined as the Profit divided by the Maximum Drawdown (maximum loss of trading capital).

In the GA formulation [13], a population of possible solutions is encoded as a set of **bit strings** (known as **parameter strings**), each of the same fixed length. The fitness of each string in the population is estimated and the basic GA operatos (crossover, selection and mutation) are then applied. This provides a second-generation population whose average fitness is greater than that of the initial population. The GA operators are now applied to the second-generation population and the process is repeated, generation after generation, until some stopping criterion is met. The string with maximum fitness in the final popoulation is then selected as the solution. In applying a GA to FX traing, each string represents a possible solution for the trader. Figure 3.2 illustrates the trading process.

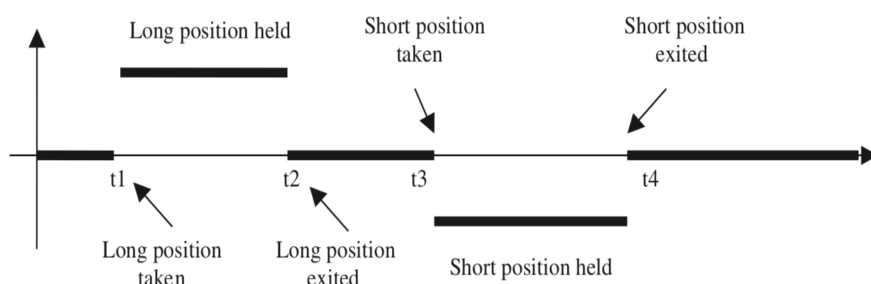


Figure 3.2: Genetic Trading process

The trader bases his decision upon the values of a set of indicators, implying that the bit strings must include indicator values. In a typical implementation, a population of 150 rules of each strategy is generated randomly. Then out of these 150 rules, the authors randomly combine 150 pairs consisting of one entry rule and one exit rule. At the end there are 150 trading strategies. Reproduction is applied by using the Roulette mechanism [13] to generate a new population. Pairs are ranked accorting to fitness (calculated by the Stirling Ratio) and are chosen in proportion to their rank to be involved in crossover and mutation.
The method of choosing the indicators to feed to the RL module is an improvement on the one in Dempster and Romahi [4] where the RL algorithm is itself employed to determine which of the indicators has the greatest fitness.
Because of the vast number of combinations of indicator values and connectives, the GA is unable to search the whole space of strategies to find the optimum. Although with high probability only one set of instantiations of rule values has been considered, the fact that the GA identified this strategy as a profitable one shows that the indicators used in the strategy are capable of making useful market predictions. Because of this, it is worthwhile to condier the other possible instantiations of the rule values in the strategy and this is the role of the RL algorithm. Figure 3.3 shows the basic structure of the trading system. The GA in-sampe module selects the most profitable strategy

(the best pair of entry and exit rules) and feeds the set of indicators making up these rules to the RL engine. The RL engine is base on the Q-learning algorithm proposed by Watkins [25] for partially observable Markov decision processes. Once there is the set of

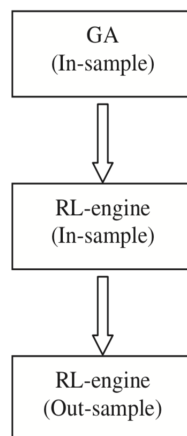


Figure 3.3: Optimization by Reinforcement Learning - The most predictive indicators are taken from the GA in-sample module and fed to RL engine

the most profitable indicators, they can be used to represent states of the environment. At each moment of time the trader has to make a decision whether to take a short, long or neutral (flat) position; thus the set of actions is (buy, flat, sell). Combining these indicators and actions leads to the Q-table.
In this GA-RL approach the data are divided into three parts: GA (in-sample), RL (in-sample) and RL (out-sample, testing) periods.

3.4 Support Vector Machine Stock Market Forecasting

S. Shen, H. Jiang and T. Zhang [24] proposed the use of global stock data in association with other financial products as input features for a SVM algorithm meant to predict stock market prices. Their idea is based on the assumption that between the different financial market exists an high degre of correlation, and thus financial data available before the target financial market opens can be used as explanatory features for the latter. They used forward feature selection to obtain optimal feature subsets with regard to accuracy of prediction, and found that a combination of daily market trend and long term movement provided the best results.
The selected SVM algorithm resulted very sensitive to the size of the training data and more robust with low-dimensional features sets, since using only the top 4 identified

features they were able to obtain better results than using all the available features. They performed a trading simulation using their output predictions and relying on a simple investment policy: buy a fixed amount of shares if the prediction of the day is positive, or sell all the owned shares if the prediction id negative. heir model outperformed the *Buy&Hold* benchmark in 4 simulations over 5.

3.5 Q-Learning and Sharpe Ratio Maximization

X. Gao and L. Chan [7] formalized a trading framework that operates on the Forex market combining Q-learning and Sharpe ratio maximization algorithms. They trained the RL system with absolute profit as reward and used supervised learning technique for a second system that aims to maximize Sharpe Ratio (SR) return. SR is defined as the expected value of the return divided by the standard deviation of the return (with the assumption of zero risk-free).
Target portfolio weights can be optimized defining SR as a functino of asset prices and portfolio's weights an then maximizing it. The RL framework provides, instead, a Q-function estimation of long and short position on the selected currency exchange. A final investment decision is then taken combining the results of the two algorithms:

$$\begin{cases} a_t = 1 & \text{if } (\beta_1 a_t^Q + \beta_2 a_t) > 0.5, \\ a_t = 0 & \text{if } (\beta_1 a_t^Q + \beta_2 a_t) \leq 0.5. \end{cases}$$

where a_t^Q is the action suggested by the learned Q-function, 1 and 0 are respectively long and short positions, and β_1, β_2 are positive parameters such that $\beta_1 + \beta_2 = 1$.
Their experimental results were obtained trading on a single Forex asset. The results outperformed a prediction-based trading strategy and the strategies output by the two proposed algorithm taken sincularly.
Taking in consideration the risk measure offered by a SR maximization in single-asset investment decision seems a promising intuition, but it is not clear how to merge the multi-weight outputs of the portfolio optimization system with the indications of the RL algorithm when taking into consideration multiple assets trading.

3.6 Multiagent Q-learning Framework

J.W. Lee and J.O [17] proposed a RL framework for stock trading systems with cooperative multiple agents to integrate more effectively long-term information and intra-day price movements of stocks. The framework, represented in Figure 3.4 is composed by four agents:

- **Buy signal agent** performs prediction by estimating the long-term and short-term information of the states of stocks to produce buy signals;
- **Buy order agent** determines prices for bids by estimating the short-term information of the states of stocks;
- **Sell signal agent** performs prediction by estimating the long-term and short-term information of the states of stocks and the current profits, to produce sell signals;
- **Sell order agent** determines the prices for offers by estimating the short-term information of the states of stocks.

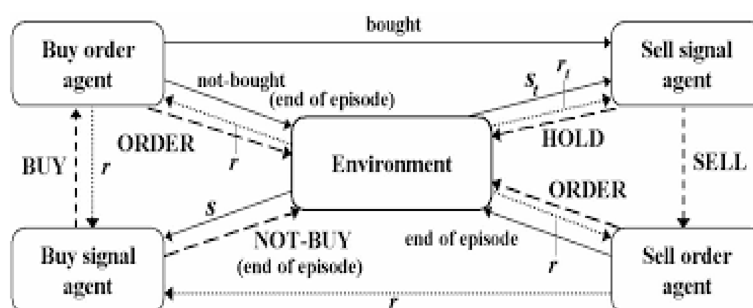


Figure 3.4: Structure of Multiagent Q-learning Framework

Signal agents are trained to output buy or sell signals for a given day, while order agents are responsible for placing the order at the correct time during the day to maximeize the profit. An interesting training feature is that the reward of buyin agents is calculated only when sell agent effectively sold the stock. Update rules and function approximation are obtained by means of regular Q-learning.
Even if the framework has multiple agents, the trading problem that is meant to solve dows not incorporate a policy over multiple stocks.

Chapter 4

Problem Formulation

4.1 Original Data

Initial data are €/$ exchange rate per minute (provided by AGS, from Bloomberg), from Monday to Friday (fx market is closed on Saturday and Sunday).
The data available are complete for the years 2014, 2015, 2016, 2017 and 2018.
In our datasets we have 1230 observations per day, considering a time slot from 00:00 to 20:30. Considering the Central European Standart Time - CET, the hours between 20:30 and 00:00 have been excluded because it is a time zone with few trading volume.

A small sample of the original dataset is provided in Table 4.1:

Date	Time	Close	Open	High	Low
01 - feb	02:00:00	1,2017	1,2016	1,2017	1,2016
01 - feb	02:01:00	1,2018	1,2017	1,2018	1,2017
01 - feb	02:02:00	1,2017	1,2018	1,2018	1,2017
01 - feb	02:03:00	1,2018	1,2017	1,2018	1,2017
01 - feb	02:04:00	1,2019	1,2018	1,2019	1,2018
01 - feb	02:05:00	1,202	1,2019	1,202	1,2019
01 - feb	02:06:00	1,2022	1,202	1,2022	1,2012
01 - feb	02:07:00	1,2022	1,2022	1,2023	1,2022
01 - feb	02:08:00	1,2021	1,2022	1,2022	1,2021
01 - feb	02:09:00	1,2021	1,2021	1,2021	1,2021
01 - feb	02:10:00	1,2021	1,2021	1,2023	1,2021

Table 4.1: sample of 2018 €/$ data

4.2 FQI Data

FQI needs a large input table, containing the current state, the actions, the next state and the reward.
The current state consists of the features (prices and/or price differences, time, position).
The position (or portfolio) consists of three values (-1, 0, 1), corresponding respectively to the 'Sell', 'Flat', 'Buy' movements. It consists in the portfolio position held previously by the agent.
The actions are three (-1, 0, 1) and represent the 'next' position (next portfolio).
The reward is calculated following the details described in the section 4.5.
Since the dataset necessary for FQI is made of tuples containing the previous state, the actions performed, the reward and the next state, the original dataset must be extended in order to consider all possible combinations of the previous position, and consequences of every possible action (Figure 4.1). Considering for example a one-year dataset, with data for 258 days, we will have a matrix with 2,856,060 (= $258 \cdot 1230 \cdot 9$) rows, and with a number of columns that will depend on the selected features.

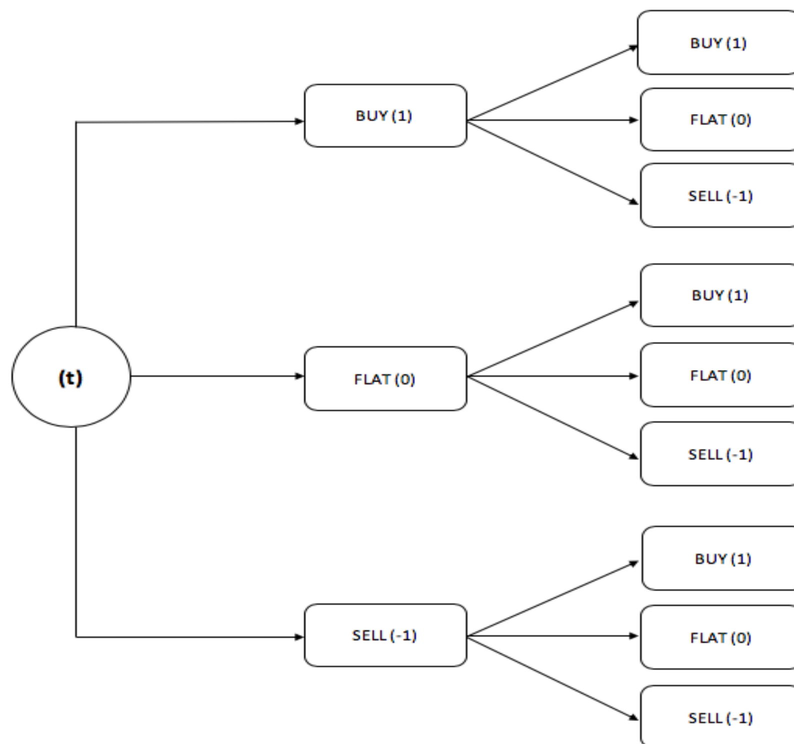


Figure 4.1: Actions - Positions Combination

4.3 Features

As features we used: portfolio, time, count, actions, prices, differences. Count represents the globally number of trades per minute.
As prices we used the shifted value with respect to the beginning of the trading day: the shifts compared to the first price of the day are recorded. We used this type of prices because we are interested in price changes during the day; in this way any model can generalize over several days. As difference we used the differences between two consecutive prices because we are also interested to consider the change in price instant by instant.
In addition to the shifted prices and differences, we also added those up to the previous 60 minute (one hour), therefore from instant (t-1) to (t-60).
This has been done because we wanted to consider the past and make the state as Markovian as possible. In order to be a real Markov process, the current state must provide all the necessary information. Providing only one price (that of the previous minute) gives us too little information, while having the trend of the previous 60 minutes gives us a better approximation of what is going on.
So, the total FQI features we used were:

FQI Features
Portfolio
Time
Count
Action
Lagged Shifted Prices [(P(t), ... ,P(t-60))]
Lagged Differences [(D(t), ... ,D(t-60))]

Table 4.2: FQI Features

4.4 Fees

We assumed to using a capital of € 100.000 per trade.
The fee (commission) is the amount of money to be paid for each transaction.
In our case (€/$), the fee for each transaction is assumed to be se at $ 2.
In the code we didn't consider the investment of € 100.000, but we considered it unitary, so instead of € 100.000 of investment and $ 2 of fee, we considered € 1 of investment and 2/1e5 of fees.
This choice was related to the way we considered the reward: We expressed the reward as a gain/loss in $ per single € invested; therefore the fee was divided by the single €; then the final reward was multiplied by the number of € invested; Next session contains the details of the reward calculation.

4.5 Reward

In order to compute the **Reward**, we need the **Reference Price**, which is an estimate of the buy/sell price.
The formulae for the Reference Price and the Reward we used are:

$$R_t = [A_t \cdot (RP_{t+1} - RP_t)] - |A_t - PTF_t| \cdot F \tag{4.1}$$

$$RP_t = O_t + [(H_t - L_t) \cdot \text{sign}(C_t - O_t)] \cdot DL \tag{4.2}$$

where:

- R_t = Reward
- A_t = Action
- RP_t = Reference Price
- PTF_t = Position (Portfolio)
- F = Fee
- O_t = Open Price
- C_t = Close Price
- H_t = Highest Price
- L_t = Lowest Price
- DL = Delay

The **signum** function of a real number x is defined as follows:

$$sign(x) = \begin{cases} -1 & \text{if } x < 0, \\ 0 & \text{if } x = 0, \\ 1 & \text{if } x > 0. \end{cases}$$

In order to better understand equation 4.1 we remind to Section 4.5.2

4.5.1 Reference Price

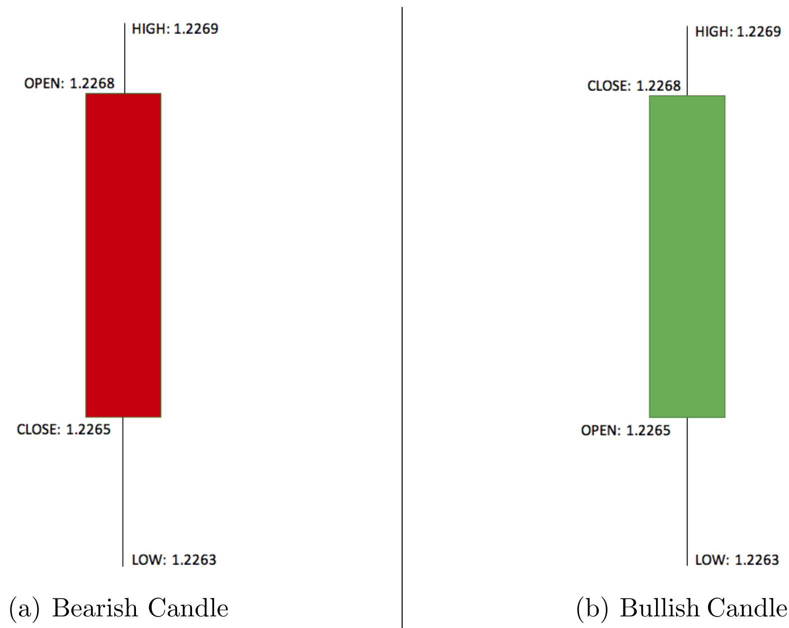
Since we do not have data with a frequency lower than the minute, we cannot know the exact buy or sell price. We must therefore estimate it.

The method we applied is the following: We took the High/Low difference (H-L) and made a proportion of it. For example let's assume that $O_t = 1.2268$, $C_t = 1.2265$, $H_t = 1.2269$, $L_t = 1.2263$, (Figure 4.2 (a)). We compute the difference H_t - $L_t = 0.0006$ and make a proportion to decide the variation (for example if we decide to use 10 seconds out of 60 as delay, we will multiply H_t -L_t by 1/6) obtaining a delta = 0.0001; finally we add this quantity to the open price O_t.

Obviously, we must consider the fact that the price has varied more in negative or positive direction and therefore if (H_t - O_t) < (O_t - L_t) we subtract the previous quantity from the open price, otherwise we add it; this is the reason of sign(C_t - O_t) in formula (4.2). In our example the final *trade price* would be 1.2267.

With no delay ($DL = 0$) the second term in (4.2) is null and the reference price becomes the open price; this means **instantaneous** buy/sell.

In our analysis we evaluated also the effects of different delays in evaluating the buy/sell price.



(a) Bearish Candle

(b) Bullish Candle

Figure 4.2: Bear-Bull Candlesticks: typical representation of price movements in trading (in this case they represent prices in a minute)

4.5.2 Reward Cases

Now we try to show the behavior of the reward in the different possible cases and how the fees impact on it.
The possible cases are:

- Stay in the same position (flat to flat, buy to buy, sell to sell);
- Change of position by one step (flat to buy, flat to sell and vice versa), fees are paid only once;
- Change of position by two steps (sell to buy and vice versa, fees are paid twice.

We have 9 possible combinations as shown previously in Figure 4.1.
Let's see what happens in three of these possible cases:

Flat to Sell

In this case the position $PTF_t = 0$ and the action $A_t =$ -1, so following Equation 4.1, the reward will be

$$
\begin{aligned}
R_t =& [-1 \cdot (RP_{t+1} - RP_t)] - sign(-1 - 0) \cdot F \\
=& -(RP_{t+1} - RP_t) - F
\end{aligned}
$$

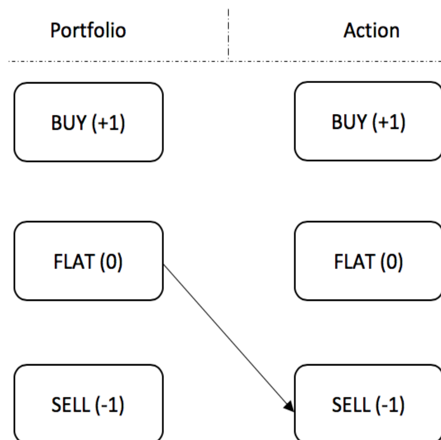


Figure 4.3: Flat to Sell

Flat to Buy

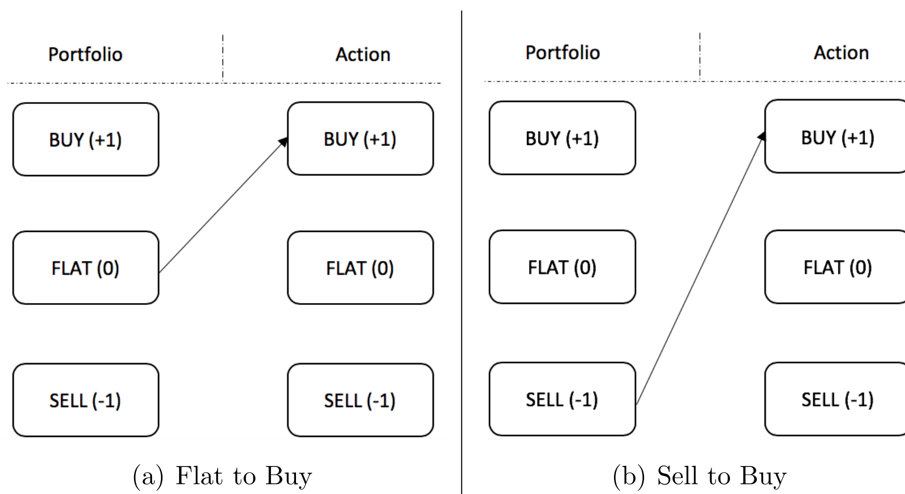
In this case the position P_t = 0 and the action A_t = +1, so following 4.1, the reward will be

$$\begin{aligned} R_t =& [1 \cdot (RP_{t+1} - RP_t)] - sign(1 - 0) \cdot F \\ =& (RP_{t+1} - RP_t) - F \end{aligned}$$

Sell to Buy

In this case the position P_t = -1 and the action A_t = +1, so following 4.1, the reward will be

$$\begin{aligned} R_t =& [1 \cdot (RP_{t+1} - RP_t)] - sign(1 - (-1)) \cdot F \\ =& (RP_{t+1} - RP_t) - 2 \cdot F \end{aligned}$$



(a) Flat to Buy (b) Sell to Buy

Figure 4.4: Flat to Buy & Sell to Buy Combinations

Chapter 5

Analysis of the dataset

5.1 Original Dataset

In this first section we do a qualitative analysis of our initial datasets available, which are exchange data of €/$ in the FX market for 2014, 2015, 2016, 2017, 2018.
In particular, we will focus on the analysis of stationarity.
We will confirm the non-stationarity of the processes that generate the time series of the €/$ exchange rate for all the years (from 2014 to 2018). And we will analyze this property.
First of all, let's look at the graphs of our original data, in particular we look at the **open** price column which is the one we use most for the creation of the FQI dataset.

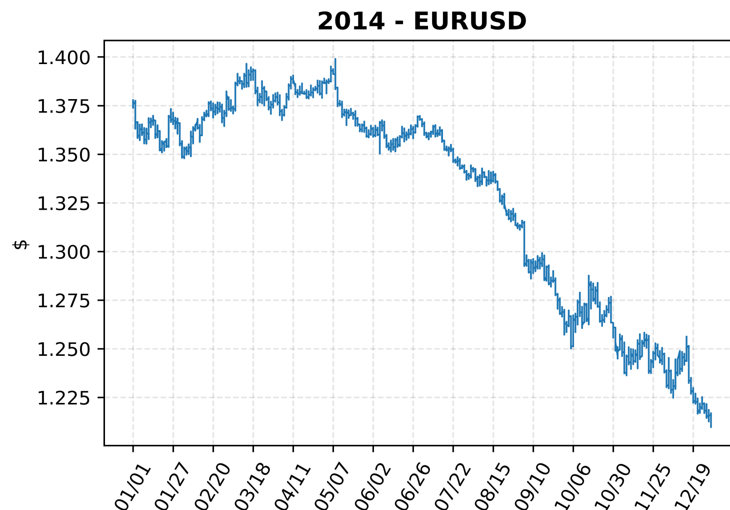
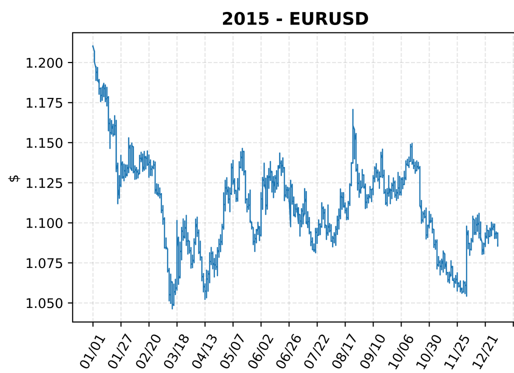
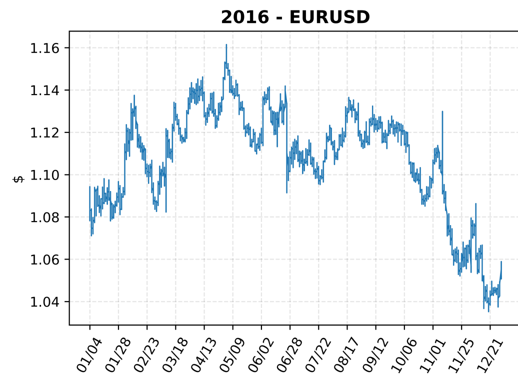


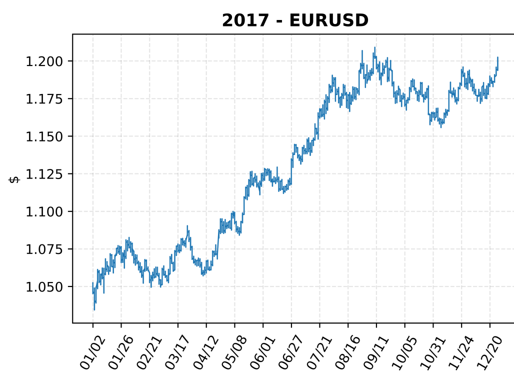
Figure 5.1: 2014 - EURUSD open price:
we can see a general bearish (downward) trend



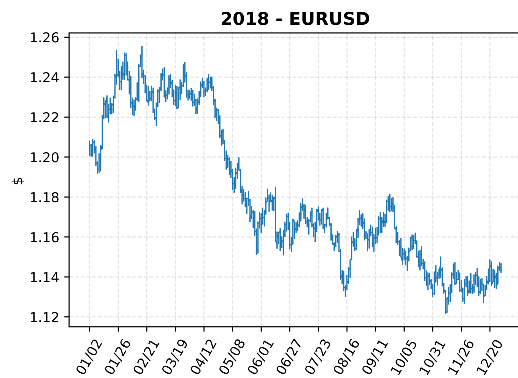
(a) 2015 - EURUSD open price: in the first three months we have a bearish trend, then a small growth and again a decrease; there are a lot of drawdowns



(b) 2016 - EURUSD open price: there are a general bullish (upward) trend for the first five months and then a downward trend; there are a lot of drawdowns



(c) 2017 - EURUSD open price: there is a general bullish trend



(d) 2018 - EURUSD open price: there is a bullish trend for the first two months, then a bearish trend

5.1.1 Stationarity

We can already see from the previous graphs, how the time series do not have a constant mean and variance over time, indicating a non-stationarity.
Stationarity means that the statistical properties of a time series (or rather the process generating it) do not change over time. It is an important concept in time series analysis because many useful analytical tools and statistical tests and models rely on it. For full details, see [12].
The ability to determine wether a time series is stationary is important. This usually means being able to ascertain, with high probability, that a series is generated by a stationary process.
For brevity, from now on we will say that a time series is stationary, meaning that it is generated by a stationary process.
In ML, we try to **learn from data** and not starting from an already established model; we cannot know how to best model unknown non-linear relationships in time series data and some methods may result in better performance when working with non-stationary observations or some mixture of stationary and non-stationary views of the problem.
We can treat properties of a time series being stationary or not as another source of information that can be used in feature selection on our problem.

There are many methos to check whether a time series is stationary or non-stationary.

1. **Summary Statistics:** We can review the summary statistics for the data for seasons or random partitions and check for obvious or significant differences.

2. **Statistical Tests:** We can use statistical tests to check if the expectations of stationarity are met or have been violated.

Above, we have already seen the plots of our time series, showing an evident presence of trend and seasonality components.

Summary Statistics

A quick check to see if the time series is non-stationary is to review summary statistics. We can split the time series into two (or more) partitions and compare the mean and variance of each group.
In our case we can split our time series into two contiguous sequences and then calculate the 6 months mean and variance of each group of numbers and compare the values.

Year	Mean 1	Mean 2	Variance 1	Variance 2
2014	1.3709	1.2866	0.000126	0.001978
2015	1.1166	1.1037	0.001145	0.000494
2016	1.1164	1.0975	0.000365	0.000705
2017	1.0825	1.1758	0.000624	0.000239
2018	1.2103	1.1519	0.000817	0.000201

Table 5.1: Mean and Variance Splitting

From Table 5.1 we can see the mean and variance look very different for all the years.

We can also check if assuming a Gaussian distribution makes sense, by plotting the values of the time series as a histogram. If the distribution of values is far from being Gaussian, therefore the mean and variance values are less meaningful, and it may be another indicator of a non-stationary time series.
This is the case for all our five time series, and we can see this in the following histograms:

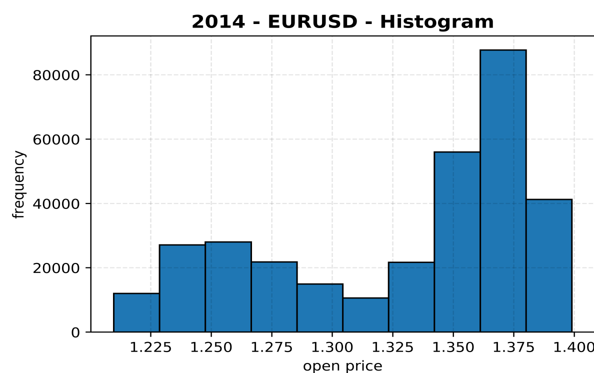
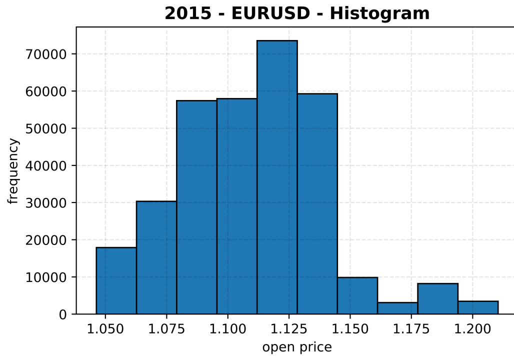
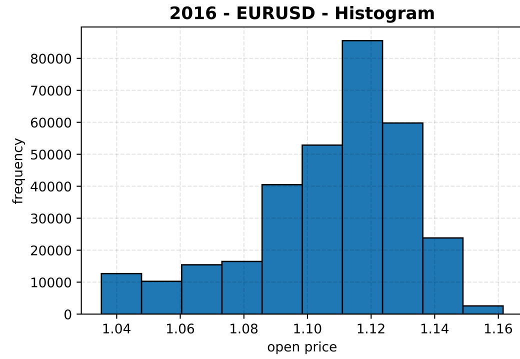


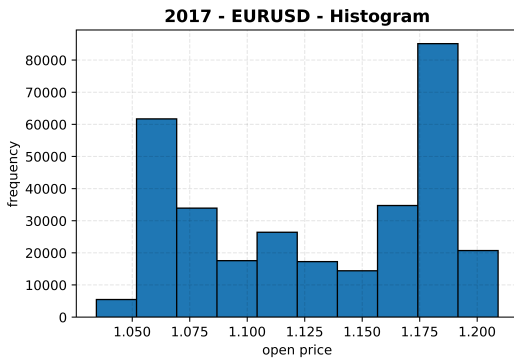
Figure 5.2: 2014 - EURUSD Histogram: data (in a range between 1.2 and 1.4) are not distributed like a Gaussian; many data are concentrated between 1.3 and 1.4 (on the right)



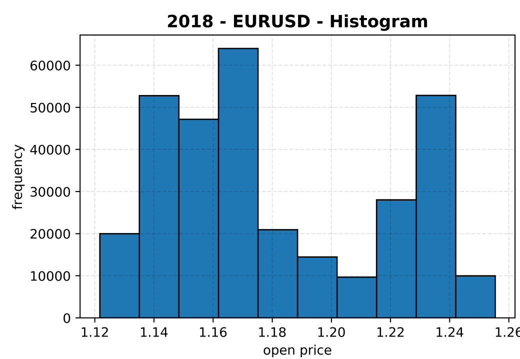
(a) 2015 - EURUSD Histogram: data (in a range between 1.0 and 1.225) are not distributed like a Gaussian; most of the data are concentrated between 1.08 and 1.13 (on the left)



(b) 2016 - EURUSD Histogram: data (in a range between 1.035 and 1.16) are not distributed like a Gaussian, with most of the distribution concentrated between 1.1 and 1.13 (on the right)



(c) 2017 - EURUSD Histogram: data (in a range between 1.025 and 1.225) are not distributed like a Gaussian; we can see two peaks in the histogram, corresponding to the first four months and the last three months of the year with values concentrated around 1.06 and 1.18 respectively



(d) 2018 - EURUSD Histogram: data (in a range between 1.122 and 1.258) are not distributed like a Gaussian; most of the data are concentrated between 1.14 and 1.17 (on the left)

Statistical Test: Augmented Dickey-Fuller Test (ADF)

Statistical tests, like the Augmented Dickey-Fuller (ADF), make strong assumptions about our data. They can only be used to inform the degree to which a null hypothesis can be rejected or fail to be rejected. They can provide a quick check and confirm evidence that our time series is stationary or non-stationary.
The ADF test is a type of statistical test called a unit root test. It is an augmented version of the Dickey-Fuller test for a larger and more complicated set of time series models (see [12]).
The intuition behind a unit root test is that it determines how strongly a time series is defined by a trend.
There are a number of unit root tests and the Augmented Dickey-Fuller may be one of the more widely used. It uses an autoregressive model and optimizes an information criterion across multiple different lag values.
The null hypothesis of the test is that the time series can be represented by a unit root, that it is not stationary (has some time-dependent structure). The alternative hypothesis (rejecting the null hypothesis) is that the time series is stationary.

- **Null Hypothesis (H_0):** If failed to be rejected, it suggests the time series has a unit root, meaning it is non-stationary. It has some time dependent structure.
- **Alternative Hypothesis (H_1):** The null hypothesis is rejected; it suggests the time series does not have a unit root, meaning it is stationary. It does not have time-dependent structure.

We interpret this result using the p-value from the test. A p-value below a threshold (such as 5%) suggests we reject the null hypothesis (stationary), otherwise a p-value above the threshold suggests we fail to reject the null hypothesis (non-stationary).

- **p-value $>$ 0.05:** Fail to reject the null hypothesis (H_0), the data has a unit root and is non-stationary.
- **p-value $\leq$ 0.05:** Reject the null hypothesis (H_0), the data does not have a unit root and is stationary.

The **statsmodels** library in *Python* provides the **adfuller()** function that implements the test.
From this test we will print the ADF statistic, the p-value and the 1%, 5% and 10% critical values (CV). The more negative the ADF statistic, the more likely we are to reject the null hypothesis, so we have a stationary dataset.
If the absolute value of our ADF statistic is greater than the critical value, we can declare statistical significance and reject the null hypothesis (H_0).
Running the ADF test, in our cases we optained:

Year	ADF Statistic	p-value	CV 1%	CV 5%	CV 10%
2014	-1.23	0.98	-3.43	-2.86	-2.56
2015	-2.12	0.53	-3.43	-2.86	-2.56
2016	-1.61	0.46	-3.43	-2.86	-2.56
2017	-0.92	0.78	-3.43	-2.86	-2.56
2018	-1.03	0.74	-3.43	-2.86	-2.56

Table 5.2: ADF Test - open price

From Table 5.2, looking at the p-values and comparing the test statistics to the critical values, it looks like we would have to fail to reject the null Hypothesis (H_0) that the time series are non-stationary and does have a time-dependent structure, for all the five years.

In the FQI Dataset, among our features, in addition to the shifted prices at the first price of the day, we used the differences.
In addition to the reasons illustrated in Chapter 4 about this choice, the differences represent a stationary time series, and this can improve the performance of our model. Looking at the plots and applying the ADF test to the time series of the differences in the original *open* price dataset, we can see how these time series are stationary.

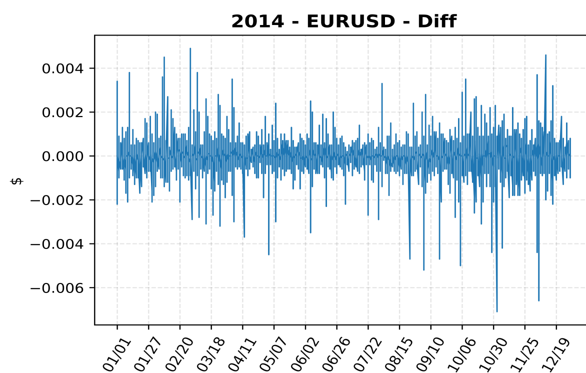
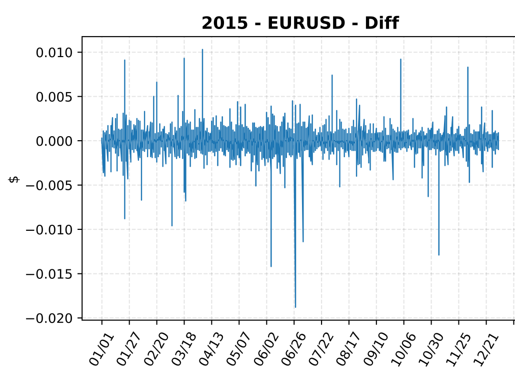
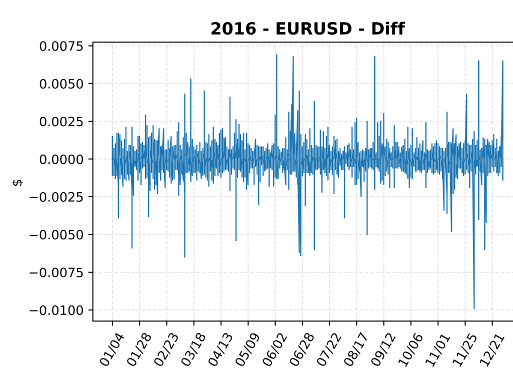


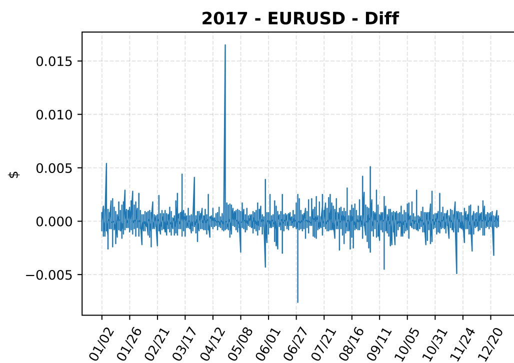
Figure 5.3: 2014 - EURUSD Differences: $O_{t+1} - O_t$ $\forall t$ where O_t is the open price in t



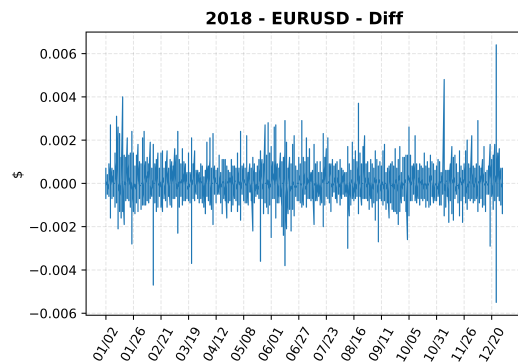
(a) 2015 - EURUSD Differences: $O_{t+1} - O_t$ $\forall t$ where O_t is the open price in t



(b) 2016 - EURUSD Differences: $O_{t+1} - O_t$ $\forall t$ where O_t is the open price in t



(c) 2017 - EURUSD Differences: $O_{t+1} - O_t$ $\forall t$ where O_t is the open price in t



(d) 2018 - EURUSD Differences: $O_{t+1} - O_t$ $\forall t$ where O_t is the open price in t

Year	ADF Statistic	p-value	CV 1%	CV 5%	CV 10%
2014	-90.41	0.0	-3.43	-2.86	-2.56
2015	-59.14	0.0	-3.43	-2.86	-2.56
2016	-58.22	0.0	-3.43	-2.86	-2.56
2017	-58.53	0.0	-3.43	-2.86	-2.56
2018	-59.54	0.0	-3.43	-2.86	-2.56

Table 5.3: ADF Test - open price differences ($O_{t+1} - O_t$ $\forall t$ where O_t is the open price in t)

From table 5.3 we can see that all the p-values are < 0.05 and all the absolute values of the ADF statistics are strongly greater than the critical values, therefore we can reject the null hypothesis (H_0) that the time series are non-stationary.

5.2 FQI Dataset

5.2.1 Regressor and Classification Analysis

In this part we did some analysis of the FQI table. In particular we tried to understand the trend of the R^2, the MSE and the distribution of the rewards (true and predicted), by varying the *min split* parameter of the Extra-Trees regressor. The same using *accuracy* score and look also at the $confusion\ matrix$.
The *min split* can vary between considering at least one sample at each node to all of the samples at each node. When we increase this parameter, each tree in the forest becomes more constrained as it has to consider more samples at each node.
We will therefore see how the minsplit impacts the quality of reward prediction and the reward sign (classification).
The results that we will see in this chapter have been obtained with the hypothesis of **immediate operation** (with $DL = 0$, where DL is the delay in Equation 4.2).

Cross Validation

To evaluate the performance of any machine learning model we need to test it on some unseen data.
Based on the models performance on unseen data we can say wether our model is Under fitting / Over fitting / Well generalised.
Cross validation (CV) is used to test the effectiveness of a machine learning model, it is also a re-sampling procedure used to evaluate a model if we have limited data.
We used the **K-Folds Cross Validation**: it ensures that every observation from the original dataset will be used in training and validation set. The steps of this method are:

1. Split the entire dataset randomly into K folds (Figure 5.4 shows how the dataset is split);
2. Fit the model using the $K - 1$ folds and validate the model using the remaining Kth fold. Store the scores/errors.
3. Repeat this process until every K-fold serve as the validation set. Then take the average of the recorded scores.

In our case we started doing this analysis with only a dataset of 25 days, using 5-fold cross validation in order to check if everything works; then we used 2 year dataset, with 2-fold cross validation.

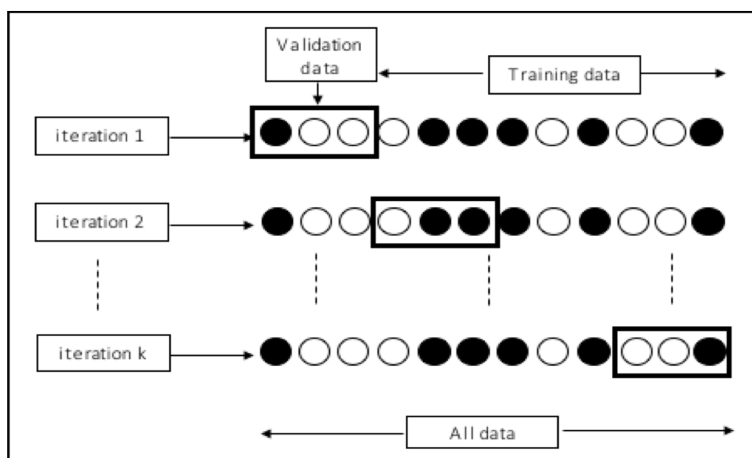


Figure 5.4: Cross Validation Scheme

R^2 and MSE

The **coefficient of determination**, denoted R^2, represents the proportion of the variance in the dependent variable (y) that has been explained by the independent variable(s) in the model.
It provides an indication of goodness of fit and therefore a measure of how well unseen samples are likely to be predicted by the model. It is a comparison of the overall error obtained by the model, with the error that I would make estimating each value as the empirical average.
If $\hat{y}_i$ is the predicted value of the i-th sample and y_i is the corresponding true value for total n samples, the estimated R^2 is defined as:

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \overline{y})^2}$$

where $\overline{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$.
Best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

The **mean squared error (MSE)** of an estimator measures the average of the squares of the errors - that is, the average squared difference between the estimated values and the actual value.
The MSE is a risk function, corresponding to the expected value of the squared error loss. It is a measure of the quality of an estimator; it is always non-negative, and values closer to zero are better.

If $\hat{y}_i$ is the predicted value of the i-th sample and y_i is the corresponding true value, then the mean squared error (MSE) estimated over n samples is defined as

$$MSE(y, \hat{y}) = \frac{1}{n} \sum_{i=0}^{n-1} (y_i - \hat{y}_i)^2$$

Accuracy and Confusion Matrix

We tried to understand how reward is predicted in terms of sign.
We have divided the real and predicted reward into 3 groups: positive reward, null reward and negative reward. We then analyzed the accuracy and confusion matrix of these results.

Tha **accuracy** ($0 \leq accuracy \leq 1$) represents the fraction or the count of correct predictions.
If the entire set of predicted labels for a sample strictly match with the true set of labels, then the subset accuracy is 1.0.

If $\hat{y}_i$ is the predicted value of the i-th sample and y_i is the corresponding true value, then the fraction of correct predictions over n samples is defined as:

$$accuracy(y, \hat{y}) = \frac{1}{n} \sum_{i=0}^{n-1} \mathbb{1}(\hat{y}_i = y_i)$$

where $\mathbb{1}(x)$ is the indicator function.

The **confusion matrix**, is a matrix that allows visualization of the performance of an algorithm in ML, specifically in the problem of classification.
Each row of the matrix represents the instances in a predicted class while each column represents the instances in an actual class (or vice versa).
By definition, entry i, j in a confusion matrix is the number of observations actually in group i, but predicted to be in group j.

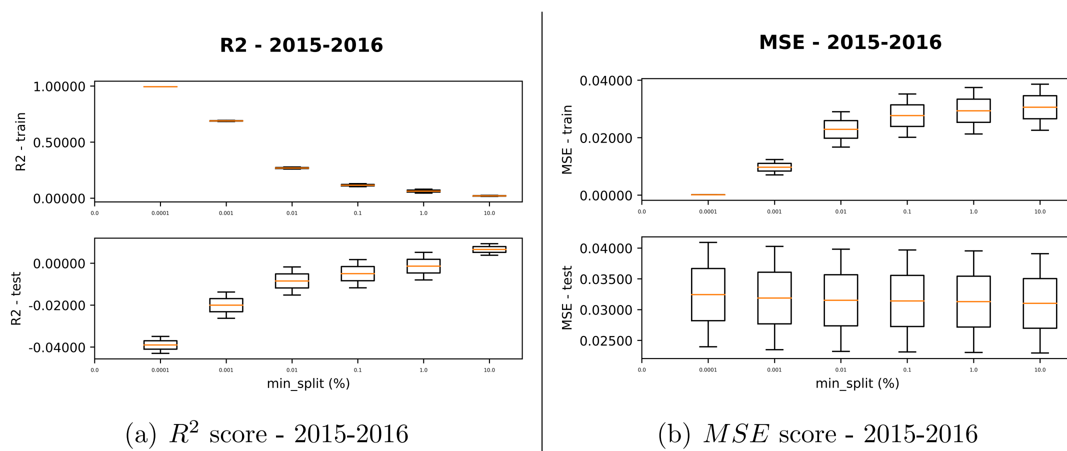
Regression and Classification Results

To see clearly how the different *min split* impact, we have chosen to observe the results for *min split* values ranging from 0.0001 % to 10 % of the size of the dataset.
The following tables show the precise values of *min split* used in this analysis, in the case of datasets for 1 year and datasets for 2 years, considering the convention chosen by $scikit - learn$.
We remember that the length of the 1 year FQI table is 2853738 rows and the 2 year FQI table is 5762781 rows.

	1 year FQI table	2 year FQI table
0,0001 %	3	6
0,001 %	29	58
0,01 %	286	577
0,1 %	2854	5763
1 %	28538	57628
10 %	285374	576279

Table 5.4: *min split* values

We did the analysis for all the combination of the years (from 2014 to 2018) and also for different K-fold choices; here we show only one combination because the results were similar with the other combinations.
We used 2 year dataset (2015-2016), with 2-fold cross validation (in order to use one year of data for train and one year odf data for validation).



(a) R^2 score - 2015-2016

(b) MSE score - 2015-2016

We can see how R^2 is very high with a decreasing trend as the *min split* in the train increases while it is increasing (but with negative values) up to values near 0.

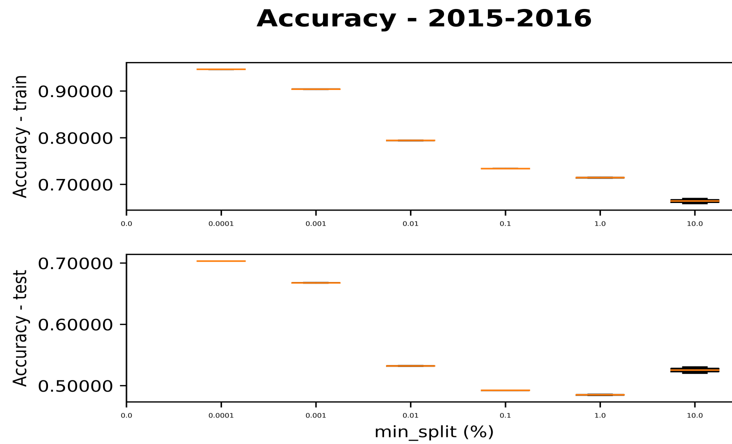


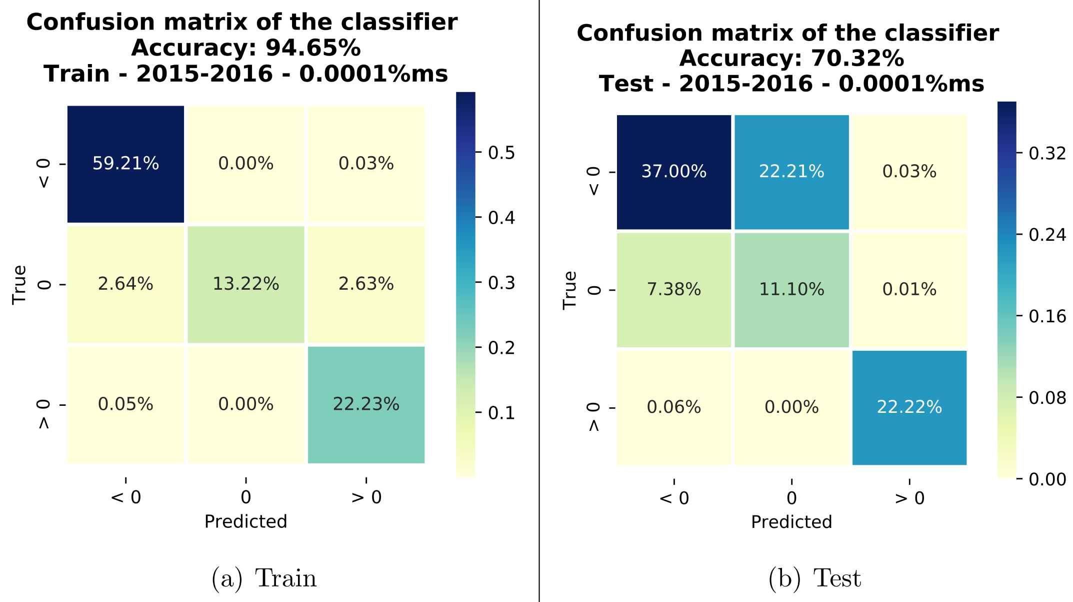
Figure 5.5: Accuracy Score - 2015-2016

The MSE in train increases as the *min split* increases, but always remains in a range of small values, while in the test there is no particular increasing or decreasing trend, remaining around a value of 0.032.

Accuracy decreases in train (from 0.9 to about 0.7), while in test it starts from a value of about 0.7, decreases until it reaches a minimum for *min split* = 1%, with a value around 0.5 and then rises slightly for the last minsplit.

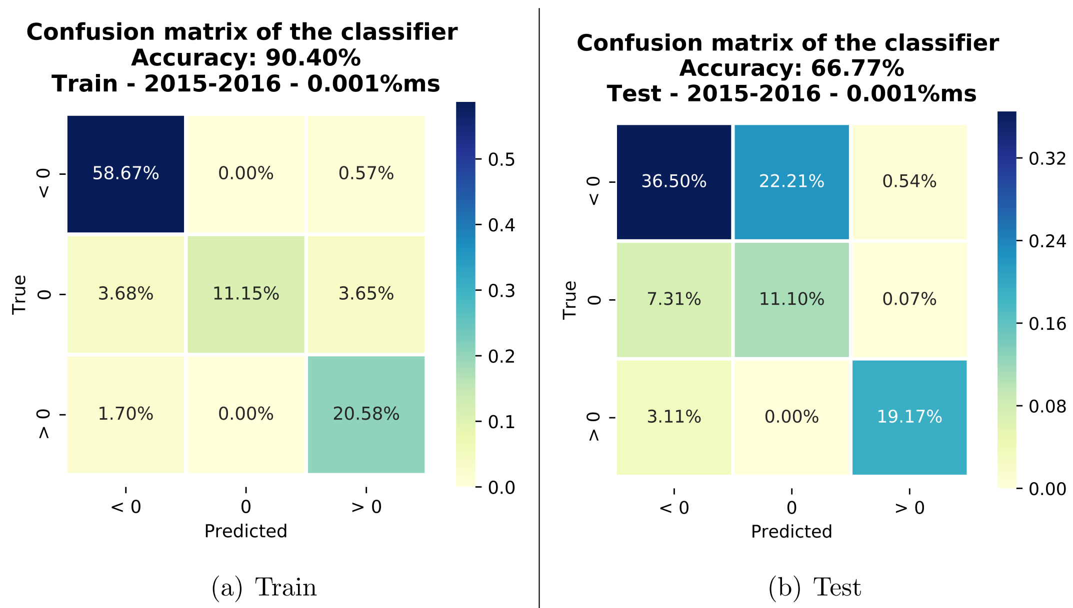
It therefore seems that the model is more accurate in predicting the reward sign, rather than variance, in fact we have low and negative R^2, while accuracy above 0.5.
It seems that it is more important to understand if the price will go up or down in the next step, rather than knowing for sure how much. We cannot know how much this behavior depends on the type of reward or the fees.

Since the accuracy values are good, it is interesting to understand in detail how the true reward is distributed against the real one and to do this we first look at the confusion matrix (where the sum of the values on the diagonal corresponds to the accuracy) and then the reward scatter plot true against that predicted.



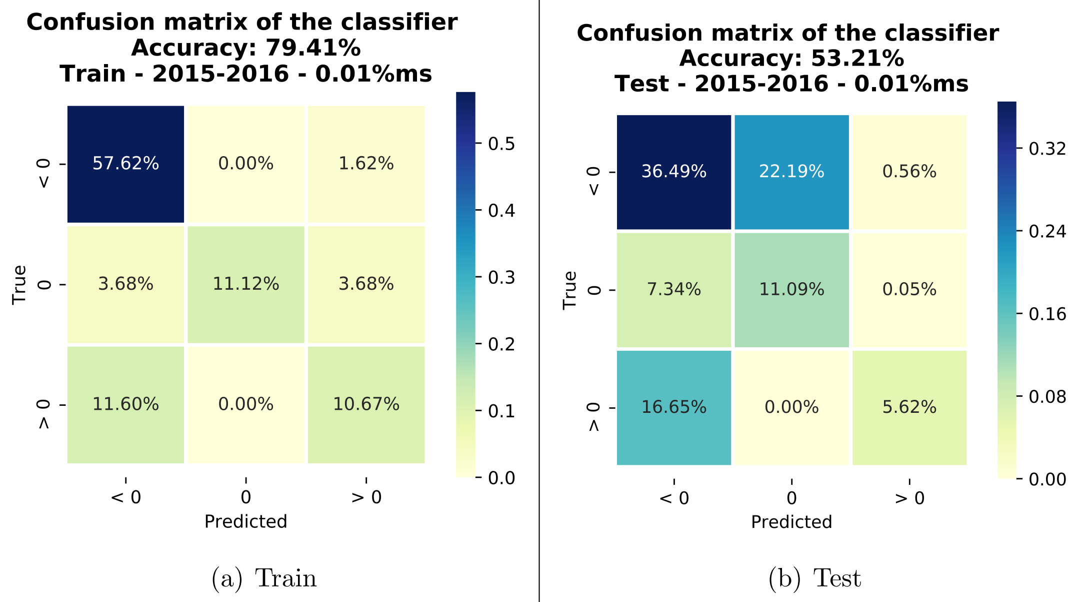
(a) Train (b) Test

Figure 5.6: Confusion Matrix - 0.0001% *min split* - 2015-2016



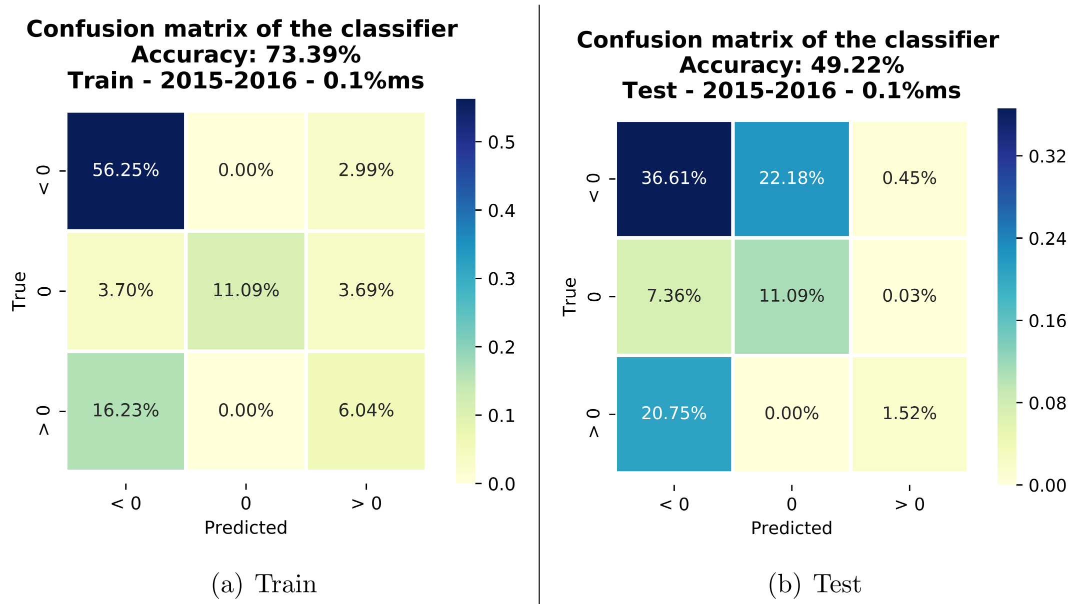
(a) Train (b) Test

Figure 5.7: Confusion Matrix - 0.001% *min split* - 2015-2016



(a) Train (b) Test

Figure 5.8: Confusion Matrix - 0.01% *min split* - 2015-2016



(a) Train (b) Test

Figure 5.9: Confusion Matrix - 0.1% *min split* - 2015-2016

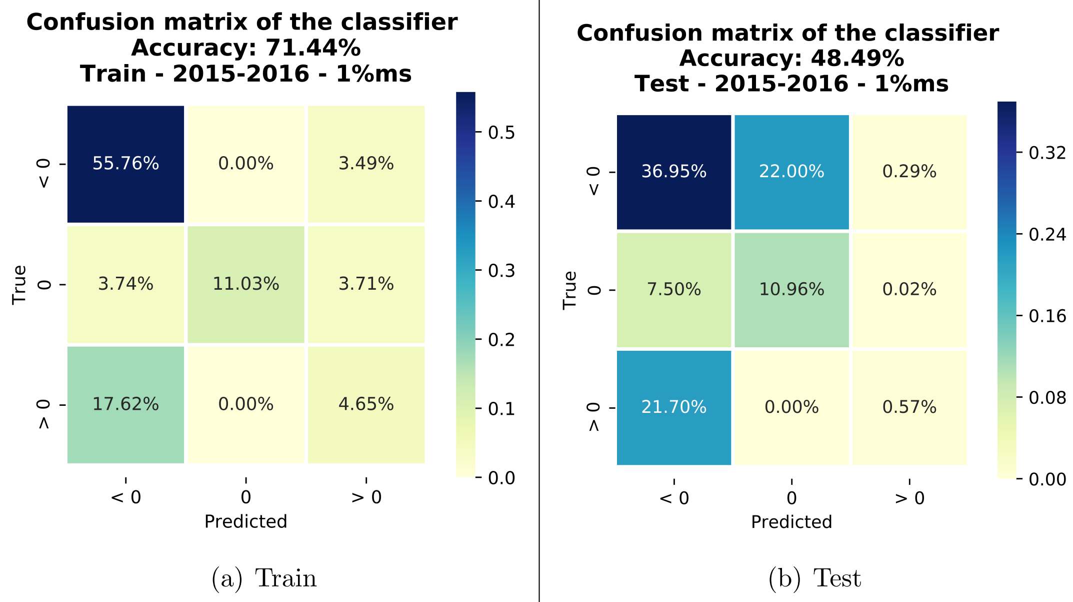


Figure 5.10: Confusion Matrix - 1% *min split* - 2015-2016

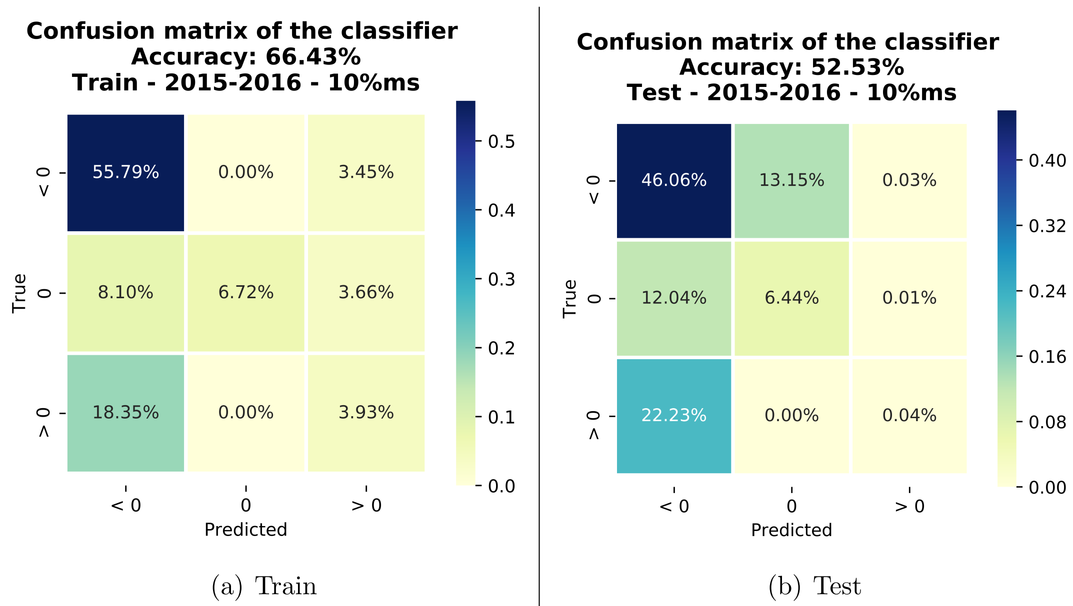


Figure 5.11: Confusion Matrix - 10% *min split* - 2015-2016

To see even better how the true and predicted rewards are distributed, we can look at the true reward scatter plot against the predicted reward.
We have scatter plot Train for Fold 1 and Fold 2, and scatter plot Test for Fold 1 and Fold 2; these for each different *min split* value. Here we show the plots for Fold 1, because those for Fold 2 are similar and do not add any further information.
We show the original plot and the plot with a zoom for x and y values in (-2,2).

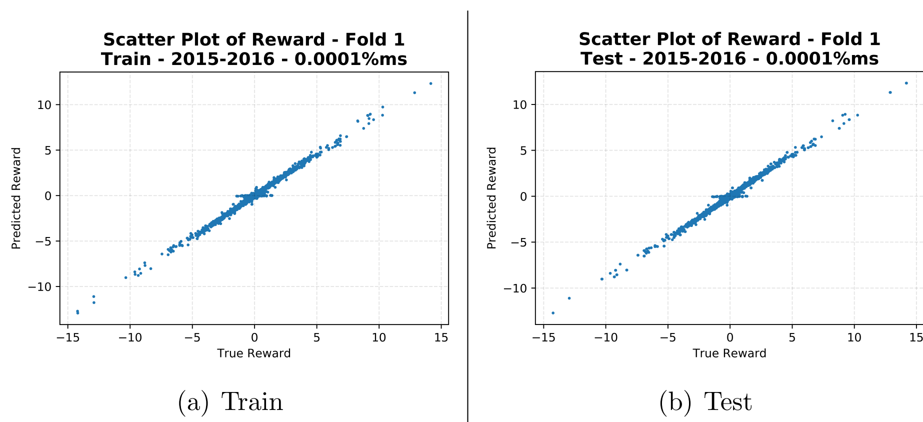


Figure 5.12: Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 0.0001% *min split*

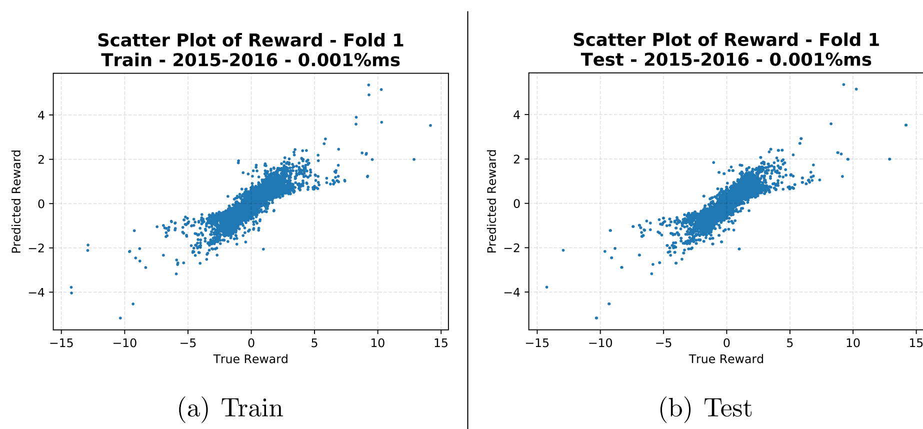
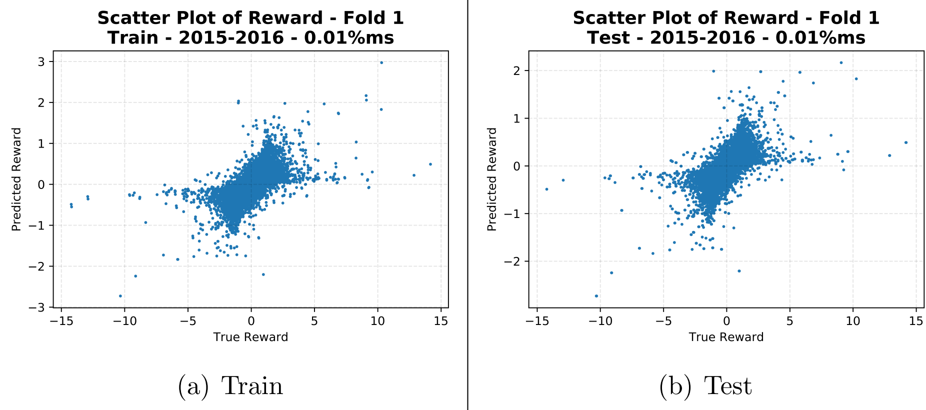


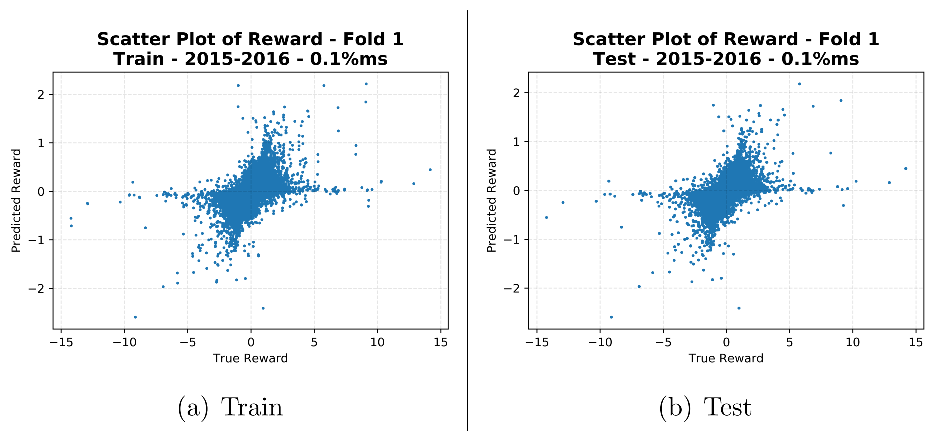
Figure 5.13: Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 0.001% *min split*



(a) Train

(b) Test

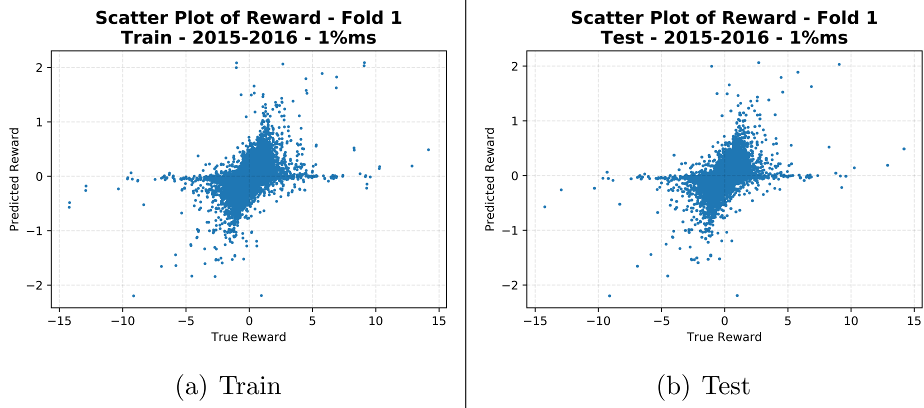
Figure 5.14: Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 0.01% *min split*



(a) Train

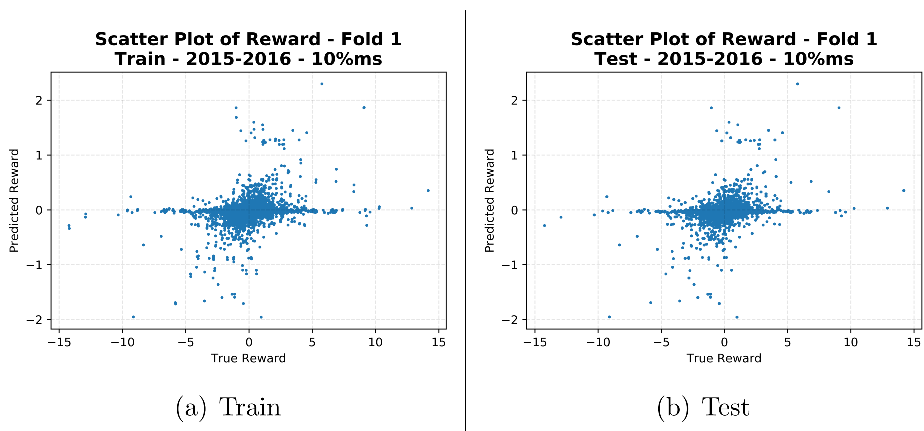
(b) Test

Figure 5.15: Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 0.1% *min split*



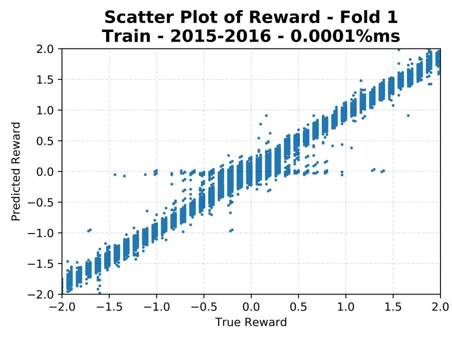
(a) Train (b) Test

Figure 5.16: Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 1% *min split*

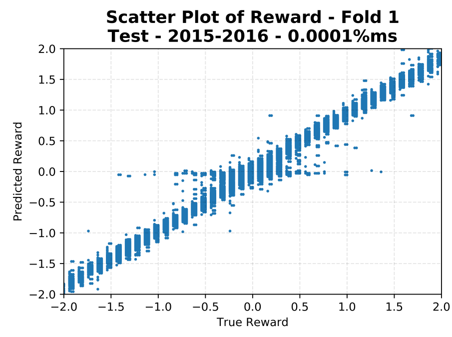


(a) Train (b) Test

Figure 5.17: Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 10% *min split*

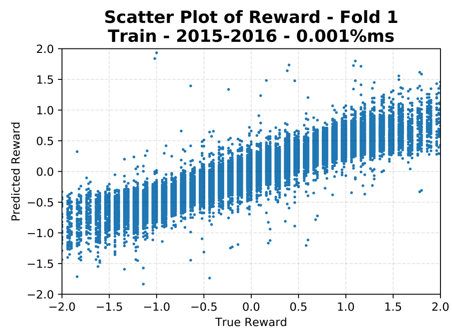


(a) Train

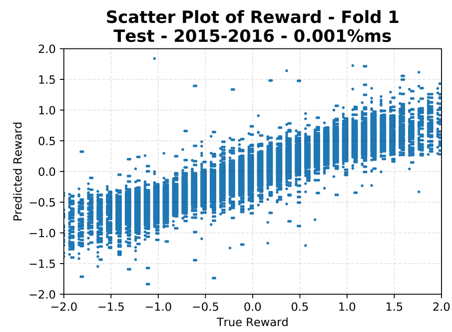


(b) Test

Figure 5.18: zoom Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 0.0001% *min split*

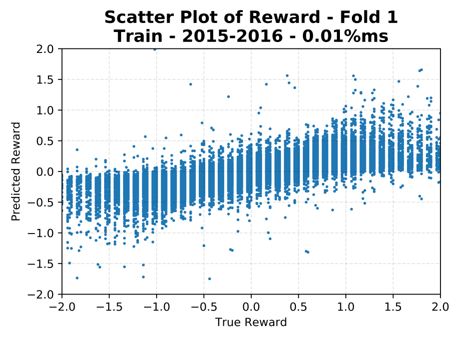


(a) Train

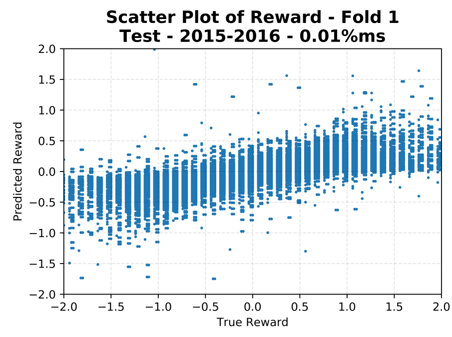


(b) Test

Figure 5.19: zoom Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 0.001% *min split*

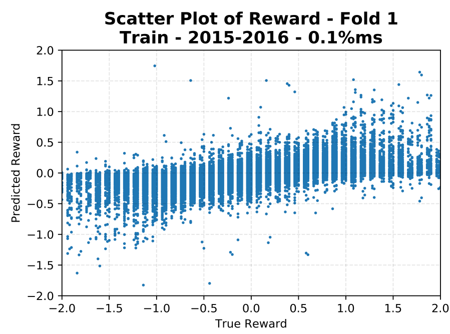


(a) Train

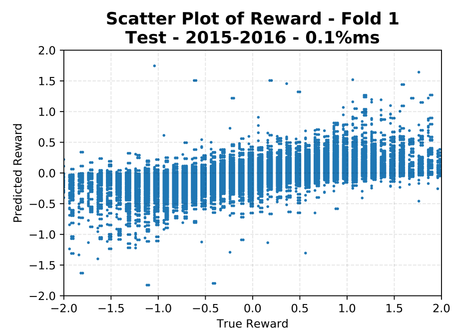


(b) Test

Figure 5.20: zoom Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 0.01% *min split*

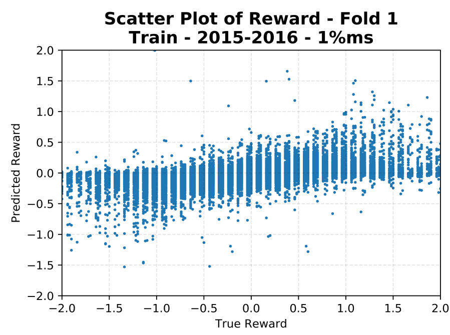


(a) Train

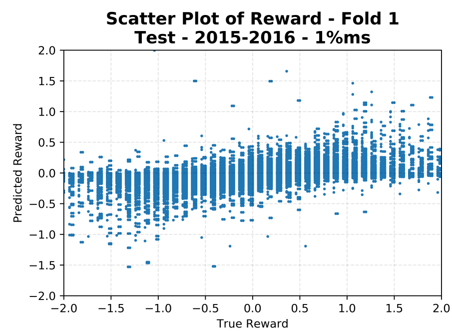


(b) Test

Figure 5.21: zoom Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 0.1% *min split*



(a) Train



(b) Test

Figure 5.22: zoom Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 1% *min split*

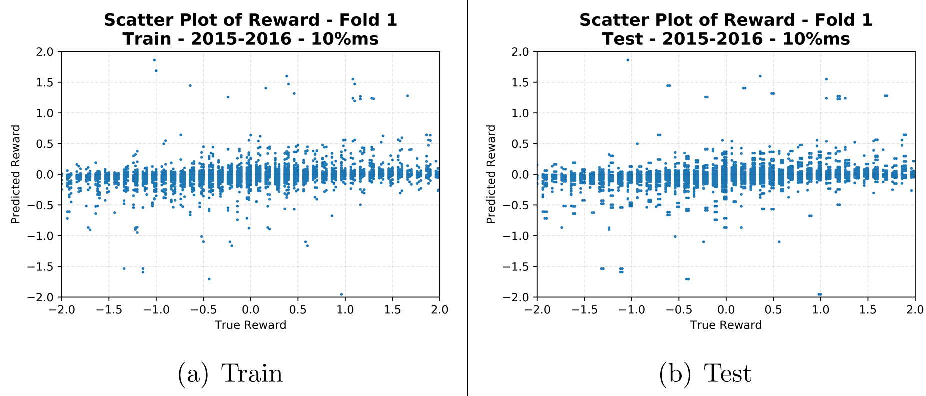


Figure 5.23: zoom Scatter Plot True vs Predicted Reward - Fold 1 - 2015-2016 - 10% *min split*

From the confusion matrices we can see how as the *min split* increases, the model tends to predict negatively more positive rewards, both in train and in tests.
Looking then at the true vs predicted reward scatter plots we can confirm what we obviously see from the confusion matrices and we also see that:

- the real reward is predicted with a tolerance that starts from about 0.3 for low *min split* (0.0001 %), reaches about 0.6 for intermediate *min split* (0.1 %) and drops to 0.2 for high *min split* (10 %);
- the range of values of the predicted reward is wider for low *min split* (for example, the reward predicted for *min split* = 0.001 % is between -1 and 1), and decreases as the *min split* increases; for *min split* = 10 % the predicted reward is distributed between -0.3 and 0.3.

5.2.2 Feature importance Analysis

Once we created the big table for FQI, we analyzed the importance of the features that we are using.
To do this we used the **IVS** (*Iterative variable selection*) approach [2], a model-free, forward-selection algorithm which is summarized in **Algorithm** 2.
Given the output variable to be explained V^o and the set of candidate variables (see Table 4.2), the IVS algorithm first globally ranks the variables according to a statistical measure of significance (the $feature\ importance$ in our case).
For every iteration, to account for variable redundancy, only the most significant variable V^* is then added to the set of previously selected variables V_{sel}, which is used for building a model $\hat{f}$ to explain V^o.
The algorithm proceeds by repeating the ranking process using as new output variable the residuals of the model built at the previous iteration ($\hat{V}^o \leftarrow V^o$ - $\hat{f}(V_{sel})$). The algorithm iterates these operations until the best variable returned by the ranking algorithm is already in the set V_{sel} or the accuracy of the model built upon the selected variables does not significantly improve.
The accuracy is computed as the coefficient of determination R^2 between the value of the output variable V^o and the value $\hat{V}^o$ predicted by the model.
The IVS approach could be coupled with any VR (Variable Ranking) and MB (Model Building) algorithm, but in our case we use a class of tree-based regression methods, named extremely randomized trees (Extra-Trees) as MB, and the $feature\ importances$ as VR.
For our problem, we apply the FQI algorithm, a batch-mode model-free RL algorithm. FQI translates the RL problem in a sequence of H supervised learning problems, where H is the length of the optimization horizon. Since the supervised learning problems can be solved using any regression algorithm, we have chosen the Extra-Trees, in line with the choices of this thesis.

Algorithm 2 IVS(D, V^o): Iterative Variable Selection

Input: A dataset D, the variable to be explained V^o

Output: V_{sel} : set of variables selected to estimate V^o

Initialize: $V_{sel} \leftarrow \emptyset$, $\hat{V}^o \leftarrow V^o$, $R^2_{old} \leftarrow 0$

repeat

$V^* \leftarrow \underset{V \in D}{\operatorname{argmax}} \, \mathbf{VR}(D, \hat{V}^o, V)$

if $V^* \in V_{sel}$ **then**

return V_{sel}

end if

$V_{sel} \leftarrow V_{sel} \cup V^*$

$\hat{f} \leftarrow \mathbf{MB}(V_{sel}, V^o)$

$\hat{V}^o \leftarrow V^o$ - $\hat{f}(V_{sel})$

$\Delta R^2 \leftarrow R^2(D, V^o, \hat{V}^o)$ - R^2_{old}

$R^2_{old} \leftarrow R^2(D, V^o, \hat{V}^o)$

until $(\Delta R^2 < \epsilon)$

return V_{sel}

In our case the *variables* are the *features* columns in the FQI table (see Table 4.2) and V^o is the reward column of the table.

Variable Ranking (VR)

We used the $scikit-learn$ **feature_importances** method of the Extra-Trees model. Feature importance is calculated as the decrease in node impurity weighted by the probability of reaching that node. The node probability can be computed as the number of samples that reach the node, divided by the total number of samples. The higher the value the more important the feature.
In the Extra-Trees we have a forest of trees (random forest). For each decision tree, $scikit-learn$ calculates a nodes importance using $Gini$ index, assuming only two child nodes (binary tree):

$$ni_j = w_j C_j - w_{left(j)} C_{left(j)} - w_{right(j)} C_{right(j)}$$

- ni_j = the importance of node j
- w_j = weighted number of sample reaching node j
- C_j = the impurity value of node j
- $left(j)$ = child node from left split on node j
- $right(j)$ = child node from right split on node j

The Gini impurity is computed as:

$$\sum_{i=1}^{C} [f_i(1 - f_i)]$$

where f_i is the frequency of label i at a node and C is the number of unique labels.
The importance for each feature on a decision tree is then calculated as:

$$fi_i = \frac{\sum_{j \in \mathcal{F}_i} [ni_j]}{\sum_{j \in \mathcal{N}} [ni_j]}$$

where

- fi_i = the importance of feature i
- ni_j = the importance of node j
- $\mathcal{N}$ = set of all nodes
- $\mathcal{F}_i$ = set of nodes that split on features i

These can then be normalized to a value between 0 and 1 by dividing by the sum of all feature importance values:

$$norm_{fi_i} = \frac{fi_i}{\sum_{j \in \mathcal{F}}[fi_j]}$$

where $\mathcal{F}$ is the set of all features.
The final feature importance, at the Random Forest level, is the average over all the trees. The sum of the features's importance value on eah trees is calculated and divided by the total number of trees:

$$FI(fi_i) = \frac{\sum_{j \in \mathcal{T}}[norm_{fi_{ij}}]}{T}$$

- $FI(fi_i)$ = the importance of feature i calculated from all trees in the Random Forest model (Extra-Trees in our case)
- $norm_{fi_{ij}}$ = the normalized feature importance for i in tree j
- T = total number of trees
- $\mathcal{T}$ = set of all trees

Feature Importance Results

The next plots show how features are chosen following **Algorithm** 2, therefore higher values indicate that the feature was chosen first in the algorithm loop (considering its feature importance).
The values on the ordinates are not between 0 and 1 (like those for the feature importance), but are between 1 and the length of the features.

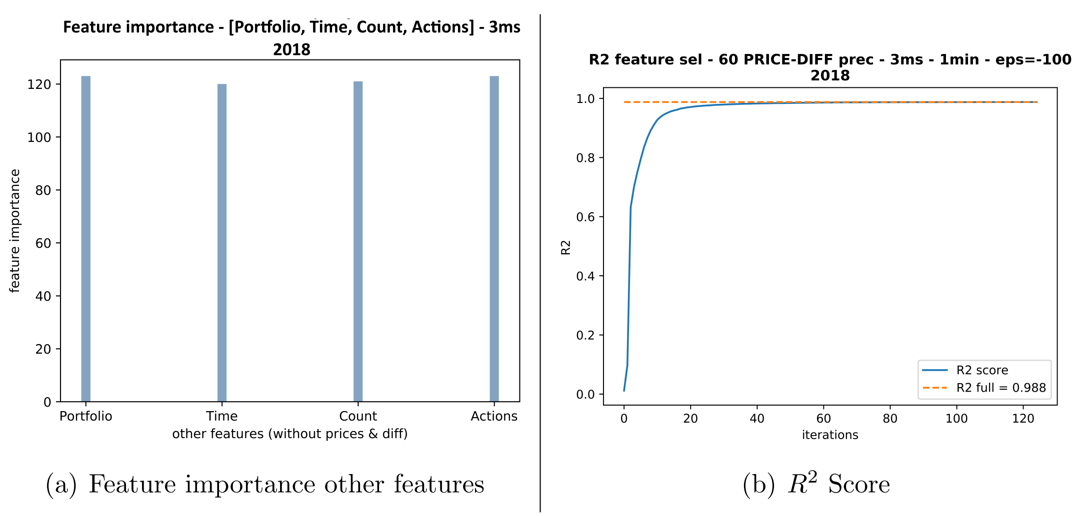
We represent the results for the 2018 dataset, with three *min_split* values of the Extra-trees Regressor, corresponding to 0.0001 %, 0.001 % and 0.01 % of the size of the dataset. Cross-validating with 2 folds, the two R^2 scores obtained were averaged.
We obtained similar results for the other 4 datasets (2014, 2015, 2016 and 2017).
For each of the three *min_split* we have two groups of plots:

- the first contains the feature importance plot for the 60 lagged shifted prices and for the 60 lagged price differences;
- the second contains the feature importance plot for the other features (Portfolio, Time, Count, Actions) and the R^2 plot.

(a) Feature importance 60 prices

(b) Feature importance 60 differences

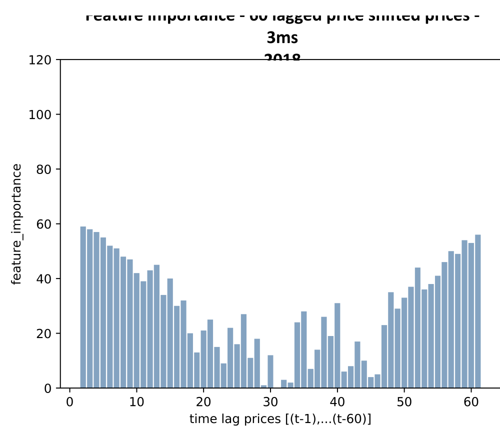
Figure 5.24: Feature importance prices & Feature importance differences - 3 ms - 2018



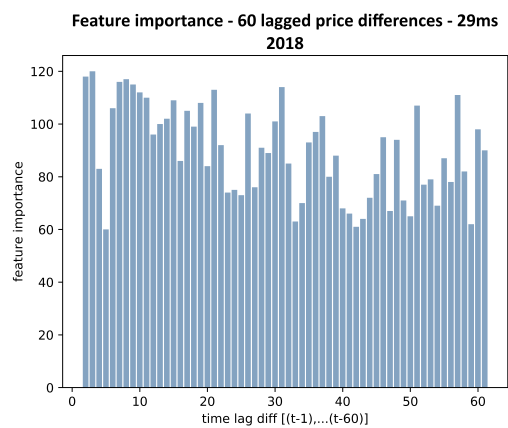
(a) Feature importance other features

(b) R^2 Score

Figure 5.25: Feature importance other features & R^2 Score - 3 ms - 2018: in the R^2 plot the dotted line is the value of the R^2 when we use all the features in the fitting

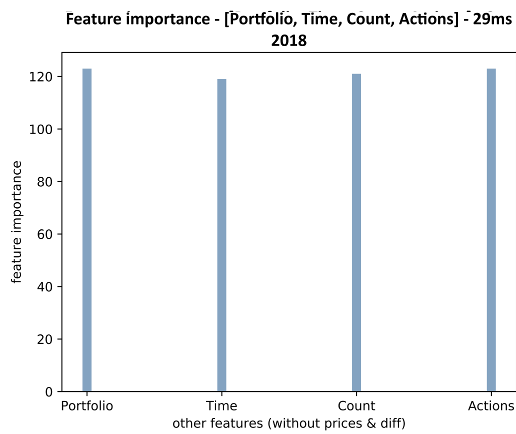


(a) Feature importance 60 prices

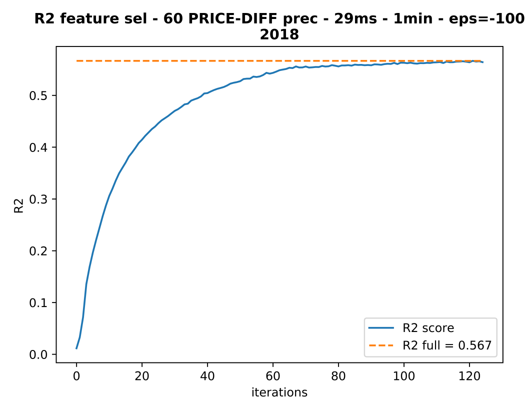


(b) Feature importance 60 differences

Figure 5.26: Feature importance prices & Feature importance differences - 29 ms - 2018

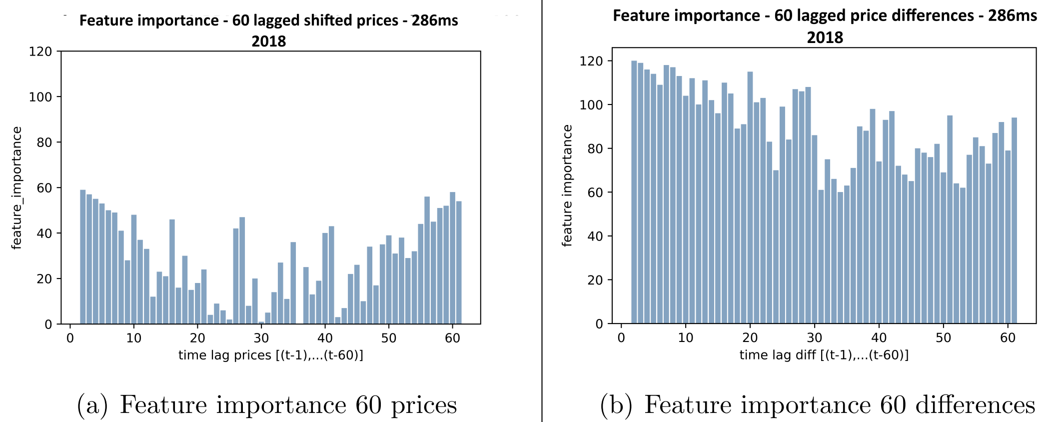


(a) Feature importance other features



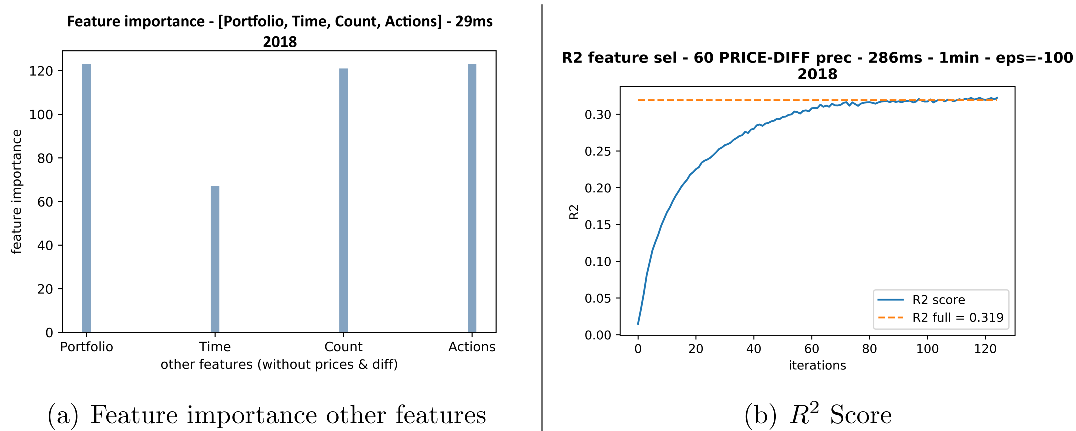
(b) R^2 Score

Figure 5.27: Feature importance other features & R^2 Score - 29 ms - 2018: in the R^2 plot the dotted line is the value of the R^2 when we use all the features in the fitting



(a) Feature importance 60 prices (b) Feature importance 60 differences

Figure 5.28: Feature importance prices & Feature importance differences - 286 ms - 2018



(a) Feature importance other features (b) R^2 Score

Figure 5.29: Feature importance other features & R^2 Score - 286 ms - 2018: in the R^2 plot the dotted line is the value of the R^2 when we use all the features in the fitting

We note that the [Portfolio, Time, Count, Actions] features other than shifted prices and price differences are very important in terms of *feature importance*.
We can see that the price differences are more important than the shifted prices. So in terms of *feature importance*, the change in price instant by instant is more important than the change in price compared to the start of the day.
It is also very important to observe (remembering the results of the stationarity analysis 5.1.1) how the *feature importance* in this case is greater for the features that are stationary (the price differences).

Looking at the R^2, we see how this grows by adding features to the model. For low *min_splits* the R^2 is high, while it tends to go down for higher *min_splits*; this is also confirmed by the analysis of R^2 in train seen earlier (Figure 5.5(a)).

Chapter 6

Experimental Results

In this chapter we will show the main results we obtained with FQI.
Also we will compare these results with others using a standard financial strategy in the FX Market (the daily Buy&Hold) and using a different ML technique (Feed Forward Neural Networks - FFNN).

6.1 Programming Language

Regarding the choice of the programming framework, we opt for the use of Python, because, in the most recent years, it has been established as the main programming language for machine learning projects.
Python offers a wide group of open source libraries that allow to address many practical issues in a smart and efficient way. Furthermore, being an open source language, it allows everyone to exploit the experience of a large and active community of developers. This choice is also motivated by the possibility of using Scikit-Learn, a renowned Python package, which provides a set of useful computational tools and state-of-the-art machine learning algorithms. For more information about those libraries, refer to the official website and http://scikit-learn.org, which offers well documented user guides as well as a large set of code examples.

6.2 FQI Results

6.2.1 Train and Validation

In order to choose the best model, we needed to choose the best combination of the *min split* for the Extra Trees Regressor and the FQI Iterations for the FQI. We vary

only this pair of parameters, because they have a major impact on the model's performance.
The ideal method would be to choose the optimal *min split* for each FQI iteration (in terms of better performance, which in our case we choose to be the average over the whole year of the daily reward) and then choose the best combination of *min split* and FQI iterations. Since this is very expensive in terms of time and computational power, we chose to carry out a series of experiments looking at 5 different *min split* values (from 0,0001% to 10% of the number of rows in the FQI dataset) and 10 values of the FQI iterations (from 1 to 30). Using a large number of fqi iterations would introduce some approximation errors for the Regressor (see [6]).

The key methodology we used is the following:
we trained the model on a year (for example 2016), then we validated the model on the previous year (2015) (so we chose the best combination of *min split* and FQI iterations that give us the maximum of the average daily reward) and finally we tested this model on the next year (2017).
We also followed the same procedure with two years of data for train: for example 2016-2017 for train, 2015 for validation and 2018 for test.
This scheme is illustrated in the following Figure 6.1 and 6.2, in the case of train used only one year of data and in the case of train using two years of data:

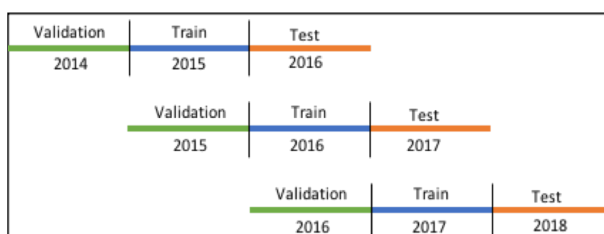


Figure 6.1: Performance evaluation scheme - Train 1y

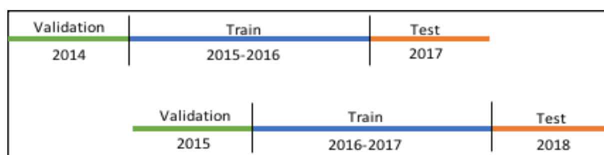


Figure 6.2: Performance evaluation scheme - Train 2y

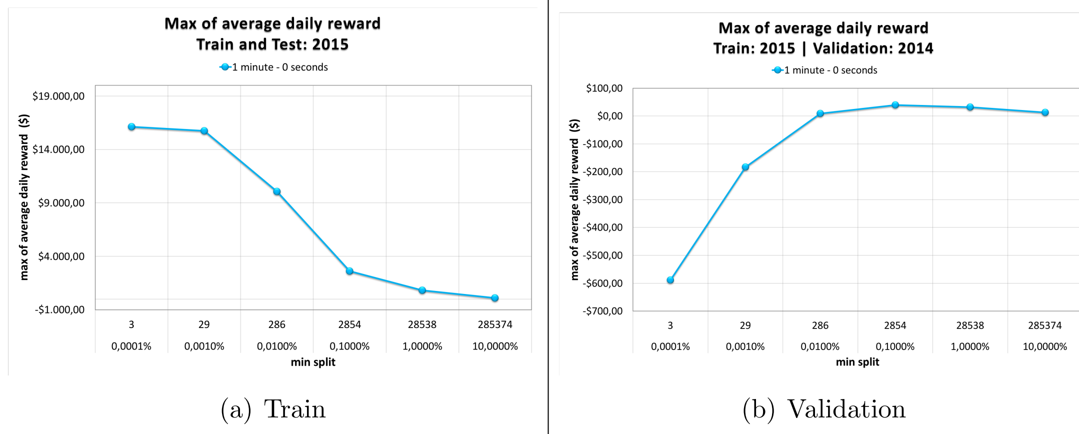
With this methodology we have totally excluded the choice of training the model over years that have similar trends, thus obtaining a model that can generalize more in differnt situations. In particular because of non-stationarity in financial series, it is reasonable to assume that the closer the train set to the test (and validation) set, the

better the performance.
We have chosen to use a maximum of 2 years for train to limit computationl times
About the **testing methodology**, we tested the policy obtained, i.e. the greedy policy for the learned Q-function, on the whole dataset of testing; in particular for each state we applied the Q-function obtained with FQI in the current state for every possible action and we chose the action that gives us the highest Q-function.

Performances have also been evaluated considering a possible **Delay** (DL in Equation 4.2) in the estimation of the trading prices (5, 10 seconds) and a different time discretization in the process, where next states and rewards are observed not only on a 1-minute basis, but also every 5 and 10 minutes (we called this parameter **Action Frequency**) in order to understand the impact of these different time windows. In this way we can be more realistic with the real trading times and also with wider discretizations on one side we decrease the possibility of control, but we make it easier for the algorithm to recognize trends and patterns and learn the best policies (less volatile, incurring less transaction fees).

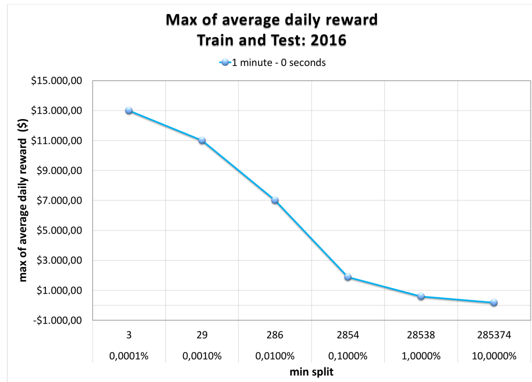
6.2.2 Validation Results with 1 year of Train

Let's first see the performance graphs in train and validation by varying the *min split* and the graphs in validation (for the optimal *min split*) by varying the fqi iterations, with the hypothesis of **instananeous action** (Delay = 0) and Action Frequency = 1 minute, using **1 year** of data for train:

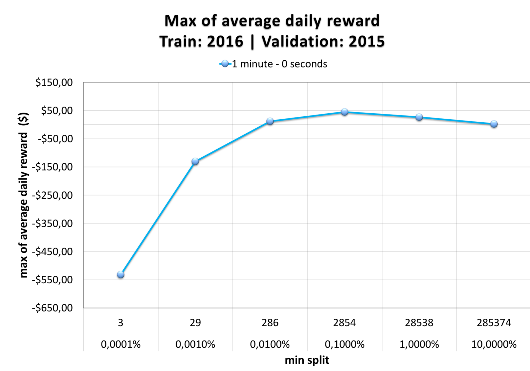


(a) Train (b) Validation

Figure 6.3: Max of average daily reward - Train: 2015 - Validation: 2014 - We took, for each *min split*, the average daily reward and select the best one

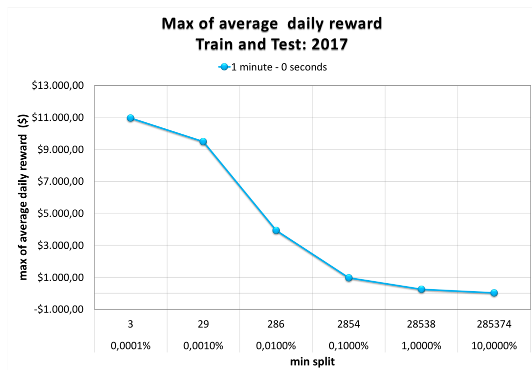


(a) Train

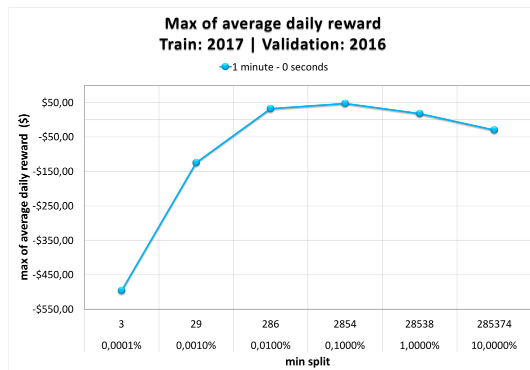


(b) Validation

Figure 6.4: Max of average daily reward - Train: 2016 - Validation: 2015 - We took, for each *min split*, the average daily reward and select the best one



(a) Train



(b) Validation

Figure 6.5: Max of average daily reward - Train: 2017 - Validation: 2016 - We took, for each *min split*, the average daily reward and select the best one

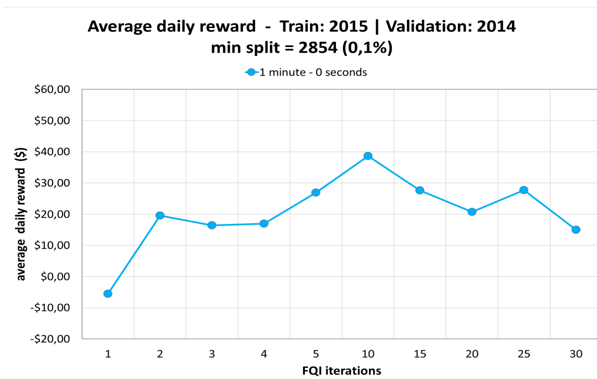


Figure 6.6: Average daily reward - Train: 2015 - Validation: 2014 - minsplit = 2854 (0,1%)

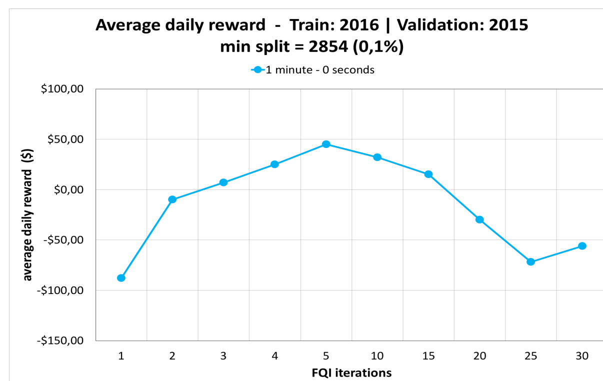


Figure 6.7: Average daily reward - Train: 2016 - Validation: 2015 - minsplit = 2854 (0,1%)

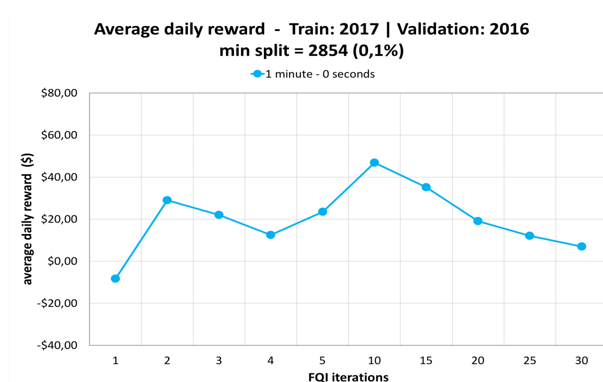


Figure 6.8: Average daily reward - Train: 2017 - Validation: 2016 - minsplit = 2854 (0,1%)

In the case of **1 year dataset**, we can see how for low values of *min split* there is the overfitting effect: very high performance (in terms of max of average daily reward)

in train and low (and negative) in validation. As the *min spli* increases, we have a decreasing trend in train performances and in validation the trend grows, reaches a maximum and then slightly decreases. So, the overfitting effect is attenuated as the *min split* increases.
For all three years in consideration, we observe a maximum of average daily reward when *min split* = 2854, corresponding to 0,1% of the number of rows of the fqi dataset.
Looking then at the graphs of the average reward, for that particular *min split* (0,1%), as the FQI iterations vary, we observe an almost increasing and then decreasing trend with a maximum peak between 5 and 10 iterations.

6.2.3 Validation Results with 2 year of Train

Now we show the performance graphs in validation by varying the *min split* and the graphs in validation (for the optimal *min split*) by varying the FQI iterations, with the hypothesis of **instananeous action** (Delay = 0) and Action Frequency = 1 minute, using **2 year** of data for train:

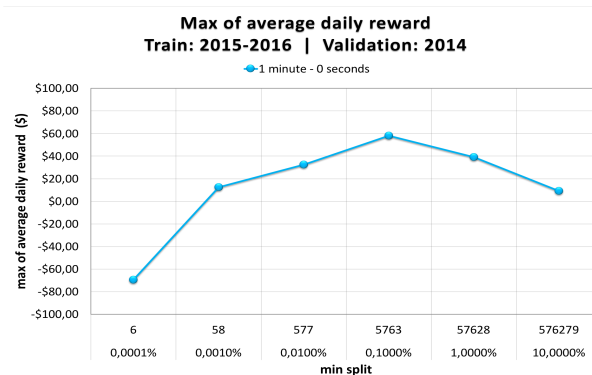


Figure 6.9: Max of average daily reward - Train: 2015-2016 - Validation: 2014 - We took, for each *min split*, the average daily reward and select the best one

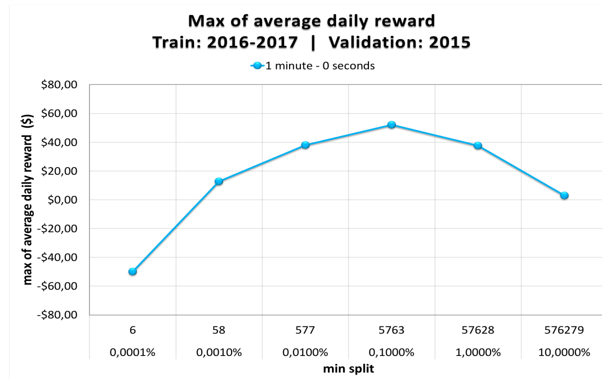


Figure 6.10: Max of average daily reward - Train: 2016-2017 - Validation: 2015 - We took, for each *min split*, the average daily reward and select the best one

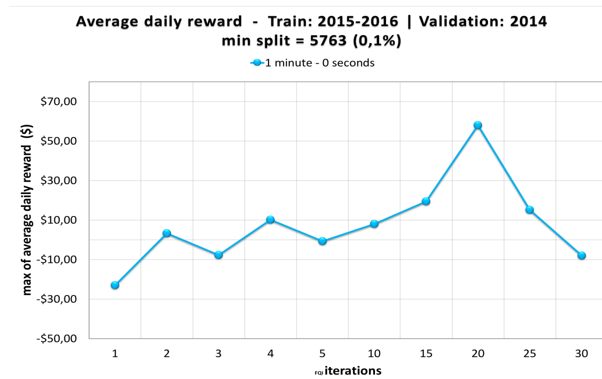


Figure 6.11: Average daily reward - Train: 2015-2016 - Validation: 2014 - minsplit = 5763 (0,1%)

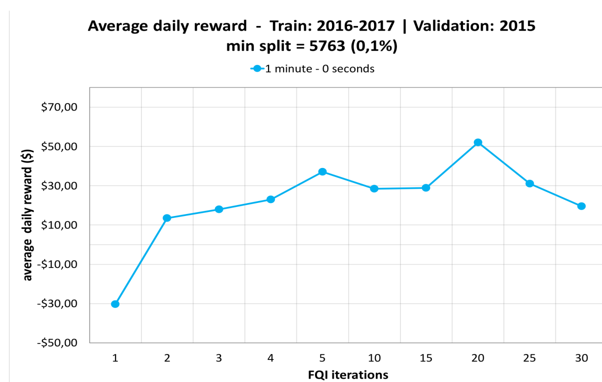


Figure 6.12: Average daily reward - Train: 2016-2017 - Validation: 2015 - minsplit = 5763 (0,1%)

In the case of **2 years dataset**, we observe the same trend in validation (increasing and then decreasing, as the *min split* increases), therefore the performances (in terms of

max of average daily reward) are low for low *min splits*, they grow and reach a maximum peak for *min split* = 5763 (corresponding to 0,1% the number of rows of the dataset fqi, which now is about double that for 1 year) and then decrease by *min split* > 0,1%. Looking at the graphs of the average reward, for *min split* = 0,1% as the FQI iterations vary, we observe an increasing and then decreasing trend with a maximum peak for iterations = 20.

So, with this range of *min split* and FQI iterations, it is observed that 0,1% of the length of the FQI dataset and FQI iterations between 5 and 10, (for 1 year train), and FQI iterations = 20 (for 2 year train), seem to be the combinations of *min split* and FQI iterations which allow us to obtain good performances (in terms of average daily reward) in validation. We summarize these results in the following Table 6.1:

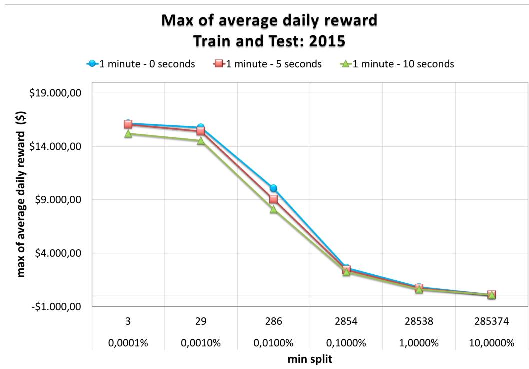
	min split	FQI iterations
Train: 2015 - Validation: 2014 %	2854 (0,1%)	10
Train: 2016 - Validation: 2015 %	2854 (0,1%)	5
Train: 2017 - Validation: 2016 %	2854 (0,1%)	10
Train: 2015-2016 - Validation: 2014 %	5763 (0,1%)	20
Train: 2016-2017 - Validation: 2015 %	5763 (0,1%)	20

Table 6.1: optimal (*min split*, FQI iterations) pairs

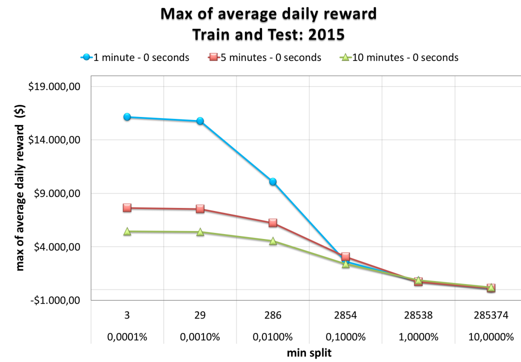
The optimal *min split* and FQI iterations values with 2 year train are almost double those with 1 year train. Computational times double in the 1 year train case, compared to the 2 year case.

Results with different Delay and Interaction Frequency

Now we show the performance graphs in train (only for 2015) and validation (only for 2014) by varying the *min split* and the graphs in validation (for the optimal *min split*) by varying the FQI iterations, for different values of the Delay (0, 5, 10 seconds) and the Action Frequency (1, 5, 10 minutes):

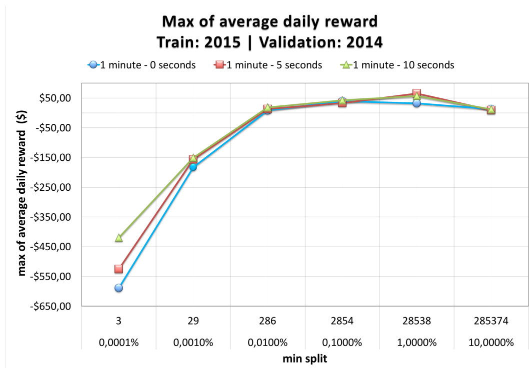


(a) different Delay

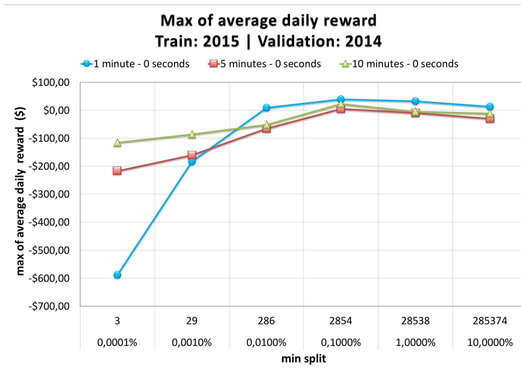


(b) different Action Frequency

Figure 6.13: Max of average daily reward - Train and Test: 2015 - different Delay and Action Frequency - We took, for each *min split*, the average daily reward and select the best one



(a) different Delay



(b) different Action Frequency

Figure 6.14: Max of average daily reward - Train: 2015 - Validation: 2014 - different Delay and Action Frequency - We took, for each *min split*, the average daily reward and select the best one

We consider the **standard case** the one with Delay = 0 seconds (instant action, without delay) and Action Frequency = 1 minute (the agent looks at the next state and compute the reward at the next minute).
From these last four graphs we can see how the trends (decreasing in train and domed in validation) also remain for Action Frequency > 1 minute and Delay > 0 seconds. So the overfitting effect remains for low *min splits*.
In train we see how the performance (in terms of max of average daily reward) for *min splits* < 0,1% are worse both for Action Frequency = 5 and 10 minutes and for Delay = 5 and 10 seconds, compared to the standard case, while they tend to be almost equal for *min split* ≥ 0,1%; this makes us understand that in train the fact of considering the next state and the reward not every minute (with Action Frequency > 1 minute)

and to delay the action (assuming a non-instantaneous action, but at the next 5 and 10 seconds) only affects for low *min split* values, at which, in test there are low and negative performances.
In validation we observe slightly better performances (in terms of max of average daily reward) with Delay = 5 and 10 seconds, moving the maximum to *min split* = 1%, while for *min split* equal to 0,01%, 0,1% and 10% they have the same performances.
With Action Frequency = 5 and 10 minutes, compared to the standard case, we observe better (but always negative) performances for the first two *min split* values and then worse (but close) perforamnces.

So overall we see how a delay in the estimate of the buy/sell price does not significantly affect performance, while looking at the next state and calculating the reward at 5 and 10 minutes slightly impacts negatively on performance. Hence, even incurring in higher computational times, the benefits of selecting a finer time discretization are clear.

6.2.4 FQI Test Results

Now we show the results obtained by testing the best models chosen in validation (synthetized in Table 6.1). In particular, we show the actions chosen and the cumulative rewards obtained from the best agents
These results were obtained by not considering the drawdown[1], not stopping the actions if a certain level of loss is reached. We decided to plot the actions with an heatmap in order to compare the actions at the same times of different days.

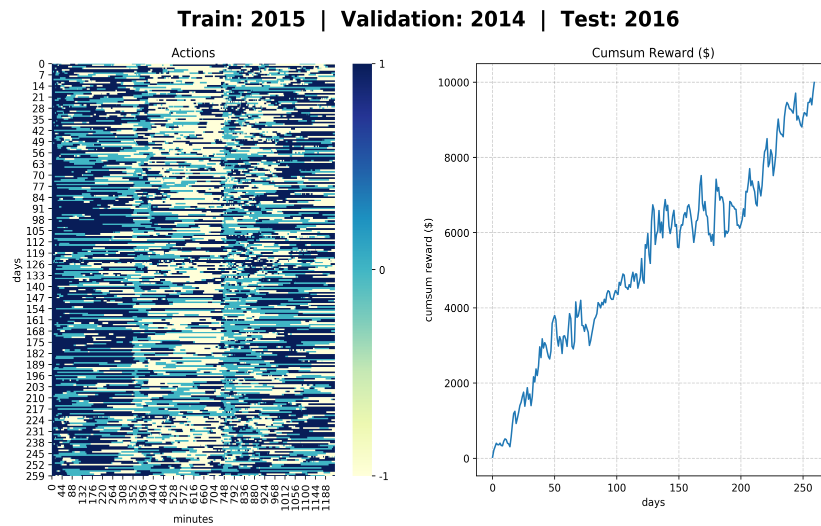


Figure 6.15: Train: 2015 - Validation: 2014 - Test: 2016 - minsplit = 2854 (0,1%) - FQI iteration = 10 - From the Actions we can see how there are similar trends in certain time slots; for most days of the year, at the beginning of the day the agent tends to buy, then remains flat for a few minutes, then sells and returns to buy at the end of the day. This behavior is typical in the FX market and it is known as **intraday seasonality**. The cumulative reward seems to be stable and growing, with a bit of variance after the middle of the year. We have a final cumulative return of almost 10% (10000 \$ over an invested capital of 100000 €($\simeq$ 100000 \$)).

[1] drawdown is the measure of the decline from a historical peak in some variable (in our case it specified the maximum loss of trading capital)

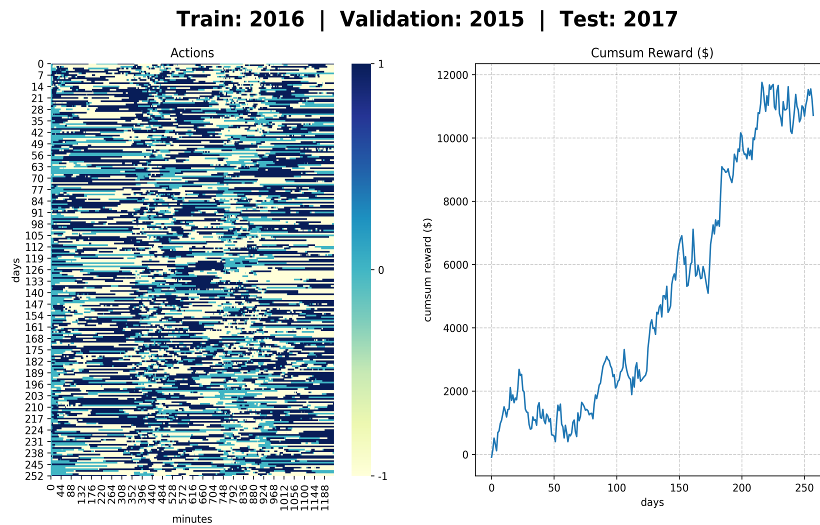


Figure 6.16: Train: 2016 - Validation: 2015 - Test: 2017 - minsplit = 2854 (0,1%) - FQI iteration = 5 - We can observe similar actions at the beginning of the day, for a few minutes (almost always flat), then it is not possible to recognize a particular pattern common to almost every days (as happened in the preiovus case in Figure 6.15). The cumulative reward has a strong growth in the first month, then remain almost stable and constant for the following month and then strongly grows until the penultimate month of the year; in the last month there is some variance without growth. We have a final cumulative return of almost 11% (11000 $ over an invested capital of 100000 €($\simeq$ 100000 $)).

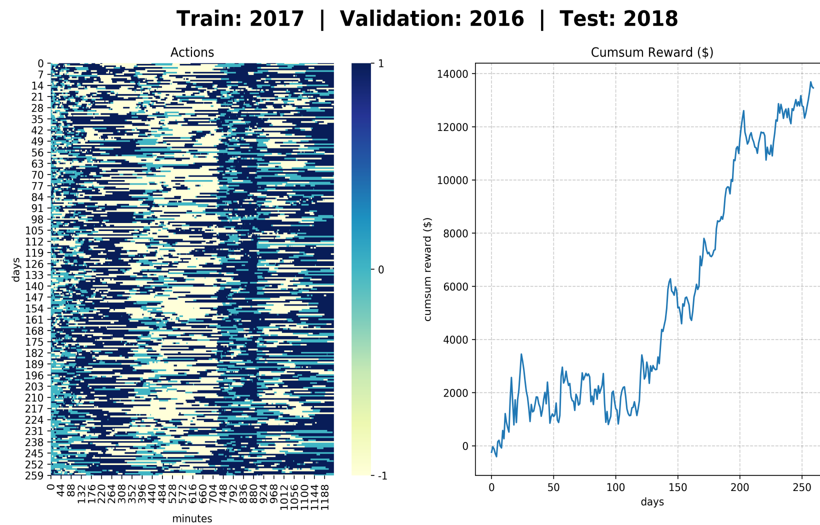


Figure 6.17: Train: 2017 - Validation: 2016 - Test: 2018 - minsplit = 2854 (0,1%) - FQI iteration = 10 - We can see a more intraday seasonality behavior. In the cumulative reward there is a bit of variance in the first three months and then a strong growth, with a final cumulative return of almost 14% (14000 $ over an invested capital of 100000 €($\simeq$ 100000 $)).

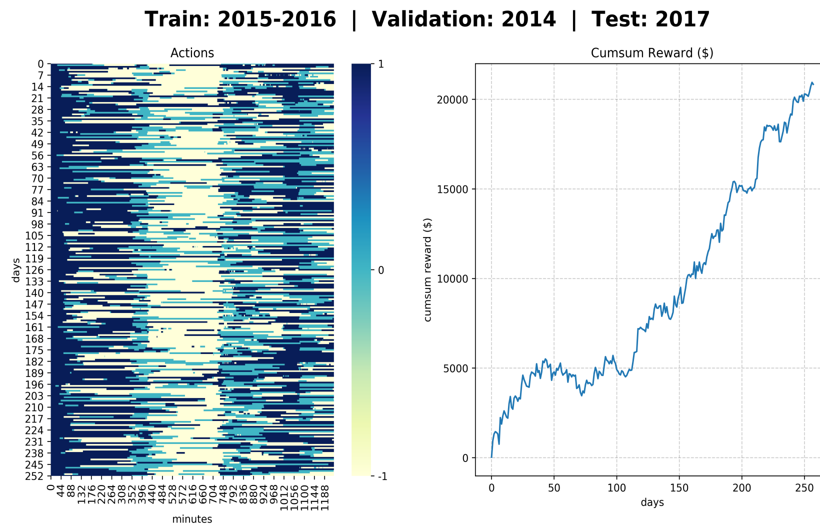


Figure 6.18: Train: 2015-2016 - Validation: 2014 - Test: 2017 - minsplit = 5763 (0,1%) - FQI iteration = 20 - We can see a clear intraday seasonality behavior in the actions which tend not to change as often as in the case with train over 1 year. We can see a stable growth in the cumulative reward with a final cumulative return of almost 21% (21000 \$ over an invested capital of 100000 €($\simeq$ 100000 \$)), which compared with the previous results with train over 1 year (10%, 11% and 14% final cumulative return) shows that the train over 2 year allows to obtain better performances in terms of final cumulative return.

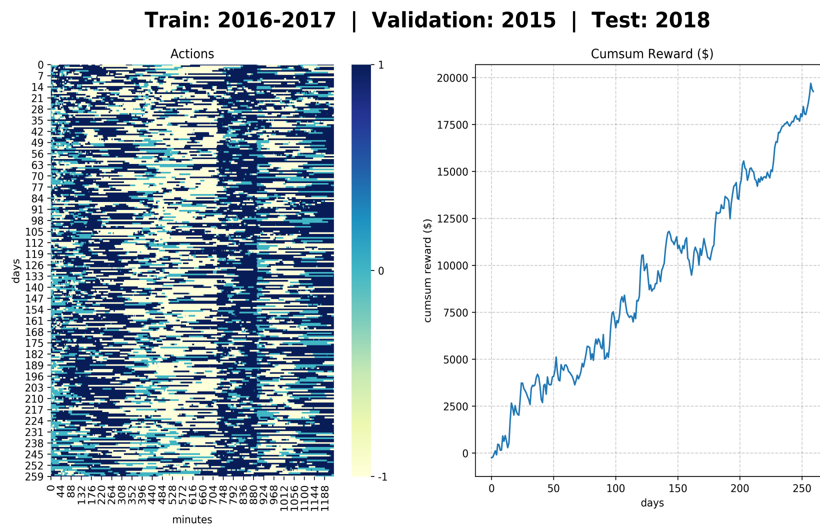


Figure 6.19: Train: 2016-2017 - Validation: 2015 - Test: 2018 - minsplit = 5763 (0,1%) - FQI iteration = 20 - We can see a intraday seasonality behavior in the actions with some noise compared with the previous case in Figure 6.18. Trend of the cumulative reward is growing and more linear than in the previous case in Figure 6.18, reaching a final cumulative return of almost 20% (20000 \$ over an invested capital of 100000 €($\simeq$ 100000 \$)).

6.2.5 FQI vs Buy&Hold Results

To understand how good our results are with FQI, we decided to compare them with those obtained by applying a standard strategy in the FX Market, the daily Buy&Hold. In the **daily Buy&Hold** strategy, a single purchase (buy order) is made at the beginning of the day and the position is closed at the end of the day; this is repeated for all days of the year.
In our case we assume that we are investing the same 100000 € capital every day.
We compare the results in terms of cumulative returns and we also look at the average and the standard deviation of the daily returns (see Table 6.2).

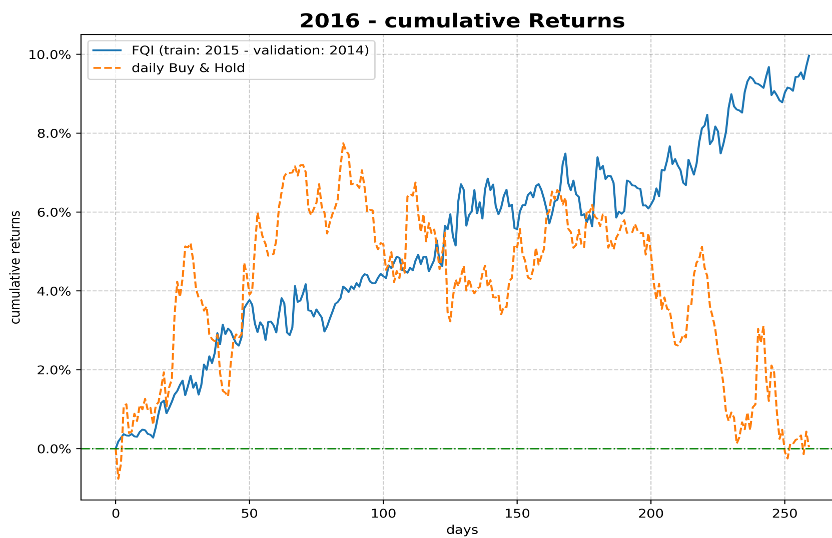


Figure 6.20: FQI vs daily Buy&Hold - Test: 2016 (Train 1y) - FQI outperforms daily B&H. We have a cumulative return of almost 10% with FQI and almost 0% with daily B&H.

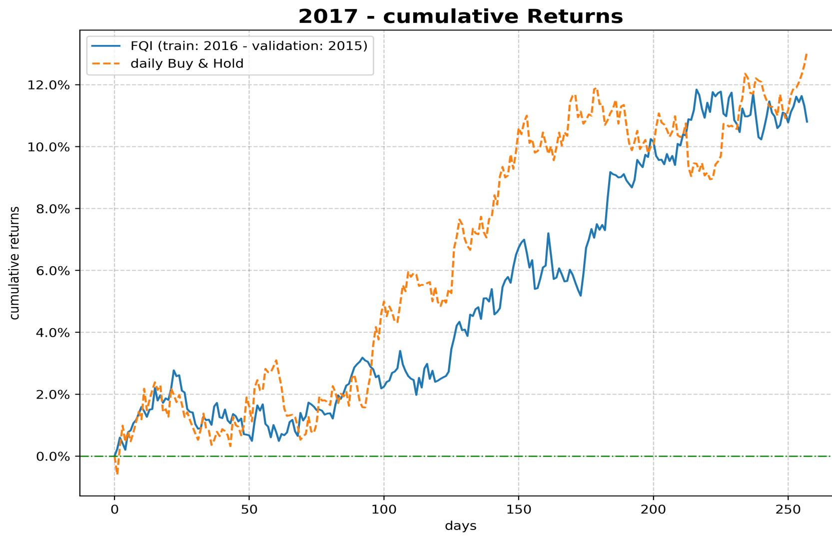


Figure 6.21: FQI vs daily Buy&Hold - Test: 2017 (Train 1y) - FQI slightly underperforms B&H, even if the final cumulative returns are very close (almost 11% for FQI and almost 13% for daily B&H).

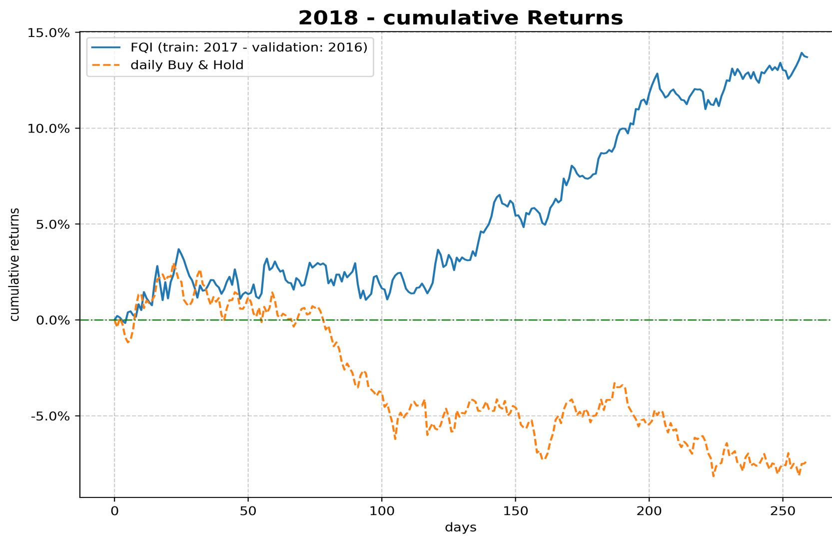


Figure 6.22: FQI vs daily Buy&Hold - Test: 2018 (Train 1y) - FQI outperforms daily B&H, which performs negatively since the second month of the year.

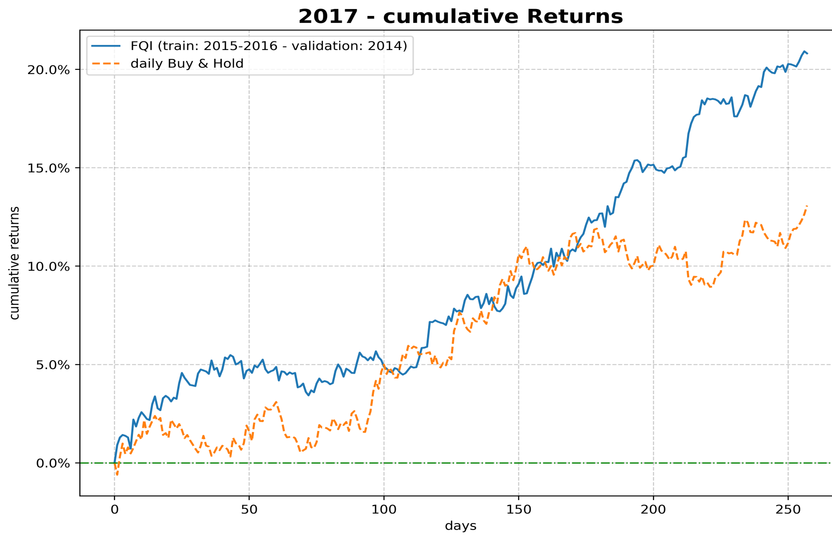


Figure 6.23: FQI vs daily Buy&Hold - Test: 2017 (Train 2y) - FQI outperforms daily B&H by almost 7% of cumulative returns.

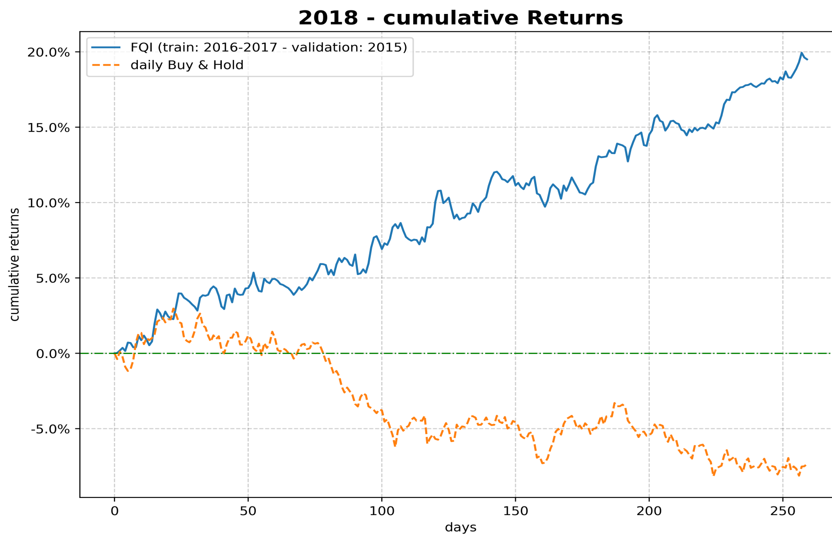


Figure 6.24: FQI vs daily Buy&Hold - Test: 2018 (Train 2y) - FQI strongly outperforms daily B&H, which performs negatively since the second month of the year.

	mean FQI	mean B&H	std FQI	std B&H
Train: 2015 - Test: 2016	0,038 %	-0,00039 %	0,31 %	0,51 %
Train: 2016 - Test: 2017	0,041 %	0,048 %	0,34 %	0,43 %
Train: 2017 - Test: 2018	0,051 %	-0,027 %	0,41 %	0,44 %
Train: 2015-16 - Test: 2017	0,081 %	0,048 %	0,36 %	0,43 %
Train: 2016-17 - Test: 2018	0,074 %	-0,027 %	0,42 %	0,44 %

Table 6.2: mean and std daily reward - FQI vs daily Buy&Hold

Overall FQI performs better than daily B&H; from the previous graphs and Table 6.2 we can observe that with train on 1 year, FQI performs slightly worse than daily B&H only in the case with test on 2017. 2017 had an overall growing trend and training only on 2016 and validation on 2016 is not enough to overcome such baseline. With train on 2 years, FQI always performs better than daily B&H both in the case of test on 2017, taking advantage of the growing trend in the environment and both in the case of test on 2018 where the trend is decreasing and negative.
In both cases of train on 1 year and train on 2 years, the standard deviation of the daily returns with FQI is always lower than the one with daily B&H. This is very important from a financial point of view, allowing for more stable returns on an entire year of investment. Having a low standard deviation allows an investor to start the trade at any time of the year, obtaining positive, growing and stable returns.

6.2.6 FQI vs FFNN Results

In this section we will show the performances achieved using FFNN (Feed Forward Neural Networks) on the same €/$ pair. The results are part of the Thesis [11]. The goal of that work was to assess to what extent deep learning methodologies are able to perform whtn dealing with highly liquid financial assets' returns prediction. In the application of the FFNN to the problem of predicting financial returns, they wanted to compare the performance achieved adopting a non-standard heteroskedastic loss function. They adopted a supervised learning approach, where the system outputs a prediction on the asset return on a specific time horizon, rather than an entire trading strategy: they decouple the forecast of expected future return from the choice of the optimal trading strategy conditioned to this forecast, which is managed separately. The time horizon over which to make the forecasts is a model parameter to be set externally. They tried to assess the network's performance including transaction costs and emulating what the real-life trading activity wuold have been through a walk forward backtesting procedure. Using this procedure the overall time span is split into many consecutive windows of

training, validation and test periods: they have a time window on which the training of different learning models is carried out, followed by a validation period that allows the selection of the most suitable model and, finally a test window, which represents the time span in which the actual trading activity wuold have taken place. The time windows are advanced after each train-validation-test cycle, so that test windows are consecutive and not overlapped. They used a 40-week long training windows followed by 4-week long validation and test windows.
In particular, they trained a set of neural networks, using reasonable hyperparameters configurations. Specifically, all the hyperparameters of these networks are fixed, except for the L2 regularization coefficient: these models, more or less regularized, are trained on each training window and evaluated on the following validation periods. At this point, the validated model is used to carry out the trading activity on the corresponding test window. The entire trading period goes from 2010-11-08 to 2018-01-21. Refer to [11] for more details.
Because in our work we have test results from 2016 to 2018 and the FFNN work had results from 2011 to 2017, we can compare the resuls (in terms of end of the year cumulative returns) only for 2016 and 2017 and we show them in Table 6.3:

	FQI (train 1y)	FQI (train 2y)	FFNN
2016	∼10%	∼21%	∼15%
2017	∼11%	∼20%	∼0%

Table 6.3: Results (end of the year cumulative returns) - FQI vs FFNN

As we can see in Table 6.3, FQI outperforms FFNN in the case of train on 2 years. Looking at Chapter 3 of [11] there are no results about the standard deviation of the daily return, but from the plot of the cumulative returns we can see that in 2016 and 2017 they had big drawdowns, unlike FQI where cumulative returns have a growing and stable trend. Since they used almost 7 years of testing, they computed the average yearly return, obtaining that the best performing model was able to reach almost 5% yearly return.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis (part of a project in collaboration with the AGS SpA - Advanced Global Solution company) we applied Fitted Q Iteration (FQI) algorithm, a Reinforcement Learning technique to Forex trading, using the €/$ pair datasets from 2014 to 2018.
In a first analysis of the original dataset, by looking at the mean and variance of two subsets of the series and by using the ADF (augmented Dickey Fuller) test, we observed that all the series were not stationary, while the differences (current price minus previous price) were stationary.
As the FQI algorithm needs an input table, we created it and we did a regression and classification analysis (using the cross validation technique) and a feature importance analysis. We compare the results varying the *min split*, an important parameter of the regressor in the FQI. The R^2 was decreasing in train, with high values for low *min split* and low values for high *min split*; the opposite in test, but with the difference that in test the R^2 values were negative and near zero. The MSE values were increasing in train and almost costant in test, with in general very low values. The *accuracy* score values were decreasing in train and test, with lower values in test; but overall these values were greater than 0.5. Therefore the model was more accurate in predicting the reward sign, rather than variance. So it is more important to understand if the price will go up or down in the next step, rather than knowing for sure how much. Looking then at the confusion matrices and the scatter plots (true reward vs predicted one) we saw how in general the results get worse as the *min split* increases.
From the feature importance analysis we understood that the price differences were more important than the shifted prices; so it seems that the model prefers to use stationary features (remembering that the price differences series were stationary). Furthermore, we have seen that in train the R2 increases as the number of features in the model

increases.
In order to choose the best model we trained and validated each models obtain in train, for different values of *min split* and FQI iterations, choosing the best combination, computing the average of daily reward and then take the maximum. We found that the best *min split* was 0.1% of the length of the FQI dataset and the best FQI iterations were between 5 and 10 for the case with train on 1 year and equal to 20 for the case with train on 2 year.
We also evaluated the performances (in terms of maximum of average daily reward) varying two parameter: the Delay in the estimate of the buy/sell price and the Action Frequency that represents a different discretization of the MDP. We saw that delay not significantly improve the performance and the action frequency slightly impacts negatively, preferring a finer time discretization.
We then tested (without stopping the actions to avoid losses, so without a stop-loss strategy) the best models. We obtained positive results (in terms of cumulative returns) in the case of train on 1 year with final cumulative return of 10% testing on 2016, 11% testing on 2017 and almost 14% testing on 2018. These results were better in the case of train on 2 years, with final cumulative return of almost 20% in both cases of testing on 2017 and 2018. In addition, these FQI results exceed (especially in the case of train on 2 years) those obtained by applying a standard strategy of daily Buy&Hold and, except in one case of train on 1 year, those obtained by applying a different ML technique (Feed Forward Neural Network - FFNN).
Both in the case of train on 1 year and especially in the case of train on 2 years we observed the intraday seasonality behavior in the actions chosen by the agent, a behavior already found in the FX market (see [15]).

7.2 Future Work

We propose a series of future improvements of this work:

- Since we have seen the effect of seasonality in the actions, it would be interesting to investigate this behavior in detail in order to create investment strategies that exploit this seasonality in the FX market.
- An improvement in order to reduce losses and obtain better performances in terms of cumulative returns could be to add a stop-loss strategy; therefore consider a drawdown threshold and block the actions if this threshold is reached.
- In this thesis we have performed a tuning of the hyperparameters (*min split* and FQI iterations) only looking at some combinations, in order to avoid huge computational time. Having more computational power available, we could perform

a tuning with greater discretization and maybe get combinations of parameters (which with our tuning we have not been able to see) that could lead to better performances.

- In this work we used only a pair (€/$); one or more pairs could be added (in particular on markets operating in different time slots), carrying out different combinations of trains and validations; this could lead to better performances, taking advantage of the different trading hours.

Bibliography

[1] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[2] Andrea Castelletti, Stefano Galelli, Marcello Restelli, and Rodolfo Soncini-Sessa. “Tree-based variable selection for dimensionality reduction of large-scale control systems”. In: *2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*. IEEE. 2011, pp. 62–69.

[3] Michael AH Dempster and Vasco Leemans. “An automated FX trading system using adaptive reinforcement learning”. In: *Expert Systems with Applications* 30.3 (2006), pp. 543–552.

[4] Michael Alan Howarth Dempster and Yazann S Romahi. “Intraday FX trading: An evolutionary reinforcement learning approach”. In: *International Conference on Intelligent Data Engineering and Automated Learning*. Springer. 2002, pp. 347–358.

[5] Damien Ernst, Pierre Geurts, and Louis Wehenkel. “Iteratively extending time horizon reinforcement learning”. In: *European Conference on Machine Learning*. Springer. 2003, pp. 96–107.

[6] Damien Ernst, Pierre Geurts, and Louis Wehenkel. “Tree-based batch mode reinforcement learning”. In: *Journal of Machine Learning Research* 6.Apr (2005), pp. 503–556.

[7] Xiu Gao and Laiwan Chan. “An algorithm for trading and portfolio management using Q-learning and sharpe ratio maximization”. In: *Proceedings of the international conference on neural information processing*. 2000, pp. 832–837.

[8] Pierre Geurts, Damien Ernst, and Louis Wehenkel. “Extremely randomized trees”. In: *Machine learning* 63.1 (2006), pp. 3–42.

[9] Carl Gold. "FX trading via recurrent reinforcement learning". In: *2003 IEEE International Conference on Computational Intelligence for Financial Engineering, 2003. Proceedings.* IEEE. 2003, pp. 363–370.

[10] Geoffrey J Gordon. *Approximate solutions to Markov decision processes.* Tech. rep. Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1999.

[11] Stefano Grassi. "Artificial Neural Networks Application to Financial Markets". 2018.

[12] William H Greene. "Econometric analysis 4th edition". In: *International edition, New Jersey: Prentice Hall* (2000), pp. 201–215.

[13] Frederick Hayes-Roth. "Review of" Adaptation in Natural and Artificial Systems by John H. Holland", The U. of Michigan Press, 1975". In: *ACM SIGART Bulletin* 53 (1975), pp. 15–15.

[14] Tin Kam Ho. "Random decision forests". In: *Proceedings of 3rd international conference on document analysis and recognition.* Vol. 1. IEEE. 1995, pp. 278–282.

[15] Takatoshi Ito and Yuko Hashimoto. "Intraday seasonality in activities of the foreign exchange markets: Evidence from the electronic broking system". In: *Journal of the Japanese and International Economies* 20.4 (2006), pp. 637–664.

[16] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[17] Jae Won Lee and O Jangmin. "A multi-agent Q-learning framework for optimizing stock trading systems". In: *International Conference on Database and Expert Systems Applications.* Springer. 2002, pp. 153–162.

[18] Andy Liaw, Matthew Wiener, et al. "Classification and regression by randomForest". In: *R news* 2.3 (2002), pp. 18–22.

[19] Tom M Mitchell et al. *Machine learning.* 1997.

[20] John Moody and Matthew Saffell. "Learning to trade via direct reinforcement". In: *IEEE transactions on neural Networks* 12.4 (2001), pp. 875–889.

[21] John Moody and Lizhong Wu. "Optimization of trading systems and portfolios". In: *Proceedings of the IEEE/IAFE 1997 Computational Intelligence for Financial Engineering (CIFEr).* IEEE. 1997, pp. 300–307.

[22] Dirk Ormoneit and Śaunak Sen. “Kernel-based reinforcement learning”. In: *Machine learning* 49.2-3 (2002), pp. 161–178.

[23] Bank for international settlements. “Triennal Central Bank Survey - Foreign exchange turnover in April 2019”. In: (2019).

[24] Richard S Sutton and Andrew G Barto. “Reinforcement learning: An introduction”. In: (2011).

[25] C Watkins. *Learning from Delayed Rewards. PhD thesis, University of Cambridge, Cambridge, England.* 1989.