UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Facultat d'Informàtica de Barcelona

POLYTECHNIC UNIVERSITY OF CATALONIA (UPC) - BARCELONATECH

MASTER THESIS

A machine learning approach to stock screening with fundamental analysis

Author:
Pol ÁLVAREZ VECINO

Supervisor:
Prof. Argimiro A. ARRATIA QUESADA

A thesis submitted in fulfillment of the requirements for the degree of Master in Innovation and Research in Informatics

in the

Barcelona School of Informatics (FIB)

April 15, 2019

Declaration of Authorship

I, Pol ÁLVAREZ VECINO, declare that this thesis titled, "A machine learning approach to stock screening with fundamental analysis" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it."

Frank Herbert

POLYTECHNIC UNIVERSITY OF CATALONIA (UPC) - BARCELONATECH

Abstract

Barcelona School of Informatics (FIB)

Master in Innovation and Research in Informatics

A machine learning approach to stock screening with fundamental analysis

by Pol ÁLVAREZ VECINO

We present *HPC.FASSR*, a High-Performance Computation Fundamental Analysis Stock Screening and Ranking system built on PyCOMPSs, to compare the performance of various supervised learning algorithms like neural networks, random forests, support vectors machines, or AdaBoost, and the criteria of famous expert trader Benjamin Graham for selecting stocks based on fundamental factors. We perform three experiments using financial data from companies of the S&P 500 Index. First, we compare Graham's criteria with classification models in a stock screening scenario trading only long positions. Second, we examine the performance of regression against classification models, also in stock screening but allowing short positions. Finally, we use the predictions of the regression models to perform stock ranking instead of just stock screening. The results show that finding the right parametrizations for the models is critical to get the highest returns, but this requires significant amount of computing resources. Without the proper configuration, some models do not outperform the index, as Graham consistently does, or even manage to get into debt in scenarios where shorting is allowed. On the other, most models outperform both Graham's criteria and the index and the best configurations multiply the initial investment tenfold in stock screening, and almost by 200 in stock ranking. The parallelization of *HPC.FASSR* with PyCOMPSs allows us to explore a vast number of configurations in a short time. We evaluate its performance in MareNostrum 4, the main supercomputer in the Barcelona Supercomputing Center, running with up to 1500 CPUs and training more than 50K models.

Acknowledgements

I wish to express my sincere thanks to Argimiro Arratia, supervisor of this project, for giving me the freedom to explore as I pleased while always finding the correct piece of advice to set me back into the right track when I went astray, and putting up with my spontaneous periods of disconnection.

Besides my advisor, I would like to thank Prof. Rosa Maria Badia Sala for supporting my work on this project and trusting my criteria.

I take this opportunity to express also gratitude to all of the Workflows and Distributed Computing members for their help and support, specially Ramon Amela and Sergio Rodríguez whose work on the scheduler and the python bindings has been crucial for the awesome results of the scalability tests.

Finally, I also want to express my gratitude towards my parents, Andreu Alvarez and Carmen Vecino, and friends, Jordi Molina and Quim Romero, who suffered my occasional rants and encouraged me enough to cross the finishing line.

Contents

List of Figures

List of Tables

List of Abbreviations

AI	**A**rtificial **I**ntelligence
API	**A**pplication **P**rogramming **I**nterface
BVPS	**B**ook **V**alue **P**er **S**hare
EBIT	**E**arnings **B**efore **I**nterest and **T**axes
EPS	**E**arnings **P**er **S**hare
IB	**I**nteractive **B**rokers
LBFGS	**L**imited-memory **B**royden-**F**letcher-**G**oldfarb-**S**hanno algorithm
MN4	**M**are**N**ostrum 4
ML	**Machine Learning**
OSS	**O**pen-**S**ource **S**oftware
P/E	**P**rice to **E**arnings ratio
P/B	**P**rice to **B**ook ratio
RBF	**R**adial **B**asis **F**unction
ReLU	**Re**ctified **L**inear **U**nit
ROA	**R**eturn **O**n **A**ssets
ROC	**R**eturn **O**n **C**apital
ROE	**R**eturn **O**n **E**quity
ROIC	**R**eturn **O**n **I**nvested **Capital**
SIC	**S**tandard **I**ndustrial **C**lassification
S&P 500	**S**tandard and **P**oor **500** index
SVM	**S**upport **V**ector **M**achines
TANH	**H**yperbolic **TAN**gent
USD	**U**nited **S**tates **D**ollar
WNSO	**W**eighted average **N**umber of **S**hares **O**utstanding

Chapter 1

Introduction

1.1 Motivation

Machine learning has seen a steady increase in industrial application in recent years. The amount of useful information in many knowledge work disciplines is too large to be analyzed without ML models or similar techniques. However, currently used ML techniques are part of the so-called weak artificial intelligence (weak AI) or narrow AI. Weak AI techniques are focused on solving a very narrow task, so they do not generalize well. This kind of AI models need to be specially tailored to the specific task they try to solve. Stock price prediction is no exception. The financial gains promised by improving existing methods have led to a high number of scientific papers applying ML techniques to stock market forecasting and automating financial decision making in general [11] [21] [5]. However, each paper proposes a different set of features, targets, and evaluation methods, so its difficult to compare their results.

Explanatory variables for financial prediction are often drawn from technical and fundamental analysis, two disciplines of quantitative analysis which require a solid base knowledge and significant experience in order to draw accurate conclusions. On these types of information, rely most criteria of expert traders. In this project, we offer a comparison between the criteria of the great Benjamin Graham, the father of fundamental analysis investment methodology, and the most common ML models. In order to address the issue of finding the best configuration for the model, we need a system capable of evaluating a massive number of models and be extensible enough to try different techniques and approaches.

We propose *HPC.FASSR* , a High-Performance Computing Fundamental Analysis Stock Screening and Ranking system, powered by PyCOMPSs, to compare the performance of various supervised learning algorithms like neural networks, random forests, support vectors machines, or AdaBoost, and well-known human expert trader's criteria for selecting stocks based on fundamental factors. The parallelization of *HPC.FASSR* with PyCOMPSs allows us to explore a vast number of configurations in a short time.

The project objectives are:

1. Compare stock screening results using human expert's rules against machine learning models.

2. Compare the performance, in terms of revenue, when considering the task of stock screening either as a regression or classification problem.

3. Determine if using regression information to do stock ranking yields better revenues results than stock screening for trading.

4. Develop a High-Performance Computing system able to explore a significant number of models and datasets.

5. Use the HPC system to test the contribution to performance (in results and computational resources) of different configurations for the models, data sets and trading.

1.2 Document Structure

The rest of the thesis is structured as follows. Chapter 2 discusses some related work. Chapter 3 describes the data sources and preprocessing, the models explored, and the experimental setup and evaluation. Chapter 4 contains a short overview of PyCOMPSs parallelization framework, how it has been used to distribute the execution of *HPC.FASSR* , and which performance tests have been executed to evaluate it. Chapter 5 presents and discusses the results of the performance and trading experiments. Chapter 6 contains the conclusions and, in Chapter 7, we point out some research lines and possible features for further work on *HPC.FASSR* .

1.3 Contributions

Part of the work in this thesis has been accepted for presentation at the International Conference on Computational Finance (ICCF 2019) A Coruña (Spain), July, 8-12th, 2019. `http://iccf2019.udc.es`

Chapter 2

Related Work

Much effort has been put into trying to reject the Efficient Market Hypothesis (EMH) because, if true, it would mean that using machine learning models to gain an advantage in stock markets is not possible. However, initial models used for prediction were simple linear statistical models [24], and many relied on expert's criteria to guide the model construction. With the increase in computing power more sophisticated models became practical and trainable in relatively short times. Recent works like [11] apply state-of-the-art time series forecasting models to successfully predict future prices as well as using the model's criteria to gain further insight into the best/worst investments. However, the most often used traditional time series forecasting models do not include any kind of macroeconomic information [32].

It has become more a trend in this new millennium to combine historical prices and financial indicators (exogenous variables) to predict stock prices and make automatic financial decisions. As in [28], where a multifactor model is built using the correlation between performance and financial health for stock ranking; or in [2], where the authors take advantage of the interpretability of decision trees to build a forecasting model based on technical and fundamental financial indicators and analyze the importance of each financial factor.

However, explainable tree-based methods are being superseded by new and improved deep learning models. Krauss et al. [21] compare the performance of deep neural networks, gradient-boosted trees, and random forests to perform statistical arbitrage on the S&P 500. Their results are promising and challenge the semi-strong form of market efficiency. Moreover, they find that pooling together all models' results in an equal-weighted ensemble produces the best returns.

Aside from research, there are websites such as Quantopian [27] which offer the possibility of coding, evaluating, and generally conducting algorithmic finance research online. These platforms usually provide easy interfaces, such as Jupyter notebook [20], so that users can avoid all the hassle of downloading and curating the data, as well as setting up all the training and paper trading evaluation. In exchange, many of them offer some kind of agreement so that they can take profit out of the best algorithms designed. However, these platforms offer limited computing resources (often allowing only to evaluate a single model each time), and they cannot be modified if some desired feature is missing. Moreover, they tend to offer the data only inside their platform, so usually, it is not even possible to reproduce the results obtained.

Chapter 3

Methodology

This chapter describes the system designed to conduct the experiments, the data sources and preprocessing used, the models trained, and how they are evaluated in terms of economic performance. The description of the parallelization is in Chapter 4. All *HPC.FASSR* code is available at Github [1]

The whole project was developed in Python [13] because, thanks to being interpreted, allows fast prototyping, it is one of the most used languages for ML, and has a large number of great OSS libraries. This project relies mostly on Pandas [25], Numpy [30], and Scikit-learn [26] libraries.

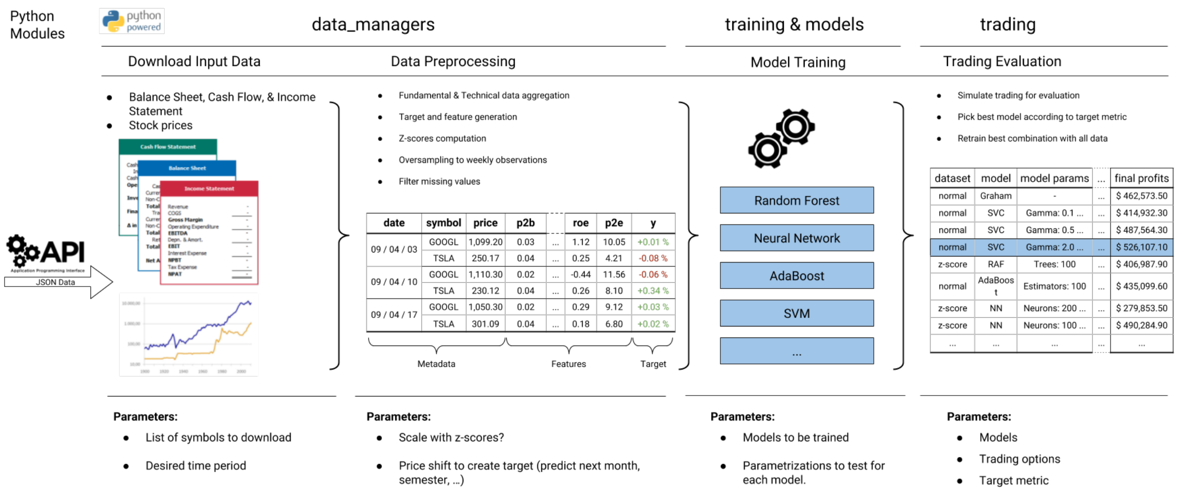
The *HPC.FASSR* system involves the integration of many techniques required to explore different models, parametrizations, and datasets. Such a system has to deal with many different tasks, ranging from the low-level ones such as downloading the data programmatically via API, to higher level ones like simulating a trading environment, passing through parallelization requirements. To isolate responsibilities, the code has been divided into four main python modules: *data_manager*, *training*, *models*, and *trading*. Figure 3.1 shows which part of the execution pipeline handles each module and which are its parameters. The *data_manager* is responsible for downloading the data from Intrinio and merge it into the desired dataset. The models and parametrizations to be trained are defined in the *models* module. The *training* module uses the data and models provided by the *data_manager* and *models* modules and trains them. Finally, the *trading* module evaluates the models using their predictions to invest in a paper trading environment. The set of all parameters used in a model execution will be called a *configuration*. A *configuration* is the triplet $\langle P_d, P_t, P_e \rangle$, where P_d are the parameters used to create the dataset; P_t are the parameters used for training (the model, its arguments, and the amount of training data); and P_e are the trading evaluation parameters like the trading frequency or the target metric to be used.

3.1 Data sources

For this study, we used data from the S&P 500 index. The S&P 500 is an American stock market index formed by the 500 leading companies of the U.S. stock market. Our selection does not include old S&P500 constituents so the results may suffer from survivorship bias. However, as all models are tested with the same dataset, the comparisons among them are valid.

The data was provided by Intrinio platform. Intrinio offers many Data Feeds which can be accessed via web API or bulk download. We used the web API of the US Company Fundamentals feed for the fundamental data, and the US Stock

[1]http://github.com/kafkasl/hpc.fassr



Download Input Data

- Balance Sheet, Cash Flow, & Income Statement
- Stock prices

Parameters:

- List of symbols to download
- Desired time period

Data Preprocessing

- Fundamental & Technical data aggregation
- Target and feature generation
- Z-scores computation
- Oversampling to weekly observations
- Filter missing values

date	symbol	price	p2b		roe	p2e	y
09 / 04 / 03	GOOGL	1,099.20	0.03	...	1.12	10.05	+0.01 %
	TSLA	250.17	0.04	...	0.25	4.21	-0.08 %
09 / 04 / 10	GOOGL	1,110.30	0.02	...	-0.44	11.56	-0.06 %
	TSLA	230.12	0.04	...	0.26	8.10	+0.34 %
09 / 04 / 17	GOOGL	1,050.30	0.02	...	0.29	9.12	+0.03 %
	TSLA	301.09	0.04	...	0.18	6.80	+0.02 %

Parameters:

- Scale with z-scores?
- Price shift to create target (predict next month, semester, ...)

Model Training

Parameters:

- Models to be trained
- Parametrizations to test for each model.

Trading Evaluation

- Simulate trading for evaluation
- Pick best model according to target metric
- Retrain best combination with all data

dataset	model	model params	...	final profits
normal	Graham	-	...	$ 462,573.50
normal	SVC	Gamma: 0.1 ...	...	$ 414,932.30
normal	SVC	Gamma: 0.5 ...	...	$ 487,564.30
normal	SVC	Gamma: 2.0 ...	...	$ 526,107.10
z-score	RAF	Trees: 100	...	$ 406,987.90
normal	AdaBoost	Estimators: 100	...	$ 435,099.60
z-score	NN	Neurons: 200 ...	...	$ 279,853.50
z-score	NN	Neurons: 100 ...	...	$ 490,284.90
...	...	...	...	...

Parameters:

- Models
- Trading options
- Target metric

FIGURE 3.1: Overview of the different modules, stages, data, and parameters of the proposed stock evaluation framework.

Prices feed for daily shares' price. Thanks to the web API *HPC.FASSR* downloads automatically the data required for the desired training.

For the fundamental data, *HPC.FASSR* downloads from Intrinio the income statement, balance sheet, and cash flow statement for each quarter and stock. The three documents are merged and their dates adjusted to create a dataset where each row represents the financial information of a company on a given date. For the historical prices, we build a hashmap where the key (s, d) contains the price of the stock s the date d.

3.2 Data preprocessing

3.2.1 Features

From the fundamentals dataset we compute the following indicators to be used by the learning models and the human expert (further details of these indicators can be found in [4, Ch. 6.2]):

- EPS $= \frac{NI - DivP}{WNSO}$
- EPS growth $= \frac{\text{Current year } EPS}{\text{Previous year } EPS}$
- BVPS $= \frac{\text{Assets} - \text{Liabilities}}{WNSO}$
- Price to Earnings $= \frac{P}{EPS}$
- Price to Book $= \frac{P}{BVPS}$
- Price to Revenue $= \frac{P}{R}$
- Dividends to price $= \frac{DivP}{P}$
- Dividend payout ratio $= \frac{DivP}{NI}$
- ROE $= \frac{NI}{\text{Shareholder's equity}}$

- ROIC $= \frac{NI+\text{ (after-tax interest)}}{\text{Invested Capital}}$
- ROA $= \frac{NI+\text{ (after-tax interest)}}{\text{Total assets}}$
- Asset turnover $= \frac{Sales}{\text{Total assets at start of year}}$
- Inventory turnover $= \frac{\text{Costs of assets sold}}{\text{Inventory at start of year}}$
- Profit margin $= \frac{NI}{Sales}$
- Debt ratio $= \frac{\text{Total liabilities}}{\text{Total assets}}$
- Times-interest-earned ratio $= \frac{EBIT}{\text{Interest payments}}$
- Revenue = Operating revenue
- Working capital = Current assets − Current liabilities
- Working capital to assets $= \frac{\text{Current assets}-\text{Current liabilities}}{\text{Current assets}}$
- Current ratio $= \frac{\text{Current assets}}{\text{Current liabilities}}$

where,

$$\begin{aligned} NI &= \text{net income,} \\ DivP &= \text{total amount of dividends} \\ WNSO &= \text{weighted average number of shares outstanding} \\ EBIT &= \text{earnings before interest and taxes} \\ P &= \text{share price} \\ R &= \text{revenue} \end{aligned}$$

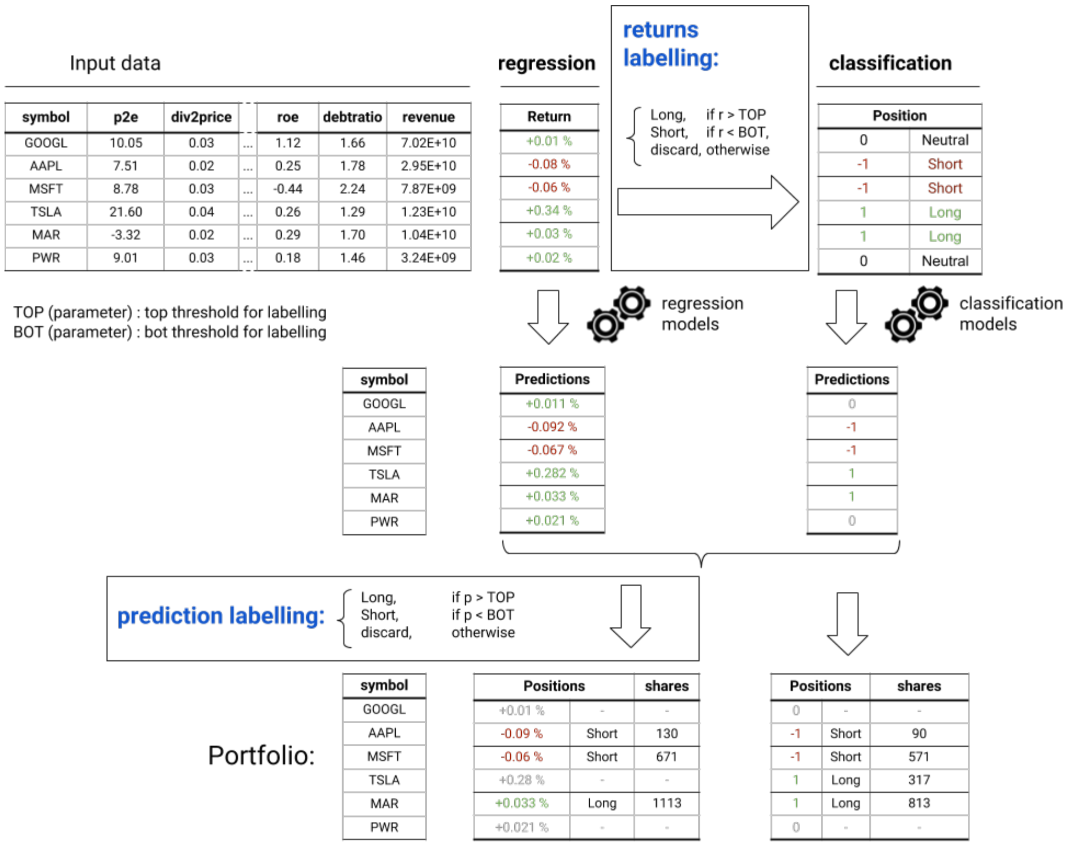
If the financial statements of a company in a given quarter lack any field required to compute a fundamental indicator, the company is removed from that quarter data. This results in an average of 190 companies per period. Since financial statements are available every quarter, one ends up with a small number of yearly training samples. To mitigate this problem, we applied oversampling as follows. Most of the indicators relate fundamental data with the stock price, the latter being available on a daily period. Hence, to generate new samples for a given date, we compute the indicator using the previous quarter's fundamental data adjusted to the stock price of the given date. For example, for the *price to book value* indicator, we would divide the current stock price by the book value of the previous quarter. For the indicators which are not related to stock price (e.g., revenue) we use the latest available value.

We considered two scaling methods: standard scaling and z-scores. When applying the standard scaling, we scale together all the data that will be used for training a model on a given period (i.e., one/two years of data). For the z-scores, we scale the data of each day separately. The idea behind our z-scores, is to normalize the indicators with respect to the mean and standard deviation of stocks within the same industry group, given by the first three digits of their Standard Industrial Classification (SIC) code, because they may have different trends.

3.2.2 Targets

For regression, we used the simple return, i.e., $y_{i,s} = \frac{r_{i,s}}{r_{i-1,s}}$, where $r_{i,s}$ is the return of stock s for trading session i. For classification, we categorized the returns into *long*, *neutral*, *short*. The thresholds were chosen using backtesting from the ranges $[-0.03, 0]$ and $[0, 0.03]$ with steps of 0.005 for the top and bottom thresholds respectively. Stocks with a return higher than the top threshold are labeled as long, stocks with a return under the bottom threshold are labeled as short, the rest as neutral.

When we compare the regression and classification, we use the same thresholds to create the classification training data, and the ones used in regression to convert predictions into positions. Figure 3.2 shows the two spots at which these thresholds are applied for each model.



symbol	p2e	div2price		roe	debtratio	revenue
GOOGL	10.05	0.03	...	1.12	1.66	7.02E+10
AAPL	7.51	0.02	...	0.25	1.78	2.95E+10
MSFT	8.78	0.03	...	-0.44	2.24	7.87E+09
TSLA	21.60	0.04	...	0.26	1.29	1.23E+10
MAR	-3.32	0.02	...	0.29	1.70	1.04E+10
PWR	9.01	0.03	...	0.18	1.46	3.24E+09

Return
+0.01 %
-0.08 %
-0.06 %
+0.34 %
+0.03 %
+0.02 %

Position	
0	Neutral
-1	Short
-1	Short
1	Long
1	Long
0	Neutral

symbol	Predictions	Predictions
GOOGL	+0.011 %	0
AAPL	-0.092 %	-1
MSFT	-0.067 %	-1
TSLA	+0.282 %	1
MAR	+0.033 %	1
PWR	+0.021 %	0

symbol	Positions		shares	Positions		shares
GOOGL	+0.01 %	-	-	0	-	-
AAPL	-0.09 %	Short	130	-1	Short	90
MSFT	-0.06 %	Short	671	-1	Short	571
TSLA	+0.28 %	-	-	1	Long	317
MAR	+0.033 %	Long	1113	1	Long	813
PWR	+0.021 %	-	-	0	-	-

FIGURE 3.2: Overview of the different data pipelines for classification and regression.

3.3 Forecasting models

We used scikit-learn [26] implementations of the most popular machine learning predictors in literature: Neural Networks, Support Vector Machines, Random Forests, and AdaBoost.

The system is designed to work with models that follow scikit-learn model's API [9] so users can integrate their custom models by just implementing sklearn's interface. Moreover, the system supports models that can run in more than one CPU (intra-node parallelism). Using the standard scikit-learn *n_jobs=number_cpus* parameter and PyCOMPSs *@constraint(ComputingUnits=number_cpus)* decorator, a model can be trained with any desired number of CPUs. This highly increases the performance of embarrassingly parallel models like random forests.

3.3.1 Graham

The expert criteria that we programmed as a rule-based decision algorithm, and that is to compete against the above classifiers, is that of famous value investor Benjamin Graham. His rules for selecting value stocks, updated for inflation to present time, are the following (see [4, S. 6.2] for details): the company should have at least USD 1.5B in revenues; 2-to-1 assets to liability ratio; positive earnings in each of the past ten years; uninterrupted payment of dividends; a 3% of annual average growth in earnings; moderate P/E and P/B ratios.

3.3.2 Support Vector Machines

For SVM, we used the RBF kernel for all tests. For the parameter, we explored the $C \in [2^{-5}, 2^{15}]$ and $\gamma \in [2^{-15}, 2^{3}]$ multiplying both of them by 2^2 each step.

3.3.3 Neural Networks

For the neural networks, we used the basic Multilayer Perceptron (MLP) with a single hidden layer. We evaluate the following hidden layer sizes $[15, 50, 100, 500, 1000]$, the solvers Adam [19] and LBFGS [23], and the activation functions *hyperbolic tangent* and *ReLU*.

3.3.4 Random Forests

For random forests, we vary the different number of trees in the ensemble. The sklearn version has a parameter *n_jobs* which controls how many threads the algorithm can use to run in parallel in a single node. We set this value to 12 CPUs, which one-quarter of a node in MN4, because it reported the best *total* times empirically.

3.3.5 AdaBoost

For AdaBoost [14], we also vary the number of estimators in the ensemble. Unfortunately, AdaBoost version cannot be trained in parallel.

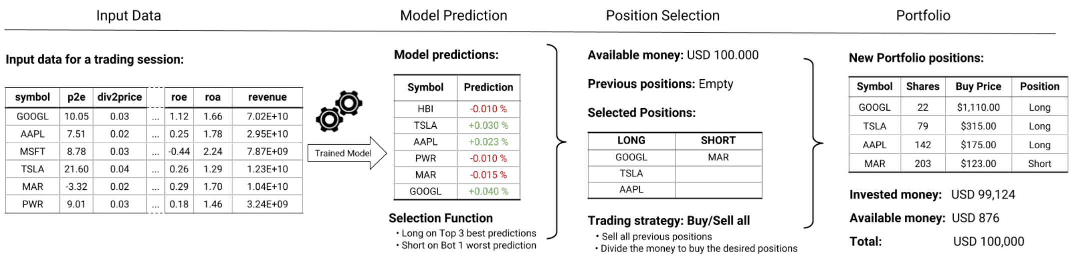
Further details about parametrizations can be found in Appendix A.

3.4 Experimental set-up and evaluation

We have evaluated the economic performance of all models taking long-short positions, trading every semester and year, using different scaling methods and models, different amounts of training data, and with different trading strategies.

The economic performance is evaluated by the *trading* module. The *trading* module simulates a backtesting paper trading environment for a given time interval.

We start by dividing the trading time interval into *n trading sessions*. Figure 3.3 shows the execution flow of a single trading session. The portfolio data structure and handling that stores the positions and available money of each session for posterior analysis is not shown for clarity's sake.



Input data for a trading session:

symbol	p2e	div2price		roe	roa	revenue
GOOGL	10.05	0.03	...	1.12	1.66	7.02E+10
AAPL	7.51	0.02	...	0.25	1.78	2.95E+10
MSFT	8.78	0.03	...	-0.44	2.24	7.87E+09
TSLA	21.60	0.04	...	0.26	1.29	1.23E+10
MAR	-3.32	0.02	...	0.29	1.70	1.04E+10
PWR	9.01	0.03	...	0.18	1.46	3.24E+09

Model predictions:

Symbol	Prediction
HBI	-0.010 %
TSLA	+0.030 %
AAPL	+0.023 %
PWR	-0.010 %
MAR	-0.015 %
GOOGL	+0.040 %

Selection Function
- Long on Top 3 best predictions
- Short on Bot 1 worst prediction

Available money: USD 100.000

Previous positions: Empty

Selected Positions:

LONG	SHORT
GOOGL	MAR
TSLA	
AAPL	

Trading strategy: Buy/Sell all
- Sell all previous positions
- Divide the money to buy the desired positions

New Portfolio positions:

Symbol	Shares	Buy Price	Position
GOOGL	22	$1,110.00	Long
TSLA	79	$315.00	Long
AAPL	142	$175.00	Long
MAR	203	$123.00	Short

Invested money: USD 99,124
Available money: USD 876
Total: USD 100,000

FIGURE 3.3: Example of a single iteration of the trading evaluation. This process is repeated for each trading session. The final portfolio and available money are the input for the next trading session.

For each *trading session*, we apply the model to the input data to obtain the next period predictions. Depending on the type of model we get different information. For the categorical models, we get if we should short or long the stock, or remain neutral. For the regression models, we get the expected simple return.

Next, we apply a *selection function* (see 3.4.2) to convert the predictions into recommendations.

The last step is to create a new portfolio using a *trading strategy* (see 3.4.3). The trading strategy takes as input the long-short recommendations, the previous portfolio, and the available money to decide which old positions should be closed and which new ones should be bought.

After evaluating all models, we choose the best one according to the desired *target metric*. Some examples of target metrics are the total profits, the average invested money, or a combination of them with the execution time (if we want a trade-off between accuracy and performance).

3.4.1 Transaction fees

Our goal is to create an automatic trading system that does not rely on humans, so we decided to use the fixed rate pricing of Interactive Brokers [16]'s (IB) python API which can be integrated with this project with minimal work. Figure 3.4 shows how the IB fees are computed given the number of shares and its price. Essentially, the fees are \$ 0.005 per share, with a minimum of \$ 1 and a maximum of 1% of the trade volume. IB does not allow to buy fractions, so all the trading is done buying only whole shares. Buying shares by units creates a residue of money. This remainder is used differently depending on the trading strategy.

3.4.2 Selection function

The *selection function* is used to convert the model's predictions into desired positions. We introduced this function because depending on the model (regression or classification) and the task type (ranking or screening) the input and desired output are different. Moreover, we wanted to provide the flexibility of applying different policies to the model's predictions (like going long for all positive stocks for stock ranking or just the best k).

For example, when are performing stock screening, the regressors' predictions are converted into a long positions if the expected returns are higher than the top threshold, to a short positions if the returns are lower than the bottom threshold, and the rest are discarded. For classifiers, the predictions are already the desired positions, and we only have to discard the neutral ones.

$$Fees(s,p) = \begin{cases} 1, & \text{if } f * s < 1 \\ 0.01 * (s * p) & \text{if } f * s > maxFees \\ f * s, & \text{otherwise} \end{cases}$$

where,

$$\begin{aligned} s &= \text{number of shares}, & & \\ p &= \text{price per share}, & & \\ f &= \text{fee per share in USD} & &= 0.005 \\ maxFees &= \text{max fees per transaction in USD} & &= 0.01 * (s * p) \\ minFees &= \text{min fees per transaction in USD} & &= 1 \end{aligned}$$

FIGURE 3.4: Interactive Brokers fixed rate fees. Applied to each transaction during the paper trading evaluation of the models.

However, when we are performing stock ranking, the *selection function* is responsible for ranking the stocks by their expected returns, and converting the top k stocks into long positions, the bottom k to short positions and discard rest.

3.4.3 Trading strategies

We designed different strategies to be used on paper trading. All strategies start with USD 100K. The difference between the strategies is how they choose the new positions of the portfolio, given the positions of the previous portfolio, the available money, and the recommendations. Appendix B contains the pseudocode of both trading strategies.

Avoid fees

This strategy is focused on having a portfolio of 20 stocks (which is between the 10 and 30 as suggested by Graham [15]) while trying to minimize the fees paid.

To start - or when the last portfolio is empty - the available money is used to buy 20 stocks (dividing the capital equally among the chosen stocks). If the models recommend less than twenty stocks, the remainder is not spent, it is kept for the next trading session.

For trading sessions when the last portfolio is not empty, the strategy is as follows. If a stock recommended by the model is already present in the portfolio, the position is maintained (thus avoiding any transaction and fee). The rest of the positions are sold. The money obtained from selling the positions is used to buy stocks (from the recommended ones) up to a maximum of 20 for stock screening and to $2 * k$ for stock ranking. Again, if there are not enough recommendations to buy up to the maximum, the remainder is kept for the next session. If all the portfolio's positions are among the recommended ones, there are no transactions.

If at any step, the available money is not enough to open some position, then it is stashed for future sessions.

Buy-Sell all

The previous *avoid fees* strategy can hide many performance details (e.g., if the initial 20 stocks are always recommended, all the other predictions will be ignored). This strategy tries to address that possible issue by selling and buying all the recommendations on each trading session. The strategy will incur in higher fees, but the relation between the recommendations and the actual positions is clear: all recommendations always become positions (if there is enough money available).

This will ease the performance analysis of the models. For example, if a model tends to overestimate returns, this will result in a high number of open positions.

Chapter 4

Parallelization

4.1 PyCOMPSs overview

PyCOMPSs [10] is a framework aimed to ease the development of parallel and distributed Python applications. The primary purpose is to abstract users from both infrastructure management and data handling. PyCOMPSs is composed of two main parts: the programming model and the runtime. The programming model provides a set of simple annotations to indicate which functions can be run in parallel. The runtime handles the data dependencies between the tasks and distributes the computation among the available resources.

PyCOMPSs code is portable because it is infrastructure unaware and it can be run in a wide number of platforms, such as clouds and grids while providing a uniform interface for the user.

4.1.1 Programming model

PyCOMPSs offers a sequential programming model to specify the parts of the code that can be run in parallel. This differs from other models and paradigms that require the developer to have a deep knowledge of the hardware executing the code such as MPI interfaces or OpenMP pragmas.

To indicate which parts of the code execution can be distributed, PyCOMPSs takes advantage of the python function decorators [12]. The basic PyCOMPSs decorator is *@task()* and it states that a given function should be treated as a task.

Tasks are run distributed, and their parameters need to be instrumented to transfer them between the nodes. All the inputs and outputs of tasks are placeholders to allow them to run asynchronously while the main code continues to be executed. When the value of some return is needed, we need to synchronize it to the master using the API call *compss_wait_on*.

Figure 4.1 shows an example application that computes the sum of the squares of the numbers in the list $[0, 10]$. The *multiply* function is decorated as a task and all of them are thus run distributed. The output of these tasks is then passed to the *mean* function which computes the average. Finally, we synchronize the value and print it.

4.1.2 Runtime

The PyCOMPSs framework is mostly a set of bindings to interact with the Java runtime. The runtime has two primary responsibilities: analyzing tasks' dependencies and scheduling them, and resources and data management.

To execute in distributed the functions decorated as tasks, the runtime needs to compute the data dependencies between them. That information is then used to

```python
from pycompss.api.task import task
from pycompss.api.api import compss_wait_on

@task(returns=int)
def multiply(num1, num2):
    return num1 * num2

@task(returns=float)
def mean(*numbers):
    return sum(numbers) / len(numbers)

s = [multiply(i, i)) for i in range(10)]
avg = compss_wait_on(mean(*s))
print(avg)
```

FIGURE 4.1: PyCOMPSs application example that squares each number in $[0, 10)$ and then computes the mean.

build a task dependencies graph which determines the order of the execution (e.g., if a task A generates the input of a task B, then B cannot be run until A is finished). Once tasks are free of dependencies in the graph, they are scheduled to be executed in some of the available computing resources.

Concerning resources management, the runtime follows a master-worker paradigm. All available computing resources are represented uniformly as workers. The master node is responsible for executing the user code. When it encounters a task, the task is scheduled (following the dependencies graph) to a given worker, its required input is transferred, and the results, if any, are gathered and sent back.

Thanks to the pluggable connectors PyCOMPSs code is infrastructure-agnostic and can be run in a wide number of platforms without requiring any change. These include clouds, cluster and grid nodes, or docker/singularity containers, using connectors like Slurm [29] or Apache Mesos [3]. Furthermore, PyCOMPSs supports heterogeneous architectures allowing the user to mix GPUs and FPGAs [1] with traditional CPU computing resources.

4.1.3 Tools

PyCOMPSs also offers some useful tools for execution analysis: the monitor and the tracing system. The monitoring offers information about executions such as diagrams of data dependencies, resources state details, statistics and easy access to the framework logs. The tracing system uses Extrae [6] to generate post-mortem execution trace files. The trace files can be analyzed with the graphical tool Paraver [8].

Figure 4.2 shows an example of a tracefile from Figure 4.1. This view shows only the executed tasks. Each horizontal bar represents a thread of the application. It is worth noting, that despite being called *threads*, each line corresponds to an independent python process, and that there are as many python processes as CPUs. The threads are named *Thread x.y.z* where x is the application the thread belongs to, the y is the node, and the z is the thread number in the node. In this example, we have two nodes 1.1 and 1.2. The first three threads (1.1.1 - 1.1.3) correspond to the master node. The other five threads belong to a single worker with four computing

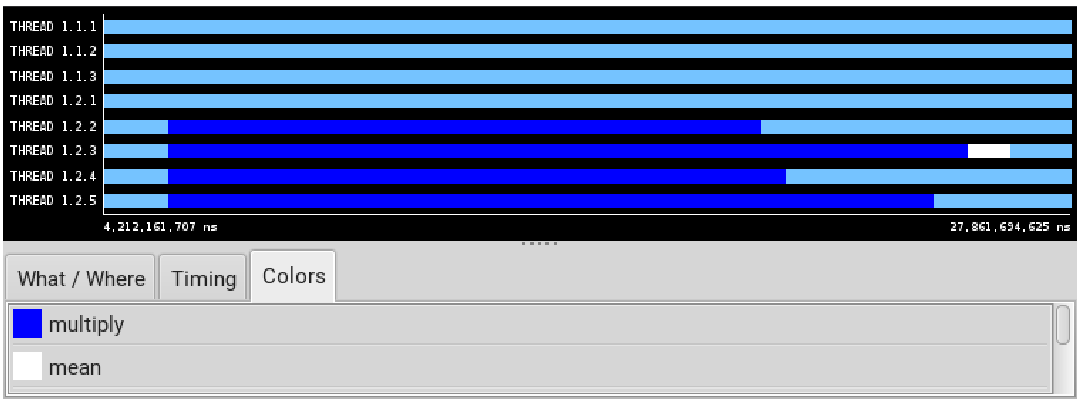


FIGURE 4.2: Tracefile of the sample application from Figure 4.1.

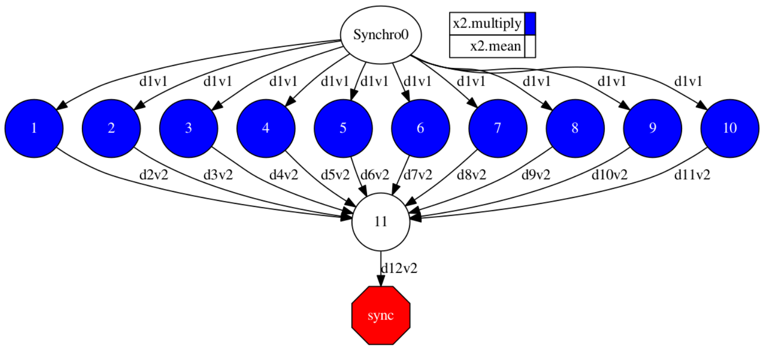


FIGURE 4.3: Task dependencies graph of the sample application from Figure 4.1.

units (1.2.2 - 1.2.5). The first thread of each worker (1.2.1) is used to show data transfers and other worker internal events. Both the master node and the worker's main thread are empty because they do not execute any task directly.

Figure 4.3 shows the data dependencies graph of the sample application from Figure 4.1. The input arrows mark the data dependencies between tasks. All the multiply tasks can be run in parallel as they only depend on their input parameter. The mean task depends on all the objects of the list.

4.2 Hardware

To test the parallelization efficiency, we run the experiments of the project in the MareNostrum 4 supercomputer [7].

MareNostrum 4 has 3456 computing nodes housing a total of 165.888 cores, 390 Terabytes of main memory, and a peak performance of 11.15 Petaflops. Each node contains:

- 2 Intel Xeon Platinum 8160 CPU with 24 cores each at 2.10 GHz
- 96 GB of main memory
- 200 GB of local SSD storage.
- L1d 32K; L1i cache 32K; L2 cache 1024K; L3 cache 33792K
- 100 Gbit/s Intel Omni-Path HFI Silicon 100 Series PCI-E adapter

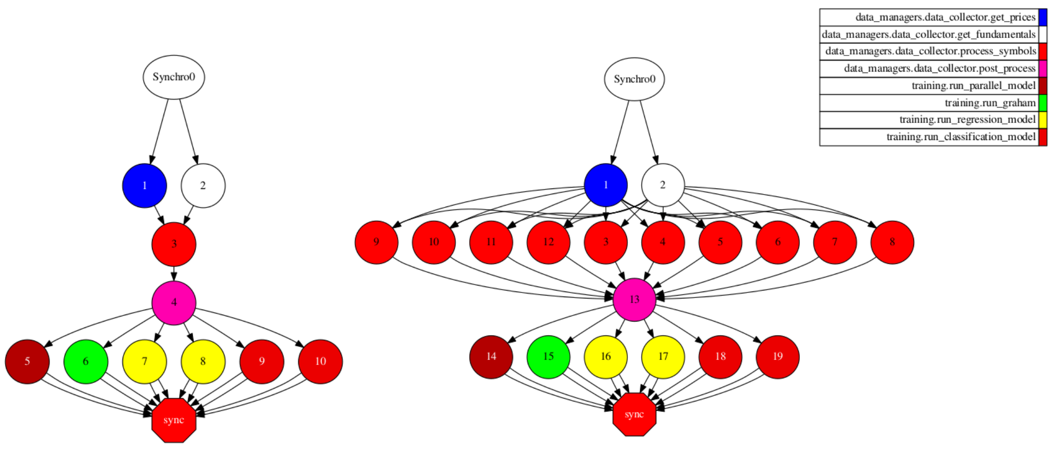

data_managers.data_collector.get_prices
data_managers.data_collector.get_fundamentals
data_managers.data_collector.process_symbols
data_managers.data_collector.post_process
training.run_parallel_model
training.run_graham
training.run_regression_model
training.run_classification_model
Synchro0
sync

FIGURE 4.4: Task dependencies graphs for the versions where all the symbols are processed in a single task (left) and the one where each symbol is processed in a different task (right).

- 10 Gbit Ethernet

4.3 Implementation

HPC.FASSR should be able to easily explore and train a considerable number of ML models, parametrizations, and inputs in a relatively short amount of time. To speed up the computing we distributed the execution of the stages of *HPC.FASSR*'s pipeline (see Figure 3.1) *Download Input Data*, *Data Preprocessing*, *Model Training*, and *Trading*. All *HPC.FASSR* code is available at Github [1]

The *Download Input Data* is parallelized at a very low-level. Downloading all the data through an API is slow, hitting at times 50K API calls. To overcome that, we do each API call in a task. Intrinio has a daily API limit. To avoid decrease the number of requests, all API calls are cached locally to avoid duplicated calls. These cached results are shared across executions.

For the *Data Preprocessing* stage, the two tasks *get_fundamentals* and *get_prices* build the fundamentals dataset and the historical prices respectively.

Next, we tried parallelizing the features and target building of each symbol individually. However, in our tests with the S&P 500, the time of processing the 500 symbols is not large enough to warrant parallelization and all symbols are processed together in a single *process_symbols* task. Figure 4.4 shows the data dependencies graph when processing all symbols together or one in each task.

Finally, the *post_process* task creates the z-scores if needed, filters some invalid values, and caches the dataset to disk for reuse.

The *Model Training* and *Trading* are grouped into four types of different tasks:

1. *run_graham*
2. *run_classification_model*
3. *run_regression_model*
4. *run_parallel_model*

[1]http://github.com/kafkasl/hpc.fassr

The first task evaluates Graham's criteria. The second and third tasks evaluate classification and regression models respectively. We differentiated between regression and classification models only for visualization purposes. Both are run identically; only the input data is different. If the target model supports intra-node parallelization, defined by the *n_jobs* parameter, we execute the fourth task with hardware constraints equal to number *n_jobs*. We explored different values for *n_jobs* and the best times were obtained setting it to use a quarter of the node: n_jobs = 12 CPUs.

4.4 Checkpointing

We aim to run very large executions so we developed a checkpointing system to be able to reuse data from a finished execution or another currently running one. Our aim is twofold: on the one hand, this allows users to reuse data from failed executions, and, on the other, to reuse data from overlapping configurations across different executions.

The strategy followed is to cache the intermediate trained models into disk. For each configuration, the *run* tasks train as many models as trading sessions exist. This is the most time-consuming part of the application so every time one of the tasks finishes training the model of a session, the model is saved to disk. The name for the file is a hash derived of the configuration that is being evaluated and the trading session. This naming convention allows other executions to reuse the model directly if they use the same cache directory. If the cache directory is in a shared file system, all other executions can reuse it. Otherwise, the models will be only usable by executions on the same node.

Thanks to this, if an application is stopped during the execution (e.g., canceled due to time limit by the queuing system in supercomputers), rerunning it will reuse all the models and continuing where it left off. Also, in executions with overlapping configurations (e.g., one execution trading each year, the other each semester), each model will be trained by the first execution and reused by the others. We do not implement any contention policy because the probability of two configurations looking for the same file simultaneously is negligible in our tests.

The amount of space required to cache big executions is not trivial. In order to keep it to a minimum, all models are compressed with *gzip* python package after being serialized.

4.5 Evaluation

We evaluate the performance of *HPC.FASSR* under strong and weak scaling conditions. Strong scaling measures how the execution time changes with the number of CPUs for a fixed problem size. On the other hand, weak scaling measures the execution for different numbers of CPUs when the problem size is fixed *per CPU* (i.e., the total problem size increases with the number of CPUs).

For the strong scaling tests, we use a sample execution of 4000K configurations. For the weak scaling, we use a ratio of $\frac{25}{24}$ configurations/CPU to match our smallest execution of a single node with 48 CPUs. To keep the workload constant, we use a base sample of 10 configurations and replicate it to obtain sample sizes in $[50, 150, 350, 750, 1550, 3150]$.

In both scenarios, we run our tests with 1, 2, 4, 8, 16, and 32 nodes. Each node has 48 CPUs except when using a single node which uses only 24 CPUs because half of the node is used by the worker, the other half by the master. MareNostrum

4 supercomputer does not allow outbound internet connections, so the executions were done using the data cached by the section *Download Input Data* from Figure 3.1.

For each test, we report the execution time of the python application. The speedup of a given application a and a base application b is computed as $S_{a,b} = \frac{t_a}{t_b}$, where t_x = execution time of application x.

Chapter 5

Results and Discussion

This section presents the results of evaluating *HPC.FASSR* . The first section exposes the execution performance through weak and strong scaling tests, and models' execution time. The second section is composed of three experiments, each with a different trading scenario, and presents the economic performance of each configuration.

5.1 Execution performance

We evaluate the parallelization performance in both weak and strong scaling scenarios and the executions time and size of the different models.

5.1.1 Strong scaling

Figure 5.1 shows the execution time and speedup of the strong scaling experiments. The speedup is linear for all the experiments. The superlinear speedup observed between 2 and 32 nodes stems from the fact that, when using a single node, the worker has 24 CPUs, when adding a second node, the full node is used (48 CPUs) so, when scaling from one node to two, you have three times more CPUs despite only having twice the amount of nodes.

Figure 5.2 shows the tracefiles of the strong scaling experiments. The initial red tasks are responsible for training the parallel models with 12 CPUs so only four of them can be run at the same time in a given node, but they use the whole node. Note that each tracefile has a different time scale, roughly half the time when doubling the resources. In the first tracefile, each thread evaluates around 33 classification and regression models. In the last one, each thread evaluates either 1 or 2

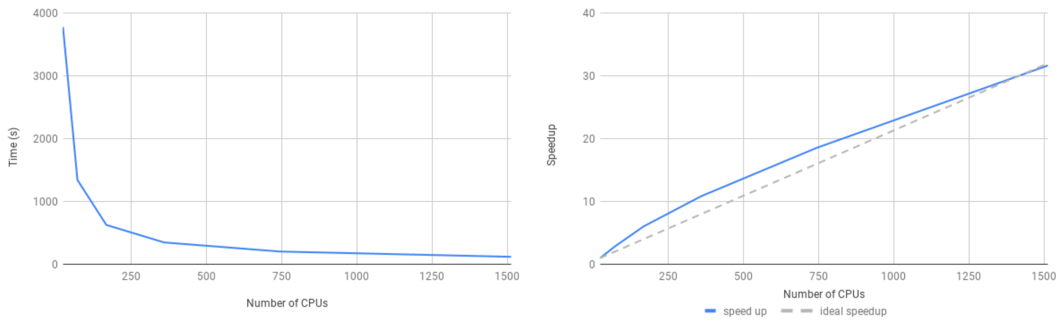


FIGURE 5.1: Execution times (left) and speedup (right) of the strong scaling experiments evaluating 2000 configurations trading yearly from 2009 to 2018.

of each model type. *HPC.FASSR*'s performance depends on the number of configurations to test and there two situations when increasing the number of resources does not improve the performance. First, when there are not enough classification and regression tasks for each thread, because their different times leave some CPUs idle while longer tasks finish. The second cause happens when the number of classification tasks (which last twice as much as the regression ones) is slightly higher than the number of available threads. If there are c classification tasks, n available threads, and $2 > c > n$, then there are going to be $2 * n - c$ idle threads at the end of the execution. As the number of resources increases (and total time decreases), the fraction of the total time that a single evaluation task represents also increases extending the time wasted by idle threads. Whenever the number of configurations' number is large enough, *HPC.FASSR* should scale almost linearly. The weak scaling tests objective is to validate this thesis.

5.1.2 Weak scaling

Figure 5.3 shows the execution time and speedup of the weak scaling experiments. The weak scaling speedup is very close to the ideal for all sizes so *HPC.FASSR* scales almost linearly when the number of configurations is large enough. The first steeper increase of the execution time is caused by going from 1 node, where there are no network communications, to 2 nodes, which have network transmissions that add overhead. From 2 nodes onwards, the cost of communication between two or more nodes is very similar so the overhead increases much more slowly. Figure 5.4 shows tracefiles of of the weak scaling executions. In this situation, we increase the problem size and the computing power at the same rate. The tracefiles are in the same time scale. The time until the first task (blue) starts increases slightly with more nodes due to the overhead of initializing a more significant number of workers. Despite that, the overhead of doubling the resources between tests is almost negligible. The ratio of tasks to train per thread is constant so, despite all executions being slightly unbalanced towards the end, this does not cause any significant performance loss.

5.1.3 Models

Table 5.1 shows the average and the standard deviation of the execution time required to evaluate each model. The random forests would be the worst models if they were not trained using 12 CPUs because they would be around $x12$ slower; we also tried also using 24 (half node), and 48 CPUs (full node) but empirically 12 CPUs yielded faster *total* execution times. It is worth noting that we are able to thanks to HPC capabilities of MareNostrum which has 48 CPUs per node and 96 GB of RAM, allowing all tasks to load the training data into memory at the same time and interleave the executions of models with 1 and 24 CPUs. Next, the neural networks are the slowest on average, but they also have the highest variance (probably because their parameter, *hidden layer size*, directly affects the training time). SVMs take on average similar times to the neural networks but their variance much lower. Finally, AdaBoost is the fastest method.

The execution times between models vary a lot and these times can be further increased for larger datasets. Having very slow and fast methods coexisting causes work unbalance. In our experiments, the unbalance overhead was never too critical. However, for larger datasets this might become a bottleneck and degrade the scalability. We leave as future work trying a finer task granularity where each training step is done in different tasks not together in a single one.

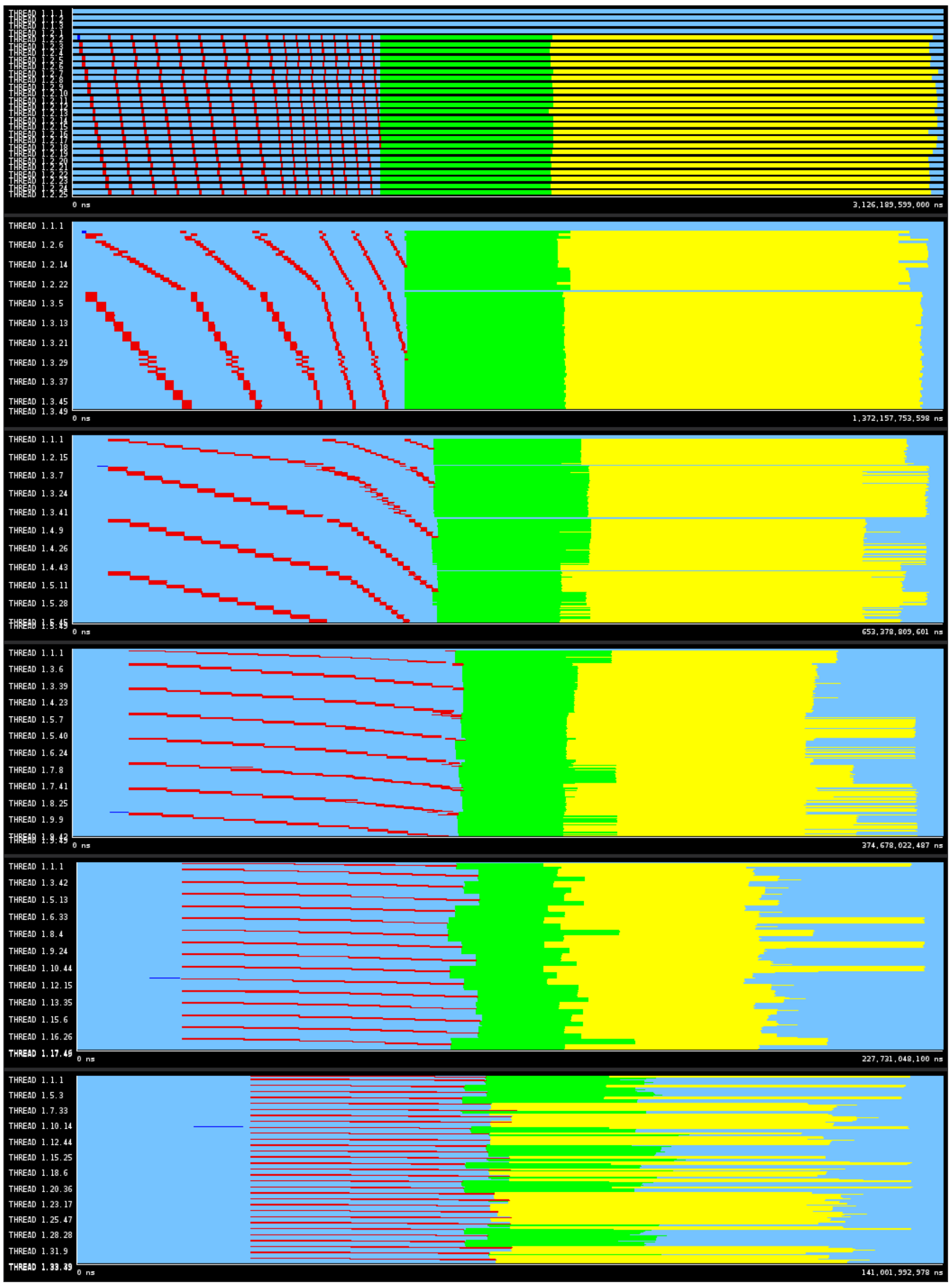


FIGURE 5.2: Tracefiles of the strong scalability tests reported in Figure 5.1 from 1 node (top) to 32 nodes (bottom). Please note that each tracefile has a different time axis (roughly halving for each execution from top to bottom). Red tasks evaluate parallel models with 12 CPUs, green and yellow tasks evaluate the regression and classification models, with a single CPU, respectively.

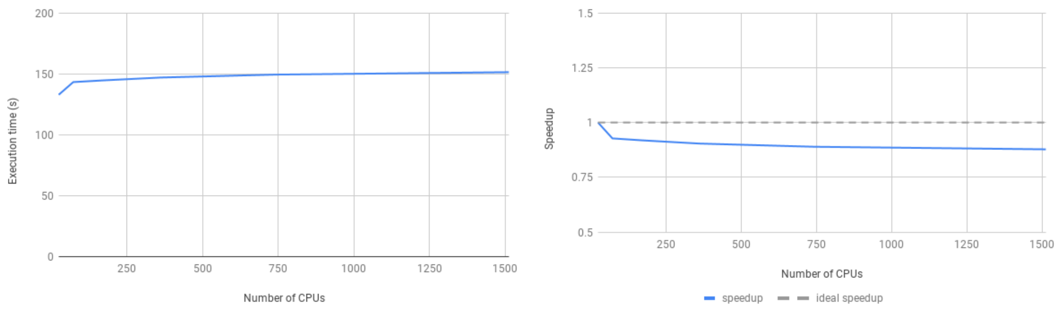


FIGURE 5.3: Execution times (left) and speedup (right) of the weak scaling experiments evaluating 50, 150, 350, 750, 1550, and 3150 configurations with 1, 2, 4, 8, 16, and 32 nodes respectively trading yearly from 2009 to 2018.

Model	Execution time (s)
Linear Regression	0.93 ± 0.12
Random Forest Classification	17.06 ± 9.51
Random Forest Regression	58.12 ± 44.02
Graham's Criteria	37.13 ± 3.35
AdaBoost Regression	39.87 ± 16.49
AdaBoost Classification	45.43 ± 30.13
SVM Classification	89.80 ± 30.50
NN Regression	91.59 ± 90.29
SVM Regression	99.25 ± 11.81
NN Classification	150.01 ± 115.72

TABLE 5.1: Mean and standard deviation of the time required to evaluate each model throughout all three trading performance experiments 5.2 ordered from fastest to slowest. Random forests training is multithreaded ($n\_jobs = 12$ CPUs) while all others models use a single CPU.

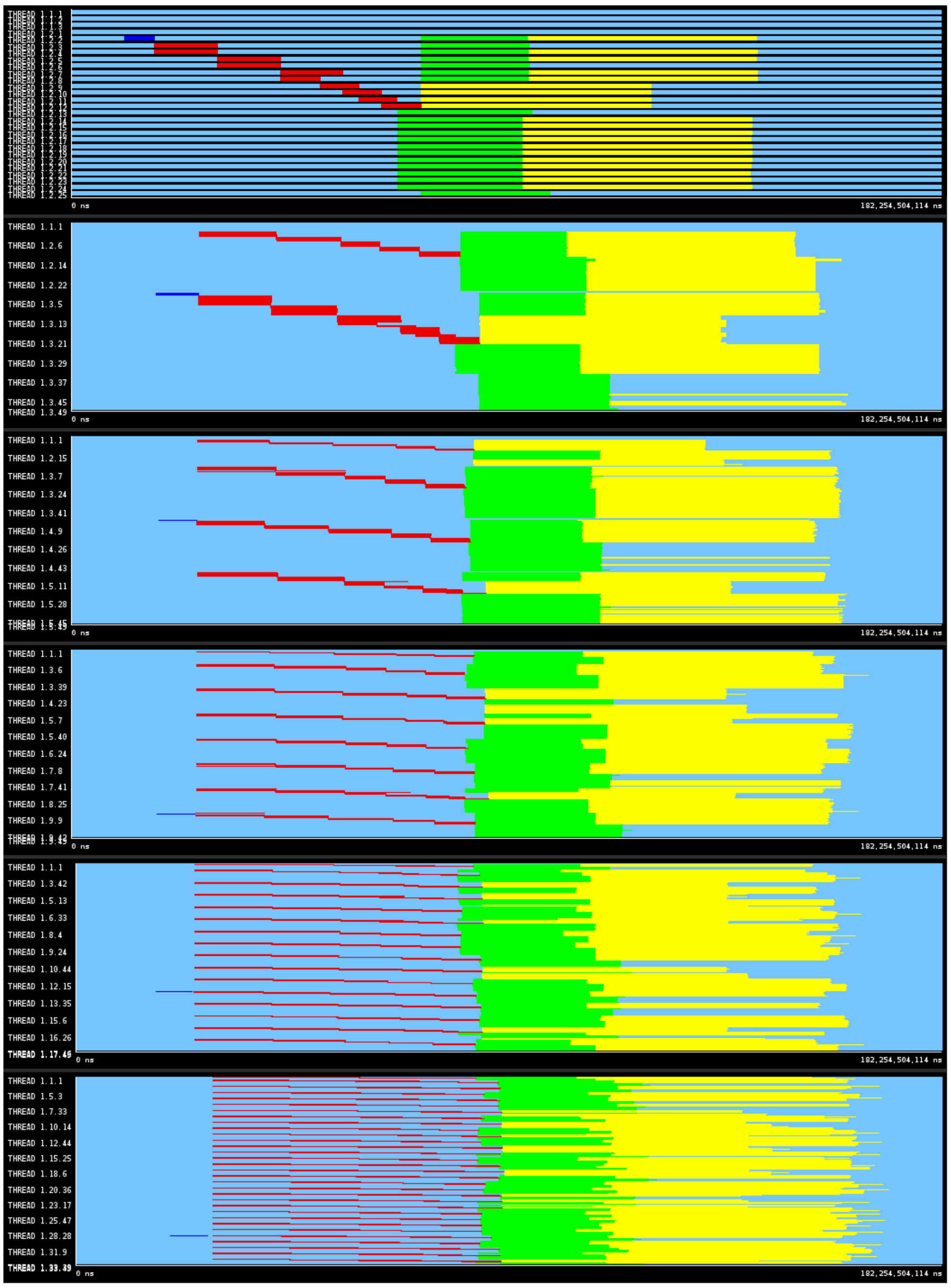


FIGURE 5.4: Tracefiles of the weak scalability tests reported in Figure 5.3 from 1 node (top) to 32 nodes (bottom). Please note that all tracefiles have the same time axis. Red tasks evaluate parallel models with 24 CPUs, green and yellow tasks evaluate the regression and classification models, with a single CPU, respectively.

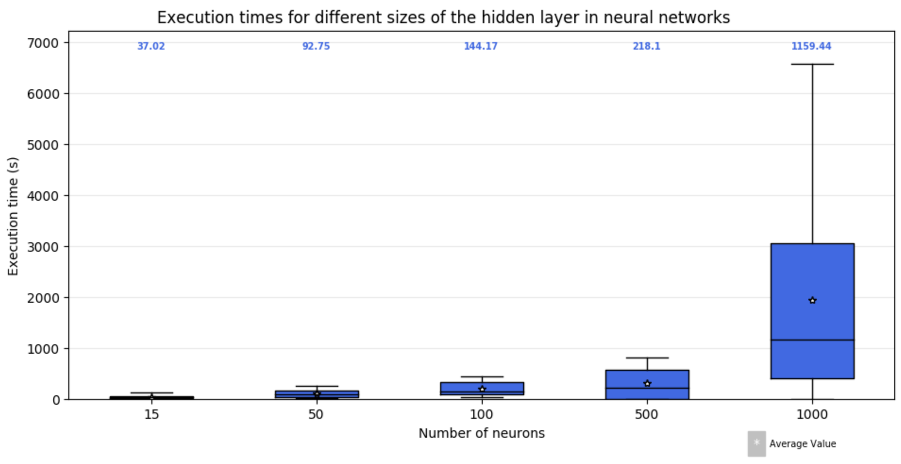


FIGURE 5.5: Boxplot of the execution time of neural networks for different sizes of the hidden layer.

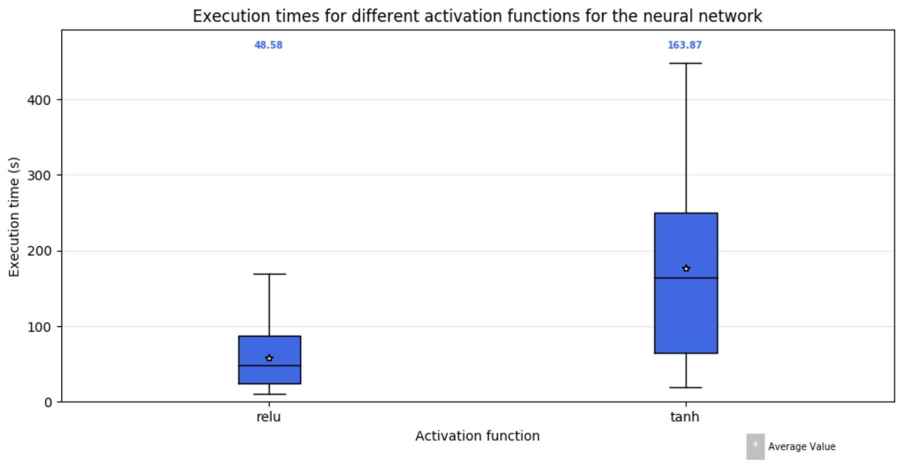


FIGURE 5.6: Boxplot of the execution time of neural networks for different activation functions.

Figures 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, and 5.11 show how the execution time varies with different values of the model's parameters. The results reported are from the three trading experiments from section 5.2. For the AdaBoost, random forests, and neural networks, the number of estimators, trees, and neurons is directly proportional to the model's execution time. With respect to the solvers, both have similar average; LBFGS has a lower median but Adam has a smaller IQR. For the network's activation function, ReLU function is much faster than *hyperbolic tangent* and has less variance. The ReLU is, as of 2017, the most used activation function thanks to learning much faster in deep networks [22].

For the checkpointing feature, Table 5.2 shows the sizes of each model after being serialized and compressed. The size of linear regression is the smallest and it is constant as it only needs to save a small constant number of coefficients and the intercept. The random forests are the heaviest models, with an average of 48MB per model, and vary a lot as the number of parameters to save grows linearly with the number of trees. The next heaviest model is the SVM which has to save all support vectors that define a model. The neural networks and AdaBoost are much lighter and have similar averages. They also present significant deviations attributed to the changes in the number of neurons and estimators.

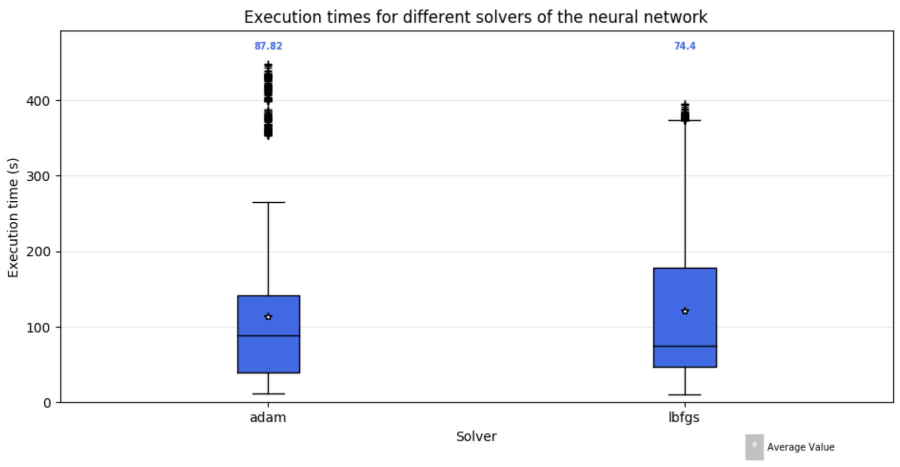


FIGURE 5.7: Boxplot of the execution time of neural networks for different solvers.

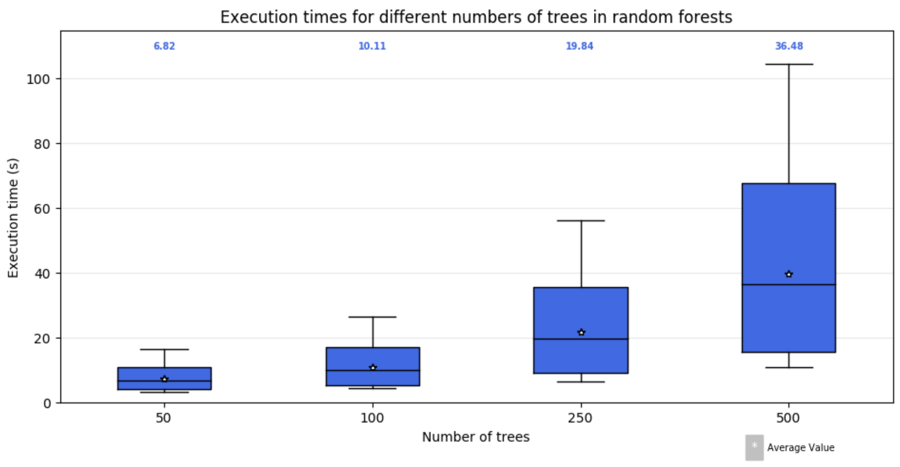


FIGURE 5.8: Boxplot of the execution time of random forests for different number of trees.

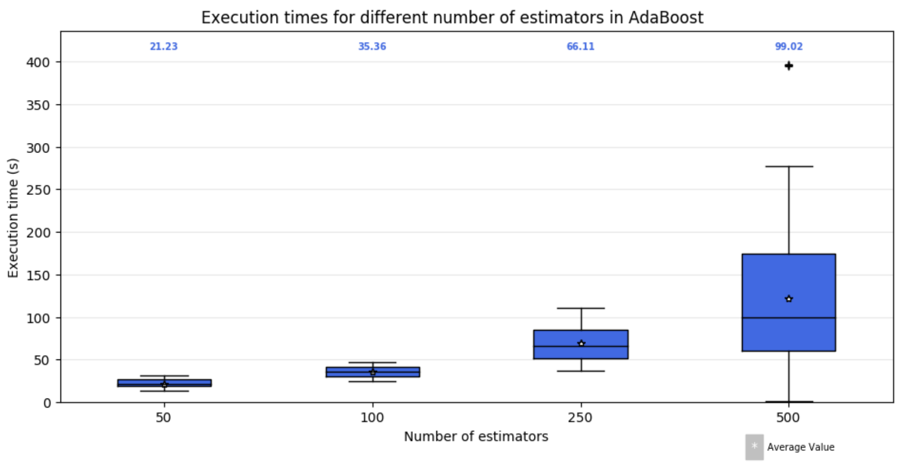


FIGURE 5.9: Boxplot of the execution time of AdaBoost for different number of estimators.

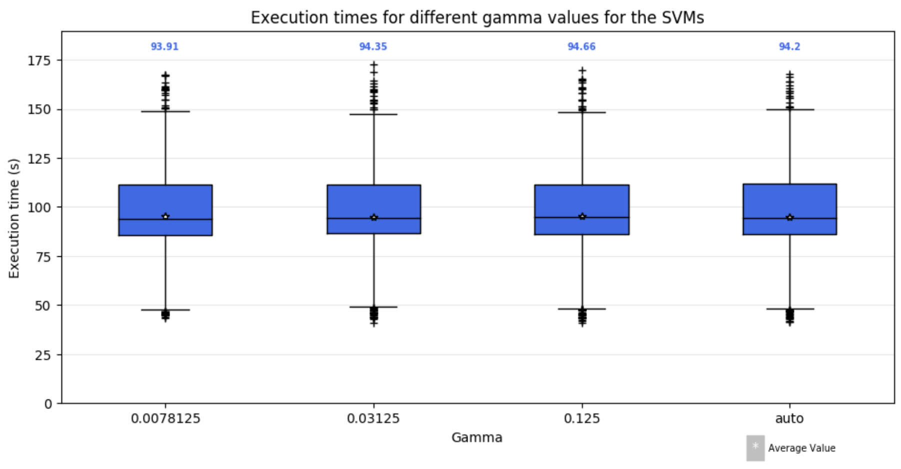


FIGURE 5.10: Boxplot of the execution time of SVMs for different values of γ.

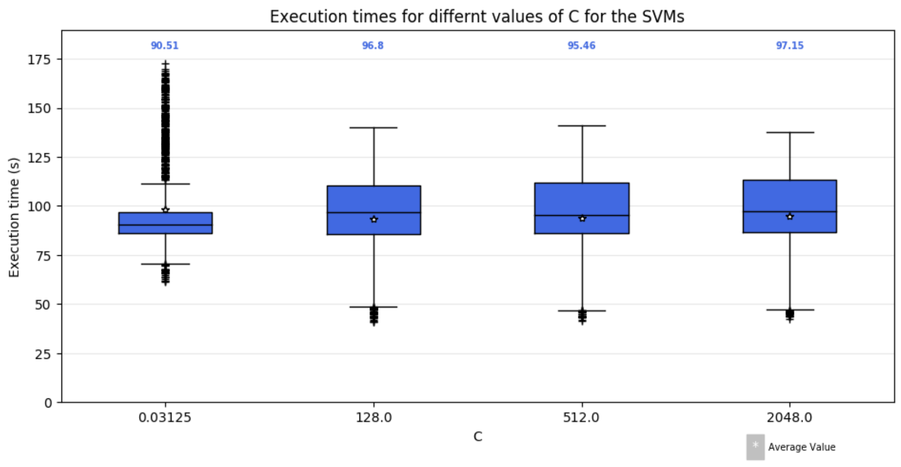


FIGURE 5.11: Boxplot of the execution time of SVMs for different values of C.

Model	Size (KB)
Linear Regression	0.6 ± 0.0
Random Forest	$48,546.1 \pm 72,040.7$
AdaBoost	37.8 ± 40.0
SVM	355.6 ± 192.0
Neural Network	39.0 ± 60.2

TABLE 5.2: Mean and standard deviation of the size of each model after serialization and compression, with pickle and gzip respectively, throughout all the trading performance experiments.

The mean size of all models (without taking into account linear regression which is only used for reference), is 12KB. The total number of models saved for each configuration is $m = tf_y * tp$, where tf_y is the yearly trading frequency (1 if we trade each year), and tp is the trading period length in years. We save a model for each trading session per configuration. For the first and third trading experiments, around 170K models saved, and close to 340K for the second experiment. This means that the average disk space required to cache experiments 1 and 3 is 20.8GB, and around 42GB for the second one. These sizes can greatly increase with higher trading frequencies. The total sizes can be quite large and may this technique may not suitable outside HPC environments with limited amounts of storage. However, in our experiments, sacrificing this amount of disk space was preferable to losing all data if the 10 hours time limit of MareNostrum 4 cancelled a job. Moreover, there are overlapping configurations in the three experiments so the total size is smaller and the overlapping models are only trained only once.

5.2 Trading performance

We set up three experiments to evaluate the performance of the different models. The first experiment compares Graham's criteria and classification models in the most restricted scenario out of the three experiments: we only allow long positions because Graham's criteria can only be used for stock screening. In the second experiment, we allow both long and short positions, and we compare classification and regression models. In this experiment, we use the regression models as a stock screening method. In the final experiment, we only evaluate the regression models. In this setting, we use the models' predicted returns to build a stock ranking and pick the best/worst k stocks in each trading session. The classification models and Graham's criteria are excluded from the third experiment because neither can be used to build a ranking.

The threshold value used for long positions ranges from $0 - 0.03$ with increments of 0.005 (i.e., from returns above 0 to returns above 3%). We also evaluate both the standard scaling dataset and the one using z-scores. Appendix A lists all parametrizations explored for each model. All models start with an initial budget of USD 100K. The plots that contain an S&P 500 Index revenue marker or threshold refer to the revenue obtained by investing these initial 100K USD in the index. All the experiments are evaluated in the trading period ranging from 2009 to 2018. Finally, unless stated otherwise, all the boxplots contain the total revenue of an investment without taking into account the fees of selling the positions. On the other hand, the chronological plots of the revenue, represent the total money available if, at each trading session, we sell all the current positions.

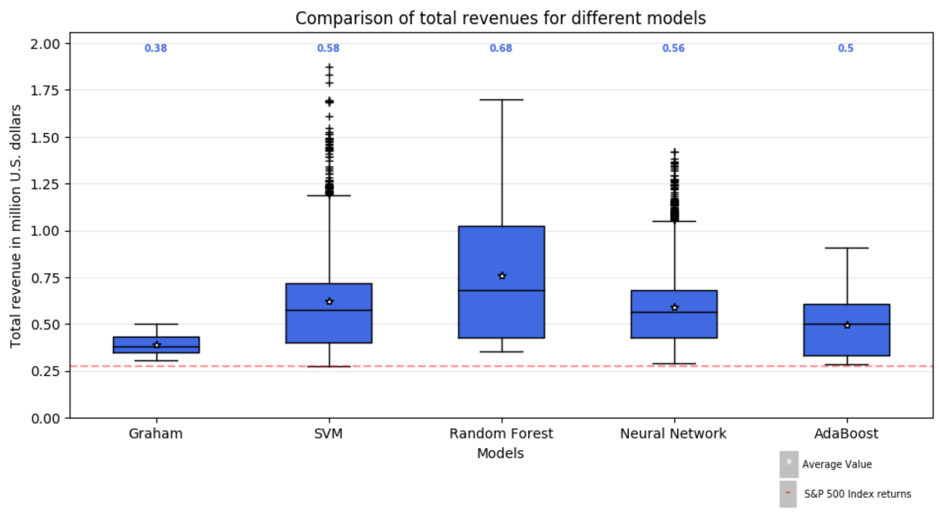


FIGURE 5.12: Boxplot of the first experiment's total revenue for different parametrizations grouped by model.

5.2.1 Experiment 1

This experiment pitches Graham's stock screening criteria against the ML models in stock screening. In this experiment, only long positions are allowed because Graham criteria cannot be used for shorting.

Figure 5.12 reports the total money obtained from selling all positions (including fees) at the end of the trading period. We see that random forest (RF) has the best mean and median. We explored a large number of parametrizations for SVMs which is probably the cause of having a more significant deviation (and outliers) w.r.t other models. Correct parametrizations of SVM lead to the best results despite having a similar mean to NN. We believe that exploring more network topologies and parameters, instead of using the basic MLP, could increase the performance.

Figure 5.13 shows the returns when scaling the data with z-scores or normalization. The results are worse when using z-scores. We believe that this is caused because the set of available stocks at each trading session is not constant. The goal of the z-scores is to normalize the data intelligently by grouping them by industries. In this scenario, we use the mean and standard deviation of the group instead of the ones from the whole training dataset as we do in normalization. However, the group means and standard deviations can vary wildly from one trading session to the next one due to the absence or presence of big stocks thus adding noise instead of information when scaling the data.

Figure 5.14 is the total revenue obtained by different threshold values. If the return of a sample is greater than the threshold it is labeled as a long position and neutral otherwise (no shorting allowed in this experiment). The higher the threshold, the higher the number of outliers (the best results). Despite this increase, the average and median values do not change much.

Figure 5.15 shows the revenue when training the models with last year's data or the previous two years. Training with only last year far outperforms using two years. This indicates that underlying data contains concept drift and that data older than a year adds noise and decreases accuracy. Moreover, using two years of data increases notably the training time of the models.

Figure 5.16 shows the returns when trading each semester or each year. Trading each semester obtains almost a 50% more returns than yearly. Trading each semester

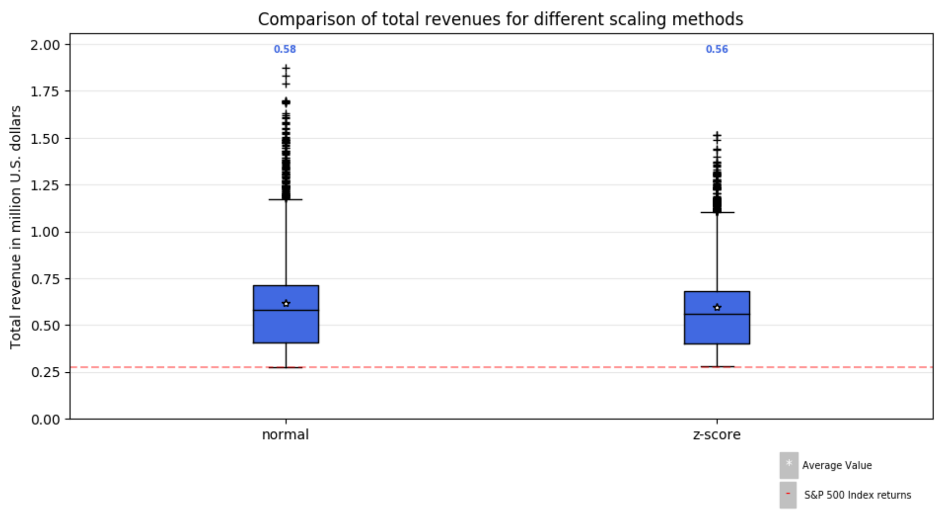


FIGURE 5.13: Boxplot of the first experiment's total revenue for normal scaling or z-scores.

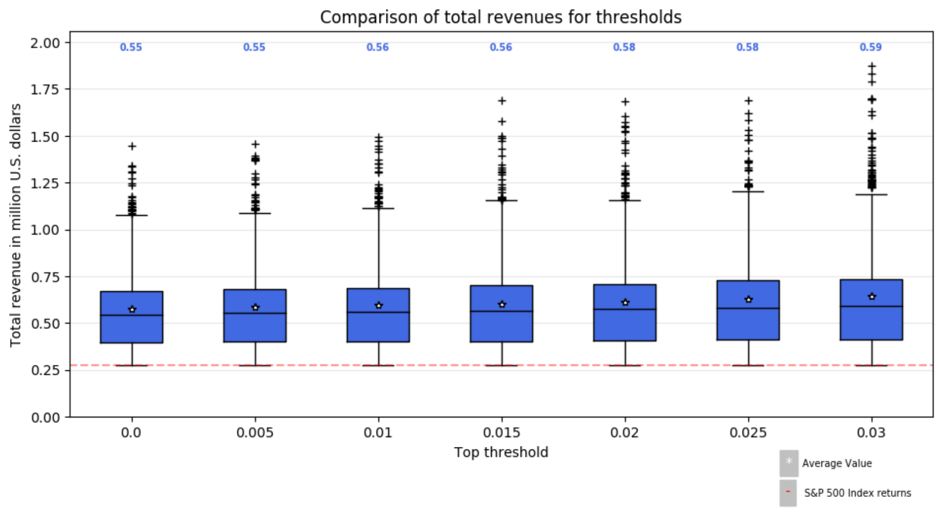


FIGURE 5.14: Boxplot of the first experiment's total revenue for different thresholds.

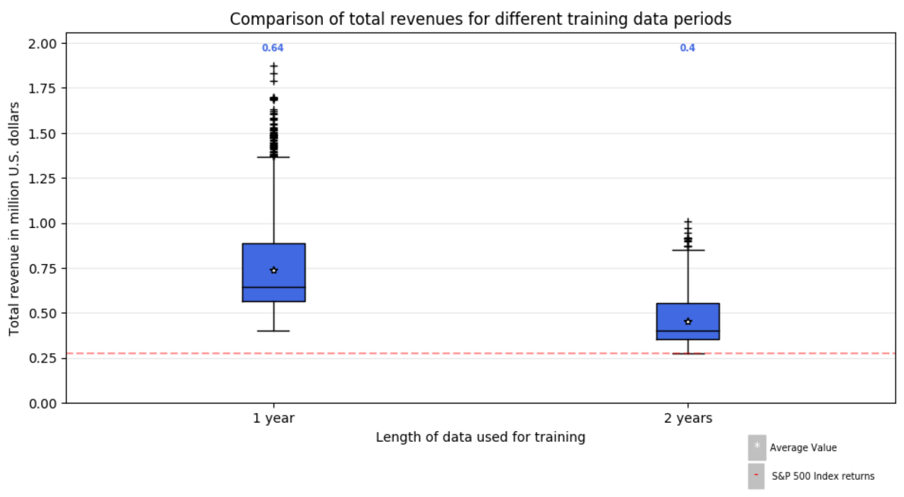


FIGURE 5.15: Boxplot of the first experiment's total revenue when taking either one or two previous years as training data.

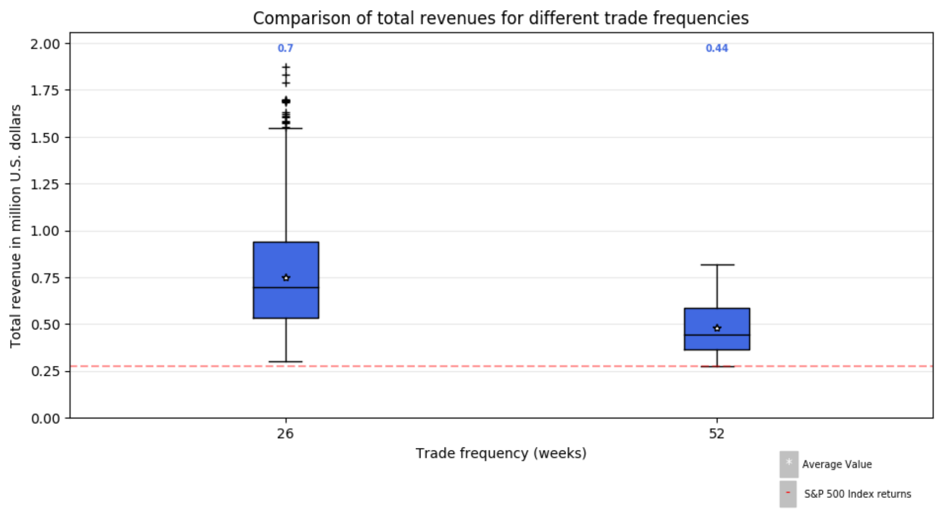


FIGURE 5.16: Boxplot of the first experiment's total revenue when trading every semester and year. Trading every six months yields far better results than yearly.

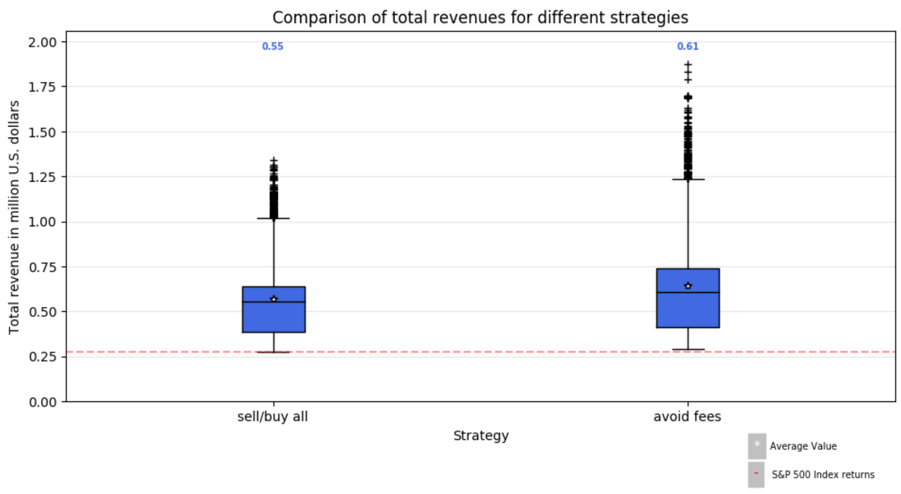


FIGURE 5.17: Boxplot of the first experiment's total revenue for trading strategies: *sell_all* and *avoid_fees*.

incurs into higher fees but predicting the returns of a stock a year ahead is harder than only a semester. Moreover, we have seen that using a single year to train the models is better than two due to concept drift. The same problem applies here.

Figure 5.17 shows the revenue obtained by each of the two proposed strategies: *sell_all* and *avoid_fees*. The strategy of trying to avoid fees by holding onto the stocks already owned whenever they are recommended again seems to work well, getting half million more of average revenue and close to a 60% more in the best cases.

Figure 5.18 shows that the more estimators used when training the AdaBoost model, the better results it gets. However, as seen in Figure 5.9 the execution time increase of AdaBoost increases far faster than the accuracy with respect to the number of estimators.

Figure 5.19 shows how the choice of solver affects the total revenue obtained by neural networks. LBFGS performs better than Adam. Our number of training samples is quite small (around 10K). In these small scenarios is usual that LBFGS converges faster and performs better.

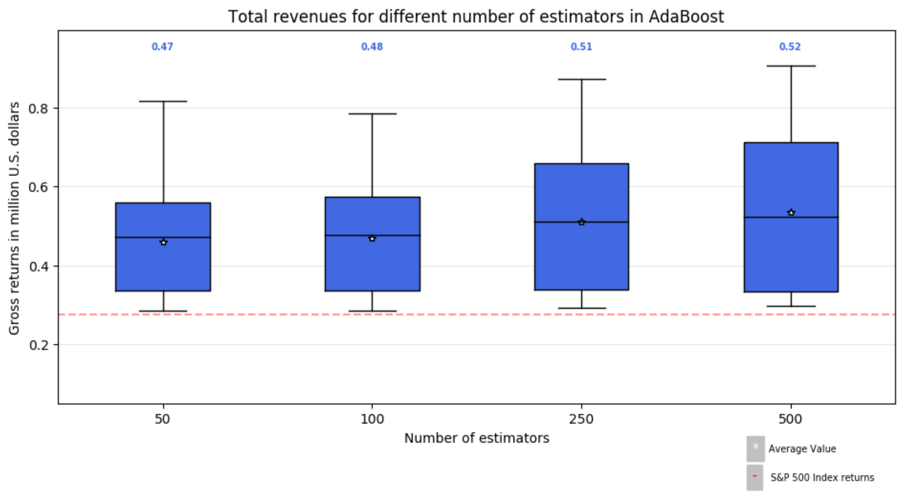


FIGURE 5.18: Boxplot of the first experiment's total revenue for different number of estimators for AdaBoost model.

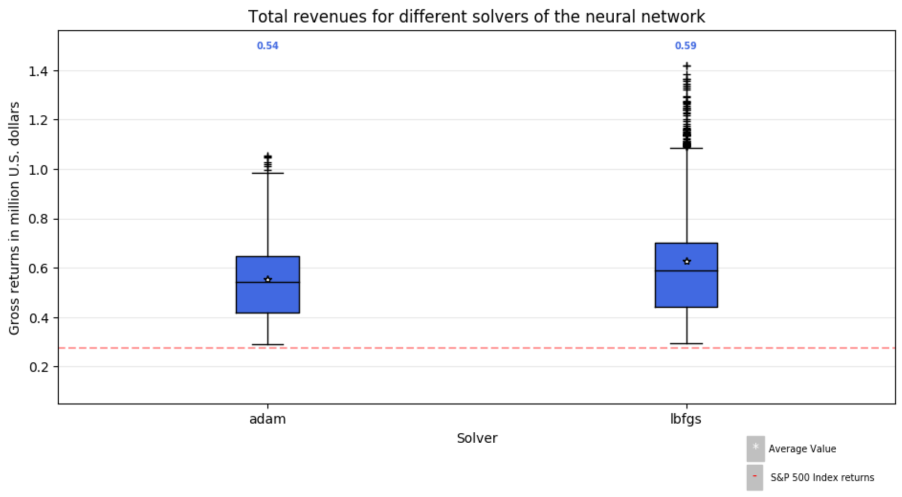


FIGURE 5.19: Boxplot of the first experiment's total revenue when training the neural network with different solvers.

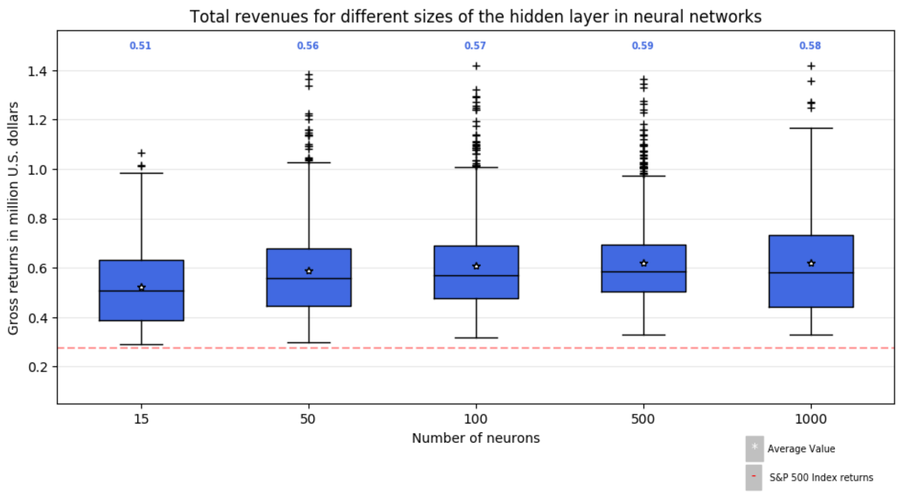


FIGURE 5.20: Boxplot of the first experiment's total revenue for different hidden layer sizes of the neural network.

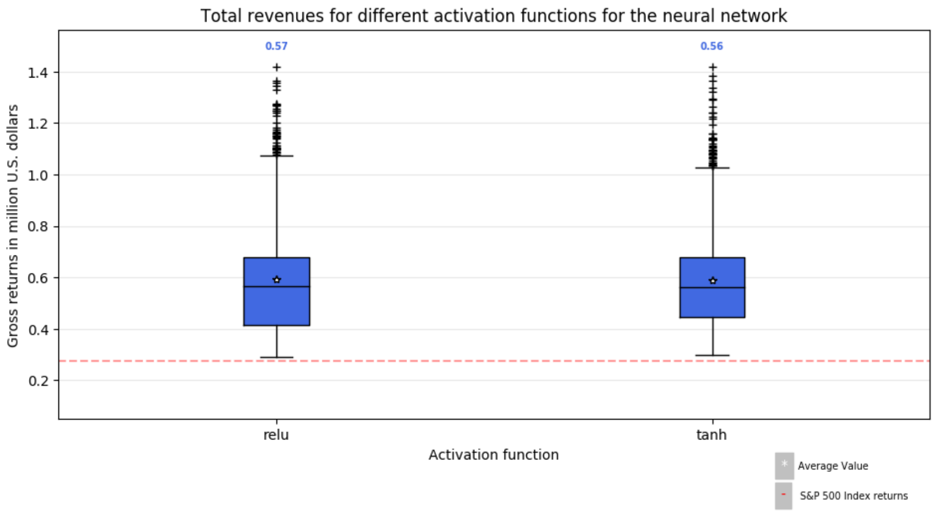


FIGURE 5.21: Boxplot of the first experiment's total revenue for different activation functions of the neural network.

Figure 5.20 shows how the number of neurons in the hidden layer affects the revenue obtained. The average return increases up to 500 neurons. For 1000 neurons, the mean revenue starts to decrease hinting that the optimal number of neurons must be in $[500, 1000)$. It is worth noting that the higher the number of neurons, the longer it takes to train the models. However, as we see here, it is worth spending more computing power examining large sizes because they can indeed improve the performance.

Figure 5.21 shows the results of using two different activation functions for the neural network. The ReLU activation has a slightly better mean than the hyperbolic tangent function but also more variance.

Figure 5.22 shows how the number of trees in the random forest affects the total revenue obtained. The average revenue slightly decreases with the number of trees. On the other hand, the lowest variance is obtained with 250 trees, more or less than that, the variance increases.

Figure 5.23 and 5.24 show how the γ kernel coefficient and the C penalty term

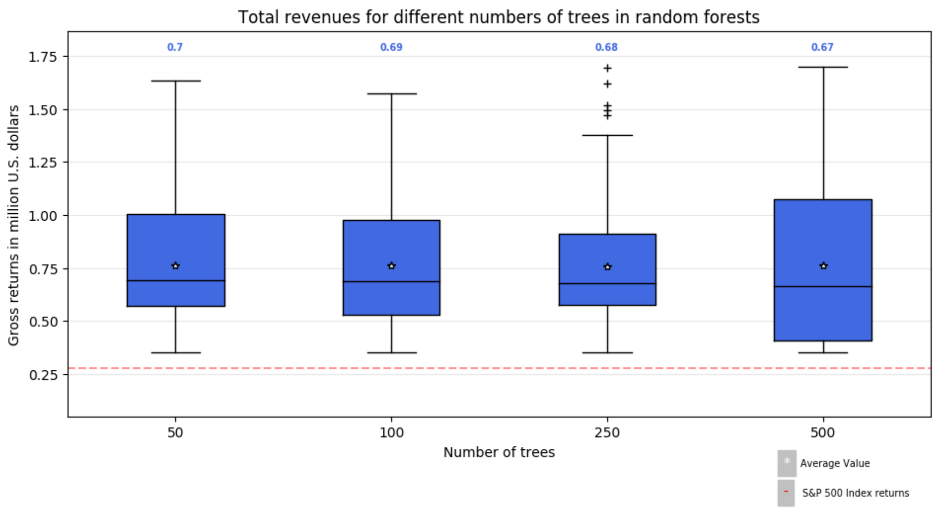


FIGURE 5.22: Boxplot of the first experiment's total revenue for different numbers of trees in each random forest.

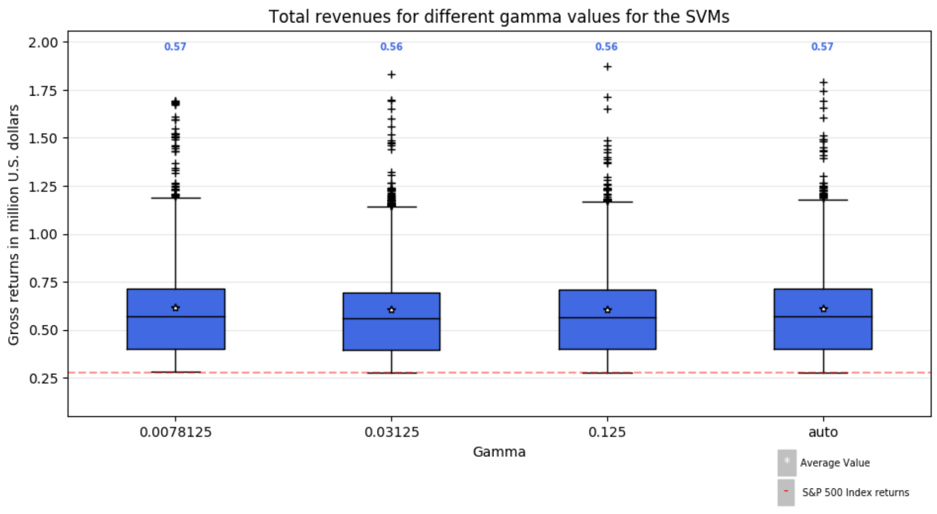


FIGURE 5.23: Boxplot of the first experiment's total revenue for different kernel coefficients gamma when using SVMs with RBF kernel. *Auto* gamma value defaults to $1/n_{features}$, in our example $1/20 = 0.5$.

of the SVMs affect the performance. For γ, we see that neither the average revenues nor the variances change much for most models, using the auto value ($\frac{1}{\text{n\_features}}$) is good enough. Concerning C, the optimal value appears to be around 2048.

5.2.2 Experiment 2

This experiment compares regression and classification models in stock screening. Both types of models use the same threshold values used to label the target in classification and to label the prediction in regression (see Figure 3.2). The extra information of the regressors (predictions can be ordered) is ignored to make the comparison fair.

Figure 5.25 shows the distribution of the net returns for each model and task type. Classification models obtain slightly higher best results but with higher variance. The introduction of short positions allows the worse parametrizations to lose

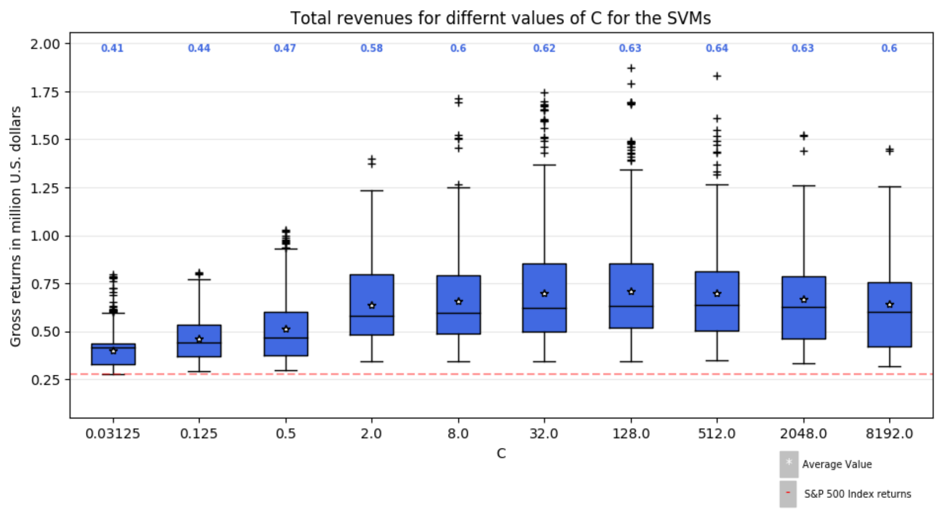


FIGURE 5.24: Boxplot of the first experiment's total revenue for penalty values C for the SVM model.

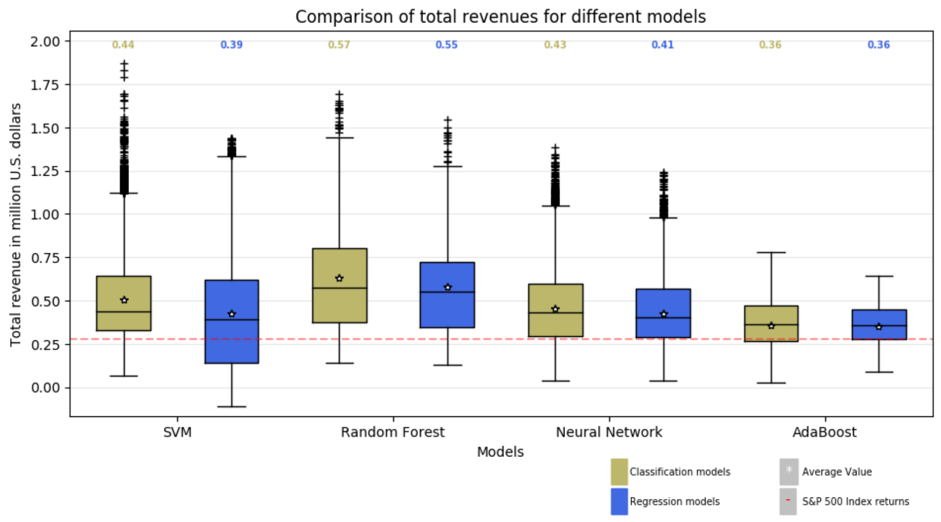


FIGURE 5.25: Boxplot of the second experiment with the total revenue of different parametrizations grouped by model.

money at a higher rate than with long positions, even finishing with debts (the model can not buy the already opened short positions). Finally, all the models' mean returns outperform the index and all IQRs, except the SVM regressor's, are above the index's returns.

Figure 5.26 shows the revenue at each session for the best models if all open positions are sold. The SVMs and random forests obtain the best results in both categories yielding around USD 1.75 million, outperforming the S&P 500 Index by a vast margin. The MLP neural network comes close in third place with a total revenue of around 1.3 million. AdaBoost is the last but still manages to get almost three times more revenue than investing on the index.

5.2.3 Experiment 3

This experiment evaluates the performance of regression models for stock ranking. We use the predicted return of the models to rank the stock in each session. When

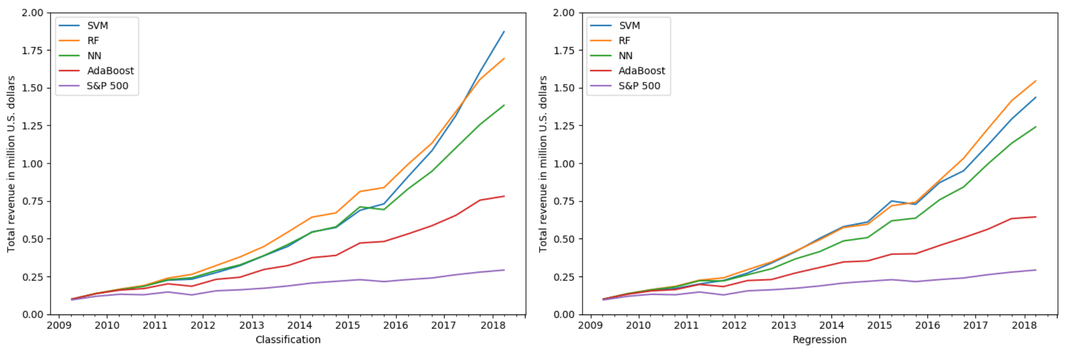


FIGURE 5.26: Evolution of the net revenue (after selling positions) for the second experiment by the best regressors and classifiers.

selecting the stocks, only the best/worst k stocks in the ranking are picked. Once selected, they are traded following one of the two strategies 3.4.3.

Again all models start with 100k. The thresholds used for trading are range from $0 - 0.03$ with increments of 0.01. The values tested for k are 10, 25, and 50 to build portfolios of size 20, 50, and 100 respectively.

Figure 5.27 shows the total money achieved by each model at the end of the trading sessions. Using the regressors for ranking improves the performance of the models dramatically. Almost all models outperform the index except some that lose money or get into debt. The best models are far above the average underlining how important it is to explore a large number of parametrizations to find the best ones.

The bests models are random forests, which can turn the initial 100K into around USD 17.5 million. It is important to note that medians and averages are quite similar to those in Figure 5.12 and Figure 5.25 from previous experiments, but the best models outperform these averages by order of magnitude. The best models are consistently reported by the random forests models. The best result is 17.6 USD millions from the initial 100K underlining how important it is to be able to explore as many configurations as possible to choose the best one.

Figure 5.28 adds to the economic theory on the goodness of diversification that the ampler the portfolio, the more consistent the revenue growth and the less relevant the sophistication of the model: there is no need to tune parameters to the best possible for the results are in line with the average model. For the values $[10, 25, 50]$ of the parameter k, we select that number of best and worst stocks building a portfolio of 20, 50, and 100 stocks respectively. However, if there are not enough predictions above the thresholds the portfolio size might be smaller. For the cases where either short or long positions are deactivated the portfolio's maximum size is k.

Figure 5.29 shows the evolution of the best and worst models of this experiment. The random forest's worst execution is the only model that does not lose money. Without careful tuning of their parameters, all the other models lose money or go into debt like SVMs.

Table 5.3 shows the parameters of the trend lines from Figure 5.29. The best results for all models are obtained trading every semester and without shorting while the worst results are obtained trading each year only short positions. Concerning the scaling method, its effect is not as clear and appears to depend on the model used. The best portfolio size is 20 ($k = 10$) for all models while worst results use $k = [50, 100]$, except the neural networks whose worst result is also with $k = 10$. For the trading strategy, even though *avoid_fees* has a better average return than *sell_all* in

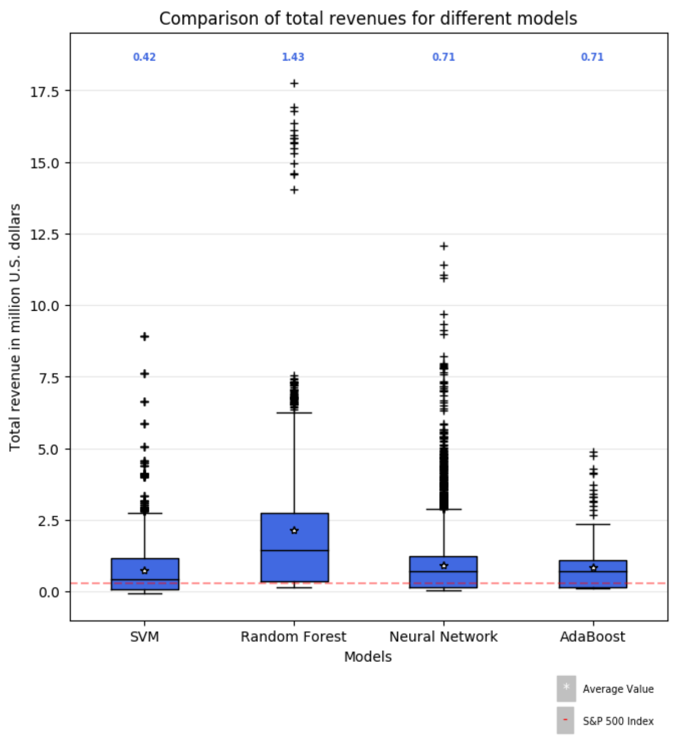


FIGURE 5.27: Boxplot of the third experiment with the total revenue obtained by each model.

Experiment 1, here *avoid_fees* is not used by any of the best models. For both the random forest and AdaBoost models, the best and worst model use 50 trees/estimators so the model's size is not as relevant as the other parameters. Moreover, the execution time of the random forest grows linearly with the number of trees, making it very slow without using HPC-grade computing nodes that allow parallel training. For the neural networks, the best activation and size are *ReLU* and 500 neurons; the worst, *tanh* and 15 neurons. This means that increasing the number of neurons is worth it. However, the time required to train larger networks increases from around 30 seconds with 15 neurons to an average of 20 minutes for 1000 neurons (see Figure 5.5). Fortunately, *ReLU* activation, which is the fastest activation function, also gets the best results so it can ease a bit the computational burden. For SVM, larger penalty parameters C produce worse results, probably because it causes the model to overfit. For the γ parameter of the RBF kernel, smaller values are better also. Finally, for the reference linear regression, the results follow the general trend: best, trading each semester long positions; worst, trading each year only short positions. The fact that the best models only trade long positions, and the worst only short positions, is caused by the fact that the S&P 500 index has a growing trend throughout all our trading period. In this situation the expected average return of long positions is positive, and the one from short positions is negative so, going long yields higher returns.

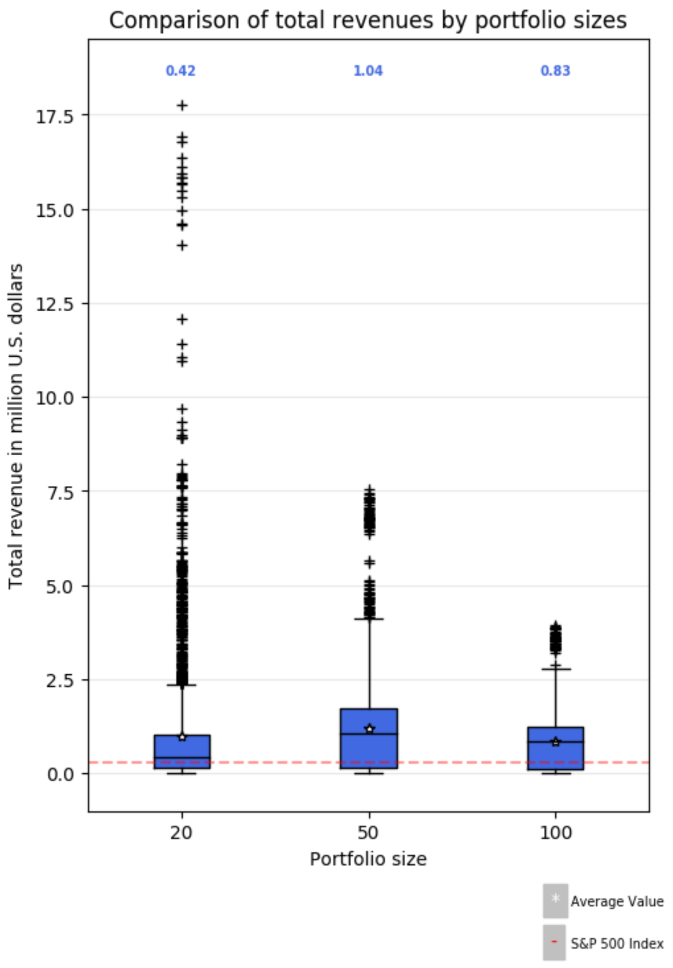


FIGURE 5.28: Boxplot of the third experiment total revenues grouped by different portfolio sizes.

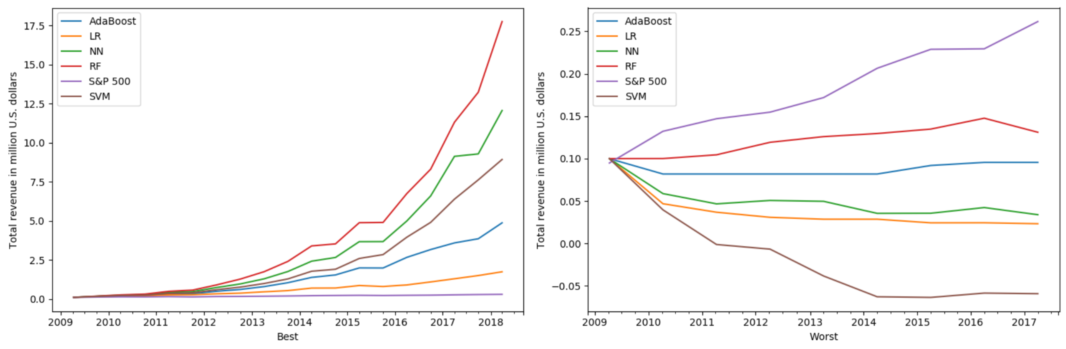


FIGURE 5.29: Net revenues of the third experiment's best and worst models.

TABLE 5.3: Best and worst configurations of each model for experiment 3 using one year of training data. An infinity value as top/bottom threshold means that long/short positions are deactivated respectively. The revenue is in USD.

Mode	Scaling	Bot	Top	Trade freq.	Strategy	Model parameters	K	CPUs	Revenue
Random Forest	z-score	$-\infty$	0.01	6 m	sell_all	Trees: 50	10	12	17.759.224,80
Neural Network	z-score	$-\infty$	0	6 m	sell_all	Hidden Layer Sizes: 500 Solver: lbfgs Activation: relu	10	1	12.061.462,50
SVM	normal	$-\infty$	0.03	6 m	sell_all	C: 0.03125 Gamma: 0.03125	10	1	8.927.041,53
AdaBoost	normal	$-\infty$	0.01	6 m	sell_all	Estimators: 50	10	1	4.870.173,30
Linear Regression	z-score	$-\infty$	0.03	6 m	sell_all	-	10	1	1.736.454,84
Random Forest	z-score	0	∞	1 y	avoid_fees	Trees: 50	50	12	131.105,80
AdaBoost	normal	-0.01	∞	1 y	sell_all	Estimators: 50	100	1	95.609,18
Neural Network	normal	-0.01	∞	1 y	sell_all	Hidden Layer Sizes: 15 Solver: lbfgs Activation: tanh	10	1	33.913,25
Linear Regression	z-score	-0.01	∞	1 y	sell_all		50	1	23.294,86
SVM	normal	-0.03	∞	1 y	avoid_fees	C: 2048 Gamma: 0.125	100	1	-59.198,42

Chapter 6

Conclusions

We have evaluated the parallelization performance of *HPC.FASSR* through the weak and strong scaling experiments. The results show that our system has linear speedup up to 1532 CPUs. However, workload imbalance caused by either too few models to train per core or by slow models might degrade the performance in some scenarios. We have also seen that the best results are reported by the slowest models showcasing the importance of having HPC-like computing resources to train them and find the best configurations in a reasonable amount of time.

For the trading performance, we set up three experiments: first, to compare Graham's criteria with classification models in stock screening with long positions; second, to compare classification and regression models with long/short positions; and finally, to evaluate regressors in stock ranking. The results of the first experiment show that all models have better average returns than Graham's criteria and all of them outperform S&P 500 the index. The comparison in Experiment 2 showed that classifiers and regressors perform similarly and that SVM and random forests are the best in both tasks with the best SVM gaining a total revenue of around USD 1.8M. In the last experiment, we used regression information to build a stock ranking to then pick up the best and worst *k* stocks. We found that using the regression information for stock ranking yields much higher revenues and with the best revenues obtained with 20 stocks in the portfolio. The best model is by a large margin the random forests which converts the initial USD 100K to 17.5 million.

Chapter 7

Further Work

In this last chapter we lay out some ideas for future work which were excluded from the project due to lack of time or because they were out-of-scope.

We have compared our models to Graham's criteria, but it would be interesting to add more expert's criteria to the benchmark, like Warren Buffett's, and evaluate them under different market regimes.

It would also be interesting to add more sklearn models to the mix and evaluating them. This could test the performance of *HPC.FASSR* under much larger experiments. We would also like to try time-series forecasting models like LSTM networks [18] and evaluate how to integrate this different kind of models into the evaluation pipeline.

The checkpointing system, although simple, was crucial to reusing data of the most massive executions whenever problems like time limit's or failure happened. Some great improvements to be done are: cache management options, like removing models once the execution is successful; and a naming system where the hash is computed from some fingerprint of the data thus removing the need of knowing all the parameters' values used to create the dataset.

The performance of the parallelization may become unbalanced if the model's training times vary too much between them. To mitigate this problem, it would be interesting to set a finer granularity and run the training of each session in a separate task. Then the resulting model of each training task should be fed to a trading/evaluation task. To this end, we have recently come across the distributed computing library [31] (dislib). This library is also built on top of PyCOMPSs and offers a grid search method out-of-the-box where each parametrization is evaluated in a separated task. The grid search could be used to, in each session, train all models and parametrizations and get the best model *per trading session* instead of just the best at the end. On the one hand, this would delegate all the grid search handling to the dislib. On the other, this new structure would allow us to create a new model which automatically use the model reporting the best results with the training data of the session.

As a final note, we would like to see *HPC.FASSR* become a worth and usable open-source project for research. The first step would be to lay out a clearer specification and design a cleaner API so that users can plug in their custom data managers and models more easily. Anyone interested in contributing or using *HPC.FASSR* will be very welcome and appreciated.

Appendix A

Parametrizations

This Appendix lists the most important parameters used to train each of the machine learning models. All implementations come from scikit-learn version 0.20.3, so the parameters not listed are the default values provided by the library.

When expressing ranges, we use the python notation $range(min, max, step)$ where min and max are the lower and upper bounds respectively, and step is the distance between two consecutive items.

The following parameters are common for all models:

- Scaling = *standard*, *z-scores*
- Trading frequency = *semester*, *year*
- Training data = $1, 2$ years
- Trading strategies = *avoid_fees*, *sell_all*
- K (stock ranking) = $10, 25, 50$

A.1 SVM

- Kernel = *RBF*
- C = $[2^i \; \forall i \in range(-5, 15, 2)]$
- Gamma = $[2^i \; \forall i \in range(-15, 3, 2)]$

A.2 Random Forest

- Number of trees = $[50, 100, 250, 500]$
- Criterion = *Gini Index*
- Max features tested = $\sqrt{\text{n\_features}}$
- Sample Bootstraping = *True*

A.3 Neural network

- Size of the hidden layer = $[5, 10, 15, 20, 30, 50, 100, 200, 500]$
- Activation function = *tanh*, *ReLU*
- Solver = *LBFGS*, *Adam*

- L2 penalty regularization = 0.0001
- Early stopping = *False*

A.4 AdaBoost

- Number of estimators = $[50, 100, 250, 500]$
- Boosting algorithm = *SAMME.R* [17]
- Base estimator = *decision tree* $(max\_depth = 1)$

Appendix B

Trading strategies pseudocode

Figure B.1 shows how the portfolio positions are updated with the recommendations with the trading strategy that focuses on avoiding fees.

Figure B.2 shows how the portfolio positions are updated with the *avoid fee* with the recommendations with the trading strategy that sells and buys all positions at each session.

```
update_positions(available_money,
                 old_positions,
                 recommendations,
                 portfolio_size):

    new_positions = [] # empty list
    for each position in old_positions:
        if position $\in$ recommendations:
            new_positions.append(position)
        else:
            profits += sell(position)
    pending_stocks = portfolio_size - length(new_positions)
    money_per_stock = available_money / pending_stocks

    while pending_stocks > 0
        stock = choose next(recommendation) in alphabetical order
        position, cost = buy(stock, money_per_stock)
        if money_per_stock is enough to open the position:
            new_positions.append(position)
          available_money - cost
          pending_stocks -= 1

    return new_positions
```

FIGURE B.1: Pseudocode showing how the positions are updated with the *avoid fees* strategy each trading session given the available amount of money, the old positions, a set of recommendations, and a desired portfolio size.

```
update_positions(available_money,
                 old_positions,
                 recommendations,
                 portfolio_size):

    new_positions = [] # empty list
    for each position in old_positions:
        profits += sell(position)

    pending_stocks = portfolio_size - length(new_positions)
    money_per_stock = available_money / pending_stocks

    while pending_stocks > 0
        stock = choose next(recommendation) in alphabetical order
        position, cost = buy(stock, money_per_stock)
        if money_per_stock is enough to open the position:
            new_positions.append(position)
          available_money - cost
          pending_stocks -= 1

    return new_positions
```

FIGURE B.2: Pseudocode showing how the positions are updated with the *buy/sell all* strategy each trading session given the available amount of money, the old positions, a set of recommendations, and a desired portfolio size.

Bibliography

[1] Ramon Amela et al. “Enabling Python to Execute Efficiently in Heterogeneous Distributed Infrastructures with PyCOMPSs”. In: *Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing*. 2017, 1:1–1:10.

[2] Anton Andriyashin, Wolfgang K Härdle, and Roman Vladimirovich Timofeev. “Recursive portfolio selection with decision trees”. In: (2008).

[3] *Apache Mesos*. URL: https://mesos.apache.org/.

[4] Argimiro Arratia. “Computational finance”. In: *An Introductory Course with R, Atlantis Studies in Computational Finance and Financial Engineering* 1 (2014).

[5] Michel Ballings et al. “Evaluating multiple classifiers for stock price direction prediction”. In: *Expert Systems with Applications* 42.20 (2015), pp. 7046–7056.

[6] Barcelona Supercomputing Center (BSC). *Extrae Tool*. URL: https://tools.bsc.es/extrae.

[7] Barcelona Supercomputing Center (BSC). *MareNostrum 4 User Guide*. 2019. URL: https://www.bsc.es/support/MareNostrum4-ug.pdf.

[8] Barcelona Supercomputing Center (BSC). *Paraver Tool*. URL: https://tools.bsc.es/paraver.

[9] Lars Buitinck et al. “API design for machine learning software: experiences from the scikit-learn project”. In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, pp. 108–122.

[10] Jesús Labarta et al. Enric Tejedor Rosa M. Badia. “PyCOMPSs: Parallel computational workflows in Python”. In: *The International Journal of High Performance Computing Applications (IJHPCA)* 31 (2017), pp. 66–82. URL: http://dx.doi.org/10.1177/1094342015594678.

[11] Thomas Fischer and Christopher Krauss. “Deep learning with long short-term memory networks for financial market predictions”. In: *European Journal of Operational Research* 270.2 (2018), pp. 654–669.

[12] Python Software Foundation. *PEP 318 – Decorators for Functions and Methods*. URL: https://www.python.org/dev/peps/pep-0318/.

[13] Python Software Foundation. *Python Language*. URL: https://www.python.org/.

[14] Yoav Freund and Robert E Schapire. “A decision-theoretic generalization of on-line learning and an application to boosting”. In: *Journal of computer and system sciences* 55.1 (1997), pp. 119–139.

[15] Benjamin Graham. *The intelligent investor*. Prabhat Prakashan, 1965.

[16] Interactive Brokers Group. *Interactive Brokers*. URL: https://www.interactivebrokers.co.uk.

[17] Trevor Hastie et al. “Multi-class adaboost”. In: *Statistics and its Interface* 2.3 (2009), pp. 349–360.

[18] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[19] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[20] Thomas Kluyver et al. "Jupyter Notebooks - a publishing format for reproducible computational workflows". In: *ELPUB*. 2016.

[21] Christopher Krauss, Xuan Anh Do, and Nicolas Huck. "Deep neural networks, gradient-boosted trees, random forests: Statistical arbitrage on the S&P 500". In: *European Journal of Operational Research* 259.2 (2017), pp. 689–702.

[22] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), p. 436.

[23] Dong C Liu and Jorge Nocedal. "On the limited memory BFGS method for large scale optimization". In: *Mathematical programming* 45.1-3 (1989), pp. 503–528.

[24] Burton G. Malkiel and Eugene F. Fama. "EFFICIENT CAPITAL MARKETS: A REVIEW OF THEORY AND EMPIRICAL WORK*". In: *The Journal of Finance* 25.2 (1970), pp. 383–417. DOI: 10.1111/j.1540-6261.1970.tb00518.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1540-6261.1970.tb00518.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1540-6261.1970.tb00518.x.

[25] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 51 –56.

[26] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[27] Quantopian. *Quantopian*. URL: https://www.quantopian.com/.

[28] Pavel Sevastjanov and Ludmila Dymova. "Stock screening with use of multiple criteria decision making and optimization". In: *Omega* 37.3 (2009), pp. 659–671.

[29] *Slurm Workload Manager*. URL: https://slurm.schedmd.com/.

[30] S. van der Walt, S. C. Colbert, and G. Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science Engineering* 13.2 (2011), pp. 22–30. ISSN: 1521-9615. DOI: 10.1109/MCSE.2011.37.

[31] Workflows and Distributed Computing Group. *The Distributed Computing Library (Dislib)*. URL: https://dislib.bsc.es.

[32] Eric Zivot and Jiahui Wang. *Modeling financial time series with S-Plus®*. Vol. 191. Springer Science & Business Media, 2007.