Portfolio Optimization using a Big Data framework

A Passive Management, Spark and Genetic Algorithm approach

Pedro Miguel de Oliveira Barata Ferreira Caeiro

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Nuno Cavaco Gomes Horta

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves

Supervisor: Prof. Nuno Cavaco Gomes Horta

Members of the Committee: Prof. Francisco João Duarte Cordeiro Correia dos Santos

October 2018

Abstract

This work analyzes the literature regarding portfolio management and proposes to perform this task automatically. A portfolio management model – based on the seminal work of Markowitz–, a metaheuristic for mathematical optimization – genetic algorithm – and a Big Data framework – Spark – are used in the system to guide stock selection. The main goal is to design a portfolio management system able to use the extensively studied genetic algorithms to solve a portfolio optimization problem while being capable of storing and processing large datasets. The secondary goal is to provide a basis for further testing of Big Data frameworks in portfolio management. Computer performance validation was done via analyzing the impact of concurrency, iteration processing and Spark memory configurations on runtime. This validation showed that the system can reduce runtime via concurrency – more than 90% latency speedup between 1 and 4 cores – and, thus, it is viable to advance with further studies regarding scalability in clusters. The system also attained adequate financial performance by beating the SPY exchange traded fund consistently, albeit not always the buy and hold strategy, in the first two weeks after optimization. In these two weeks, the solution return on investment gains reached up to 2% above SPY.

Keywords

Markowitz, Big Data, Spark, Genetic Algorithm, Mean-Variance Analysis

Resumo

Este trabalho analisa a literatura relativa à gestão de portfólios financeiros e propõe realizar esta tarefa automaticamente. Um modelo de gestão de portfólios – baseado no trabalho seminal de Markowitz–, uma metaheurística para otimização matemática – algoritmo genético – e uma *framework* de *Big Data* – Spark – são usados neste sistema para guiar a escolha de ativos. O principal objetivo é desenhar um sistema de gestão de portfólios capaz de usar um algoritmo genético para resolver problemas de otimização de portfólio. Simultaneamente, o sistema é capaz de persistir e processar grandes quantidades de dados. O objetivo secundário é providenciar uma base para testes adicionais de *frameworks* de *Big Data* no domínio da gestão de portfólios. A validação da performance computacional foi feita através da análise do impacto de concurrência, processamento iterativo e configurações de memória do Spark no tempo de execução. Esta validação mostrou que o sistema consegue reduzir o tempo de execução através de concurrência, alcançando um *speedup* superior a 90% entre 1 e 4 cores. Este facto permite concluir que é viável avançar com estudos adicionais em relação à escabilidade em *clusters*. O sistema também obteve uma performance financeira adequada durante as primeiras duas semanas depois da otimização, tendo melhores resultados que o *exchange traded fund* SPY que mede o Standard & Poor's 500, consistentemente, com algumas configurações. Estes ganhos chegam a ser de 2% superiores ao SPY, em termos de *return on investment* nas primeiras duas semanas após a otimização.

Palavras Chave

Markowitz, Big Data, Spark, Algoritmo Genético, Análise Média-Variância

Acknowledgments

First, I would like to express my gratitude to my supervisor professor Nuno Horta for his support, advice and guidance during the periods of research and development of this thesis. I would also like to thank professor Helena Galhardas for the knowledge sharing and advice given.

I would like to thank my mother, Ana Paula, for never letting me stop believing that I can achieve my goals if I prepare and work towards them. And I thank my father Carlos Augusto, for always supporting me in my endeavors and for being understanding and patient.

I would also like to thank my grandparents José Manuel, Maria Natália, Isidro and Maria de Lourdes.

I thank my dog Ragnar for keeping me company while writing this thesis, for providing comfort and the occasional healthy distraction.

Last, but most definitely not least, I would like to thank my wonderful girlfriend Ana for all her love, support, encouragement and patience during this journey.

Table of Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
CBIPSO	Combination of Binary and Improved Particle Swarm Optimization
CDH	Cloudera Distribution for Hadoop
CPU	Central Processing Unit
DAG	Direct Acyclic Graph
DE	Differential Evolution
DJIA	Dow Jones Industrial Average
EA	Evolutionary Algorithm
EMH	Efficient Market Hypothesis
ETF	Exchange Traded Fund
ETL	Extract, Transform, Load
FA	Fundamental Analysis
GA	Genetic Algorithm
GDP	Gross Domestic Product
GFS	Google File System
HC	Hill Climbing
HDD	Hard Disk Drive
HDFS	Hadoop Distributed File System
JVM	Java Virtual Machine
MO	Multi-objective
MPT	Modern Portfolio Theory
MTGA	Memetic Tree-based Genetic Algorithm
NSGA-II	Non-dominated Sorting Genetic Algorithm II
NYSE	New York Stock Exchange
OS	Operating System
PAES	Pareto Archived Evolution Strategy

PBIL	Population Based Incremental Learning
PESA-II	Pareto Envelope-based Selection Algorithm II
POP	Portfolio Optimization Problem
PSO	Particle Swarm Optimization
QP	Quadratic Programming
RAM	Random Access Memory
RDBMS	Relational Database Management System
RDD	Resilient Distributed Dataset
ROI	Return on Investment
ROR	Rate of Return
S&P500	Standard & Poor's 500
SA	Simulated Annealing
SO	Single-objective
SPEA-II	Strength Pareto Evolutionary Algorithm II
SQL	Standard Querying Language
TA	Technical Analysis
TGA	Tree-based Genetic Algorithm
TS	Tabu Search
USA	United States of America
VEGA	Vector Evaluated Genetic Algorithm
VM	Virtual Machine

1. Introduction

Nowadays, personal finance, which is considered the act of managing one's financial assets, is an increasingly needed skill. With an ever unpredictable geopolitical and economic landscape, where even renowned private banks need government bailouts, managing personal investments for the average person is no longer as simple as making a term deposit. Whereas up to the 2008 financial crisis term deposits offered a relatively risk-free investment with decent returns, currently, a large portion of offered interest rates are so low that they fall below inflation. This effectively reflects itself in a loss of purchasing power, creating a growing need for diversified personal investments with prospects of medium to high returns at low risk levels. The collection of investments held by an entity or person is called a portfolio. Portfolios can be considered as being low or high risk depending on the assets selected and their types. For example, while bonds are considered low risk since governments tend to always pay their debts, stocks have higher risk because their value is volatile. They can even lose all value if a company goes bankrupt. Figure 1.1 depicts the constitution of generic high and low risk portfolios [1–3].

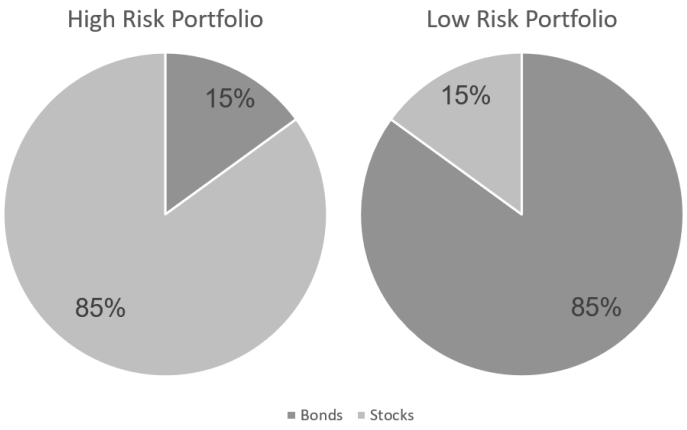

Figure 1.1: Generic examples of high and low risk portfolios according to asset types.

Portfolio management is a field of its own and usually takes one of two primary approaches: *active*, which tries to beat a chosen benchmark market index at the expense of extra risk taken, and *passive*, where the objective is to mimic the performance of the benchmark index with the lowest possible risk. Different economic theories, which try to explain how the financial market evolves, seem to both support and discredit the two different management approaches [1, 4].

Since most people do not want to learn about personal finance and portfolio management, nor are they willing to trust their finances to other people, there exists a real need for intelligent systems with the ability to suggest and manage investments. With the advent of the Internet, historical financial data required to make informed investment choices is readily available, such as opening and closing prices. This, alongside with the advances in computer hardware and artificial intelligence, have made portfolio management an attractive topic to study.

Mathematical optimization is a discipline of Mathematics and Computer Science which studies the problems of finding the best solution given a set of alternatives. Applying it to portfolio management enables the analysis of all historical data to choose the best combination of assets perceived to yield high return at low risk [4, 5].

As the amount and types of traded financial assets (or instruments) becomes larger, so does the historical datasets that portfolio management systems need to analyze. *Big Data* is a term which encompasses the study of these large datasets and the techniques needed to analyze them. Recently, there has been a growth in frameworks that were designed specifically for processing and analyzing datasets ranging from a few dozen gigabytes to hundreds of terabytes by taking advantage of distributed computing and the cloud computing paradigm.

The problem that this work aims to solve is the management of a financial portfolio through mathematical optimization using a Big Data framework.

1.1. Motivation

Managing a portfolio through mathematical optimization was first introduced by Markowitz in 1952, coining the concept of a *Portfolio Optimization Problem* (POP). His work is the basis of *Modern Portfolio Theory* (MPT) and aims at obtaining, out of a set of assets, the portfolio which minimizes risk, measured as a variance, for a specified level of minimum wanted return [6, 7]. Despite being the most used formulation in literature for portfolio management, usually extended with additional constraints to model it more realistically, MPT is not the only mathematical model available for portfolio management. Markowitz's work is traditionally considered a *passive approach* [2, 8].

In fact, although MPT often yields results that beat the benchmark index, more *active management* strategies for portfolio management exist. They usually focus on finding over or undervalued assets through *Technical Analysis* (TA) or by using *Fundamental Analysis* (FA) to find the most promising assets regarding profitability in the long term. TA and FA are the two main techniques used by financial analysts and stockbrokers to make investment choices [2]. The use of TA or FA to manage a portfolio can also be modelled as an optimization problem [2, 9].

As there is controversy regarding which management strategy attains better consistent returns, both strategies are considered valid. A key concern with recent portfolio management systems is risk-awareness, due to an ever-changing macroeconomic environment. As such, models which account for risk still provide an interesting opportunity for relevant study when combined with novel programming techniques and experimental validation [1, 2, 9].

Since optimization problems are at the core of current portfolio management systems, methods to solve them need to be understood. Methods to solve optimization problems, clearly diverge into two main categories: *exact methods*, which can guarantee a globally optimal solution but are usually

computationally taxing, and *heuristic methods*, that have no mathematical guarantee of finding a globally optimal solution but are computationally less expensive [10].

Most works in the literature are mainly worried with solving the optimization problems subjacent to portfolio management in a computationally efficient manner. Since more complex models of the problem are intractable, the focus of most recent related works has shifted from traditional exact methods to heuristic methods [6, 11–16]. While heuristic methods are computationally preferred to exact methods, they present an additional liability: some of them converge to local optimal solutions instead of global optimal solutions. *Metaheuristics* are a specific type of heuristic methods which include high level procedures that employ strategies for besting local optimality while searching in complex solutions spaces to solve optimization problems [17–19]. This specific type of heuristic method has seen an increase in published works, being used to solve both passive and active management models [2, 6, 9, 11, 19].

The performance of metaheuristics is very dependent on the fine tuning of *input parameters* and *fitness functions*, which evaluate the quality of candidate solutions w.r.t. the optimization objectives. This creates some opportunity for continuous improvement regarding computational efficiency and accuracy, which is a minor motivation for this work.

In fact, many of the analyzed works evaluate their results by how computationally efficient the system is or how accurate it is w.r.t. to exact methods. With these metrics they attain results which are very promising. But, while this is a legitimate validation methodology in Computer Science, specifically for metaheuristics, it does not align with the original goal of financial portfolios.

It is important to state that while there is an exact solution to optimization problems, the problem this work wants to solve is the management of a profitable portfolio. Thus, solving the underlying optimization problem is simply how the portfolio management is achieved. The true goal of managing a portfolio is to attain the highest possible return while maintaining the lowest possible perceived risk. As such, this work is motivated by the idea of validating the system via the profitability of the managed portfolio, as measured by financial performance metrics, following the trend established by [2] and [9]. Those works achieved encouraging results and their future work propositions are motivations for this work, specifically the inclusion of parallel processing and the ability to analyze historical data regarding more than one financial market.

Historical data can be found online, for example at [20] and [21], and is available in heterogeneous data formats and for an increasingly large number of assets. As these datasets become larger the need for distributed and parallel computing to process and analyze them increases, since traditional systems become inadequate to give solutions in an acceptable time window. Recent Big Data frameworks, like Hadoop or Spark, introduce a chance to improve computational performance via distributed and parallel computing while abstracting from the programmer many of the downsides of such a system [22, 23]. These frameworks present an opportunity to achieve this work motivations regarding the ability to process large datasets. In addition to this, since they are relatively new frameworks, there is room to study their computational efficiency with several algorithms.

1.2. Goals

The goal of this work is to develop a software system that manages a portfolio through solving optimization problems with large datasets. This will be achieved using a domain tuned metaheuristic implemented in a Big Data framework. The system needs to deliver a near-optimal solution in a reasonable time-window. The portfolio will be managed from a passive management perspective.

This goal can be divided in the following sub-goals:

- Develop a scalable software architecture able to store historical financial market data and manage a portfolio through solving an optimization problem on demand;
- Study optimization methods and understand how they are used for portfolio composition;
- Compare available Big Data frameworks, understand the one that fits the portfolio management problem better and learn its good programming practices;
- Provide a basis for further study of portfolio optimization via Big Data frameworks.

1.3. Outline of the Document

The document is organized as follows:

- In Section 2, the theoretical background needed to understand the work will be summarized. This background will encompass the fields of finance, portfolio management, mathematical optimization and an introduction to Big Data;
- In Section 3, the state of the art in the problem domain will be analyzed. This analysis will be done by comparing the most relevant characteristics for the scope of this work. Moreover, relevance will be given to the study of Big Data frameworks and optimization methods;
- In Section 4, the architecture for the proposed portfolio management system will be summarily described, as well as a description of the proposed technologies to use;
- In Section 5, the validation methodology and concrete metrics that will be used to evaluate the solution will be described;
- Finally, Section 6 will present conclusions regarding the feasibility of the proposed work.

2. Background

This document encompasses a series of multidisciplinary subjects, from financial economics and portfolio management to mathematical optimization. Thus, some theoretical topics need to be introduced. In this section, we will do precisely that, varying in depth according to the requirements of the work to be developed. In Section 2.1 the topics of finance and financial economics, with a clear emphasis on the aspects pertaining to portfolio management will be covered. Section 2.2 will provide a very brief overview of mathematical optimization. Finally, Section 2.3 will introduce some Big Data concepts and technologies.

2.1. Finance and Portfolio Management

The term *stock market* encompasses the set of all markets and exchanges where the trading and issuing of financial assets like stocks, bonds and other securities occurs. The two largest stock exchanges are based in New York city, the Nasdaq Stock Market and the New York Stock Exchange (NYSE). The best performing and most well-known Securities also have the possibility of being traded in groups, such as Exchange Traded Funds (ETFs) [4].

Stocks are securities which represent ownership share in the equity of a company. When a company earns profits, the stock holders can choose to redistribute them amongst themselves or reinvest them in the company. Companies have two ways to pay back profits: by buying back shares or by paying dividends. When dividends are distributed, each stock holder receives a certain amount per share, and the sum of those dividends is deducted from the company's equity. Since stocks are inherently related to their companies they have the risk of losing their value in a situation of bankruptcy, as creditors, like banks, have first claim on all assets still within the company. Stocks present on the market can see their value change with different degrees of volatility, since the company is usually being evaluated based on expected future returns [4].

Bonds are securities that represent a loan to a certain entity, normally a government or recognized company, and have a fixed income rate. Historically, they are a low return-risk investment option. Nevertheless, recently, bond interest rates have risen, as governments struggle to get financing in the markets. Unfortunately, these higher returns bring higher risk, since the ability of governments to sustainably pay their debts has been questioned [4].

A *stock market index* (or simply index or market index) measures the value of a subset of the stock market over time and is computed from those stock values [24]. The most renowned indexes are the Dow Jones Industrial Average (DJIA) and the Standard & Poor's 500 (S&P500). Figure 1.2 depicts the evolution of the S&P500 index during a year.

ETFs are a type of security traded in stock markets, which aim at mimicking the performance of a market index. They are traded like a traditional stock but are more tax efficient since dividends are not

distributed, but rather automatically reinvested. An investor must sell his ETFs on the market to get returns. One way to measure their risk is through the *tracking error* which is the difference between the ETFs' returns and the returns of their market index of reference [25].

In finance, a *portfolio* is a collection of securities, such as stocks, bonds and ETFs, held by an investor. To develop a portfolio, investment objectives must be set, according to each investor's preferences, to balance expected return and risk. This is the field of *portfolio management*, which encompasses all decisions pertaining to the construction of a portfolio, mainly which securities should be included [1, 2]

Figure 2.1: Evolution of S&P500 during the period of one year (from 30.12.2016 to 30.12.2017). Image retrieved from [21].

The importance of Markowitz's MPT in justifying portfolio diversification as a legitimate investment goal, as opposed to focusing solely on maximizing returns, was recognized and Markowitz was awarded a Nobel prize in Economics in 1990 [6, 7]. *Diversification* means that the portfolio should be constructed of several different securities, to spread the risk and minimize the impact of assets that decrease in value. To better comprehend this notion, two main types of risk should be understood: *systematic risk*, that is the inherent risk of a certain market or market segment, like a global economic crisis, cannot be removed through diversification; and *unsystematic risk*, which can be removed using diversification, since it is only present when holding a low variety of assets. Worker strikes in companies or bankruptcy are examples of unsystematic risks. It is now understandable that the risk of a portfolio can never be zero, due to systematic risk. Nevertheless, it is possible to reduce the risk by following guidelines, like selecting different securities and security types or varying the segments (countries and industries) of the investments [2, 26].

Portfolio management can be divided into two forms: *active management*, where the aim is to outperform a specific index making use of market trends and undervalued assets; and *passive management*, whose goal is to track an index and mimic its performance, focusing on creating a well-diversified portfolio as opposed to taking advantage of value irregularities in assets. Although MPT is usually classified as a passive management method, it often beats the indexes by applying good diversification and

management of costs. Nevertheless, the use of a purely active management strategy, via the use of TA indicators, for example, increases the possibility of higher returns, albeit with increased risk [1, 2].

Constructing a portfolio requires the analysis of historical market data to invest in assets predicted to yield the best return in the future. Besides certain intrinsic attributes of the historical data time series, like opening and closing price or traded volume, it is possible to further construct useful investment data by performing fundamental or technical analysis on the market. The fundamental and technical market analysis technics are not mutually exclusive [2, 4, 9].

Fundamental analysis delves on economic indicators, like gross domestic product (GDP), inflation rate or balance of trade, and financial factors, such as income statements, balance sheets and cash flow statements, that influence companies to make a forecast regarding its value [2, 4, 9, 27].

Technical analysis is performed on market activity, like historical prices and volume of transactions, since TA states that current asset prices include all information needed to identify trends in the market. There also is an inherent assumption that the market can be predicted [2, 4, 9, 28].

Investment theories are the basis for decision-making regarding the choice of investments. The *efficient-market hypothesis* (EMH) has three main forms, but we will focus on two – the weak and semi-strong – since the strong form clearly states that no excess returns can be obtained. The weak form, which states that TA, with its historical prices, cannot provide a consistent investment strategy since, from period to period, prices evolve like a random walk. The semi-strong form states that FA, unless done using inside information, will not provide a significant advantage return-wise, since all information it studies is reflected in the stock value. The main consequence, if the EMH strong form is correct, is that a portfolio mimicking an index, either manually by buying its indexed stocks or through the purchase of ETFs, can yield returns comparable to one managed by professional analysts [27, 29, 30].

Paradigms like MPT or EMH, assume that all investors in the market are rational. *Behavioral finance* is another paradigm in financial economics which states that the market is not fully efficient since investors make irrational decisions based on four factors: biased judgments, overconfidence, herd mentality and loss aversion. It further states that not even the presence of investors that provide arbitrage, by making profits buying undervalued securities and then selling them, can provide full efficiency in market values. This hypothesis brings some merit to technical and fundamental analysis techniques regarding the existence of undervalued assets [27, 30, 31].

MPT uses mean return to represent expected return and variance for risk. Since this work aims to diversify the financial performance metrics used to evaluate results, an alternative metric will follow.

Portfolio return can also be evaluated using *Return On Investment* (ROI), which measures the return on investments relative to their cost. Formally, in the case of a portfolio, it is defined by Equation 2.1.

$$ROI = \frac{PV_t - PV_0}{PV_0} \tag{2.1}$$

Where PV_0 is the initial value of the portfolio and PV_t is the current value of the portfolio [2, 9].

2.2. Mathematical Optimization

Mathematical optimization encompasses the study of *optimization problems* which are, in a simple case, formulated by Equation 2.2.

$$Minimize\ f(x), \text{subject to } x \in X, g(x) \leq 0, h(x) = 0 \tag{2.2}$$

Where X, called the search space, is a subset of the Euclidean space R^n and is the domain of functions f, g and h, which map to the real space R. Functions g and h are arbitrary functions relevant to a problem domain. Function f is called the *objective function* and the relations $g(x) \leq 0$ and $h(x) = 0$ are called *constraints*. A point x is a candidate or a *feasible solution* if it belongs to X and satisfies the specified constraints. A feasible solution x^* that satisfies Equation 2.3 is called a *globally optimal solution*.

$$f(x^*) \leq f(x)\ for\ all\ x \in X \tag{2.3}$$

An optimization problem can be defined as *single-objective* (SO) or *multi-objective* (MO), depending on the number of objective functions used. The result of solving a MO optimization problem is a Pareto curve, front or set, containing all the non-dominated solutions with different trade-offs between objective functions [2, 32, 33]. Figure 2.1 depicts a generic Pareto curve.

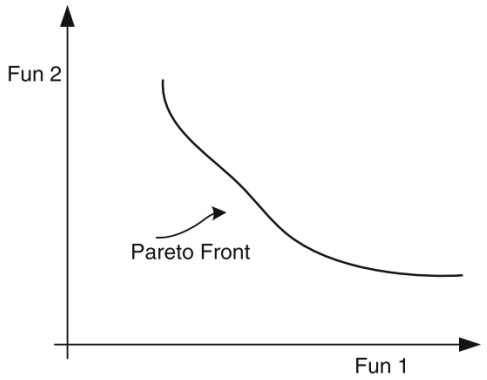

Figure 2.2: Generic Pareto Front. From [9].

2.3. Big Data

The term *Big Data* usually refers to very large datasets and to the aggregate of strategies, techniques and tools needed to process and analyze that data, so that, insights or information can be extracted. Generally, a system that deals with Big Data must be able to perform the following tasks: (i) extraction and ingestion of large quantities of data; (ii) persistent data storage; (iii) data processing and analysis; and (iv) visualization of the results. A brief analysis will follow on the most relevant components to assemble such a system: data storage and processing engine [22, 34, 35].

Traditional *data storage* is not scalable or fast enough to encompass the needs of big data, hence relational databases and non-distributed file system were deprecated in favor of technologies that enable storage of these large datasets with fault tolerance, some of the ACID properties and scalable throughput in read/write operations. These technologies are usually divided into non-relational databases, specially tuned for scalability, and distributed file systems.

Processing frameworks or engines are responsible for making computations over Big Data in the system. These types of systems can usually be divided into three subgroups: batch processing frameworks, where the system handles data in batches; stream processing frameworks, where data is processed in a continuous manner as it streams into the system; and hybrid frameworks, that allow for both kinds of processing. Since the goal of this work is to implement a system that only requires batch processing, the analysis will be restricted to these types of frameworks and hybrids.

Hadoop was the first well-established framework to appear and it started most of the current Big Data trend [22]. It is an Apache project, started in 2008, which started as a cooperation between Yahoo and the University of Michigan. There are two main elements in Hadoop: the Hadoop Distributed File System (HDFS) for data storage and Hadoop MapReduce for processing data. Both components are based on systems and research done by Google: the Google File System (GFS) and the MapReduce programming model, respectively [34].

HDFS provides an abstraction for a file system that, while scalable and distributed in nature, allows a transparent use for the final user. It provides a similar interface as UNIX file systems. HDFS also stores data and metadata separately, as the metadata is stored in a NameNode server, which is dedicated, and the remainder of data is distributed through other servers, DataNodes. This can be seen in Figure 2.3. As HDFS is a robust solution to store up to petabytes of data and is incorporated with Hadoop, it is often used as the main storage site for Big Data [22, 23, 36, 37].

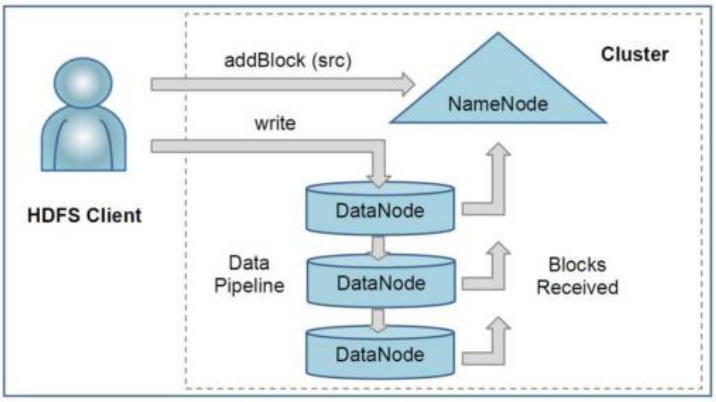

Figure 2.3: A simplified overview of the HDFS architecture. Obtained from [37].

The *MapReduce Programming Model* was created in 2004 to abstract parallel and distributed processing of large datasets. It is responsible for fault tolerance, while hiding its details, as well as the details for load balancing, distributed storage and parallelization. As the name implies, this model is

based on two main functions: the Map function and the Reduce function. The pipeline of a MapReduce program encompasses the following phases [22, 34, 37] and can be seen in Figure 2.4:

1. Read data: input data is read and saved as a set of key-value pairs. Then, it is split in same-size chunks and each chunk is used in an instance of the Map function;
2. Map: each single key-value chunk is received and by an instance of the Map function and each one produces a set of intermediate key-value pairs;
3. Combine: all intermediate key-value pairs with the same key are combined;
4. Partitioning: after they are combined, resulting chunks are distributed to different instances of the Reduce function;
5. Reduce: key-value pairs with the same key are merged and a final result is computed.

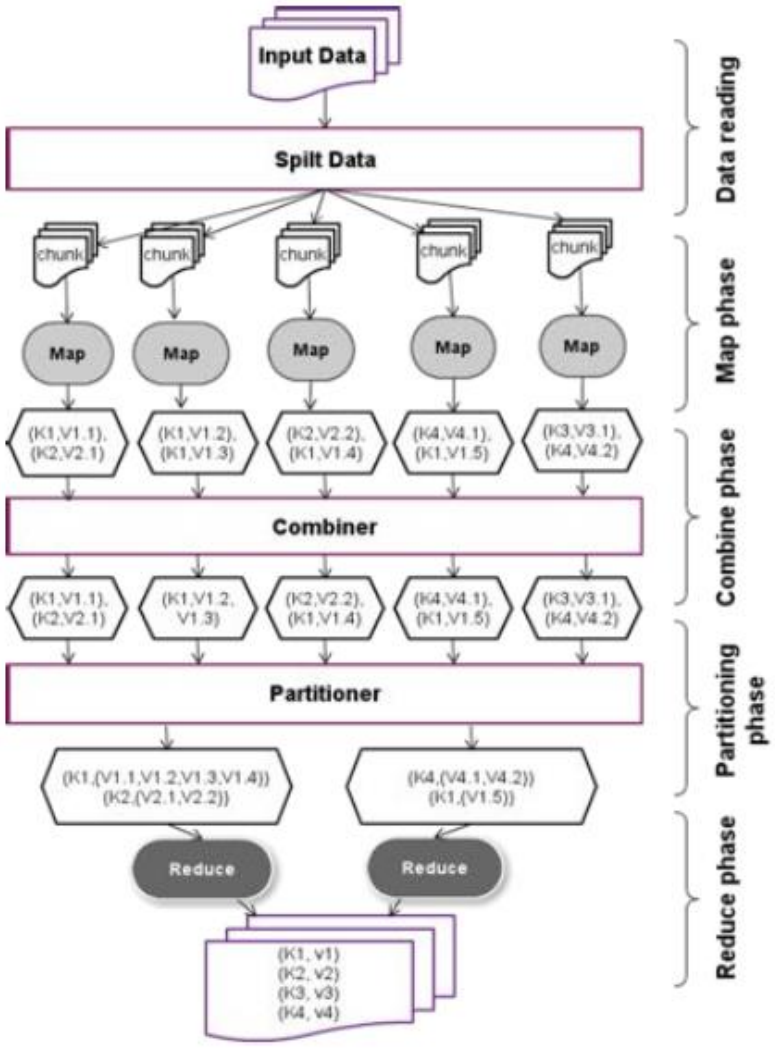

Figure 2.4: MapReduce programming model pipeline. Picture from [34].

In the latest version of *Hadoop MapReduce*, version 2, there are two main modules: the cluster resource management component, called YARN, and the typical MapReduce programming model capabilities. YARN is divided into three services: the Resource Manager, which receives and deploys applications

on the cluster (a MapReduce job, for example); the Job History Server, which logs information about completed applications; and the Node Manager, responsible for launching containers in nodes (a container can be a Map or Reduce function, for example). As for the MapReduce capabilities of the system, they exist on the MapReduce Application Master. There is one Application Master per MapReduce job and its responsible for monitoring and controlling the flow of the MapReduce pipeline. An overview of this YARN architecture can be seen on Figure 2.5 [34, 38].

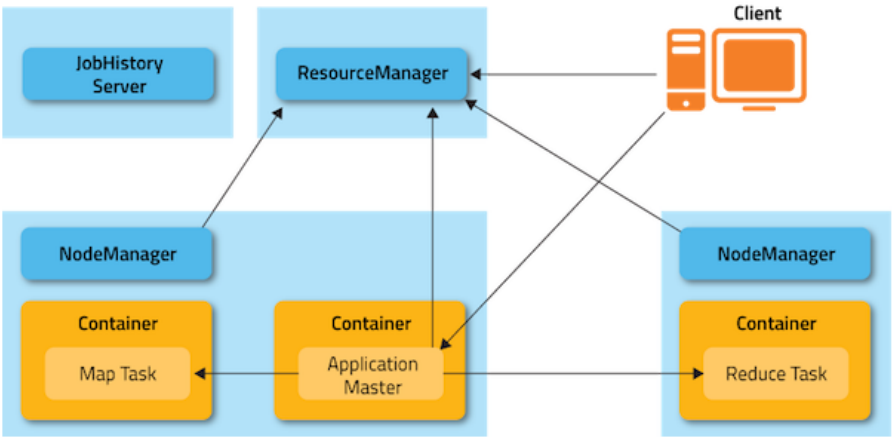

Figure 2.5: An overview of the YARN architecture in MapReduce 2. From [38].

Despite being highly popular for being bundled with Hadoop, YARN is not the only available cluster manager for Big Data frameworks. *Mesos* is a cluster manager which allows dynamic resource sharing and is based on a master-slave architecture and *ZooKeeper* which takes a different approach to YARN and Mesos, as it is based on a cooperative control architecture, since the same service is present on all nodes of a cluster [34].

In fact, the Hadoop framework basic package of MapReduce, HDFS and YARN originated a software stack which allowed for the development of several other applications, such as *Pig*, a scripting language for launching chained MapReduce programs; *HBase*, a distributed, non-relational database based on Google's Bigtable; or *Sqoop*, a data ingesting service for structured data, able to transfer data between HDFS and a Relational Database Management System (RDBMS) or data warehouse. It is also relevant to note that Hadoop was built in Java, atop the Java Virtual Machine (JVM), and all elements in the Hadoop ecosystem do the same, albeit some use Scala instead of Java, like Spark [35, 39, 40]. A complete overview of the current Hadoop ecosystem is detailed in Figure 2.6.

Hadoop was followed by the *Spark* framework, which was developed in 2009 at University of California, Berkeley [23, 35]. Although Spark uses the Hadoop ecosystem, as seen in Figure 2.6, it is intrinsically different as it provides a new way to program for easy processing in a distributed environment through the key concept abstraction of Resilient Distributed Datasets (RDDs), fault tolerant, immutable and distributed collections of objects across a cluster that can be used as a regular object to make parallel manipulations [23, 34, 35].

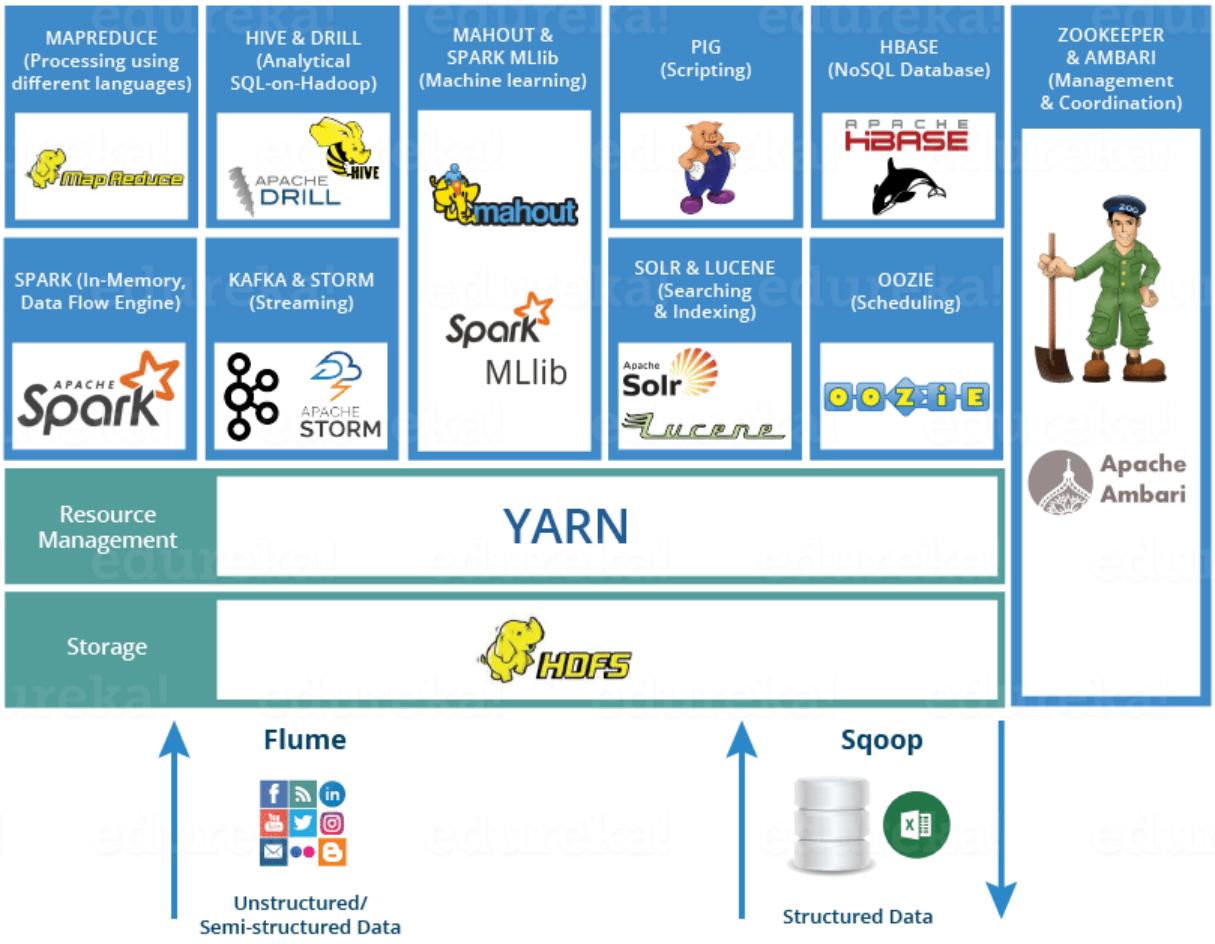

Figure 2.6: An overview of the Hadoop ecosystem. As seen in [40].

The Spark framework allows two basic types of operations on RDDs: transformations and actions. *Transformations* are deterministic lazy operations, such as join, union, filter and map, which create a new RDD from an existing one without making an immediate computation. *Actions* are the operations that trigger the computation of previously defined computation, such as: show (equivalent to print), write or collect. This lazy evaluation of RDDs can happen due to Spark's Direct Acyclic Graph (DAG) engine. When an action is called, Spark looks at the whole transformations graph and creates an optimized execution plan, for example: merging transformations or grouping them according to data partitioning [23, 34]. A Spark execution plan of the well-known WordCount example, which reads several text files and counts the number of times each word appears, can be seen in Figure 2.7.

Architecture-wise, Spark uses a master-slave paradigm, with three components: Driver, Cluster Manager and Workers. The *Driver*, is basically the master node. It maintains a SparkContext object that monitors running applications (in the most recent versions, Spark 2.x, this object is wrapped in the unified SparkSession object). The *Cluster Manager* is responsible for distributing the application

workflow defined by the Driver, as well as allocating resources throughout the cluster. *Workers* represent containers for operations during application execution [23, 34, 35]. A summary of this architecture can be seen in Figure 2.8.

Figure 2.7: Spark's execution plan of the WordCount example. From [34].

Spark supports three programming languages, Java, Scala and Python, and is divided in several main Application Programming Interfaces (APIs): *SparkCore*, the main execution engine, required for all other APIs*; SparkMLlib,* a scalable machine learning library with several pre-implement algorithms and some linear algebra operations; *SparkStreaming*, for processing streaming data – although this is achieved by processing in micro-batches instead of using a pure stream processing engine; *SparkSQL*, it allows the manipulation of high level RDD abstractions called DataFrames and DataSets for analyzing data using the Standard Querying Language (SQL) or its functional programming aliases – filter, union, join,

etc.; and, lastly, *GraphX*, for parallel graph computation, which is useful in algorithms such as PageRank [23, 34, 35].

Flink is the newest Big Data framework that allows batch processing. As Hadoop and Spark, it exists in the Hadoop ecosystem. It is actually very similar to Spark w.r.t. API syntax, but with two key differences: a pure stream processing engine, improving over Spark's micro-batching for low latency applications and accessible via the DataStream abstraction; and an intensive optimization engine which borrows concepts from RDBMS query planners to analyze the code and produce the best possible pipeline according to cluster's properties. It further allows for iterative processing to occur on the same nodes instead of having each iteration be independent from the last and even has its own memory management system on top of the standard Java garbage collect, requiring much less tuning of configurations than Spark. Nevertheless, it is still an immature framework, with very few production deployments, and less available functions and operations on its APIs, when compared to Hadoop or Spark [34, 41]. Flink's software stack is depicted in Figure 2.9.

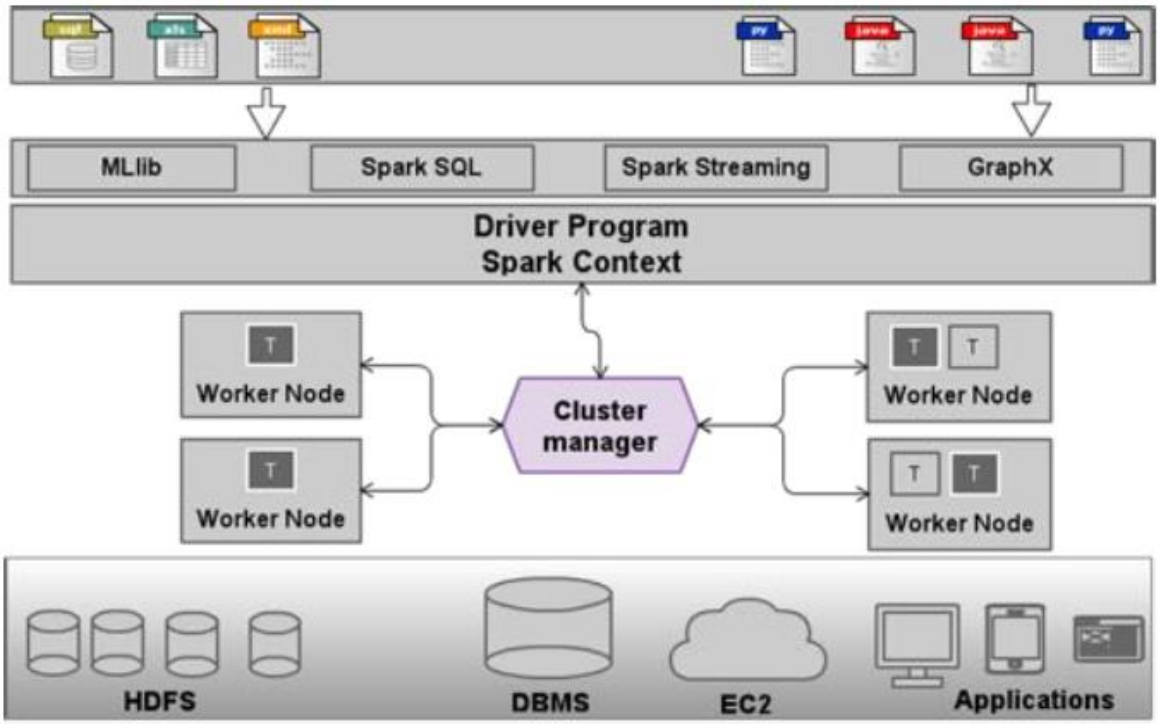

Figure 2.8: Spark's architecture overview. Obtained from [34].

At the core of Flink's processing engine is the common representation of dataflow graphs. A *dataflow graph* is a DAG that contains: stateful operators and data streams. These data streams symbolize data produced by an operator and that can be consumed by an operator [41]. Figure 2.10 shows an example dataflow graph.

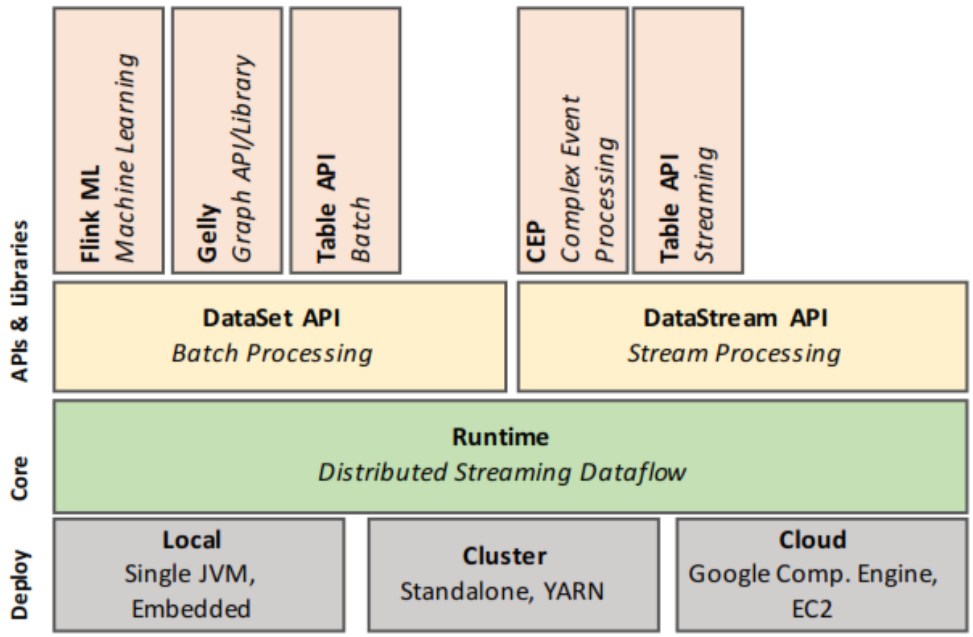

Figure 2.9: Flink's software stack. As seen in [41].

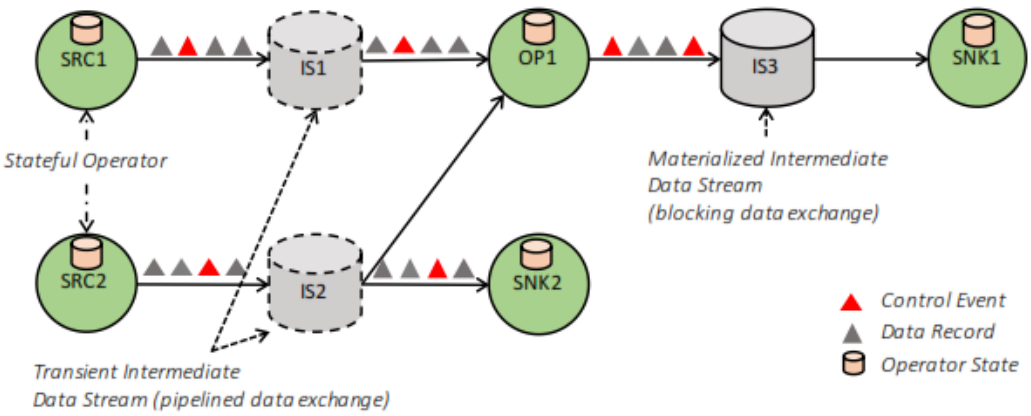

Figure 2.10: An example dataflow graph. From [41].

3. Related Work

In this section, the most relevant related work will be introduced, mostly inside the domain of portfolio management. Since the number of portfolio management systems which use Big Data techniques is insufficient, several works regarding Big Data frameworks will also be presented. In Section 3.1, the classical Markowitz's work and exact methods that can solve it are detailed. The pertinent characteristics in portfolio management systems which are relevant for this work's goals will be introduced are detailed in Section 3.2. In Section 3.3 and 3.4, works that use metaheuristics to solve complex optimization problems aimed at portfolio management are explored, first using SO and then using MO optimization. Section 3.5 describes related works using Big Data frameworks and provides a comparative analysis of these frameworks. Finally, Section 3.6, presents an overview and a brief discussion of the related work.

3.1. Modern Portfolio Theory and Exact Methods

In his work, Markowitz [7] enables the definition of a POP considering risk and expected return by maximizing the portfolio's return for a certain risk level or minimizing the portfolio's risk for a determined level of expected return. Markowitz's POP is the basis for MPT. The classical formulation uses mean return for measuring expected return and variance as a measure of risk, although different metrics are possible [4, 7].

In Markowitz's work [7], the market is modeled by n tradeable securities and the rate of return is r_n and the expected return is $E[r_n]$. The error for each security is labeled σ_n. A portfolio W is composed by n weights $(w_1, \dots, w_n)$. Each w_i represents the proportion of the available budget invested in the security i. There are two typical constraints to the value of w_i, as seen in Equations 3.1 and 3.2.

$$\sum_{i=1}^{n} w_i = 1 \tag{3.1}$$

$$0 \leq w_i \leq 1 \tag{3.2}$$

Equation 3.2 establishes that no *short-selling*, that is the selling of borrowed securities, is permitted and that an asset can occupy, at most, the whole portfolio [7, 42, 43].

Any asset has an expected return denoted R_i. The expected return of the portfolio is defined in Equation 3.3 [7].

$$R_P = \sum_{i=0}^{n} R_i w_i \tag{3.3}$$

Each asset also has a risk σ_i expressed as the variance of its returns over time. The portfolio's risk is formulated as the covariance between its assets, as formalized in Equation 3.4 [7, 42, 43].

$$\sigma_P = \sum_{i=0}^{n} \sum_{j=0}^{n} \sigma_{ij} w_i w_j \tag{3.4}$$

In which $\sigma_{ij}, j \neq i$ is the covariance between i and j and σ_{ii} is the variance of security i. The POP can then be expressed as seen in Equation 3.5 [2, 42, 43].

$$min_w \, \sigma_P, \; subject \; to \; (3.1), (3.2) \; and \; (3.3) \tag{3.5}$$

In this optimization problem, a R_p must be chosen, which represents the intended constant level of expected return [4].

A way to analyze the impact of risk and return is to view the *efficient frontier* of the problem, since it behaves like the introduced Pareto curve. The efficient frontier is composed by the optimal portfolios solution space that exist for every combination of parameters and is hyperbolic in its nature. *Efficient portfolios* are located on the efficient frontier and have the highest risk-return ratios for each combination of parameters [4].

Adding a *risk-free asset* to the POP, usually measured as the interested rates of a big government like the United States of America (USA), creates a unique case where the efficient frontier becomes linear and a *tangency portfolio* exists. The tangency portfolio is a portfolio containing the risk-free asset and stochastically dominates every other portfolio without the risk-free asset [4]. All these concepts can be seen in Figure 3.1.

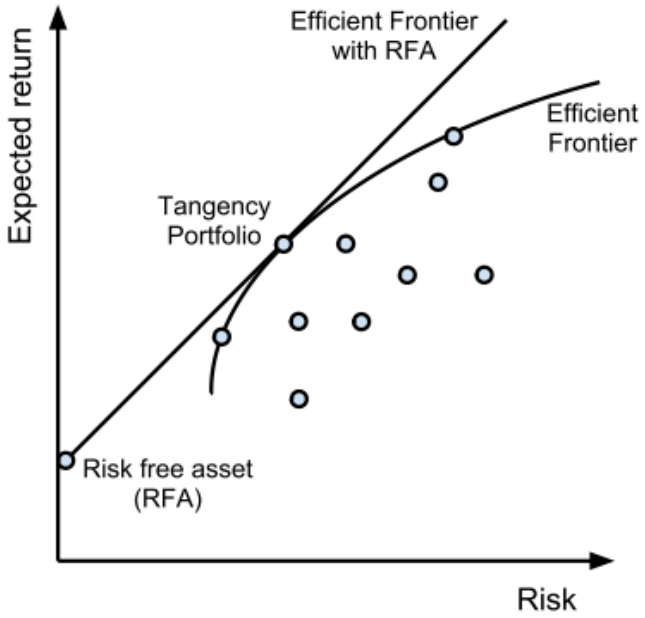

Figure 3.1: Linear and hyperbolic efficient frontiers. Figure from [4].

Even though additional constraints to the introduced Markowitz's formulation create a more computationally complex problem, they are needed to create a more realistic model of the POP. This is true because Markowitz's makes some naïve assumptions, such as: a market without taxes, no

transactions costs and no short selling [44]. Some observed constraints in literature will now be described:

- *Cardinality* – specifies the number of assets allowed in the portfolio, which can be useful to limit transaction costs and to facilitate management [44];
- *Floor* (or *buy-in thresholds*) and *Ceiling* – they limit the lower and upper bound, respectively, of the proportion of each held asset, to reduce unrealistic very low or very high proportions [2, 44];
- *Minimum transaction lots* – if the brokers establish a minimum amount of securities to trade, then the proportion of that asset in the portfolio should be a multiple of the established trading lot [2, 14];
- *Sector capitalization* – reflects the inclination of investors to buy securities in sectors with higher market capitalization to reduce perceived risk [14, 45];
- *Asset classes* – details how many and in what proportion are security types considered for inclusion in the portfolio, according to their risk and capital budget [46].

Besides the aforementioned constraints there is also the consideration of *single-period*, where the portfolio is defined once, or *multi-period*, where the portfolio is managed and readjusted periodically [47].

In the classical MPT, the POP is SO, despite taking into consideration both risk and return. This is achieved by setting a constant level of return and having the objective function minimize risk. Nevertheless, the same POP could be modelled in a MO paradigm and truly minimize risk and maximize return, without setting a constant level to any of them [7, 9].

The classical Markowitz SO POP can be solved using an exact method like *quadratic programming* (QP), however the addition of a simple cardinality constraint, limiting the number of assets in the portfolio, renders the problem NP-Hard [15, 16]. This affects the tractability of the computation, especially with supplementary constraints which are desirable to model the problem more realistically. As stated, this fact makes the use of heuristic methods, particularly metaheuristics, more attractive compared to traditional exact methods [14, 19].

3.2. Relevant Characteristics for Analysis

To analyze works that overlap some of their goals or techniques with those intended for this work, in the following sections, we will be analyzing them on these characteristics:

- The *publishing year*. The following analysis will try to focus on recent works, despite containing some older ones, due to their relevance in the field;
- The *main optimization methods* that were used to obtain solutions, mainly metaheuristics;
- The *supplementary traits* used to complement the main methods;

- The *Big Data techniques* applied, if they exist. The traditional techniques are distributed computing and parallel computing, but Big Data frameworks are increasingly present;
- The *constraints* applied to the optimization problem to better model the problem domain, as explained in Section 3.1;
- The portfolio or market analysis *model* used for management and optimization. The most frequent and well-known are Markowitz's MPT, FA and TA;
- The *number of objective functions*, which defines if the problem is *MO* or *SO*, as described in Section 3.1;
- The *objective functions* of the optimization problem. They might also be used to evaluate the fitness of candidate portfolio solutions. They can be simple risk minimization equation for risk, a lambda tradeoff function that includes both return and risk, maximization of Sharpe's ratio, maximize ROI, a linear combination of technical indicators, among others;
- The *datasets* used, namely which historical stock markets and during which years;
- The *evaluation metrics* used to validate related works. They can be accuracy of solutions w.r.t. exact methods, computer performance measured, for example, in computing time and profitability and risk measured with ROI or Sharpe's Ratio, for example.

3.3. Single-objective Optimization Metaheuristics for Portfolio Management

Solving optimization problems via exact methods usually requires modelling the problem with several very strict assumptions, like in the classical Markowitz' work. Metaheuristics can solve the classical POP extended with several constraints as well as technical or fundamental analysis-based optimization. The most preeminent metaheuristics should be known [17][48]:

- G*enetic algorithms* (GAs) belong to a large class of metaheuristics called *evolutionary algorithms* (EAs) which have their basis in natural selection. They represent problems by chromosomes and then apply rules of natural selection via a fitness function;
- *Tabu Search* (TS) uses local or neighborhood search methods to move from one feasible solution to a more optimal one in its neighborhood;
- *Simulated Annealing* (SA) is a metaheuristic inspired in the metallurgic process of annealing;
- *Particle Swarm Optimization (PSO)* is inspired in social behavior, particularly in the movement of animals, like a flock of birds for example.

The rest of this section contains a description of the most relevant proposals of portfolio management that solve SO optimization problems through metaheuristics.

One of the foremost works that applied metaheuristics to solve a POP was Chang et al. [49]. The authors use GAs, SA and TS and their findings indicated that the *Genetic Algorithm* (GA) implementation

was the best alternative when trying to approximate the efficient frontier of the original Markowitz's model. Nevertheless, when adding a simple cardinality constraint, the difference in performance was found to be negligible between the three methods. The authors depicted the POP as a SO problem using a lambda trade-off function between return and variance. In the same year, Busetti [13] stated that in a POP with nonlinear transaction costs, cardinality, floor and ceiling constraints the GA method outperformed the TS, detailed as a SO optimization problem with a lambda trade-off return variance function.

Another metaheuristic approach to the SO POP was used in 2009, by Cura [12], where PSO was implemented. Using the same dataset as Chang et al. [49] a comparison was made between PSO, GA, TS and SA. The results obtained by Cura [12] were rather inconclusive, showing that no specific metaheuristic clearly outperformed the others with a cardinality constraint. In the same year, Soleimani et al. [14] showed that, for their specific dataset, GA could solve a SO POP with an objective function which minimizes variance within a 3% margin of error to the exact global optimum solution. Soleimani et al. [14] considered cardinality, minimum transaction lots, floor, ceiling and sector capitalization as constraints.

In 2011, Golmakani and Fazel [45] developed a new PSO approach via a combination of binary PSO and improved PSO (CBIPSO). The authors used the dataset of Soleimani et al. [14] to perform a comparison between the GA used in that work and their novel CBIPSO. The results showed that the GA was outperformed by this novel solution. Golmakani and Fazel modeled their work with cardinality, minimum transaction lots, floor, ceiling and sector capitalization constraints. In the same year, Woodside et al. [11] continued the work of Chang et al. [49] by implementing a SO POP with GA, TS and SA and comparing the metaheuristics. GA and TS were the best individual solutions, with TS showing less mean error but larger median error and a much larger computation time when compared to GA. The main conclusion of Woodside et al. [11] is that their metaheuristics implementations were more efficient than in Chang et al. [49] and that pooling the results of different metaheuristics further improves their performance. The considered constraints were cardinality, floor and ceiling. Woodside et al. [11] also used an interesting supplementary trait, by applying an exact method to solve subsets of assets via a quadratic mixed-integer problem (QMIP) formulation to deal with the constraints.

A new solution using EAs was proposed by Lwin and Qu [50] during 2013. It used a combination of Population Based Incremental Learning (PBIL) and Differential Evolution (DE), two variants of EAs. This hybrid algorithm was named PBILDE. The authors perform a comparative benchmark using the dataset of Chang et al. [49], comparing the obtained results with the results reported in Woodside et al. [11], which also use the same dataset. While PBILDE mostly outperformed all the compared metaheuristics, it failed to outperform, on many tests, the GA implemented by Woodside et al. [11]. Other hybrid approaches, like the one created by Cui et al. [44], where PSO and exact methods are mixed, continued to be developed in an effort to achieve a lower error compared to exact solutions and succeeded.

An interesting SO POP work was introduced in 2014. Di Tollo et al. [51] differs from the remaining literature by trying an hybrid portfolio management, both active and passive. It does so by adding index tracking error to the criteria of the POP, using a constant level of return or risk.

3.4. Multi-objective Optimization Metaheuristics for Portfolio Management

While the start of metaheuristic usage to solve the POP might have been dominated by SO formulations, MO formulations were developed parallelly to the SO formulations. MO formulations evaluate solutions based on distinct objectives, like mean return and variance for example, that are not in single objective function, as in SO formulations. There is no major indicator that MO is superior to SO, or vice-versa. Nevertheless, when dealing with EAs, MO EAs have the advantage of generating an approximate efficient frontier in a single run [2].

One of the first attempts of using MO formulations is the work of Streichert et al. 2003 [52] that models the problem with MPT and cardinality, floor and minimum transaction lots constraints. The authors used the two traditional objective functions: minimization of variance and maximization of mean return. Streichert et al. 2003 [52] applied an EA, which was complemented with a local search algorithm before the evaluation of a generation via the fitness function, to improve the selected chromosomes. This local search algorithm updates the decision variables, which are the weights of each asset, so they can be passed onto the offspring. This is known as Lamarckism inheritance. The aim of the local search method is to convert infeasible solutions to feasible ones. The results obtained by Streichert et al. 2003 [52] were promising, since the local search algorithm presented better result than applying a penalty to infeasible chromosomes [2, 52].

Streichert et al. 2004 [53], studied the representation of chromosomes in MO EAs POPs. Their conclusions deemed that a hybrid representation using a bit array B, determining if the variable belongs to the portfolio or not, and an array of real values W, where each element contains the weight of each asset, delivered the best performance. Their representation is depicted in Figure 3.2, alongside the representation that only uses an array W.

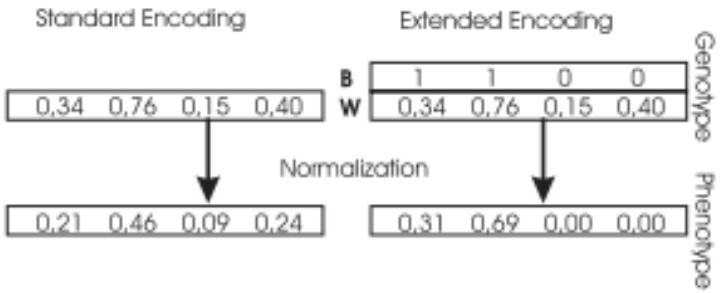

Figure 3.2: Hybrid representation of a chromosome. Retrieved from [53].

Skolpadungket et al. [32] benchmarked the performance of several MO EAs with cardinality, floor and minimum transactions lots constraints. The authors used: (i) a vector evaluated GA (VEGA), which is an improvement upon a simple GA, to deal with multiple objectives, extended with fuzzy logic; (ii) a MO GA algorithm; (iii) the Strength Pareto EA II (SPEA-II); and (iv) the Non-dominated Sorting GA II (NSGA-II).

A fuzzy logic algorithm specifies the probability of inclusion in a portfolio, since the inclusion variable is not a Boolean, but a real value between 0 and 1. Results obtained by Skolpadungket et al. [32] seem to favor the performance of the SPEA-II algorithm.

A tree-based GA (TGA) was developed and then extended by Aranha and Iba [54]. This new algorithm, called Memetic TGA (MTGA) uses the TGA tree structure, depicted in Figure 3.3, to represent assets and a Hill Climbing (HC) algorithm to achieve optimal weights for these assets.

An hybrid version of the NSGA-II algorithm was introduced by Deb et al. [55]. This algorithm developed by Deb et al. [55] was used to solve a POP with cardinality constraints. It enhanced NSGA-II by applying k-mean clustering to filter the amount of solutions near the efficient frontier and a SO GA to search the previously made clusters for the best intra-cluster portfolio. Maguire et al. [56] developed a MO EA to solve a POP by running a set of parallel GAs, each one trying to get the fittest population for a different objective function. These objective functions are depicted in Figure 3.4 as metrics. Despite trying to innovate, the results were not significant since they were not better than a random portfolio.

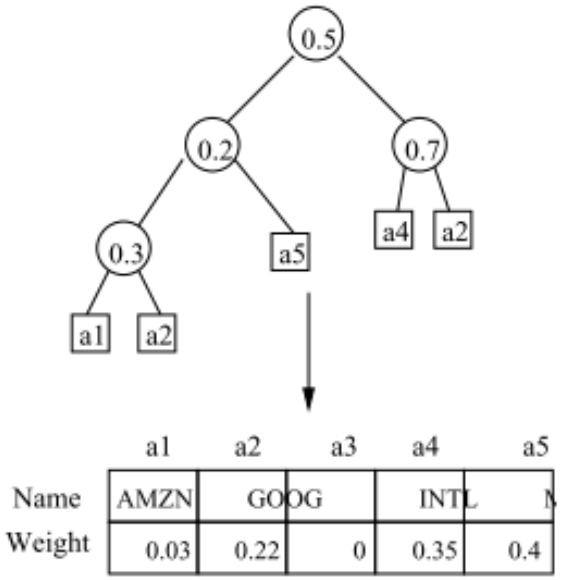

Figure 3.3: A tree-based genome structure. The values in the intermediate nodes store the weight of the assets in the leaf nodes. Obtained from [54].

Lwin et al. [57] built upon the work of Skolpadungket et al. [32] by comparing several MO EAs. They modeled the POP with cardinality, minimum transaction lots, floor and ceiling constraints. The comparison was performed using NSGA-II, SPEA-II, Pareto Archived Evolution Strategy (PAES), Pareto Envelope-based Selection Algorithm (PESA-II) and an innovative algorithm, the Learning-guided MO EA (MODEwAwL). The results obtained by Lwin et al. [57] show that MODEwAwL has the best performance of all the compared algorithms, but it somewhat contradicts the conclusions of Skolpadungket et al. [32], since NSGA-II sometimes surpasses SPEA-II in performance.

The previously mentioned works all revolve around Markowitz's MPT to model the portfolio management, representing it as a POP. As we have seen before in Sections 1 and 3.1, this is a passive

approach towards portfolio management. Works that focus on a different model of the problem and on a more active management perspective will now be introduced.

	Concept	Formula	Range
Metric 1	Return-to-standard deviation ratio	$\frac{R - R_f}{s_x}$	> 0 indicates positive return, < 0 indicates loss
Metric 2	Return-to-variance ratio	$\frac{R - R_f}{s_x^2}$	> 0 indicates positive return, < 0 indicates loss
Metric 3	Maximum fall	$\max_{i<j}(x_i - x_j)$	0% indicates no daily falls, 100% indicates total loss
Metric 4	Maximum duration of any fall	$\max_i(\max_j\{x_{i+1} < x_i, \ldots, x_{i+j} \leq x_i\})$	0% indicates no daily falls, 100% indicates investment never recovers first day value

Figure 3.4: Objective functions considered in [56].

Exclusive active management contributions are rare. Gorgulho et al. [2] detail a work that uses TA to model the portfolio management problem. The work allows short selling and uses the chromosome representation showed in Figure 3.5, which includes TA indicators and the strength of the buy and short signals towards an asset. The optimization problem in Gorgulho et al. [2] is SO and does not aim at optimizing the weights of each asset owned in the portfolio. Rather, it aims at optimizing the weight of each TA indicator so that the portfolio gives the best ROI possible during a back-testing period. The TA perspective introduced by these authors seems to have some credit to it, since the work showed that the system could effectively beat the buy and hold and random strategies in some time periods.

1st Rule	2nd Rule	…	Last rule	Buy limit	Short limit	Close buy position	Close short position
[0, 1]	[0, 1]	…	[0, 1]	[0, 1]	[−1, 0]	[−1, 1]	[−1, 1]

Figure 3.5: Chromosome representation of the TA method in [2].

Another innovative work regarding the usage of an active management approach was done by Silva et al. [9], which emphasis the use of FA, but with a small number of TA indicators. It uses a SO method to maximize ROI and a MO method to maximize ROI and minimize variance. The results obtained were very promising, effectively beating the S&P500 index for the considered back-testing period.

3.5. Big Data Frameworks

There are very few portfolio management systems using Big Data frameworks. The proposal of Jothimani et al. [58] is a good example of a suggested software architecture using these tools, although theoretical. This solution suggests Data Envelopment Analysis (DEA), a non-parametric linear programming tool, in addition to the Hadoop framework to perform sentiment analysis and obtain perceived efficient stocks. It then uses machine learning techniques, k-means clustering and Artificial Neural Networks (ANNs), for stock diversification and ranking of assets. Finally, it suggests a MPT formulation for optimization using metaheuristics, like GAs, PSO and Ant Colony Optimization (ACO).

Since we cannot compare Big Data frameworks in the context of portfolio management domain, it is important to better understand these frameworks to properly choose the most adequate for this work's solution.

Table 3.1: Comparative analysis of Big Data frameworks. Obtained from [34].

	Hadoop	Spark	Storm	Flink	Samza
Data format	Key-value	Key-value, RDD	Key-value	Key-value	Events
Processing mode	Batch	Batch and Stream	Stream	Batch and Stream	Stream
Data sources	HDFS	HDFS, DBMS and Kafka	HDFS, HBase and Kafka	Kafka, Kinesis, message queus, socket streams and files	Kafka
Programming model	Map and Reduce	Transformation and Action	Topology	Transformation	Map and Reduce
Supported programming language	Java	Java, Scala and Python	Java	Java	Java
Cluster manager	YARN	Standalone, YARN and Mesos	YARN or Zookeeper	Zookeeper	YARN
Comments	Stores large data in HDFS	Gives several APIs to develop interactive applications	Suitable for real-time applications	Flink is an extension of MapReduce with graph methods	Based on Hadoop and Kafka
Iterative computation	Yes (by running multiple MapReduce jobs)	Yes	Yes	Yes	YES
Interactive Mode	No	Yes	No	No	No
Machine learning compatibility	Mahout	SparkMLlib	Compatible with SAMOA API	FlinkML	Compatible with SAMOA API
Fault tolerance	Duplication feature	Recovery technique on the RDD objects	Checkpoints	Checkpoints	Data partitioning

In the work of Inoubli et al. [34] a survey on Big Data frameworks is done, which encompasses the five most used frameworks: Hadoop, Spark, Storm, Samza and Flink [34]. As stated in Section 2.3, low latency portfolio management systems are out of this work's scope and, as such, Storm and Samza are not relevant for this analysis since they are purely stream processing engines. The first step in this comparison effort by the authors is done by categorizing the previously mentioned frameworks according to the following characteristics: data format used, processing engine type, data sources, programming model, supported programming languages, allowed cluster managers, machine learning APIs, strategy for fault tolerance and if the framework is prepared for iterative computation. The summary of this analysis can be seen in Table 3.1 [34].

The second step in these authors' analysis is to conduct empiric experiments comparing scalability, resource usage and effect of different configurations of parameters. The author's experimental environment consisted in a cluster of 10 nodes operating Linux Ubuntu 16.04. Each node has 4 cores. 8GB of main memory and 500GB of local storage in a Hard Disk Drive (HDD). Additionally, each node uses Hadoop 2.9.0, Flink 1.3.2, Spark 1.6.0, Samza 0.10.3 and Storm 1.1.1. Every framework is deployed with a YARN cluster manager [34].

The authors created batch and streaming case studies, but since streaming is outside the scope of this work, only batch results will be presented. In the batch case studies, the examples used by Inoubli et al. [34] were: the WordCount example, K-means and PageRank algorithms. For the WordCount example the dataset is composed of random tweets stored in HDFS. More specifically, 10 billion tweets spread in files ranging from 250MB to 100GB. K-means used a dataset comprised of 10000 to 100 million generated learning examples. Finally, regarding PageRank, since it tests graph processing its results will be omitted [34].

To test how the size of the dataset impacts application runtime, Inoubli et al. [34] performed two simulations: one with small datasets, from 250MB to 2GB, and another with big datasets, from 1GB to 100GB. These simulations are shown in Figures 3.6 and 3.7, for small and big datasets respectively [34].

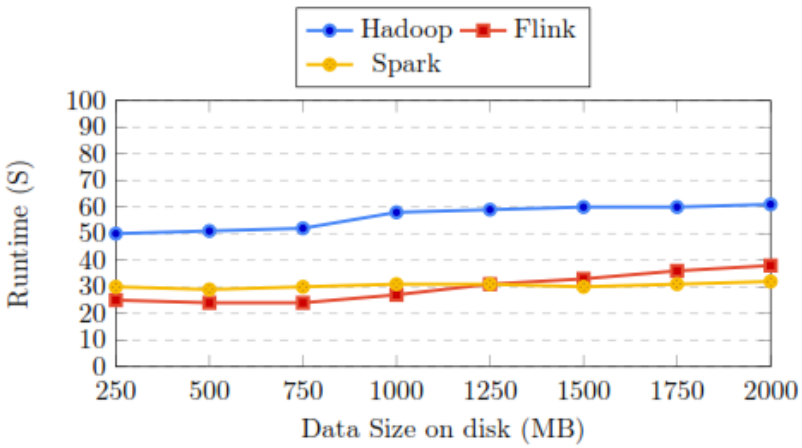

Figure 3.6: Inoubli et al. [34] scalability simulation for small datasets. Depicts the average runtime variation w.r.t. data size in the WordCount example.

The authors remark that in the case of small datasets Spark is the fastest framework, followed by Flink and then Hadoop, except in the case where the dataset is smaller than 1250MB: in this range Flink is slightly faster than Spark. When the dataset size is big, Spark remains the fastest, followed by Hadoop and, finally, Flink. The exceptions are datasets smaller than 2GB, where Flink is faster than Hadoop [34].

The next experiment Inoubli et al. [34] performed was aimed at studying the scalability and processing time of these frameworks. The authors achieved this by varying the number of nodes in the cluster, as can be seen in Figure 3.8 [34].

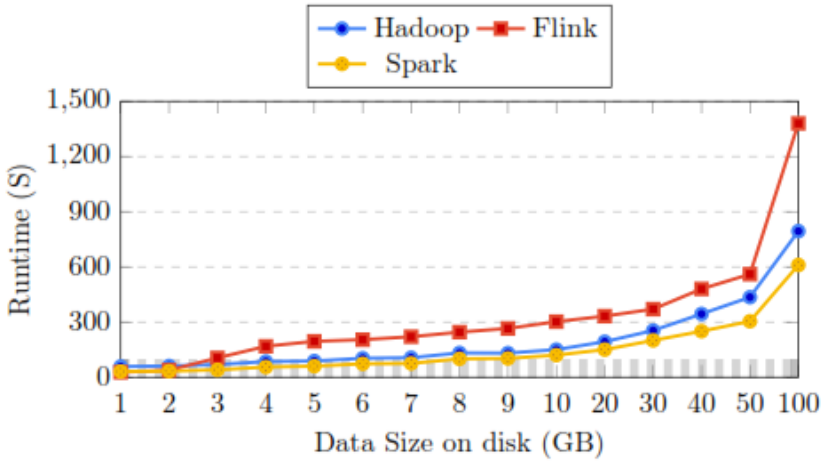

Figure 3.7: Inoubli et al. [34] scalability simulation for big datasets. Depicts the average runtime variation w.r.t. data size in the WordCount example.

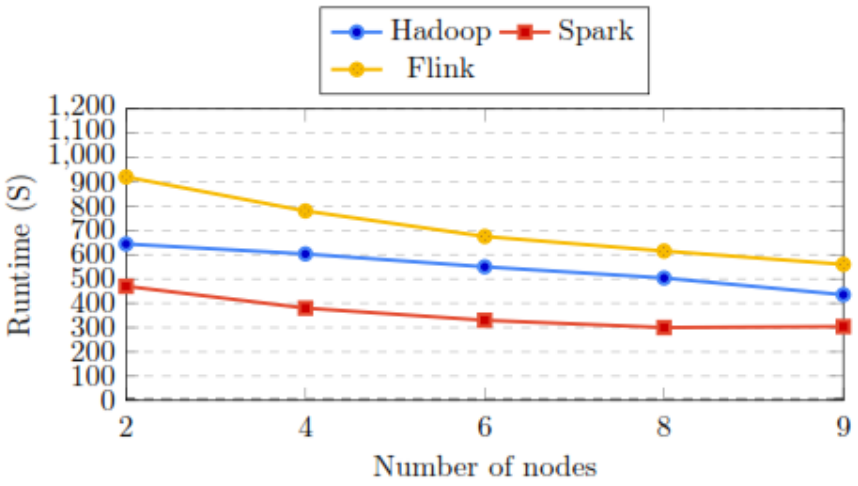

Figure 3.8: Inoubli et al. [34] simulation depicting how the number of nodes in the cluster impact runtime in the WordCount example.

Another experiment relevant to this work's scope in the work of Inoubli et al. [34] analyses how the number of iterations impacts average processing time of the K-Means algorithm with a dataset containing 10 million learning examples.

Despite mostly using the YARN cluster manager in their experiments, Inoubli et al. [34] performed another empiric test which studies how the chosen cluster manager affects runtime. The studied cluster managers were: YARN, Mesos and the standalone managers of the Big Data frameworks. The used datasets were: 50GB of data for the WordCount example and 10 million rows for the K-Means algorithm. As can be seen in Figure 3.10, the authors state that the standalone mode is faster than both YARN and

Mesos, since it uses all resources while running the applications. Whereas YARN and Mesos allocate memory to their schedulers to manage cluster resources.

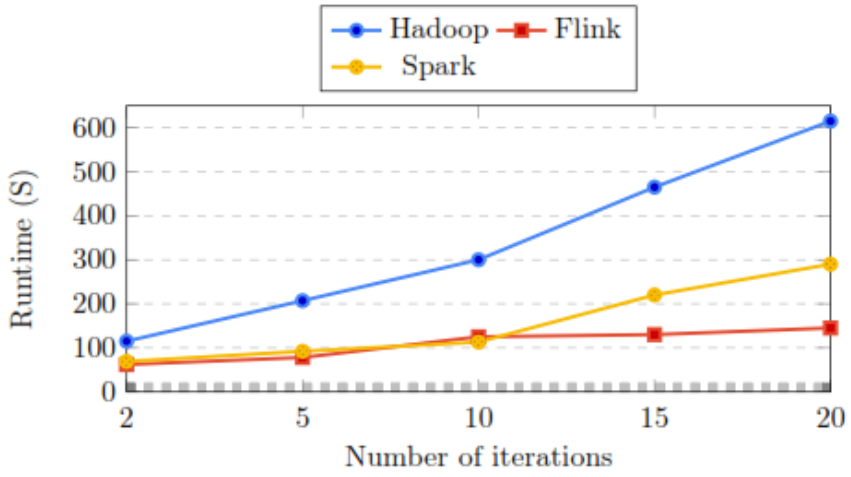

Figure 3.9: Inoubli et al. [34] simulation depicting how the number of iterations in the K-Means algorithm impacts average runtime.

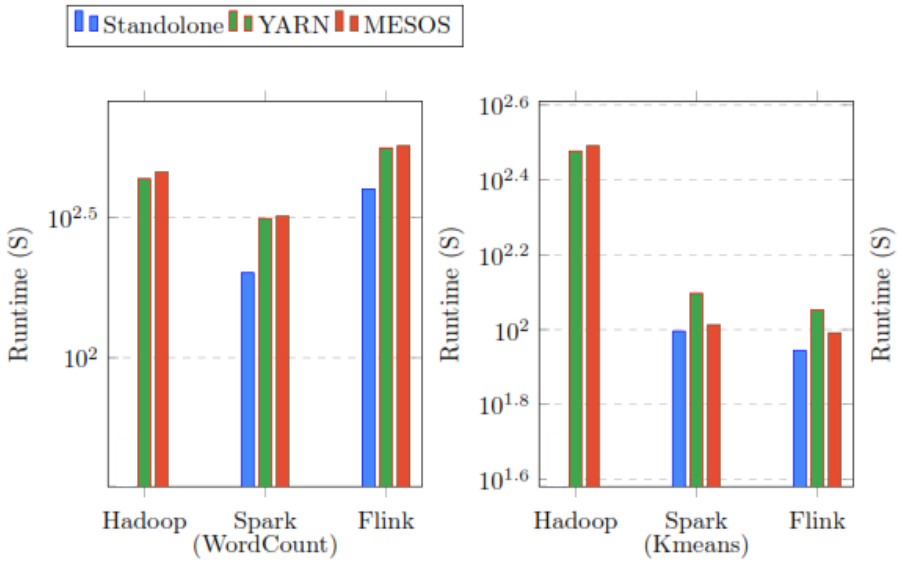

Figure 3.10: Inoubli et al. [34] simulation depicting how the choice of cluster manager impacts average runtime in the K-Means algorithm and WordCount example.

3.6. Overview and Discussion

Table 3.1 summarizes the portfolio management approaches described in Section 3.1 and Sections 3.3 to 3.5 by analyzing them on the characteristics detailed in Section 3.2

Analyzing the literature in Table 3.1, it is noticeable that it is very hard to compare existing solutions via benchmarks, w.r.t. accuracy, computational performance or profitability, due to the heterogeneity of evaluation metrics, datasets and time periods used in back-testing. This makes it hard to analyze which is the best overall *main optimization method* and remains true even if we take into consideration that a substantial number of works, like Cui et al. [44], Woodside et al. [11] or Lwin and Qu [50], try to create a platform for performance comparison between metaheuristic algorithms by using the *dataset* of Chang et al. [49]. Nevertheless, the most frequently used *main optimization method* is the GA metaheuristic, which presents good overall results in every work that implements it and in benchmarks. As such, GA will be the core of the *main optimization method* considered in this work.

A very straightforward observation regarding the state of the art in portfolio management is the notorious lack of approaches using modern Big Data frameworks, like in Jothimani et al. [58], and even more traditional parallel or distributed approaches, such as the work of Maguire et al. [56]. The lack of Big Data techniques for portfolio management is very puzzling, since the ability to feed more historical data to a portfolio management system enables the analysis of an increased number of assets to choose from. The larger number of available assets can increase the likelihood of finding undervalued ones, which provide higher portfolio returns. Although a Big Data system would have to deal with larger quantities of data, it can achieve results in a reasonable amount of time due to the extensive use of parallel and distributed computing techniques, taking advantage of the highly available and affordable cloud computing paradigm, as described by Yee [59].

Taking into consideration the comparative analysis provided by the work of Inoubli et al. [34] regarding batch processing Big Data frameworks, Spark seems to generally outperform both Hadoop and Flink in scalability. Nevertheless, it is interesting to note that Flink outperforms Spark in iterative processing. Since GA is an iterative algorithm, this fact could suggest that Flink would be the best alternative. Despite this, the scalability of Spark allied with its maturity in terms of APIs development and strong presence in production environments make Spark the Big Data framework chosen for this work.

While Spark is still not efficient enough to beat dedicated domain-specific systems using parallelization technologies like OpenMP and MPI, as shown by the work of Reyes-Ortiz et al. [60], the ease of implementation coupled with the existence of the cloud computing paradigm make it a suitable choice. Additionally, there are already a couple of successful implementations of metaheuristics on this framework, such as GA and PSO [61, 62].

A quick glance at the contents of Table 1 denotes that the most frequent *portfolio management model* is MPT, which indicates a strong dominance of passive management approaches. Nevertheless, a few innovative works, like those of Gorgulho et al. [2] and Silva et al. [9] use an active management strategy. Gorgulho et al. [2] propose that active management, via TA and FA, is less common than passive management, not due to worse performance but mainly due to the computer intelligence's community lack of familiarity with the topics. This work's analysis supports this observation, since most of the works studied have a very strong emphasis on obtaining more accurate (w.r.t. exact methods) and faster techniques to solve Markowitz's formulation. There is a distinct lack of focus on obtaining portfolios with high returns and/or lower risk by diversifying the study of portfolio management in the light of more

complex financial theories and analysis methods, like TA and FA [2]. Since there is no clear better management approach, this work will implement the predominant passive management based on Markowitz's MPT, but keeping an emphasis on evaluating the solution w.r.t. financial metrics.

Regarding *constraints* in the passive management approach, the most prevalent, as well as the ones that will be adopted in this work are: cardinality, floor and ceiling. Considering attained results, it is not clear which *number of objective functions* is best in the underlying optimization problem: SO or MO. It is apparent that MO optimization offers a more suitable overview of trade-offs, since it returns the whole Pareto curve. However, SO can achieve the same level of information with multiple runs. Another apparent difference is that in SO optimization the user defines trade-off preferences before running the algorithm, while in MO preferences are expressed after the run. Since this work will already make use of Spark as its Big Data framework, which is scarcely studied in portfolio management literature, the solution will use the more traditional SO problem formulation. As for what *objective functions* to use in the underlying optimization problems, maximization of mean return and minimization of variance, as well as the linear combination of both, seem to be the norm in passive management.

Lastly, regarding *evaluation metrics*, this work will deviate from the norm by adopting a financial performance metric, namely ROI, as opposed to accuracy w.r.t. exact methods. It will also employ the most prevalent metric, which is computation time.

Table 3.2: Overview of several recent and/or relevant approaches to portfolio optimization.

Authors	Year	Main methods	Supplementary traits	Big Data	Constraints	Model	MO or SO?	Objective functions	Data set	Evaluation Metric
Deb et al.	2011	NSGA-II	K-means clustering SOGA for local-search	-	Cardinality	MPT	MO	Min. variance Max. return	1000 random securities.	Accuracy
Cui et al.	2014	PSO	Calculates exact solutions in a sub-efficient frontier	-	Cardinality, Floor, Ceiling	MPT	SO	Min. variance	Weekly prices. Indexes: Hang Seng, DAX100, FTSE100, S&P100, Nikkei225.	Accuracy, Time
Woodside et al.	2011	GA, SA, TS	Subset optimization using QMIP	-	Cardinality, Floor, Ceiling	MPT	SO	Min. variance	Weekly prices. Indexes: Hang Seng, DAX100, FTSE100, S&P100, Nikkei225.	Accuracy, Time
Lwin and Qu	2013	Hybrid: PBIL and DE	Elitism and partially guided mutation	-	Cardinality, Floor, Ceiling	MPT	SO	Linear combination: return and variance	Weekly prices. Indexes: Hang Seng, DAX100, FTSE100, S&P100, Nikkei225.	Accuracy, Time
Lwin et al.	2014	MODEwAwL, PAES, PESA-II, NSGA-II, SPEA-II	-	-	Cardinality, Minimum transaction lots, Floor	MPT	MO	Max. return Min. variance	Weekly prices. Indexes: Hang Seng.	Inverted generational distance (IGD)
Di Tollo et al.	2014	TS, SD (Steepest Descent), FD (First Descent)	QP used to determine weights of assets	-	Cardinality, Floor, Ceiling	MPT and index tracking	SO	Min. risk using QP solver	Weekly prices. Indexes: Hang Seng, DAX100, FTSE100, S&P100, Nik-kei225.	Time, Coverage Measure

(continued)

Table 3.1: (continued)

Authors	Year	Main methods	Supplementary traits	Big Data	Constraints	Model	MO or SO?	Objective functions	Data set	Evaluation Metric
Golmakani and Fazel	2011	CBIPSO	-	-	Minimum transaction lots, Sector capitalization lots, Cardinality, Floor, Ceiling	MPT	SO	Min. variance	Three unknown sets with 9, 30 and 150 stocks	Mean Time, Mean Variance
Jothimani et al.	2014	GA, PSO, ACO (Ant Colony Optimization)	Data Envelopment Analysis (DEA), Sentiment Analysis, Diversification through clustering: K-Means, Louvian, Ranking: ANN	Hadoop	-	-	-	-	-	-
Maguire et al.	2012	GA	Parallel populations for each objective	Parallelization	Floor, Multi-period considerations	MPT	MO	In Figure 3.4	Daily prices. Indexes: S&P500	In Figure 3.4
Pai and Michel	2014	ES (Evolution strategy), DE	-	-	Floor, Ceiling, Asset classes	MPT	MO	Min. Herfindahl index Min. constraint violations	35 unknown assets: 20 equity indices, 10 bonds and 5 currencies	Convergence measure, Variance
Gorgulho et al.	2013	GA	-	-	Short-selling allowed, Multi-period considerations	TA	SO	Max. ROI	Indexes: DJI and S&P500	ROI, Sharpe Ratio, Sortino's ratio
Silva et al.	2016	GA	-	-	Cardinality, Floor, Ceiling, Transaction costs	TA and FA	SO and MO	Max. ROI Min. mensal variance of ROI Max. Sharpe's Ratio	Index: S&P500	ROI, Sharpe Ratio, Pareto Dominance

4. Portfolio Management System

This section describes the achieved solution: a portfolio management system which suggests a portfolio composition for a certain time window. In Section 4.1, there is a brief overview of the system. Section 2 presents the solution's architecture. In Section 4.3 the data flow of the system is described and depicted. Finally, in Sections 4.4 to 4.8 each system component is detailed. Annex B contains some modules full code.

4.1. Overview

In this section the proposed solution will be introduced. The features of this work are the use of a Big Data framework – Spark – in the data Extract, Transform, Load (ETL) process and in the GA used to solve a MPT-based passive portfolio management optimization problem.

Considering the established goals, the proposed architecture will have to be able to accommodate them via satisfying the following requirements:

1. Retrieve and store large amounts of heterogeneous historical market data available online;
2. Prepare extracted data for system use via an ETL process;
3. Solve an optimization problem modeled with MPT using cardinality, floor and ceiling constraints using a SO GA;
4. Evaluate the attained solutions;
5. Be scalable and minimize the amount of performance deterioration with scalability.

An overview of the components composing the system is depicted in Figure 4.1. Different components are responsible for satisfying different requirements: Download Script and HDFS are responsible for 1., Data Module for 2., Optimization Module for 3. and, finally Portfolio Module for 4. Requirement 5 is satisfied by using the Spark framework alongside good programming practices.

4.2. Architecture

To achieve this work's goals, the portfolio management system must be able to construct a valid portfolio (respecting constraints) step-by-step. Each system component, or module, combines a set of actions and represents a step in the construction of the solution portfolio. Every module is unique in its goals and responsibilities, coming together to form the whole portfolio management system.

The complete list of system components, as can be seen in Figure 4.1, is composed by five modules that fall into three different tiers:

- **Hadoop HDFS (File System):** The file system component of the system corresponds to Hadoop HDFS. This component is responsible for persistent data storage in a distributed

fashion. For testing purposes HDFS can be interchangeable with a local file system. Belongs to the Data Access Tier.

- **Download Script:** This system component is charged with periodically extracting historical data from the Yahoo! Finance online database and performing some minor transformations.
- **Data Module:** One of the most fundamental system components. It reads historical data from the file system and performs transformations in it. Additionally, it saves processed data to the file system and is also responsible for loading it to the main memory.
- **Optimization Module:** The core of the portfolio management system. This module is responsible for the SO GA iterative process that generates the solution portfolio.
- **Portfolio Module:** An auxiliary module responsible for keeping portfolio evaluation functions and generating comparison portfolios for testing.
- **User Module:** A very straightforward module, which allows the user to specify input parameters to the portfolio management system and abstracts the system complexity.

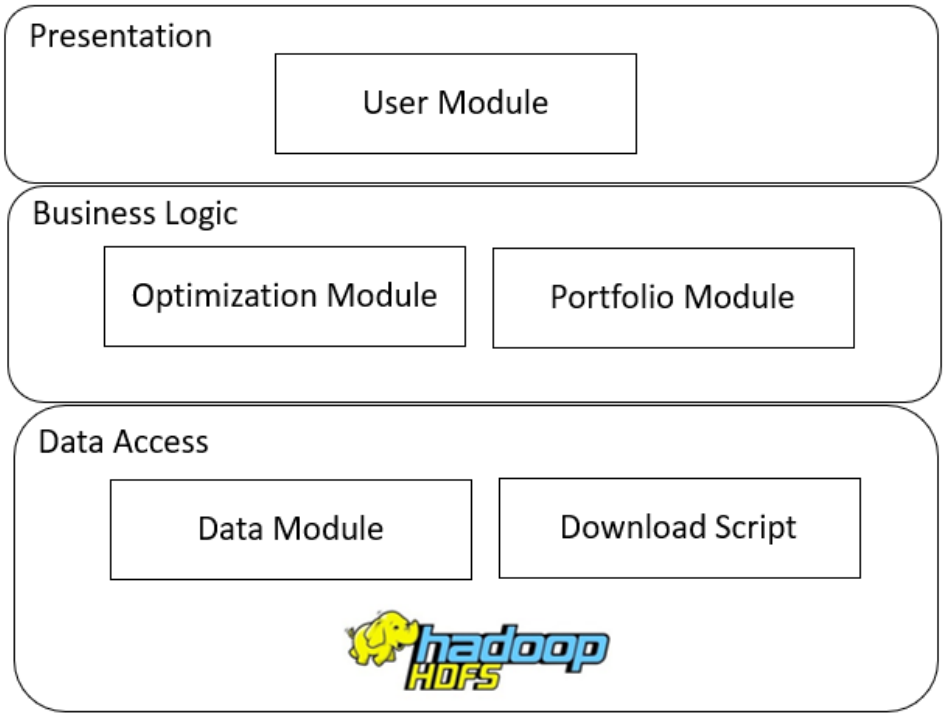

Figure 4.1: Portfolio Management System Tier Architecture.

4.3. Data Flow

The data flow within the system, how information is transferred between components, consists in two different flows: one with two main steps and another with seven main steps.

The first flow of information, which is internal to the system, is in respect to data extraction and loading into the HDFS component:

1. A scheduled job or a system administrator initiate the Download Script; which retrieves historical data from Yahoo! Finance [21] to main memory;
2. Next, the Download Script writes the retrieved data to the HDFS component.

The second data flow is the core one and concerns user interaction:

1. The user inputs preferred parameters regarding the GA: number of generations, population size and mutation rate; and regarding portfolio management: budget, cardinality (number of assets), time window to take into consideration and transaction costs. This information is inputted into the User Module which requests an optimized portfolio from the Portfolio Module;
2. Next, the Portfolio Module communicates with the Optimization Module to initiate the GA iterative process;
3. Before the Optimization Module can initiate its iterative process, it communicates with the Data Module to request prepared historical data;
4. The Data Module receives the request and starts a Spark application. It starts by reading historical prices data from HDFS, performing transformations to it. The Data Module starts by dropping excess information from historical prices and calculating additional financial performance metrics like rate of return and mean rate of return. Then, it persists the transformed data to HDFS and loads it into memory;
5. With the historical data prepared and loaded into memory, the Optimization Module initiates the GA by creating the initial population of chromosomes, or individuals, each a portfolio with random asset composition and weights. Then it starts the iterative process of evaluating the current generation individuals with a fitness function and creating a new generation by using the selection, crossover and mutation operators until convergence is achieved. The resulting portfolio is communicated to the Portfolio Module;
6. Next, the Portfolio Module calculates the final asset allocation taking into consideration the inputted budget and transactions costs and communicates it to the User Module;
7. The User Module outputs the system solution, the recommended portfolio.

The system's data flows can be better understood by consulting Figure 4.2 and noticing how the different system components interact with each other, with the user and with the Yahoo! Finance [21] online database.

4.4. Hadoop HDFS

The Hadoop HDFS component is responsible for persisting the historical data in a distributed manner.

4.4.1. Implementation and Functionality

To implement this component a distribution of Hadoop must be installed and configured so it can communicate with the Portfolio Management System.

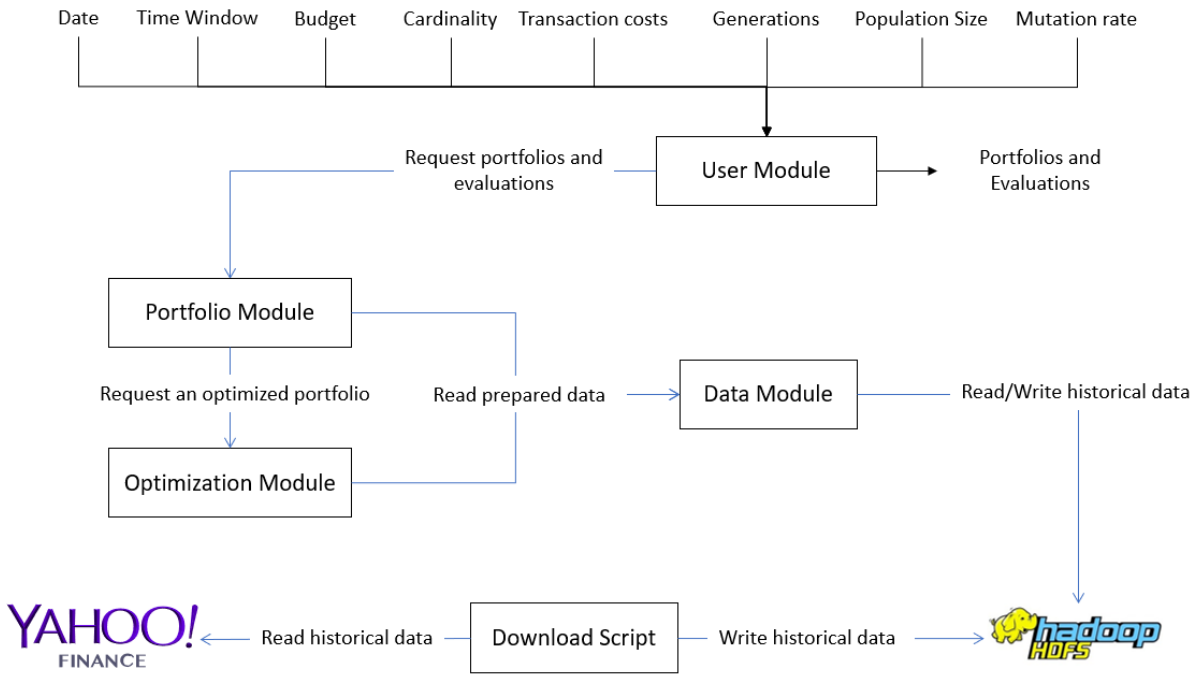

Figure 4.2: Portfolio Management Data Flow and Interaction Overview. The boxes represent the system's components. The black arrows represent user interaction via inputs and outputs. The blue arrows represent interaction between system modules.

Despite being open-source and readily available online, alongside proper documentation, the Hadoop ecosystem is highly complex to implement and configure. For this reason, there are several Hadoop distributions which were developed by companies to facilitate the deployment and creation of a Hadoop production environment [63]. Erraissi et al. [63] performed a comparative study of these distribution where the five most preeminent ones were presented: Cloudera, HortonWorks, MapR, IBM BigInsights and Pivotal HD. From this selection, Cloudera and HortonWorks are the most popular [63]. From these two, Cloudera is the one which presents the best open source distribution regarding utility and functionality. This distribution is called Cloudera Distribution for Hadoop (CDH) [64]. This was the distribution chosen for the Portfolio Management System, namely CDH version 5.13.0. CDH has the bonus of coming with a QuickStart Virtual Machine (VM) with the Linux CentOS [65] Operating System (OS) and CDH pre-installed to facilitate beginning of production. CDH comes with Hadoop 2.6.0, which includes HDFS, and includes a Spark installation. Nevertheless, the Spark version used was 2.3.0 and not the included 1.6.0 version, since after Spark 2.0 the main Spark API changed from RDD to DataFrame and DataSet JVM objects [66]. Both are higher level representations of RDD, but DataFrame is more interesting for this work due to its inbuilt SQL utilities.

The Cloudera Manager, an application which monitors the Hadoop ecosystem services deployed, can be seen in Figure 4.3 where the collection of services provided by CDH is apparent.

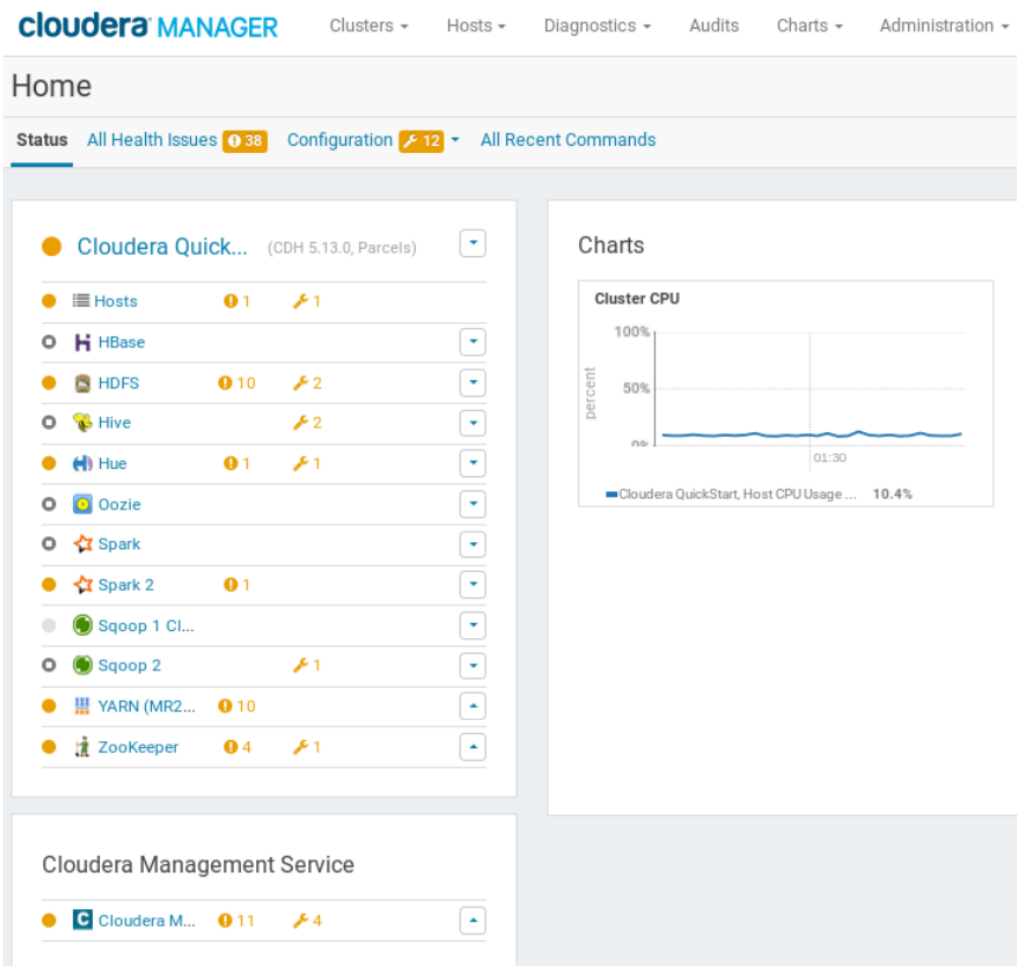

Figure 4.3: Cloudera Manager of the CDH distribution used in this work. Represents the available Hadoop ecosystem.

Regarding further Hadoop HDFS configurations, in line with the discoveries made by Inoubli et al. [34], regarding the impact of HDFS block size on the runtime of an iterative process w.r.t. Spark, the chosen block size was 16MB. We can see this behavior in Figure 4.4.

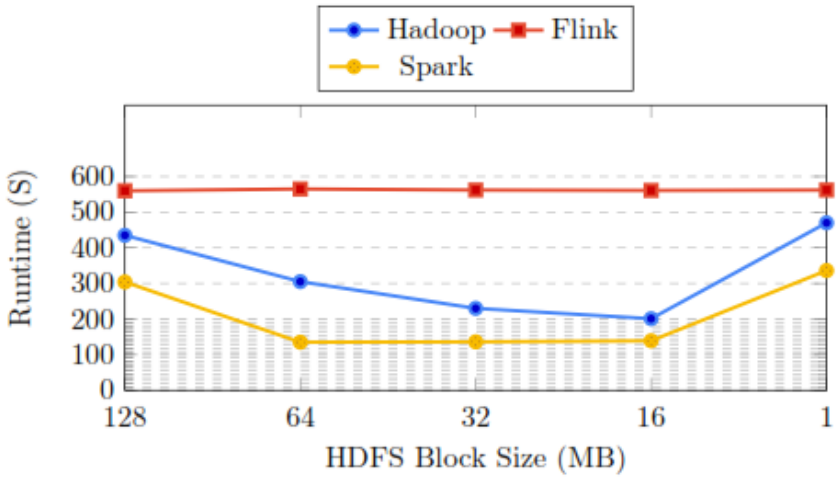

Figure 4.4: Inoubli et al. [34] simulation with a K-Means iterative process which studies the impact of HDFS block size on runtime.

4.5. Download Script

The Download Script component is responsible for three actions: the first is retrieving historical data prices for stocks, ETFs and the exchange rate for currencies the assets are traded in. Yahoo! Finance [21] online database; the second action is to perform some basic transformations on the data which are easier to perform outside the Data Module; and the third action is writing the downloaded data to the HDFS component.

4.5.1. Implementation and Functionality

This component, as the name implies, is implemented as a single script. The script was coded in the Python programming language and can be executed by a system administrator/developer or scheduled in the OS to run at a certain time.

To perform the first action, retrieving the historical data prices from Yahoo Finance! [21], it uses the Yahoo! Finance Fix for Pandas Datareader [67] library, which in turn depends on the popular Pandas [68] library to represent the data as a Pandas Dataframe.

The historical prices data for each financial asset are retrieved from Yahoo Finance! [21] by iterating through a list of wanted assets. For each asset, the script downloads historical price data to a Pandas DataFrame, with the following attributes:

- *Date:* the date of the data entry, in the “dd-mm-yyyy” format;
- *Open:* the asset’s opening price for the entry’s date;
- *High:* the asset’s highest price during the entry’s date;
- *Low:* the asset’s minimum price during entry’s date;
- *Close:* the asset’s close price for this entry’s date;
- *Volume:* the number of shares that were traded during entry’s date;
- *AdjClose:* the asset’s adjusted close price for the entry’s date, which reflects stock dividends and share splits.

Additionally, it downloads currency exchange rate data with the same attributes.

Then, for assets, it transforms the data by adding three additional attributes:

- *Symbol:* symbol that uniquely represents the asset for the data entry;
- *Index:* the market index the asset belongs to;
- *Currency:* the currency the asset is traded in.

As for currency exchange rates, it adds two columns:

- *From:* currency to be exchanged;
- *To:* currency the exchange is made for.

Finally, it writes all Pandas Dataframes to a CSV files in the Hadoop HDFS component.

4.6. Data Module

The Data Module component is responsible for: loading historical data saved in HDFS to memory, performing data preparation transformations and persisting the transformed data for future access.

4.6.1. Implementation and Functionality

The Data Module is implemented as a Scala object named DataModule which uses the singleton design pattern. The DataModule object uses the trait SparkSessionWrapper. This trait is responsible for setting the Hadoop HDFS username and initiating the SparkSession object, which is the entry point to SparkSQL. It also sets additional Spark configurations:

- *spark.driver.cores*: configuration which tells the Spark Driver how many processing cores to use;
- *spark.driver.memory*: configuration which tells the Spark Driver how much main memory it has available;
- *spark.executor.cores*: configuration which tells a Spark Executor how many processing cores to use;
- *spark.executor.memory*: configuration which tells the Spark Executor how much main memory it has available;
- *spark.yarn.jars*: configuration which tells the YARN where the Spark libraries are.

The Data Module Unified Modelling Language (UML) class diagram can be seen in Figure 4.5.

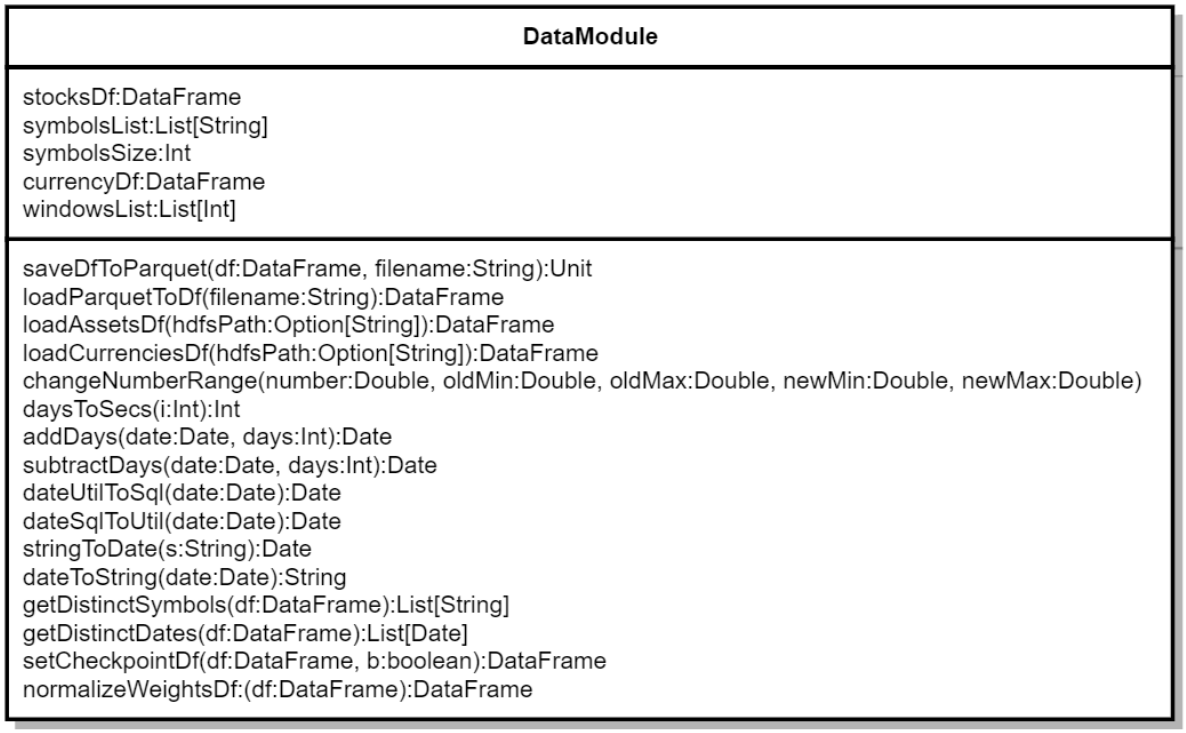

Figure 4.5: Data Module class diagram.

This module is responsible for providing access to the data files stored in Hadoop HDFS and to their main memory distributed representation, the Spark DataFrame object. The DataFrame abstraction was chosen over the low-level RDD representation since its data is in a tabular format, with named columns, imposing structure in the distributed dataset. Additionally, it provides easy access to SparkSQL functions [66].

The Data Module starts by loading the CSV files containing the financial assets and currency exchange rates historical prices, after being processed by the Download Script, into two Spark DataFrame objects: *stocksDf* and *currencyDf*, respectively.

At this point, the Data Module starts performing Spark transformations in the *stocksDf*, since the *currencyDf* data can be used as is. It starts by dropping the columns *Open*, *Close*, *High* and *Low*, since *AdjClose* presents all the price information need by the Portfolio Management System.

The *Rate of Return (ROR)* is the gain or loss of an asset over a specified time window, expressed as percentage of the asset's cost. Formally, in the case of financial assets, it is defined by Equation 4.1:

$$ROR = \frac{A_t - A_0}{A_0} * 100 \quad (4.1)$$

Where A_0 is the initial value of the financial asset and A_t is the current value of the financial asset.

This attribute is used to compute an asset's expected return by calculating its mean over an historical time window. The A_0 is obtained for a preset of six time windows:

- 5 trading days, which represent a week;
- 10 trading days, which represent two weeks;
- 20 trading days, which represent a month;
- 60 trading days, which represent a quarter;
- 120 trading days, which represent half a year;
- 252 trading days, which is the number of average trading days in a year.

Considering that *x* can take the values of the preset time windows, the Data Module takes advantage of SparkSQL Window functions, which allow calculations on a group of rows while still returning a single value for every input row [69], to make several Spark transformations that add the columns:

- *ROR_x*: which represents the ROR for an asset on the entry's date, considering that *x* is the number of trading days in the time window;
- *Mean_x*: which represents the average ROR, or the expected return, for an asset on the entry's date, considering that *x* is the number of trading days in the time window;

The final DataFrame table structure can be seen in Table 4.1. A partial snippet of the DataFrame creation can be seen in Table 4.2.

Table 4.1: Resulting DataFrame structure and its attributes.

Date	*Volume*	*AdjClose*	*Symbol*	*Index*	*Currency*	*ROR_x*	*Mean_x*

Table 4.2: Pseudo-code snippet detailing ROR_x and Mean_x column creation.

Asset DataFrame initialization – partial snippet

```
windowsList = [5, 10, 20, 60, 120, 252]]
w:WindowSpec = Window.partitionBy(“symbol”).orderBy(col(“date”))
for (i ← windowsList)
  //df is the variable holding the current assets DataFrame
  df = df.withColumn(s”adjCloseLag_$i”, lag($”adjClose”, i).over(w))-
    .withColumn( s”ror_$i”, ($”adjClose” – col(s”adjCloseLag_$i”)) / col(s”adjCloseLag_$i”)*100 )
    .drop(s”adjCloseLag_$i”)
end for
for (i ← windowsList)
  w:WindowSpec = Window.partitionBy(“symbol”).orderBy(col(“date”)).rowsBetween(-i, 0)
  df = df.withColumn( s”mean_$i”, mean(col(s”ror_$i”)).over(w) )
end for
```

Finally, it saves the resulting DataFrame in a Parquet file on HDFS. Apache Parquet is a structured column-oriented file format specially designed for the Hadoop ecosystem, inspired by Google’s Dremel [70]. It was designed to be query efficient as it reads only needed cells on the DataFrame by using vertical and horizontal partitioning [71]. Operations on Parquet files are attained via the functions: *savedDfToParquet* and *loadParquetToDf*. This persistence on the file system is needed for two reasons: to speed up further runs of the application by skipping the transformation process and to clear the DataFrame’s underlying RDD lineage.

The *Lineage graph* is used by Spark to know how all RDDs are dependent on other RDDs and what transformations where applied. Essentially, each RDD points to one or more parents and the lineage graph keeps metadata about their relationship. This lineage graph is kept for fault-tolerance reasons but takes up a lot of memory, being one of the most preeminent causes for Out of Memory errors in Spark’s development processes. As such, it is prudent to clear the lineage graph periodically by persisting the DataFrame in a performant file format.

In this work, additional iterative lineage cutting can be attained by calling the *setCheckpointDf* function, which persists the inputted DataFrame to secondary memory – local executor memory – and reloads it to main memory without lineage. In an iterative algorithm, such as the implemented GA described in the next Section, not cutting lineage results in runtime slowdowns per iteration and might even lead to memory errors, as memory is filled with lineage metadata.

Finally, the Data Module provides auxiliary data transformation functions to the Optimization and Portfolio Modules.

4.7. Optimization Module

The Optimization Module is the core of the Portfolio Management System. It is responsible for solving a MPT-based optimization problem with the following constraints: cardinality, floor and ceiling. To solve this optimization a SO GA is used, for the reasons discussed in Section 3.6. A GA is composed by several components, which will be detailed in this Section alongside its implementation.

4.7.1. Chromosome Representation

The chosen chromosome representation for the problem dictates that an individual representing a portfolio consists in an array of n assets and their respective W weights, as discussed in Section 3.1. The chromosome representation can be seen in Table 4.3.

Table 4.3: Portfolio Management System chromosome representation for a portfolio.

1st Asset	2nd Asset	3rd Asset	4th Asset	…	$\boldsymbol{n}$ th Asset
[0, 1]	[0, 1]	[0, 1]	[0, 1]	…	[0, 1]

4.7.2. Selection

The selection operator is responsible for defining how the GA will choose the individuals who will be the parents of the next generation. There are several selection operators available for a GA but truncation selection is the fastest. The tradeoff is that it is the selection operator that disallows the most amount of information variation, which could make the solution a local maximum instead of a global maximum [72]. *Truncation Selection* sorts the population is by fitness and then drops the lowest ones according to the truncation threshold. In this system the truncation threshold is set to 0.5, which considers the top half of the population as parents.

After obtaining the set of parents to choose from, pairs of parents must be chosen to generate new offspring via crossover. The selection of parent pairs is done via *Roulette Wheel Selection*, or *Fitness Proportionate Selection*. This selection method maps the chromosomes fitness values to a probability, generates a random probability – the roulette's fixed point – and then iterates over the chromosomes map calculating the cumulative probability. When the cumulative probability is bigger or equal to the random probability, the iteration stops and that chromosome is chosen [72]. A design overview of the *Roulette Wheel Selection* is depicted in Figure 4.6.

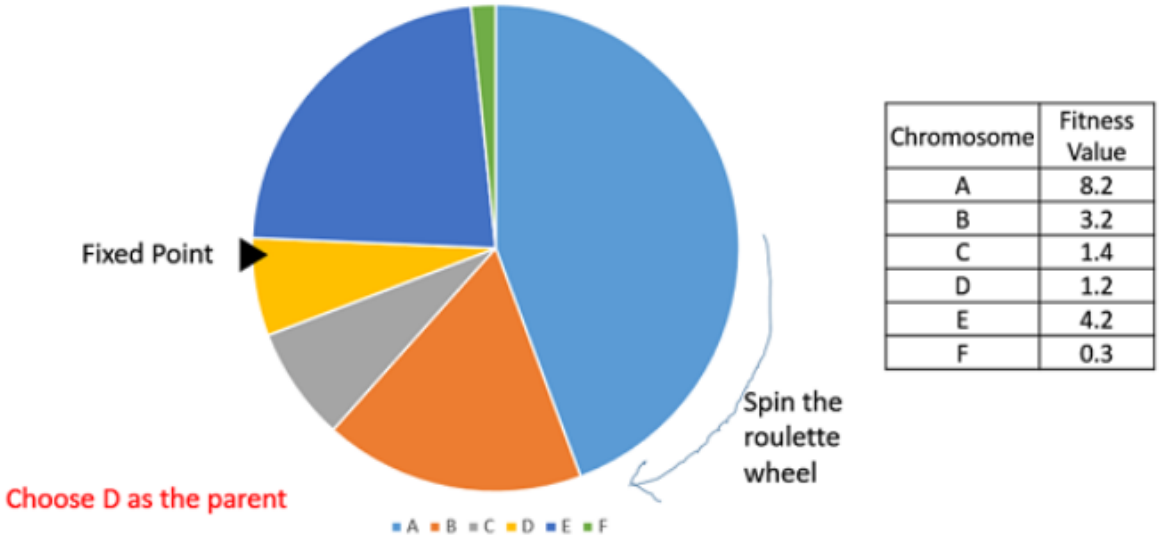

Figure 4.6: Representation of a fitness proportionate selection method. As seen in [73].

4.7.3. Crossover

The crossover, or recombination, operator is similar to how biological reproduction works. With the parents already selected by selection operator new individuals are generated by combining the chromosomes of two parents.

There are several types of crossover operators. In this work two were considered due to their straightforwardness: *One-Point Crossover*, where a random crossover point is chosen and the tails of the two parents are switched; seem in Figure 4.7 and *Multi-Point Crossover*, which is a generalization of the one-point crossover where several alternating segments of parents are swapped to generate a new individual, seen in Figure 4.8.

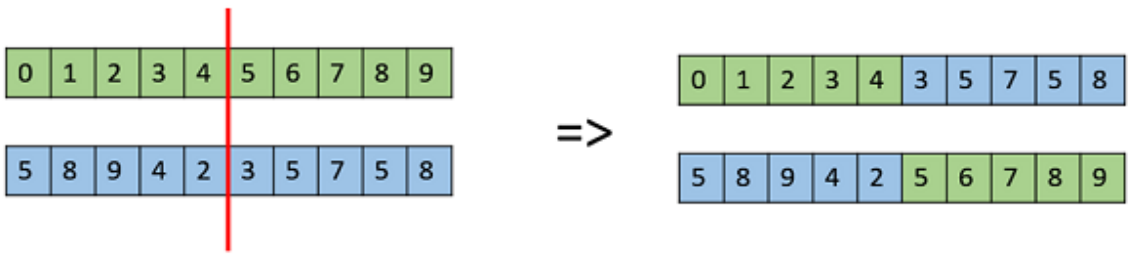

Figure 4.7: Representation of the one-point crossover operator. As seen in [74].

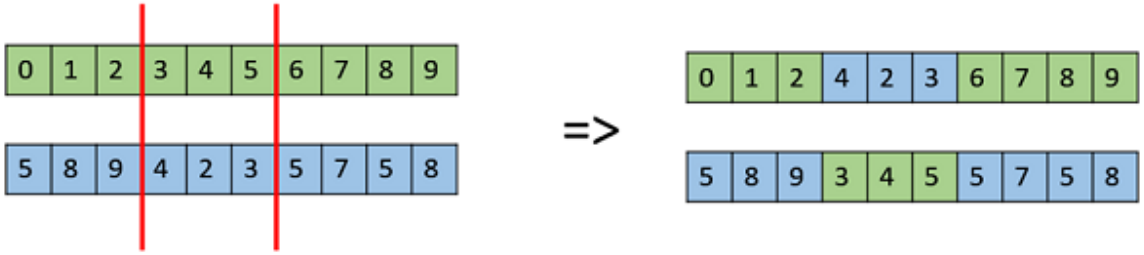

Figure 4.8: Representation of the multi-point crossover operator. As seen in [74].

After analyzing the work of Jorge Magalhães-Mendes [75], it was apparent that the one-point crossover operator tends to display the best results, as corroborated by Gorgulho et al in their GA work. [2]. For this reason, the chosen operator was the one-point crossover.

4.7.4. Mutation

The mutation operator is responsible for generating a small random tweak in the chromosome genotype to get a new solution. It is applied with probability equal to the defined mutation rate, which should be low, or the GA will simply be a random search. As with the selection and crossover operators there are several types of mutation operators. In this work the chosen was the simple *Random Resetting Mutation* which randomly chooses a gene and changes its value to a random permissible value [76]. The solution also implements *Elitism*, which saves the current generation best individual for the next generation, unmutated [2].

4.7.5. Initial Generation

The initial population, or generation, is generated randomly. To avoid infeasible solutions w.r.t. Equation 3.1 the sum of all asset weights must be equal to one. This is achieved by normalizing the weights as follows in Equation 4.2 [2]:

$$w_i = \frac{\mathrm{w}_i}{\sum_{i=0}^{N} \mathrm{w}_i} \tag{4.2}$$

4.7.6. Fitness Functions

Fitness, or evaluation, functions are used to quantify the quality of a chromosome. They are essential in the selection operator, since only the fittest ones are chosen for crossover. There are two fitness functions:

- *Evaluate Portfolio Return:* creates a fitness score equal to the portfolio's expected return R_P, calculated using Equation 4.3;

$$Fitness(p) = R_P \tag{4.3}$$

- *Evaluate Portfolio Linear Combination:* the fitness score is a linear combination of the portfolio's expected return R_P and the portfolio's variance σ_P as seen in Equation 4.4. The linear

combination's default values for coefficients (a_1 and a_2) for return and variance, respectively, are 0.5.

$$Fitness(p) = a_1 R_P + a_2 \sigma_P \quad (4.4)$$

4.7.7. Constraints Handling

To satisfy the cardinality, floor and ceiling constraints the simple *death penalty* methodology for dealing with constraints was implemented. This consists in simply discarding any infeasible individual generated during random generation of individuals [2].

Nevertheless, another strategy is applied for dealing with infeasible offspring generated by crossover and mutation. If, during crossover, the offspring generated becomes infeasible due to violating the cardinality constraint (twenty assets by default) then the top assets sorted by weight are kept and the remaining assets weights are reset to zero. If the infeasibility of an offspring occurs after mutation due to violating the cardinality constraint, then a similar approach to the one described for crossover is applied. If the offspring infeasibility is due to violating the floor and ceiling constraints, by default (0 and 0.30 respectively), then a new gene weight is generated randomly until the result is a feasible gene.

4.7.8. Implementation and Functionality

The Optimization Module is implemented as a Scala object named OptimizationModule which uses the singleton design pattern. The OptimizationModule object uses the trait SparkSessionWrapper to have access to SparkSQL via the SparkSession object. It is the main system component, as it implements the SO GA algorithm to attain an optimized portfolio. A class diagram can be seen in Figure 4.9.

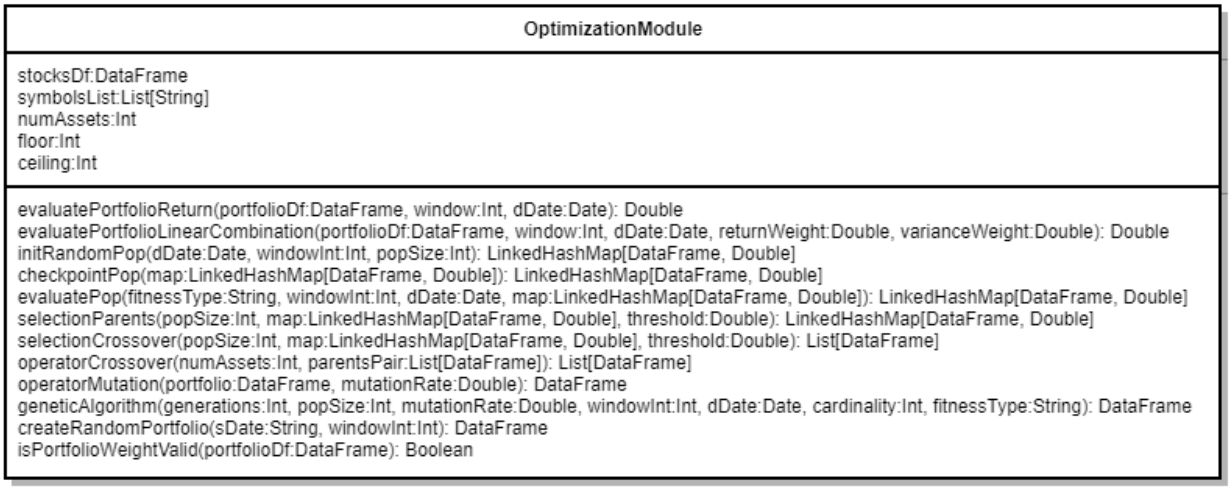

Figure 4.9: Optimization Module class diagram.

The implemented chromosome representation is a DataFrame object, joining the structure seen in Table 4.1 and the chromosome representation in Table 4.3. The final DataFrame for each individual is depicted in Table 4.4.

Table 4.4: DataFrame representation for an individual in the population.

Date	*Volume*	*AdjClose*	*Symbol*	*Index*	*Currency*	*ROR_x*	*Mean_x*	*Weight*

The use of every attribute instead of only the columns *Symbol* and *Weight*, corresponding to the simple chromosome representation in Table 4.3, might seem to impact memory management negatively. Nevertheless, the impact is negligible since the complete dataset of assets is already cached in a DataFrame in memory by the Data Module.

This DataFrame representation for an individual is simply a subset of the overall assets DataFrame, seen in Table 4.1, filtered by *Date* and with the incorporation of a *Weight* column. Additionally, it is possible to use SparkSQL transformations to access only the *Symbol* and *Weight* columns to implement the GA operators. This representation is advantageous since it allows for easy union with the full dataset, as they partially share a schema, and quick access to relevant attributes.

A pseudo-code algorithm implementation is described in Table 4.5.

The main iterative loop process, for each generation, can be described as follows:

1. The genetic algorithm starts by concurrently initiating a random initial population of DataFrame individuals via *initRandomPop*. To originate the individuals, *createRandomPortofolio* is called. Its code snipped can be seen in Table 4.6. Each random individual is generated by a set of Spark transformations and actions: orderBy(random) operation, followed by a limit operation choose the portfolio assets; and an addColumn operation with random weights finalizes the candidate creation. Then the candidate is analyzed to see if any constraint is violated. If there any violations, the death penalty method is applied, and a new candidate is generated;
2. The generated portfolios are persisted to secondary memory to cut RDD lineage via *checkpointPop* every odd generation;
3. Fitness is calculated as detailed in Section 4.7.6 using the evaluation functions provided by the Portfolio Module, that calculate portfolio return and variance. The results are saved in a LinkedHashMap[DataFrame,Double] collection containing individuals and their fitness values, which is sorted by fitness;
4. The next step is truncation selection to select the set of parents, as explained in Section 4.7.2. In this method the list of chromosomes is sorted by fitness. The top half (by default) of the sorted map are the selected parents for the next generation;
5. Next, to guarantee elitism, the best individual is kept for the next generation;
6. The following step is applying roulette wheel selection on the parent set, as detailed in Section 4.7.2 via the function *crossoverSelection*, where the more fit parents have a higher change to be selected;
7. For each pair of selected parents, one-point crossover is applied, as detailed in Section 4.7.3 via *operatorCrossover* as detailed in Table 4.7.

Table 4.5: Optimization Module SO GA pseudo-code algorithm description.

SO GA Algorithm

Input: generation *Int*, popSize *Int*, mutationRate *Double*, windowInt *Int*, dDate *Date*, cardinality *Int*, fitnessType *String*.
Output: optimized portfolio *DataFrame*.

```
for (g ← 0 until generations)
  if g = 0
    //generation is a LinkedHashMap[individual,fitness]
    generation = initRandomPop(dDate, windowInt, popSize)
  end if
  if (g+1) % 2 = 0
    generation = checkpointPop(generation)
  end if
  generation = evaluatePop(fitnessType, windowInt, dDate, generation)
  threshold = 0.5
  selectionIndex = popSize – (popSize*threshold).toInt
  parents = selectionParents(popSize, generation, threshold)
  mostFitIndividual = generation.maxBy(fitness)
  newGeneration += mostFitIndividual
  crossoverIterations = ((selection-1)/2).toInt
  for (i ← crossoverIterations)
    parentsPair = selectionCrossover(popSize, parents, threshold)
    childrenPair = operatorCrossover(cardinality, parentsPair)
    crossoveredChildren += childrenPair
  end for
  for (p ← crossoveredChildren)
    mutatedIndividual = operatorMutation(p, mutationRate)
    newGeneration += mutatedIndividual
  end for
  diff = popSize – newGeneration.size
  for (x ← 0 until diff)
  port = createRandomPortfolio(dDate.toString, windowInt)
  newGeneration += port
  end for
  generation = newGeneration
  newGeneration = new LinkedHashMap.empty
end for
generation = evaluatePop(fitnessType, windowInt, dDate, generation)
return generation.maxBy(fitness)
```

Table 4.6: createRandomPortfolio pseudo-code. Initiates a random individual which does not violate cardinality, floor and ceiling constraints.

Create Random Portfolio (Chromosome)

Input: sDate *String*, windowInt *Int*.
Output: random portfolio *DataFrame*.

```
date = DataModule.stringToDate(sDate)
string = s"mean_${windowInt}"
r = Random
b = false
c = true
l = ListBuffer.fill[Double](numAssets)(0.0)
while (b == false)
  c = true
  for (i ← 0 until numAssets)
    xy = r.nextInt(1000).toDouble
    if(xy == 0.0)
      xy = 1.0
    end if
    k = xy / 1000.0
  end for
  sum = l.sum
  for (i ← 0 until numAssets)
    l(i) = l(i) / sum
    if (l(i) < floor || l(i) > ceiling)
      c = false
    end if
  end for
  if (l.sum != 1.0)
    c = false
  end if
  b = c
end while
sampleDf = stocks.filter($"date" === date)
  .drop("weight").orderBy(rand).limit(numAssets)
lSymbols = DataModule.getDistinctSymbols(sampleDf)
lTuples = l.zip(lSymbols)
lDf = lTuples.toDF("weight", "symbol1")
sampleDf = sampleDf.join(lDf, sampleDf.col("symbol")
  .equalTo(lDf.col("symbol1"))).drop("symbol1")
return sampleDf
```

8. Next, random resetting mutation is applied to newly generated crossover offspring, as detailed in Section 4.7.4 and shown in Table 4.8;
9. Lastly, the remaining new generation vacant spots are filled with random chromosomes.

This iterative process terminates, or achieves convergence, when executed for the inputted number of generations.

It is important to note that the fitness functions are dependent on evaluation functions implemented by the Portfolio Module, which calculate a portfolio's return and risk as variation, as will be detailed in the next Section.

Table 4.7: *operatorCrossover* pseudo-code. Switches parents' chromosomes heads and tails to generate to new offspring. Finally, it re-normalizes the offspring weights

Operator Crossover

Input: numAssets *Int*, parentsPair *List[DataFrame]*
Output: childrenPair *List[DataFrame]*.

```
pA = parentsPair(0)
pB = parentsPair(1)
r = Random
duplicate = true
while (duplicate == true)
  rand = r.nextInt(numAssets+1)
  if (rand == 0)
    rand == 1
pAHead = pA.sort(desc("symbol")).limit(rand)
pATail = pA.sort(asc("symbol")).limit(numAssets-rand)
pBHead = pB.sort(desc("symbol")).limit(rand)
pBTail = pB.sort(asc("symbol")).limit(numAssets-rand)
p1 = pAHead.union(pBTail)
p2 = pBHead.union(pATail)
dup1 = p1.select("symbol").distinct.count
dup2 = p2.select("symbol").distinct.count
if (dup1 != numAssets || dup2 != numAssets)
  duplicate = true
else
  duplicate = false
end if
p1 = DataModule.normalizeWeightsDf(p1)
p2 = DataModule.normalizeWeightsDf(p2)
childrenPair += p1
childrenPair += p2
return childrenPair.toList
```

Table 4.8: *operatorMutator* pseudo-code. Randomly selects a symbol and assigns it a new random weight, until a valid portfolio mutation is found. Finally, it re-normalizes the mutates offspring weights.

Operator Mutation

Input: portfolio *DataFrame*, mutationRate *Double*, sDate *String*, windowInt *Int*

Output: mutated portfolio *DataFrame*.

```
t0 = System.nanoTime
r = Random
p = portfolio
m = r.nextInt(100).toDouble / 100.0
if (m < mutationRate)
  g = r.nextInt(numAssets)
  pList = DataModule.getDistinctSymbols(p)
  symb = pList(g)
  allowed = false
  w = p.filter($"symb" === symb).select("weight").map(r => r(0).asInstanceOf[Double]).collect(0)
  limit = 20
  curentLimit = 0
  while (allowed == false)
    t1 = System.nanoTime
    delta = t1 - t0
    mutationRuntime = delta.toSeconds.toDouble
    if (mutationRuntime >= 10)
      p2 = createRandomPortfolio(sDate, windowInt)
      allowed = isPortfolioWeightValid(p2)
    end if
    currentLimit = currentLimit + 1
    if (currentLimit == limit)
      g = r.nextInt(numAssets)
      symb = pList(g)
      w = p.filter($"symb" === symb).select("weight").map(r => r(0).asInstanceOf[Double]).collect(0)
      currentLimit = 0
    end if
    xy = r.nextInt((ceiling*100).toInt).toDouble
    if (xy == 0.0) xy = 1.0  end if
    newW = xy / 1000.0
    p1 = p.withColumn("weight", when(col("weight").equalTo(w), newW).otherwise(col("weight"))
    p2 = DataModule.normalizeWeightsDf(p1)
    allowed = isPortfolioWeightValid(p2)
  end while
  p = p2
end if
return p
```

4.8. Portfolio Module

The Portfolio Module is responsible for calculating portfolio measures: return and variance. It is also responsible for generating comparison portfolios for testing. Namely, using a random and buy and hold strategies. This component's implementation will be detailed in Section 4.8.1.

4.8.1. Implementation and Functionality

The Portfolio Module is implemented as a singleton Scala object named PortfolioModule with the SparkSessionWrapper trait. Its UML class diagram can be seen in Figure 4.10:

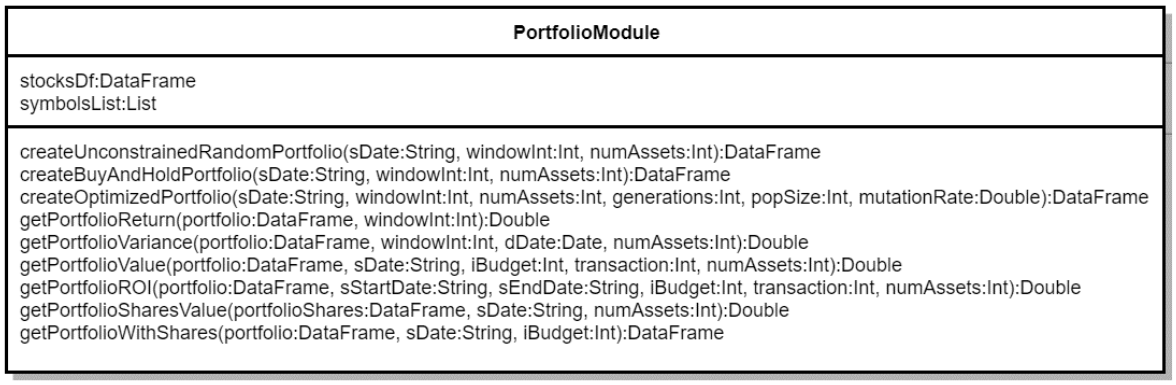

Figure 4.10: Portfolio Module class diagram.

The first set of functions of interest work with the DataFrame representation shown in Table 4.3. There are two methods which allow creation of comparative portfolios for testing. *createUnconstrainedRandomPortfolio* and *createBuyAndHoldPortfolio*. Besides these methods for comparative portfolios, there is also a method for obtaining the optimized portfolio solution by running the GA – *getOptimizedPortfolio*.

The next two methods evaluate portfolio's return and variance, respectively: *getPortfolioReturn* and *getPortfolioVariance.* The method *getPortfolioReturn* calculates the portfolio's return by implementing Equation 3.3. It reads the Spark DataFrame containing the portfolio, adds a temporary column with the result of $R_i w_i$, where R_i is attribute *ROR_x* and w_i attribute *Weight* and then performs a sum aggregation.

As for the function returning a portfolio's variation, *getPortfolioVariance,* it essentially implements Equation 3.4. To achieve this using Spark transformations and actions, the system starts by filtering the dataset by date, in order to attain only the subset of data for the inputted time window. Then it creates a DataFrame which contains only the following columns: *Date, 1st Portfolio Symbol, …, nth Portfolio Symbol.* This is achieved via the pivot transformation, which transforms the row values of a column, in this case the *Symbol* column, into columns containing the corresponding row value of another column, in this case the *ROR_x* column. Then it joins this DataFrame with the original portfolio DataFrame. Next, using the Spark window function, it calculates sample covariance between each entry's *ROR_x* and the *ROR_x* of another symbol, since they are now in the same row. After performing all these

transformations, the DataFrame is persisted to cut RDD lineage. Then, having all the necessary values, it is a simple case of calculating the formula in Equation 3.4. This is achieved by using the Spark MLlib's Row Matrix, which is a row-oriented distributed matrix with access to optimized linear algebra functions, and its *computeCovariance* method. A snippet of this function's pseudo-code can be seen in Table 4.9.

Table 4.9: *getPortfolioVariance* partial pseudo-code. Uses Spark MLlib to compute portfolio variance with asset covariance matrix values.

Get Portfolio Variance – partial snippet

```
Input: portfolio DataFrame, dDate Date, windowInt Int, numAssets Int
Output: portfolio variance Double
…
sampleDf = sampleDf.filter( (col("date") <= rightLimitDate) && (col("date") >= leftLimitDate) )
tempDf = sampleDf.union(portfolioDf).distinct.sort(col("date").asc)
pivotedRorSymbolDf = tempDf.filter($"symbol" isin (pSymbolsList:_*)).groupBy("date")
  .pivot("symbol").agg(first(s"ror_${windowInt}")).sort(asc("date")).toDF
datesDf = pivotedRorSymbolDf.na.drop("any").select("date")
df = pivotedRorSymbolDf.na.drop("any").drop("date")
df = DataModule.setCheckpointDf(df, true)
rows = new VectorAssembler().setInputCols(df.columns).setOutputCol("cov_features")
  .transform(df).select("cov_features").rdd
items_mllib_vector = rows.map(_.getAs[org.apache.spark.ml.linalg.Vector](0))
  .map(org.apache.spark.mllib.linalg.Vectors.fromML)
covMatrix = new RowMatrix(items_mllib_vector).computeCovariance
pairwiseArr = new ListBuffer[Array[Double]]()
for ( i ← 0 to covMatrix.numRows-1){
  for (j ← 0 to covMatrix.numCols-1){
    pairwiseArr += Array(i, j, covMatrix.apply(i,j))
  end for
end for
pairwiseDf = pairwiseArr.map(x => pairRow(x(0), x(1), x(2))).toDF()
outerSumUp = 0.0
outerSumDown = 0.0
outerSum = 0.0
for (i ← 0 to numAssets-1)
  innerSum = 0.0
  for (j ←0 to numAssets-1){
    cov= pairwiseDf.filter( (col("i") === i) && (col("j") === j) ).select("cov").first.getDouble(0)
    innerSum = innerSum + (cov*pWeightsList(i)*pWeightsList(j))
  end for
  outerSum = outerSum+innerSum
end for
return outerSum
```

The next set of functions, *getPortfolioValue* and *getPortfolioROI*, are responsible for calculating a portfolio value, with share allocation and transaction costs, and a portfolio ROI with the same considerations.

As can be seen, there are two other functions in the module, which are used as auxiliary functions for when budget allocation is taken into consideration. These functions use another DataFrame representation, where a *Shares* column is added to the DataFrame Table 4.3 representation. The first one is function *getPortfolioWithShares*, which is responsible for this transformation. The second function, *getPortfolioSharesValue*, is responsible for calculating the value of a portfolio with budget allocation, in EUR, at a certain date, considering currency exchange rates.

4.9. User Module

This straightforward system component allows the user to specify input parameters for the portfolio management system and abstracts the remaining system's complexity. It is responsible for returning an optimized portfolio as a DataFrame and showing the user the following information: portfolio asset allocation in number of shares, portfolio expected summary (return, variance and value in Euros) and portfolio ROI after a certain amount of days, to be defined by the user.

4.9.1. Implementation and Functionality

This component is implemented as a Scala singleton object named UserModule. Contrary to other Scala modules, it does need the SparkSessionWrapper trait. It is a simple wrapper to calls on functions present in the Portfolio Module. Its UML class diagram can be seen in Figure 4.11.

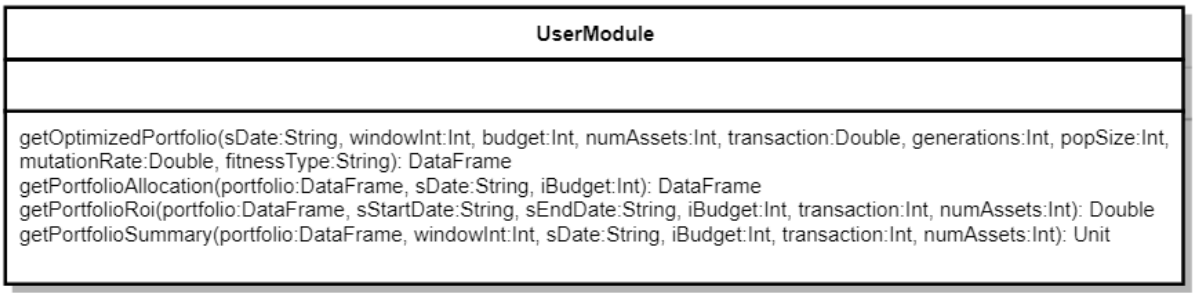

Figure 4.11: User Module class diagram.

Function *getOptimizedPortfolio* is a wrapper for calling Portfolio and Optimization Module functions to get an optimized portfolio via the SO GA. As for function *getPortfolioAllocation*, it simply adds the column *Shares* to a portfolio DataFrame, where the exact number of shares for each asset is depicted, considering the inputted budget.

Function *getPortfolioRoi* returns the specified portfolio ROI at the inputted date and, finally, *getPortfolioSummary* returns the inputted portfolio's return, variance and value in Euros.

5. Experimental Validation

In this Section the performance metrics used to validate the Portfolio Management System are explained. System results need to be understood considering previous solutions and to verify that the goals of this work were achieved. The Portfolio Management System was evaluated according the following performance metrics:

- *Return:* the solution portfolio return was evaluated via the indicator ROI;
- *Computation time/runtime:* the processing time needed to obtain the portfolio solution. The validation comprehends studying how several configurations impact runtime, namely regarding:
    - *Concurrency:* how varying the number of used cores impacts portfolio computation time;
    - *Iterative processing:* how varying the number of iterations impacts runtime;
    - *Memory:* how changing the amount of main memory available to the Spark Portfolio Management Application impacts runtime.

An explanation of the performance metrics is detailed in Section 4.1. Next, the experimental validation results, the simulations used to obtain them and the environment they were performed are described in Section 4.2. Finally, in Section 4.3 an overview and discussion of obtained results is presented.

5.1. Performance Metrics

5.1.1. Computation Time

The last metric used in this work is a computer performance metric: computation time. This is also known as the system's runtime, or how much time does it take from system start until a result is returned to the user. Usually measured in seconds.

Since this is a Big Data system it is important to understand how concurrency and memory size affect computation time. Knowing that the system uses a SO GA iterative algorithm, analyzing the impact of iterative processing on computation time is also relevant.

5.1.2. Return – ROI

The ROI financial performance metric was introduced in Section 2.1 via Equation 2.1. It was used to perform a comparative analysis comparing the portfolio's return to that of ETFs which track market indexes and other portfolios.

Although this metric is widely used and provides a very good measure of a portfolio's financial performance it does provide a certain conclusion for comparing the quality of investments. It is necessary to take into consideration the time window used, since investments might attain the same

ROI with different speeds, and the period of the investment. It also completely neglects portfolio's volatility which is important to understand an investment's stability [77].

5.2. Simulations and Environment

As explained in Section 4.3., there are several input parameters which the user must decide upon before starting the Portfolio Management System. For each simulation a table presenting input parameters is presented. Additionally, the fitness function used is stated.

The environment created to support the simulations encompasses two machines. The first machine launches a VM with CDH 5.13.0 installed. This machine is used keep the Hadoop ecosystem services running, namely Hadoop HDFS 2.6.0 and Spark 2.3.0. Machine details can be seen in Table 5.1.

Table 5.1: Specifications on the machine running CDH.

CDH Machine			
Central Processing Unit (CPU)	**Random Access Memory (RAM)**	**Secondary Memory**	**OS**
Intel i7 6700HQ @ 2.8 GHz	9.5GB DDR3	64GB on an HDD	CentOS 5.11

The second machine is where the Portfolio Management System is run. It corresponds to the Spark Driver. All simulations were performed locally in this pseudo-cluster of a single machine, while the CDH machine provided access to the needed Hadoop environment libraries and HDFS, acting as a server.

YARN and the standalone cluster manager are available in the CDH machine, so tests could use this second machine for the Spark Driver but have the CDH machine be a Spark Executor, effectively using a two-node cluster. Nevertheless, due to the high amount of memory and CPU requirements to launch the CDH distribution on the CDH machine, preliminary observations suggested that this would only degrade performance.

The tests performed by Inoubli et al. [34] with Spark suggest that for datasets smaller than 50GB the standalone cluster manager exhibits the best performance. Due to the stated limitation of not being able to launch an effective two-node cluster, Spark local mode was used for testing as it is the most similar to the standalone cluster manager. While the standalone cluster manager deploys a Driver and Executor on distinct machines with two separate JVM processes, the Spark local mode deploys a single JVM with the Spark Driver as the sole Executor.

In short, simulations were run using just the Spark Driver machine detailed in Table 5.2 using Spark local mode, that is, all computations are performed at the Driver.

Table 5.2: Specifications on the machine running Spark Driver.

Spark Driver Machine			
CPU	**RAM**	**Secondary Memory**	**OS**
Intel i5-3570k @ 4.2 GHz	11 GB DDR3	41 GB on an HDD	Ubuntu 16.04

The dataset used for every simulation can be seen in Table 5.3. This specific dataset was chosen to represent stocks from different continents and countries, traded in different currencies. The addition of volatility ETFs is there to give the algorithm a chance at better results when markets are devaluating (bearish), since the Portfolio Management System does not perform short-selling. The dataset size in the Parquet file format is 324.7MB and has a total of 582 assets. All historical data existent, for each symbol, from 01-01-2006 until 28-06-2018 was considered in this dataset.

Table 5.3: Experimental dataset details.

Assets Type	**Market Index**	**Number of Assets**	**Currency**	**Comment**
Stocks	S&P500	518	USD	Every asset that belonged to S&P500 since July 2015.
ETFs	-	14	USD	Volatility ETFs. Tend to rise in value if stocks are devaluating.
Stocks	Portuguese Stock Index (PSI)	18	EUR	-
Stocks	Deutscher Aktienindex (DAX)	30	EUR	-
Stocks	TSX Venture Exchange (TSXV)	2	USD	Canadian stock exchange.

The performed simulations will be detailed in the following Sections.

5.2.1. Computer Performance Simulations

In this Section, several tests regarding how Portfolio Management System and Spark parameter configurations affect the *Computation Time*, or runtime, metric are detailed. For the following studies, the system general system configuration is detailed in Tables 5.4 and 5.5.

While Tables 5.4 and 5.5 present the general configuration for testing. To perform the following studies some of these parameters need to vary on a study by study basis, in order to understand their impact on overall system performance. Specific GA parameters like population size and number of generations are proposedly relatively low to avoid capping the machine's resources.

Additionally, shown results are the average of five executions for each particular study configuration.

Table 5.4: Portfolio Management System general parameter configuration for computer performance simulations.

Portfolio Management System	
Parameter	**Value**
Date	15-03-2018
Time Window	120 trading days (roughly equivalent to 6 months)
Budget	50000 EUR
Cardinality	20
Transaction Costs	8 EUR
Generations	50
Population Size	16
Mutation Rate	0.03 (3%)
Fitness Function	Portfolio Return

Table 5.5: Spark general parameter configuration for computer performance simulations.

Spark	
Parameter	**Value**
spark.driver.cores	4
spark.driver.memory	10g

Concurrency

In the first simulation scenario, the system concurrency, or ability to perform parallel tasks, was evaluated. Since the number of available CPU cores directly affects the system's ability to perform concurrent tasks, the parameter of interest in this scenario is Spark's *spark.driver.cores* parameter.

By varying this parameter, the impact of the number of available cores on runtime can be studied. To better evaluate concurrency, the *Speedup in latency,* or just speedup, of multi-core executions when compared to single-core executions was calculated, as expressed in Equation 5.1:

$$S_{latency} = \frac{L_{old}}{L_{new}} \tag{5.1}$$

Where L_i is the latency, or runtime, for each execution. Results are depicted in Table 5.6.

Table 5.6: Details the impact of available CPU cores on system runtime.

spark.driver.cores	**Average Runtime (s)**	**Speedup (%)**
1	683.9	-
2	610.8	11.97
3	376.2	81.79
4	353.2	93.63

Iterative Processing

In the second simulation scenario, an evaluation is performed on the system regarding iterative processing with the implemented SO GA. Since each generation represents a new GA iteration, the

system parameter of interest in this study is *Generations*. By varying the *Generations* parameter, the impact of the number of iterations on runtime can be studied. The results are depicted in Figure 5.1.

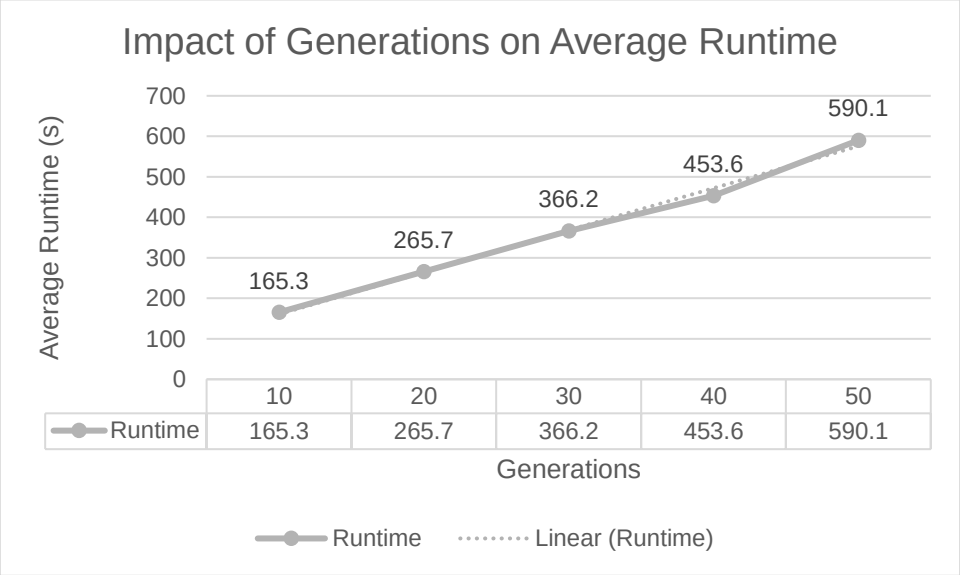

Figure 5.1: Graphic depicting how the number of generations (iterations) impacts average system runtime.

Another study which was performed with this second scenario was analyzing the evolution of each iteration's runtime for 50 iterations (generations). These results can be seen in Figure 5.2.

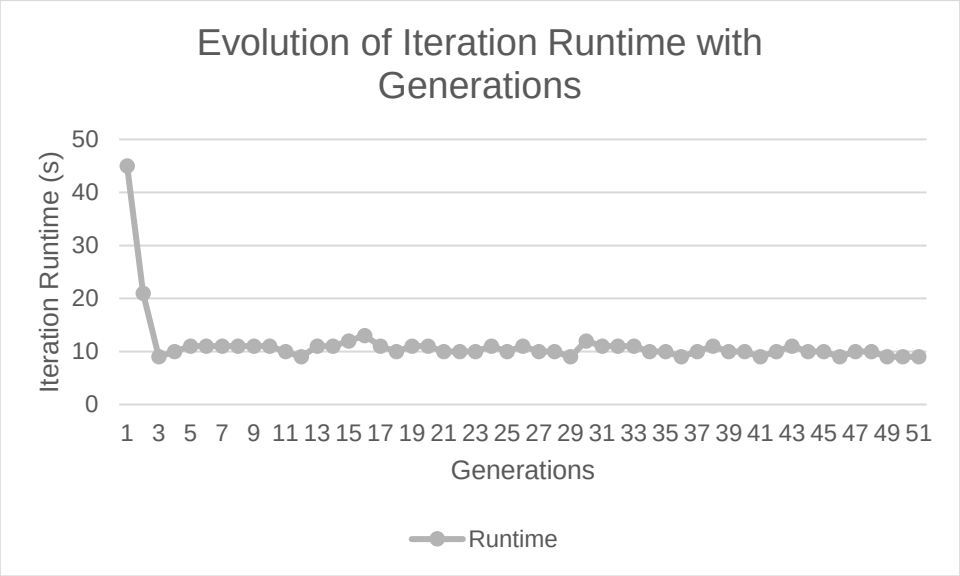

Figure 5.2: Graphic depicting the evolution of iteration runtime during system execution for 50 generations.

Memory

The third and last computer performance study scenario evaluates how varying the amount of main memory available to the Spark Driver affects the system's runtime.

Spark memory allocation has two main usages: execution, where data being processed is buffered – such as, for shuffles, joins, aggregations and sorts –; and storage, used to cache recurrently used RDDs. How contention between these two memory usages is managed and overall memory size can have an impact on Spark's performance, according to Inoubli et al. [34].

Thus, the parameter of interest in this simulation is Spark's *spark.driver.memory* parameter which defines the amount of memory allocated to the Spark Driver in the JVM heap. The impact of varying the *spark.driver.memory* parameter on runtime is depicted in Figure 5.3.

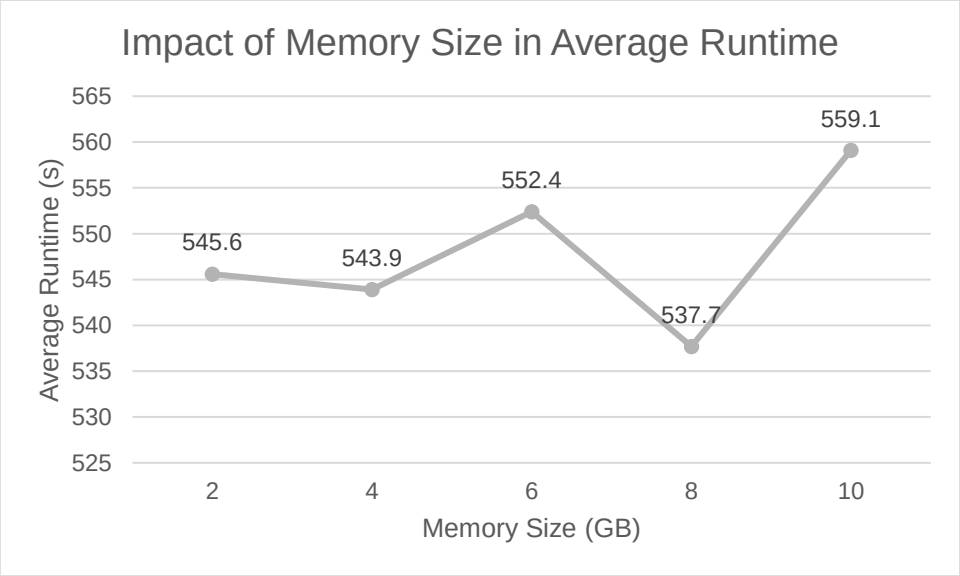

Figure 5.3: Graphic depicting how the memory size impacts average system runtime.

5.2.2. Financial Performance Simulations

In this Section, several tests regarding how Portfolio Management System and Spark parameter configurations affect the *ROI* metric are detailed.

Two simulations will be made: one during a bullish market period and the last one in a sideways market. Bearish markets are omitted, since the only mechanism the system has to deal with these markets are volatility ETFs which were unavailable in the last global bear market, in the 2008 crisis.

For the following studies, the system general system configuration is detailed in Table 5.7 and the Spark configurations remain the ones shown in Table 5.5. Additionally, shown results are the average of five executions for each case study configuration. It is also worthy to note that the user profile is of a risk-taker, since the weights of the linear combination of return and variance are 90% and 10%, respectively.

While Table 5.7 presents the general configuration for testing, to perform the following studies *Date* and *Time Window* need to vary on a study by study basis, in order to understand their impact on system performance. Both metrics will be evaluated at the moment of portfolio creation and then after the following four time windows:

- 5 trading days, which represent a week;

- 10 trading days, which represent two weeks;
- 20 trading days, which represent a month;
- 60 trading days, which represent a quarter.

System results will be compared with a random portfolio (assets and weights chosen randomly), an equally distributed buy and hold portfolio (comprised of the biggest earners in the analyzed time window) and with SPY, an ETF that mimics the growth of S&P500.

Table 5.7: Portfolio Management System general parameter configuration for financial performance simulations.

Portfolio Management System	
Parameter	**Value**
Date	03-01-2017; 01-02-2018
Time Window	120 trading days (roughly equivalent to 6 months)
Budget	100 000 EUR
Cardinality	20
Transaction Costs	8 EUR
Generations	100
Population Size	32
Mutation Rate	0.03 (3%)
Fitness Function	Portfolio Return and Variance Linear Combination (Weights: 0.90; 0.10)

Case Study 1 – Sideways Market

The system's portfolio is created at 02-01-2018, and then analyzed overall several windows until 02-04-2018.

The composition of the portfolios regarding each of the five system executions can be seen in Tables 5.8 to 5.12.

Table 5.8: Case Study 1 - Portfolio 1 composition

Index	Currency	Symbol	Weight
SP500	USD	ADS	0.026257
SP500	USD	AON	0.016397
TSXV	USD	BSK	0.286375
SP500	USD	CNX	0.11168
SP500	USD	D	0.0277
SP500	USD	DHR	0.080606
SP500	USD	EMR	0.019389
SP500	USD	ENDP	0.083146
SP500	USD	EVHC	0.022295
SP500	USD	HCP	0.019523
SP500	USD	HSIC	0.017488
SP500	USD	IFF	0.044395
SP500	USD	MDT	0.03253
SP500	USD	PM	0.019914
SP500	USD	QRVO	0.015954
LS	EUR	RAM_LS	0.065193
SP500	USD	RF	0.035024
SP500	USD	TGT	0.043765
SP500	USD	WHR	0.013266
SP500	USD	XEC	0.0191

Table 5.9: Case Study 1 - Portfolio 2 composition

Index	Currency	Symbol	Weight
SP500	USD	AAP	0.018291
SP500	USD	AMZN	0.09007
TSXV	USD	BSK	0.241322
SP500	USD	CHRW	0.054619
SP500	USD	D	0.017278
SP500	USD	ED	0.035963
SP500	USD	ESRX	0.021183
SP500	USD	EVHC	0.030639
SP500	USD	EXC	0.029289
SP500	USD	EXR	0.091117
DE	EUR	HEN3_DE	0.041194
SP500	USD	INTC	0.06259
SP500	USD	MLM	0.01035
SP500	USD	ORLY	0.012855
SP500	USD	PM	0.060832
DE	EUR	SAP_DE	0.029989
SP500	USD	STI	0.059322
SP500	USD	TDG	0.028354
SP500	USD	TMK	0.02964
SP500	USD	XRAY	0.035104

Table 5.10: Case Study 1 - Portfolio 3 composition

Index	Currency	Symbol	Weight
SP500	USD	AET	0.059105
SP500	USD	AKAM	0.037432
SP500	USD	BBT	0.069285
TSXV	USD	BSK	0.220439
SP500	USD	CINF	0.062566
SP500	USD	CPB	0.02221
SP500	USD	DO	0.06256
SP500	USD	EFX	0.031151
SP500	USD	GLW	0.021527
SP500	USD	HAS	0.01931
DE	EUR	HEI_DE	0.011016
SP500	USD	JPM	0.064653
SP500	USD	LYB	0.037638
SP500	USD	NLSN	0.045465
SP500	USD	PKG	0.071605
SP500	USD	SCHW	0.038989
SP500	USD	SLB	0.045226
SP500	USD	UNH	0.028407
SP500	USD	UTX	0.023699
SP500	USD	ZTS	0.027716

Table 5.11: Case Study 1 - Portfolio 4 composition

Index	Currency	Symbol	Weight
SP500	USD	AXP	0.036961
TSXV	USD	BSK	0.184562
SP500	USD	BWA	0.039243
SP500	USD	CSX	0.083307
SP500	USD	DISCK	0.044239
SP500	USD	DPS	0.043088
ETF	USD	EXIV	0.077504
SP500	USD	EXPD	0.014039
SP500	USD	FOXA	0.058707
SP500	USD	HD	0.022885
SP500	USD	IBM	0.067873
SP500	USD	NVDA	0.110319
SP500	USD	OI	0.03012
SP500	USD	OKE	0.016473
SP500	USD	PBCT	0.013393
SP500	USD	ROK	0.050686
DE	EUR	SAP_DE	0.025475
SP500	USD	SRCL	0.013495
SP500	USD	TSN	0.056994
SP500	USD	XRX	0.010638

Table 5.12: Case Study 1 - Portfolio 5 composition

Index	Currency	Symbol	Weight
SP500	USD	ADBE	0.048822
TSXV	USD	BSK	0.171106
SP500	USD	BWA	0.019683
SP500	USD	CHRW	0.094965
SP500	USD	DVN	0.066917
SP500	USD	EA	0.020655
SP500	USD	ESS	0.032757
SP500	USD	ETFC	0.05906
SP500	USD	FB	0.110763
LS	EUR	GALP_LS	0.041236
SP500	USD	HAS	0.012129
SP500	USD	MTB	0.028814
SP500	USD	PCAR	0.024787
SP500	USD	PYPL	0.083491
SP500	USD	ROK	0.035884
LS	EUR	SEM_LS	0.043932
SP500	USD	TDC	0.020034
SP500	USD	VRSN	0.018825
SP500	USD	WHR	0.03898
SP500	USD	YUM	0.027159

And their respective calculated ROI, in percentage for each applicable sliding window, can be seen in Table 5.13.

Table 5.13: Details the ROI for each of the solution portfolio compositions in case study 1, in each time window.

Date	ROI 1 (%)	ROI 2 (%)	ROI 3 (%)	ROI 4 (%)	ROI 5 (%)
02/01/2018	0	0	0	0	0
09/01/2018	4.2	3.7	3.6	3.3	2.9
16/01/2018	4.6	3.9	3.5	3.4	2.7
02/02/2018	-4.1	-0.9	-1.0	0.4	-0.1
02/04/2018	-4.7	-1.1	-5.5	-5.6	-3.6

Next, a ROI comparison between mean of achieved portfolios, best portfolio (Portfolio 1 in this case), SPY, Buy and Hold and Random portfolios is depicted in Figure 5.4.

Finally, a graphic showing how fitness evolution behaved in the best GA portfolio attained in this case study can be consulted in Figure 5.5.

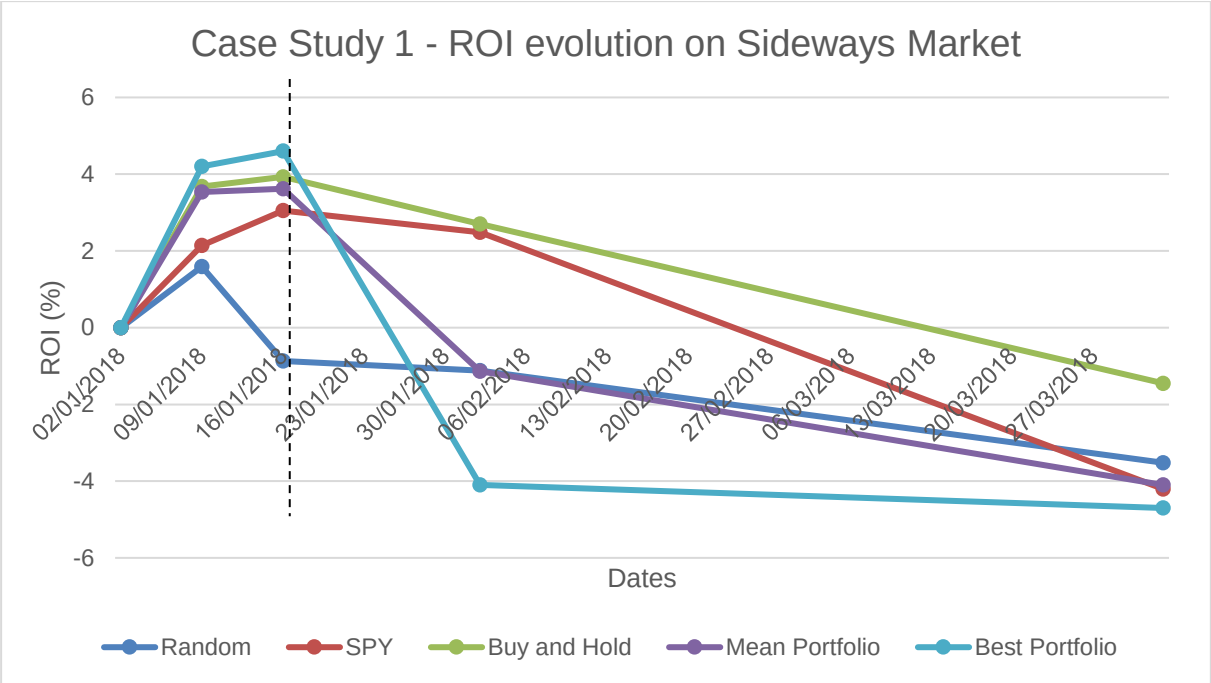

Figure 5.4: Depicts how ROI evolves in a 60 trading day period for: a random portfolio, a buy and hold portfolio, the SPY ETF, the mean of the system's attained portfolios and the system best attained portfolio in case study 1. The line in the graph marks the last point at which solution portfolios are a valid option.

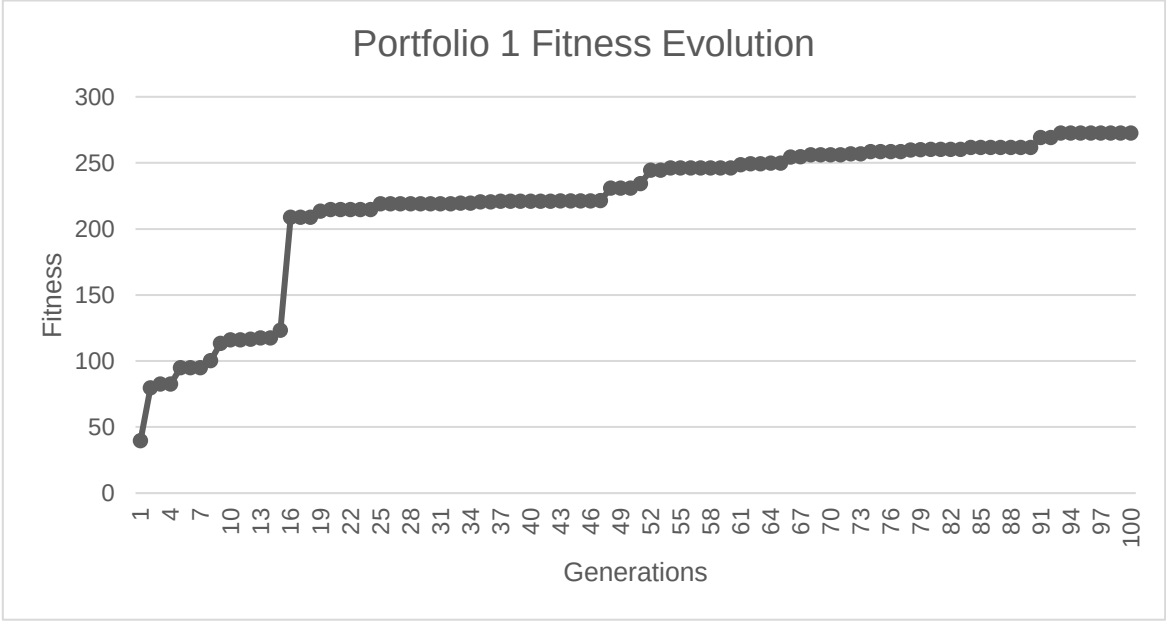

Figure 5.5: Evolution of portfolio 1 fitness with generations, in case study 1.

Case Study 2 – Bullish Market

The system's portfolio is created at 03-01-2017, and then analyzed overall several windows until 03-04-2017. The composition of the portfolios regarding each of the five system executions can be seen in Tables 5.14 to 5.18.

Table 5.14: Case Study 2 - Portfolio 1 composition

Index	Currency	Symbol	Weight
TSXV	USD	BSK	0.186741
SP500	USD	CMA	0.11099
SP500	USD	PWR	0.101637
SP500	USD	TDG	0.058356
SP500	USD	MCO	0.056997
SP500	USD	ALGN	0.052276
SP500	USD	XEL	0.048384
SP500	USD	MAC	0.047385
SP500	USD	HD	0.047287
SP500	USD	MSI	0.046508
SP500	USD	CTAS	0.044411
SP500	USD	CMI	0.043139
SP500	USD	EMR	0.04058
SP500	USD	COF	0.028445
ETF	USD	SVXY	0.020947
SP500	USD	ABT	0.015911
SP500	USD	MA	0.014139
SP500	USD	STI	0.013467
SP500	USD	WM	0.011912
SP500	USD	HBAN	0.01049

Table 5.15: Case Study 2 - Portfolio 2 composition

Index	Currency	Symbol	Weight
SP500	USD	XYL	0.055372
SP500	USD	URI	0.037146
SP500	USD	UPS	0.024609
SP500	USD	TROW	0.058683
SP500	USD	TRIP	0.050007
LS	EUR	PHR_LS	0.030047
SP500	USD	MUR	0.015003
SP500	USD	MRO	0.051149
SP500	USD	MHK	0.024366
SP500	USD	KMI	0.057467
SP500	USD	BLX	0.022988
SP500	USD	BR	0.053223
TSXV	USD	BSK	0.22921
DE	EUR	CBK_DE	0.073796
SP500	USD	CNX	0.087807
SP500	USD	DISH	0.014923
SP500	USD	ECL	0.015166
SP500	USD	FLR	0.043536
SP500	USD	GOOGL	0.021161
SP500	USD	IPG	0.034343

Table 5.16: Case Study 2 - Portfolio 3 composition

Index	Currency	Symbol	Weight
SP500	USD	XL	0.022172
SP500	USD	WLTW	0.018948
SP500	USD	WDC	0.040738
SP500	USD	SWKS	0.058026
SP500	USD	STX	0.062288
SP500	USD	PX	0.041038
SP500	USD	OKE	0.012651
SP500	USD	LLL	0.027883
SP500	USD	LB	0.037756
SP500	USD	KSU	0.044452
SP500	USD	KMX	0.077256
SP500	USD	GOOG	0.03835
SP500	USD	FIS	0.054828
SP500	USD	FCX	0.039921
SP500	USD	FBHS	0.032916
SP500	USD	CRM	0.030248
TSXV	USD	BSK	0.25197
SP500	USD	AOS	0.049863
SP500	USD	AIZ	0.03818
SP500	USD	ALGN	0.020516

Table 5.17: Case Study 2 - Portfolio 4 composition

Index	Currency	Symbol	Weight
TSXV	USD	BSK	0.188268
SP500	USD	NVDA	0.130633
SP500	USD	AMD	0.078924
SP500	USD	BBBY	0.061067
SP500	USD	NCLH	0.059021
SP500	USD	LKQ	0.055776
SP500	USD	CA	0.055242
LS	EUR	SONC_LS	0.048236
SP500	USD	ISRG	0.045876
LS	EUR	PHR_LS	0.043212
DE	EUR	IFX_DE	0.039239
SP500	USD	ALLE	0.036932
SP500	USD	NFLX	0.033421
ETF	USD	SVXY	0.031942
SP500	USD	AVGO	0.025197
SP500	USD	INTC	0.016241
SP500	USD	K	0.014904
SP500	USD	MAS	0.013756
SP500	USD	WMB	0.011543
SP500	USD	R	0.010571

Table 5.18: Case Study 2 - Portfolio 5 composition

Index	Currency	Symbol	Weight
SP500	USD	ULTA	0.019557
SP500	USD	UHS	0.024244
SP500	USD	UA	0.016909
SP500	USD	TXN	0.035894
SP500	USD	SBAC	0.011965
SP500	USD	R	0.045238
SP500	USD	OKE	0.040096
SP500	USD	NEM	0.037668
LS	EUR	IBS_LS	0.082921
SP500	USD	HPE	0.074393
SP500	USD	FOXA	0.031728
SP500	USD	ABBV	0.042807
SP500	USD	ANSS	0.036035
SP500	USD	BR	0.05634
TSXV	USD	BSK	0.242876
SP500	USD	CHK	0.080322
SP500	USD	CTXS	0.032986
SP500	USD	DHR	0.03716
SP500	USD	DLTR	0.024106
SP500	USD	EQT	0.026753

And their respective calculated ROI, in percentage for each applicable sliding window, can be seen in Table 5.19.

Table 5.19: Details the ROI for each of the solution portfolio compositions in case study 2, in each time window.

Date	ROI 1 (%)	ROI 2 (%)	ROI 3 (%)	ROI 4 (%)	ROI 5 (%)
03/10/2017	0	0	0	0	0
10/01/2017	0.91	1.9	1.94	2.31	1.57
17/01/2017	0.83	2.82	3.21	1.38	2.77
03/02/2017	-2.10	-0.96	0.82	2.92	-2.81
03/04/2017	5.11	3.50	5.62	12.6	4.65

Next, a ROI comparison between mean of achieved portfolios, best portfolio (Portfolio 3 in this case – considering only the first two weeks after composition), SPY, Buy and Hold and Random portfolios is depicted in Figure 5.7.

Finally, a graphic showing how fitness evolution behaved in the best GA portfolio attained in this case study can be consulted in Figure 5.8.

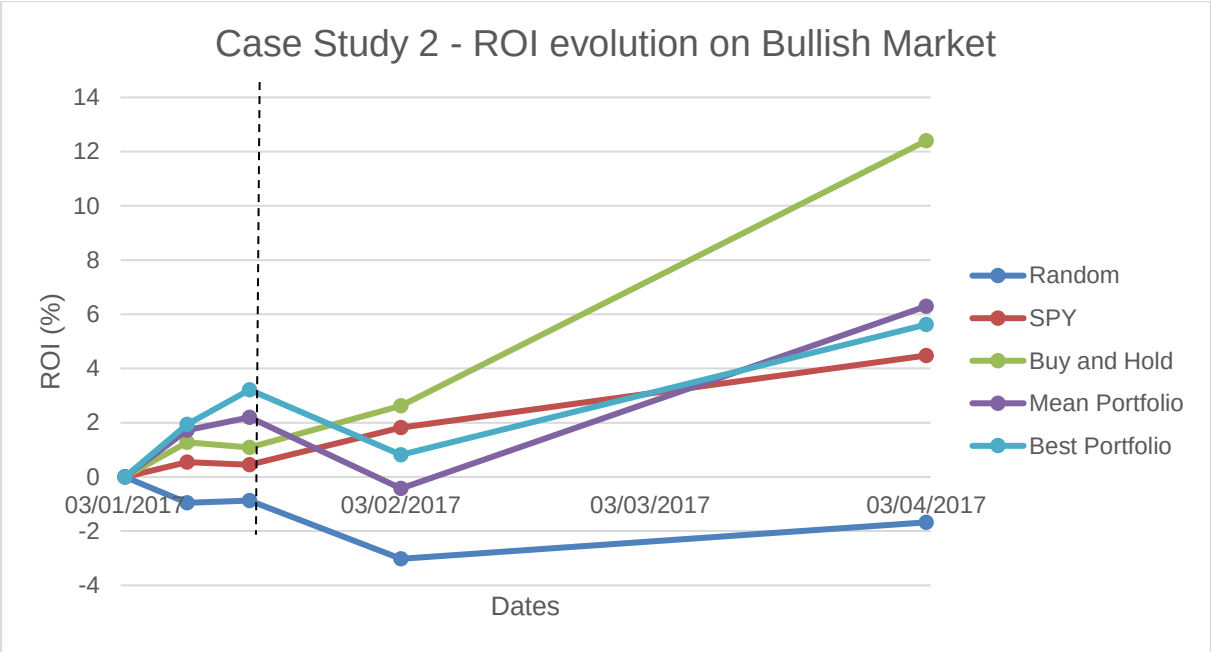

Figure 5.6: Depicts how ROI evolves in a 60 trading day period for: a random portfolio, a buy and hold portfolio, the SPY ETF, the mean of the system's attained portfolios and the system best attained portfolio in case study 2. The line in the graph marks the last point at which solution portfolios are the best option.

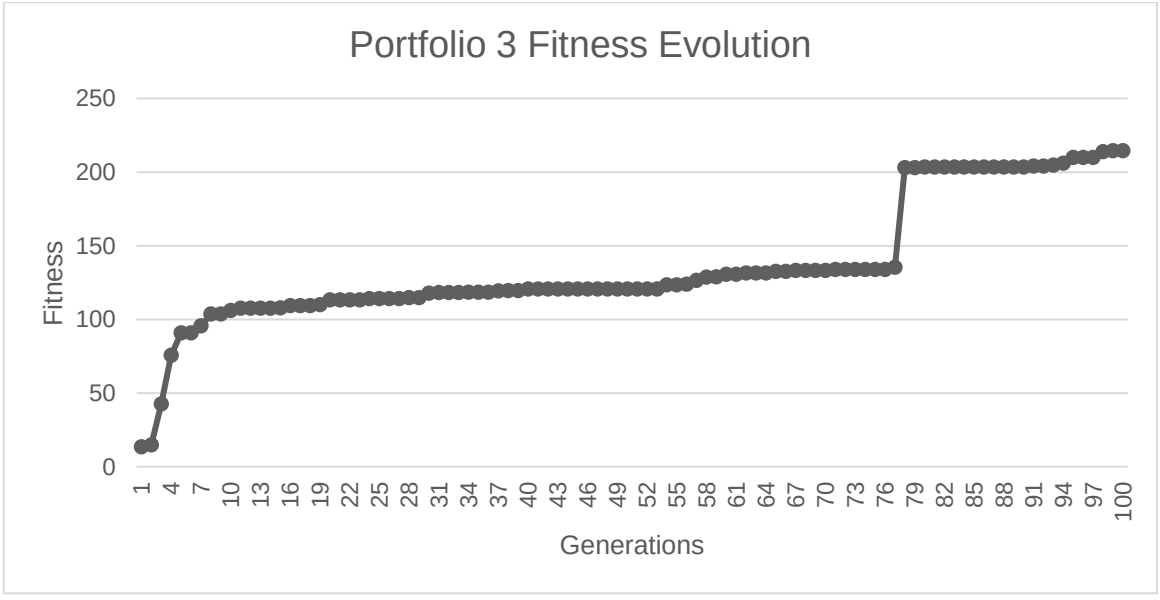

Figure 5.7: Evolution of portfolio 3 fitness with generations, in case study 2.

5.3. Overview and Discussion

Computer performance was first validated regarding concurrency. There is a clear gain in increasing the number of processing cores available to the system, with a particular large increase from two to three

processing cores, yielding more than 60% speedup with that increase, as seen in Table 5.6. Secondly, iterative processing performance was validated. It is apparent that the number of iterations has no significant influence on runtime, since Figure 5.1 shows that the system runtime is characterized by a linear slope. Concerning each iteration's runtime, the evolution study in Figure 5.2 shows that it remains relatively constant, with exception of the initial two generations. This might be due to random population initiation in the first generation. Finally, regarding memory size available to the launched Spark application, results can be seen in Figure 5.3. Despite Inoubli et al. [34] results, which imply that memory size has a dramatic effect on runtime, our study demonstrates that in, for this application design and dataset size, it does not. The memory size appears to have no impactful influence on runtime in local mode, as long as it is sufficient to cache and process the dataset.

As for financial performance, in the ROI analysis, seen in Figures 5.4 and 5.6, the best portfolio manages to beat both the Buy and Hold strategy and SPY in the first two weeks after optimization, reaching ROIs up to 2% above those of SPY. This suggest that this period, between the first and second week, is ideal for a re-optimization of asset allocation. The mean portfolio also attains promising results, beating SPY in both case studies and buy and hold in case study 2. These results show that, although financial performance is adequate, there is still margin for growth. Future studies should include other management models, like TA and FA , additional constraints and multi-period considerations, with periodic rebalancing.

Results are influenced by the used GA configurations and the used user profile: a user who favors returns over volatility. Another fact that might influence these results is convergence, or how close the algorithm is w.r.t. the globally optimal solution. In Figures 5.5 and 5.7 fitness grows at a small rate towards the end of the 100 generations. Nevertheless, there are big jumps in the graph in terms of fitness. These jumps represent the GA getting outside the scope of a local maximum and finding a new best candidate to explore. Since 100 generations is a traditional low number of generations, results could improve as the number of generations increases, possibly reaching the global maximum. Larger population sizes could also lead to faster convergence.

6. Conclusions and Future Work

The experimental validation described in Section 5 denotes that portfolio management using a GA can be successfully implemented in a Big Data framework, while attaining average computer performance results – regarding runtime in a single machine – but with great concurrency, which is very promising for scalability. Financial performance results were promising with the best run attaining higher ROI than both the SPY ETF and buy and hold strategy in the first 10 trading days after optimization, with falling performance thereafter. The mean of every run also has higher ROI than SPY, consistently, in the first 10 trading days.

Despite the results attained with this system, there is a lot of studying left to do in this type of approach to portfolio management, namely regarding the use of distinct portfolio management models and regarding scalability and optimization of computation time. However, this work provides the reader with a suitable knowledge base regarding portfolio management, genetic algorithms and Big Data frameworks and, as such, it is a good starting point for further exploration in this field.

There could be contention regarding the pertinence of using Big Data frameworks for historical price analysis due to traditional dataset sizes. However, with a current estimation of 630 thousand tradeable companies worldwide, excluding commodities, bonds and other asset types, and knowing that there will be increasingly more available historical data, traditional systems will eventually be unable to cope with the scale of computations and memory required. Simple calculations show us that considering 90 bytes per CSV file data entry and 30 years of data (252 trading days per year), a dataset considering the aforementioned amount of companies would amount to approximately 400GB. The use of a Big Data framework, when hardware resources to mount a cluster are available, essentially future proof a portfolio management system, regarding increasing historical dataset sizes, while maintaining moderate execution times. Additionally, it allows for easy transition into real-time applications due to stream processing capabilities.

Nevertheless, it would be interesting to compare this system's results with a similar system using Pandas [68] , for example, and discover at which point in a dataset size is it worth developing in a Big Data framework, where computational performance is concerned.

6.1. Contributions

This section denotes the contributions made by this work in the field of automated portfolio management:

- Successful implementation of a portfolio management system in a Big Data framework, using a GA with computational and financial performance validation, via runtime and ROI, respectively;
- Overview of the current portfolio management and Big Data frameworks literature. Discussion on the overlap of the two domains and remarks on how to design a system for this overlap,

namely on how portfolio management can be achieved with and benefit from the use of Big Data frameworks and large dataset sizes.

6.2. Future work

This Section describes the Portfolio Management System current limitations and/or possible improvements to circumvent, minimize, remove those limitations, or simply to add extra functionality.

6.2.1. Current Limitations and Possible Improvements

- The system does not allow short-selling of assets, so it has limited options for maintaining return levels when the market is bearish. It would be an interesting improvement to add the ability to perform short-selling;
- The asset types accepted are only stocks and ETFs. The system should be prepared to work with all different types of financial assets and real assets, like commodities;
- No experimental validation regarding scalability with additional Spark Executors with the Portfolio Management System specific GA iterative process. The next step would be to validate experimental results with a big enough dataset and a large cluster;
- Naïve assumptions in modelling the market, regarding the reliability of historical data in prediction the future. A possible improvement would be to try out different passive and active management models, like TA or FA, and compare their results.;
- SO optimization does not allow for a complete overview of trade-offs, unless run multiple times. It would be interesting to try out different MO GAs, such as NSGA-II or SPEA-II;
- Only three constraints: cardinality, floor and ceiling. An additional improvement would be to model the MPT-based optimization problem to handle additional constraints;
- The dataset used in this thesis is not big enough to compare needed financial performance on real production environments. Expand the dataset to hold several distinct assets and compare performance, financial and computational, w.r.t. dataset size;
- Use Spark Streaming or choose another Big Data framework amongst the streaming-able subset to extend the solution for real-time data ingestion;
- A comparative analysis on GA operators to assert which present better financial performance in this passive management model;
- Compare this Spark results with Flink results, as some existing comparative analysis suggest Flink might be faster for iterative processes;
- Extend the system to allow for multi-period considerations.

7. References

1. Maginn JL, Tuttle DL, McLeavey DW, Pinto JE (2007) Managing investment portfolios: a dynamic process, 3rd ed. John Wiley & Sons, Inc.
2. Gorgulho A, Neves RFMF, Horta NCG (2013) Intelligent Financial Portfolio Composition based on Evolutionary Computation Strategies, 1st ed. doi: 10.1007/978-3-642-32989-0
3. Pitzl Financial (2014) Investment Methodology White Paper.
4. Andersen A, Mikelsen S (2012) A Novel Algorithmic Trading Framework Applying Evolution and Machine Learning for Portfolio Optimization. Norwegian University of Science and Technology
5. INFORMS Computing Society The Nature of Mathematical Programming. https://web.archive.org/web/20140305080324/http://glossary.computing.society.informs.org/index.php?page=nature.html. Accessed 29 Nov 2017
6. Di Tollo G (2008) Portfolio Selection by Metaheuristics. Università degli Studi “G. d’Annunzio” Chieti-Pescara
7. Markowitz H (1952) Portfolio Selection. J Finance 7:77–91
8. Stein M, Branke J, Schmeck H (2005) Portfolio selection: How to integrate complex constraints. Inst. Appl. Informatics Form. Descr. Methods. Univ. Karlsruhe (TH), June 1:
9. Silva AD, Neves RF, Horta N (2016) Portfolio Optimization Using Fundamental Indicators Based on Multi-Objective EA, 1st ed. doi: 10.1007/978-3-319-29392-9
10. Rothlauf F (2011) Design of Modern Heuristics. doi: 10.1007/978-3-540-72962-4
11. Woodside-Oriakhi M, Lucas C, Beasley JE (2011) Heuristic algorithms for the cardinality constrained efficient frontier. Eur J Oper Res 213:538–550
12. Cura T (2009) Particle swarm optimization approach to portfolio optimization. Nonlinear Anal Real World Appl 10:2396–2406
13. Busetti F (2000) Metaheuristic approaches to realistic portfolio optimisation. doi: 10.2495/CF060351
14. Soleimani H, Golmakani HR, Salimi MH (2009) Markowitz-based portfolio selection with minimum transaction lots, cardinality constraints and regarding sector capitalization using genetic algorithm. Expert Syst Appl 36:5058–5063
15. Bienstock D (1996) Computational study of a family of mixed-integer quadratic programming problems. Math Program 74:121–140
16. Cesarone F, Scozzari A, Tardella F (2011) Portfolio selection problems in practice: a comparison between linear and quadratic optimization models. arXiv Prepr arXiv11053594 1–28

17. Gendreau M, Potvin J-Y (2010) Handbook of Metaheuristics, 2nd ed. Handb Metaheuristics. doi: 10.1007/978-1-14419-1665-5

18. Yang X-S (2011) Metaheuristic Optimization. Scholarpedia 15

19. Bianchi L, Dorigo M, Gambardella LM, Gutjahr WJ (2009) A survey on metaheuristics for stochastic combinatorial optimization. Nat Comput 8:239–287

20. Bloomberg. https://www.bloomberg.com/europe. Accessed 30 Dec 2017

21. Yahoo! Finance. https://finance.yahoo.com/. Accessed 30 Dec 2017

22. Bhosale HS, Gadekar DP (2014) A Review Paper on Big Data and Hadoop. Int J Sci Res Publ 4:2250–3153

23. Zaharia M, Franklin MJ, Ghodsi A, et al (2016) Apache Spark: a Unified Engine for Big Data Processing. Commun ACM 59:56–65

24. Broby D (2011) Equity Index Construction. J Index Invest 2:36–39

25. Ben-David I, Franzoni F, Moussawi R (2017) Exchange-Traded Funds. Annu Rev Financ Econ 9:annurev-financial-110716-032538

26. Kadan O, Liu F, Liu S (2012) Systematic Risk. St. Louis

27. Malkiel BG (2015) A random walk down wall street: The time-tested strategy for succesful investing, 11th ed. W. W. NORTON & COMPANY, New York - London

28. Achelis SB (2000) Technical Analysis from A to Z. Search 77:33–4

29. Fama EF (1970) Efficient Capital Markets-A Review of Theory and Empirical Work. J Finance 25:383–417

30. Shiller RJ (2003) From Efficient Markets to Behavioral Finance. J Econ Perspect 17:83–104

31. Barberis N, Thaler RH (2002) A Survey of Behavioral Finance. SSRN Electron J. doi: 10.2139/ssrn.327880

32. Skolpadungket P, Dahal K, Harnpornchai N (2007) Portfolio optimization using multi-objective genetic algorithms. 2007 IEEE Congr Evol Comput 516–523

33. Roberts MC, Dizier AST, Vaughan J Multiobjective optimization: portfolio optimization based on goal programming methods.

34. Inoubli W, Aridhi S, Mezni H, Maddouri M, Mephu Nguifo E (2018) An Experimental Survey on Big Data Frameworks. Futur Gener Comput Syst 86:546–564

35. Zhang J (2018) Exploring and Evaluating the Scalability and Efficiency of Apache Spark using Educational Datasets. York University, Toronto, Ontario

36. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I (2010) Spark : Cluster Computing

with Working Sets. HotCloud'10 Proc 2nd USENIX Conf Hot Top cloud Comput 10

37. Shvachko K, Kuang H, Radia S, Chansler R (2010) The Hadoop Distributed File System. 2010 IEEE 26th Symp Mass Storage Syst Technol MSST2010 1–10

38. Ryza S (2013) Migrating to MapReduce 2 on YARN (For Operators). In: Cloudera Blog. http://blog.cloudera.com/blog/2013/11/migrating-to-mapreduce-2-on-yarn-for-operators/. Accessed 24 Jun 2018

39. Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE (2008) Bigtable: A Distributed Storage System for Structured Data. ACM Trans Comput Syst 26:1–26

40. Sinha S (2018) Hadoop Ecosystem: Hadoop Tools for Crunching Big Data. In: Edureka. https://www.edureka.co/blog/hadoop-ecosystem. Accessed 25 Aug 2018

41. Carbone P, Ewen S, Haridi S, Katsifodimos A, Markl V, Tzoumas K (2015) Apache Flink: Unified Stream and Batch Processing in a Single Engine. Data Eng 36:28–38

42. Aranha CC, Iba H (2008) A tree-based GA representation for the portfolio optimization problem. Proc 10th Annu Conf Genet Evol Comput - GECCO '08 873–880

43. Aranha C (2007) Portfolio Management with Cost Model using Multi Objective Genetic Algorithms. University of Tokyo

44. Cui T, Cheng S, Bai R (2014) A combinatorial algorithm for the cardinality constrained portfolio optimization problem. Proc 2014 IEEE Congr Evol Comput CEC 2014 491–498

45. Golmakani HR, Fazel M (2011) Constrained portfolio selection using particle swarm optimization. Expert Syst Appl 38:8327–8335

46. Pai GAV, Michel T (2014) Metaheuristic multi-objective optimization of constrained futures portfolios for effective risk management. Swarm Evol Comput 19:1–14

47. Steinbach M (1999) Markowitz revisited: single-period and multi-period mean-variance models. 30:

48. Ab Wahab MN, Nefti-Meziani S, Atyabi A (2015) A comprehensive review of swarm optimization algorithms. PLoS One 10:1–36

49. Chang TJ, Meade N, Beasley JE, Sharaiha YM (2000) Heuristics for cardinality constrained portfolio optimisation. Comput Oper Res 27:1271–1302

50. Lwin K, Qu R (2013) A hybrid algorithm for constrained portfolio selection problems. Appl Intell 39:251–266

51. Di Tollo G, Stützle T, Birattari M (2014) A metaheuristic multi-criteria optimisation approach to portfolio selection. J Appl Oper Res 6:222–242

52. Streichert F, Ulmer H, Zell A (2003) Evolutionary Algorithms and the Cardinality Constrained Portfolio Optimization Problem. Oper Res Proc 2003, Sel Pap Int Conf Oper Res 1–8

53. Streichert F, Ulmer H, Zell A (2004) Comparing Discrete and Continuous Genotypes on the Constrained Portfolio Selection Problem. Genet Evol Comput 1239–1250

54. Aranha CDC, Iba H (2009) Using memetic algorithms to improve portfolio performance in static and dynamic trading scenarios. Proc 11th Annu Conf Genet Evol Comput - GECCO '09 1427

55. Deb K, Steuer RE, Tewari R, Tewari R (2011) Bi-objective portfolio optimization using a customized hybrid NSGA-II procedure. Lect Notes Comput Sci (including Subser Lect Notes Artif Intell Lect Notes Bioinformatics) 6576 LNCS:358–373

56. Maguire P, O'Sullivan D, Moser P, Dunne G (2012) Risk-adjusted portfolio optimisation using a parallel multi-objective evolutionary algorithm. 2012 IEEE Conf Comput Intell Financ Eng Econ 1–8

57. Lwin K, Qu R, Kendall G (2014) A learning-guided multi-objective evolutionary algorithm for constrained portfolio optimization. Appl Soft Comput J 24:757–772

58. Jothimani D, Shankar R, Yadav SS (2014) A Big Data Analytical Framework for Portfolio Optimization. 6–10

59. Yee P (2011) Large Scale Portfolio Optimization on Cloud - Technical Report. Hong Kong

60. Reyes-Ortiz JL, Oneto L, Anguita D (2015) Big data analytics in the cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf. Procedia Comput Sci 53:121–130

61. Rivera AJ, Pérez-Godoy MD, Pulgar F, Jesus MJ del (2015) GenRBFNSpark : A first implementation in Spark of a genetic algorithm to RBFN design. Actas la XVI Conf la Asoc Española para la Intel Artif 1011–1020

62. Cao B, Li W, Zhao J, Yang S, Kang X, Ling Y, Lv Z (2016) Spark-Based Parallel Cooperative Co-evolution Particle Swarm Optimization Algorithm. 2016 IEEE Int Conf Web Serv 570–577

63. Erraissi A, Belangour A, Tragha A (2017) A Comparative Study of Hadoop-based Big Data Architectures. Int. J. Web Appl. IJWA 9:

64. Cloudera. In: Cloudera Blog. https://www.cloudera.com/. Accessed 20 Dec 2017

65. CentOS. https://www.centos.org/. Accessed 24 Apr 2018

66. Damji J (2016) A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets. In: databricks Blog. https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html. Accessed 24 Mar 2018

67. Yahoo! Finance Fix for Pandas Datareader. https://pypi.org/project/fix-yahoo-finance/. Accessed 24 Jun 2018

68. McKinney W (2011) pandas: a Foundational Python Library for Data Analysis and Statistics. Python High Perform Sci Comput 1–9

69. Huai Y, Armbrust M (2015) Introducing Window Functions in Spark SQL. In: databricks Blog. https://databricks.com/blog/2015/07/15/introducing-window-functions-in-spark-sql.html. Accessed 20 Feb 2018

70. Melnik S, Gubarev A, Long JJ, Romer G, Shivakumar S, Tolton M, Vassilakis T (2010) Dremel: Interactive Analysis of Web-Scale Datasets. 36th Int Conf Very Large Data Bases 330–339

71. Vohra D (2016) Pratical Hadoop Ecosystem. doi: https://doi.org/10.1007/978-1-4842-2199-0

72. Champlin R (2018) Selection Methods of Genetic Algorithms. Bourbonnais, Illinois

73. Genetic Algorithms - Parent Selection. In: Tutorialspoint.com. https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm. Accessed 14 May 2018

74. Genetic Algorithms - Crossover. In: Tutorialspoint.com. https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover.htm. Accessed 20 Jun 2018

75. Magalhães-Mendes J (2013) A Comparative Study of Crossover Operators for Genetic Algorithms to Solve the Job Shop Scheduling Problem. WSEAS Trans Comput 12:164–173

76. Kramer O (2017) Genetic Algorithm Essentials, 1st ed. doi: 10.1007/978-3-319-52156-5

77. André Nicolau Fernandes (2016) An Evolutionary Computing Approach to Financial Portfolio Management Based on Growth Stocks & Sector / Industry Distribution. Instituto Superior Técnico

A. Appendix A

Instructions on how to use the provided VMs: one for the CDH machine and another for the Driver machine, to use the system. Due to the use of CDH cluster configuration, access to the application will be done via an Integrated Development Environment, namely, Eclipse. Both machines have the password *cloudera* on their user accounts.

1. Launch both VMs in two separate machines on the same local network;
2. Verify that the CDH machine is running on local IP 192.168.1.244, since the Driver machine is configured to communicate on that IP. This can be done by launching a command line prompt and writing *ip addr*;
3. Verify that the Driver machine is running on the same local network via command line *ifconfig*;
4. In the Driver machine, go to */home/osiris_wrath/Code/eclipse* and launch the *eclipse* executable;
5. The suggested way to run the application is by going to the UserModule object, under the portfolio project, and using its main function to make calls to UserModule methods;
6. To run the application, simply use the Run Configuration called UserModule and observe the Console tab for results.

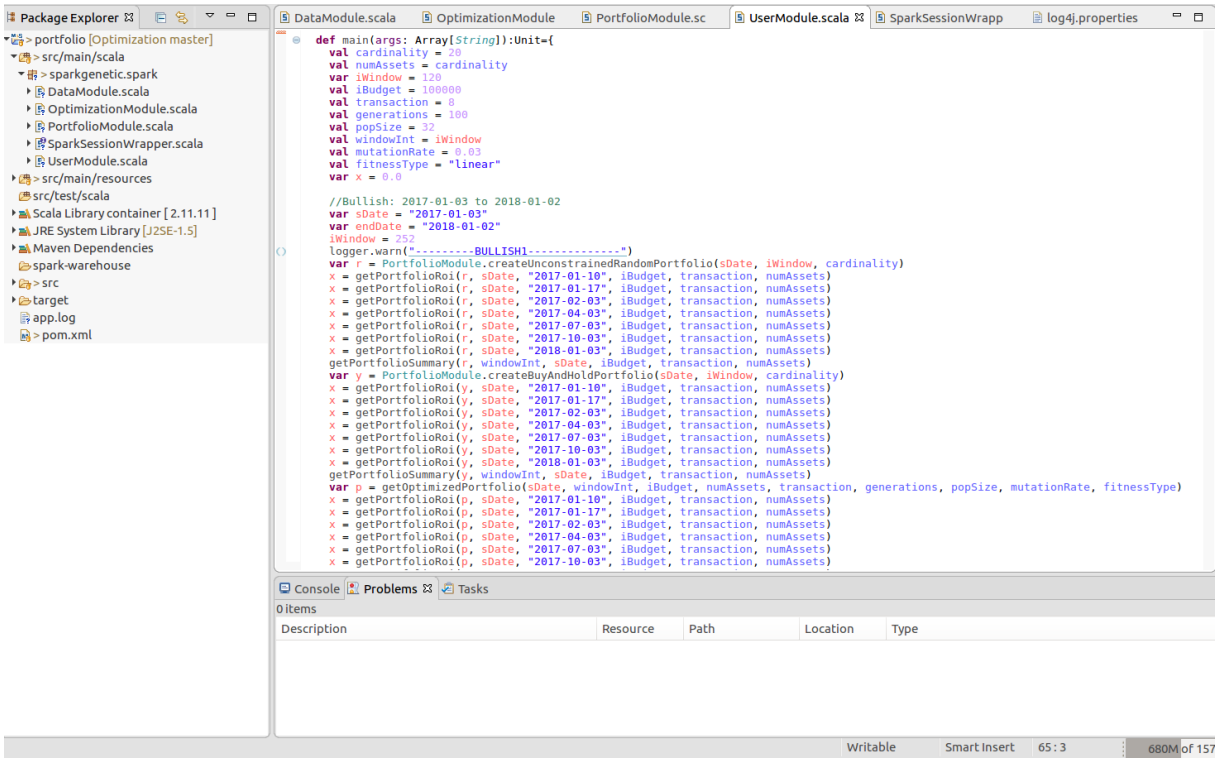

Figure A.1: Example of Eclipse's layout and of the main function used to run the program.

B. Appendix B

The complete project dependencies, via a Maven pom.xml file. Additionally, code for the main Scala modules will be shown: Data Module, Optimization Module and Portfolio Module.

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ist.spark</groupId>
  <artifactId>portfolio</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>PortfolioOptimization</name>
  <description></description>
  <inceptionYear>2018</inceptionYear>
  <!-- Example of a non-restrictive license (MIT)
   <licenses>
    <license>
      <name>MIT License</name>
      <url>http://www.opensource.org/licenses/mit-license.php</url>
      <distribution>repo</distribution>
    </license>
  </licenses>
   -->

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>1.8</java.version>
    <scala_2.11-base.version>2.11</scala_2.11-base.version>
    <scala_2.11.version>2.11.11</scala_2.11.version>
    <spark.version>2.3.0.cloudera2</spark.version>
    <scala-maven-plugin.version>3.4.2</scala-maven-plugin.version>
    <scala.tools.version>2.11</scala.tools.version>
  </properties>

  <repositories>
      <repository>
          <id>central</id>
          <name>Central Repository</name>
          <url>https://repo.maven.apache.org/maven2</url>
          <snapshots>
              <enabled>false</enabled>
          </snapshots>
      </repository>
      <repository>
          <id>cloudera</id>
          <url>https://repository.cloudera.com/content/groups/cloudera-repos</url>
          <snapshots>
              <enabled>false</enabled>
          </snapshots>
      </repository>
      <repository>
              <id>Scalaz Bintray Repo</id>
              <url>http://dl.bintray.com/scalaz/releases/</url>
      </repository>
  </repositories>

  <dependencies>
              <dependency>
                <groupId>org.scala-lang</groupId>
                <artifactId>scala-library</artifactId>
                <version>${scala_2.11.version}</version>
                <scope>provided</scope>
              </dependency>
              <dependency>
                <groupId>org.apache.spark</groupId>
                <artifactId>spark-core_2.11</artifactId>
                <version>${spark.version}</version>
                <scope>provided</scope>
```

```
            </dependency>
            <dependency>
              <groupId>org.apache.spark</groupId>
              <artifactId>spark-mllib_2.11</artifactId>
              <version>${spark.version}</version>
              <scope>provided</scope>
            </dependency>
            <dependency>
              <groupId>org.apache.spark</groupId>
              <artifactId>spark-sql_2.11</artifactId>
              <version>${spark.version}</version>
              <scope>provided</scope>
            </dependency>
            <dependency>
              <groupId>org.apache.spark</groupId>
              <artifactId>spark-yarn_2.11</artifactId>
              <version>${spark.version}</version>
              <scope>provided</scope>
            </dependency>
            <dependency>
          <groupId>org.scalaz.stream</groupId>
          <artifactId>scalaz-stream_${scala.tools.version}</artifactId>
          <version>snapshot-0.7a</version>
          <scope>compile</scope>
            </dependency>
            <dependency>
          <groupId>org.clapper</groupId>
          <artifactId>grizzled-slf4j_${scala.tools.version}</artifactId>
          <version>1.3.2</version>
            </dependency>
            <dependency>
          <groupId>org.slf4j</groupId>
          <artifactId>slf4j-api</artifactId>
          <version>1.7.25</version>
            </dependency>

</dependencies>

<pluginRepositories>
          <pluginRepository>
            <id>scala-tools.org</id>
            <name>Scala-tools Maven2 Repository</name>
            <url>http://scala-tools.org/repo-releases</url>
          </pluginRepository>
</pluginRepositories>

<build>
  <sourceDirectory>src/main/scala</sourceDirectory>
  <testSourceDirectory>src/test/scala</testSourceDirectory>

  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <excludes>
        <exclude>**/*.java</exclude>
        <exclude>**/*.scala</exclude>
      </excludes>
    </resource>
  </resources>

          <plugins>

    <plugin>
      <groupId>net.alchim31.maven</groupId>
      <artifactId>scala-maven-plugin</artifactId>
      <version>3.2.0</version>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
            <goal>testCompile</goal>
          </goals>
          <configuration>
            <args>
              <arg>-dependencyfile</arg>
              <arg>${project.build.directory}/.scala_dependencies</arg>
```

```
            </args>
          </configuration>
        </execution>
      </executions>
    </plugin>

    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.18.1</version>
      <configuration>
        <useFile>false</useFile>
        <disableXmlReport>true</disableXmlReport>
        <!-- If you have classpath issue like NoDefClassError,... -->
        <!-- useManifestOnlyJar>false</useManifestOnlyJar -->
        <includes>
          <include>**/*Test.*</include>
          <include>**/*Suite.*</include>
        </includes>
      </configuration>
    </plugin>

    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>2.3</version>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>shade</goal>
          </goals>
        </execution>
      </executions>
    </plugin>

          </plugins>
  </build>
</project>
```

DataModule.scala

```
package sparkgenetic.spark

import grizzled.slf4j.Logging
import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.Dataset
import org.apache.spark.sql.Column
import org.apache.spark.sql.expressions._
import org.apache.spark.sql.types.StructType
import scala.collection.mutable._
import org.apache.spark.sql.functions._
import java.sql.Date
import java.text.SimpleDateFormat
import java.util.Calendar
import java.io.File
import org.apache.commons.io.FileUtils

case class Stock(date:Date, open:Double, high:Double, low:Double,
                        close:Double, adjClose:Double, volume:Int, symbol:String, index:String, currency:String)

case class Currency(date:Date, open:Double, high:Double, low:Double,
                        close:Double, adjClose:Double, volume:Int, from:String, to:String)


object DataModule extends SparkSessionWrapper with Logging{
        import spark.implicits._
        import org.apache.spark.sql.catalyst.ScalaReflection

        val stockSchema = ScalaReflection.schemaFor[Stock].dataType.asInstanceOf[StructType]
        val currencySchema = ScalaReflection.schemaFor[Currency].dataType.asInstanceOf[StructType]
        //Define a list of windowIntervals
        val windowsList = List(5, 10, 20, 60, 120, 252)

        lazy val symbolsList = {
                getDistinctSymbols(stocksDf)
        }
        val symbolsSize = symbolsList.size

        lazy val currencyDf: DataFrame = {
                var df = loadCurrenciesDf(None)
                        //.cache()
                        .toDF
                //logger.warn("Dropping excess columns and switching to and from")
                df = df.drop("open").drop("close")
                        .drop("high").drop("low")
                        .drop("volume")
                        //.cache()
                //logger.warn("Adding day, month and year columns")
                df =  df.withColumn("year", year(col("date")))
                        .withColumn("month", month(col("date")))
                        .withColumn("day", dayofmonth(col("date")))
                        //.cache()
                //logger.warn("Sorting by date")
                df = df.sort(asc("date")).cache()
                df = setCheckpointDf(df, true)
                df
        }

        //Lazy initiate the Stocks DataFrame
        lazy val stocksDf = {
                try{
                        var df = loadParquetToDf("assets")
                                .cache
                        df
                }catch{
                        case _: Throwable =>
                                logger.warn("Load assets from CSV files")
                                var df = loadAssetsDf(None)
                                        //.cache()
                                logger.warn(df.show)
                                logger.warn("Save assets DF to parquet file")
                                saveDfToParquet(df, "temp")
                                df = loadParquetToDf("temp")
```

```
                .cache


            logger.warn("Replace symbol x.y to x_y - SPARK compatibility - . reserved for structs")
            df = df.withColumn("ticker", translate(col("symbol"), ".", "_"))
                .drop("symbol")
                .withColumnRenamed("ticker","symbol")
                //.cache()

            logger.warn("Dropping excess columns")
            df = df.drop("open").drop("close").drop("high").drop("low")//.cache()
            //remaining attributes: date adjClose volume symbol currency
            logger.warn("Sorting by symbol & date")
            df = df.sort(asc("symbol"), asc("date")).cache()

            logger.warn("Adding day, month and year columns")
            df =  df.withColumn("year", year(col("date")))
                .withColumn("month", month(col("date")))
                .withColumn("day", dayofmonth(col("date")))

            logger.warn("Adding adjCloseLag_x and ror_x columns")
            val w: WindowSpec = Window.partitionBy("symbol")
                .orderBy(col("date"))
            for(i <- windowsList){
                logger.warn(s"... where x = $i")
                df = df.withColumn(s"adjCloseLag_$i", lag($"adjClose", i).over(w))
                    .withColumn(s"ror_$i", ($"adjClose" - col(s"adjCloseLag_$i")) /
col(s"adjCloseLag_$i") * 100 )
                    .drop(s"adjCloseLag_$i")
            }

            logger.warn("Adding sampleStddev_x wrt ror_x columns")
            for(i <- windowsList){
                logger.warn(s"... where x = $i")
                var tempW: WindowSpec = Window.partitionBy("symbol")
                    .orderBy(col("date"))
                    .rowsBetween(-i, 0)
                df = df.withColumn( s"sampleStddev_$i", stddev(col(s"ror_$i")).over(tempW)
)
            }

            logger.warn("Adding mean_x wrt ror_x columns")
            for(i <- windowsList){
                logger.warn(s"... where x = $i")
                var tempW: WindowSpec = Window.partitionBy("symbol")
                    .orderBy(col("date"))
                    .rowsBetween(-i, 0)
                df = df.withColumn( s"mean_$i", mean(col(s"ror_$i")).over(tempW) )
            }

            logger.warn("RDD partitions")
            logger.warn(df.rdd.partitions.size)
            logger.warn("Saving ETL result to Parquet file")
            saveDfToParquet(df, "assets")
            df
        }
    }

    //FUNCTIONS TO SAVE AND LOAD DFs TO PARQUET

    def saveDfToParquet(df: DataFrame, fileName: String): Unit ={
        logger.warn("Saving DF to "+fileName+" Parquet file")
        val hdfsHost: String = "quickstart.cloudera"
        df.write
            .format("parquet")
            .save("hdfs://"+hdfsHost+":8020/user/cloudera/"+fileName+".parquet")
            //.save("file:///home/osiris_wrath/Code/hdfs/import_data/"+fileName+".parquet")
    }

    def loadParquetToDf(fileName: String): DataFrame ={
        logger.warn("Loading "+fileName+" to DF")
        val hdfsHost: String = "quickstart.cloudera"
        var df = spark.read
            .format("parquet")
            .option("header", "true") //reading the headers
            .option("mode", "DROPMALFORMED")
```

```
                .option("delimiter",";")
                .option("dateFormat", "yyyy-MM-dd")
                .load("hdfs://"+hdfsHost+":8020/user/cloudera/"+fileName+".parquet")
                //.load("file:///home/osiris_wrath/Code/hdfs/import_data/"+fileName+".parquet/*")
                df
    }

    //FUNCTIONS TO LOAD FROM CSVs

    def loadAssetsDf(hdfsPath:Option[String]): DataFrame = {
        logger.warn("Loading stocks CSV files to DataFrame")
        val hdfsHost: String = "quickstart.cloudera"

        hdfsPath match{
            case None =>
                val df: DataFrame = spark.read
                        .format("csv")
                .option("header", "true") //reading the headers
                .option("mode", "DROPMALFORMED")
                .option("delimiter",";")
                .option("dateFormat", "yyyy-MM-dd")
                .schema(stockSchema)
                .load("hdfs://"+hdfsHost+":8020/user/cloudera/data/*.csv")
                //.load("file:///home/osiris_wrath/Code/hdfs/import_data/data/*.csv")
                return df
            case Some(x) =>
                val df: DataFrame = spark.read
                        .format("csv")
                .option("header", "true") //reading the headers
                .option("mode", "DROPMALFORMED")
                .option("delimiter",";")
                .option("dateFormat", "yyyy-MM-dd")
                .schema(stockSchema)
                .load(hdfsPath.get)
            return df
        }
    }

        def loadCurrenciesDf(hdfsPath:Option[String]): DataFrame = {
        logger.warn("Loading currency CSV files to DataFrame")
        val hdfsHost: String = "quickstart.cloudera"

        hdfsPath match{
            case None =>
                val df: DataFrame = spark.read
                        .format("csv")
                .option("header", "true") //reading the headers
                .option("mode", "DROPMALFORMED")
                .option("delimiter",";")
                .option("dateFormat", "yyyy-MM-dd")
                .schema(currencySchema)
                .load("hdfs://"+hdfsHost+":8020/user/cloudera/data/currency/*.csv")
                //.load("file:///home/osiris_wrath/Code/hdfs/import_data/data/currency/*.csv")
                return df
            case Some(x) =>
                val df: DataFrame = spark.read
                        .format("csv")
                .option("header", "true") //reading the headers
                .option("mode", "DROPMALFORMED")
                .option("delimiter",";")
                .option("dateFormat", "yyyy-MM-dd")
                .schema(currencySchema)
                .load(hdfsPath.get)
            return df
        }
    }

    //AUX FUNCTIONS

    def changeNumberRange(number:Double, oldMin:Double, oldMax:Double, newMin:Double,
newMax:Double):Double={
        val oldRange = oldMax-oldMin
        if (oldRange == 0){
            return newMin
        }
        else{
```

```
                        val newRange = newMax-newMin
                        val newValue = ( ((number-oldMin)*newRange) / oldRange ) + newMin
                        return newValue
                }
        }

        def daysToSecs(i: Int): Int={
                //Hive timestamp is interpreted as UNIX timestamp in seconds
                val x: Int = i * 86400
                return x
        }

        /*
         * Obtain the date after x days
         */
        def addDays(date:java.sql.Date , days:Int):java.sql.Date={
                //logger.warn("addDays")
                val cal:Calendar = Calendar.getInstance
                val utilDate = dateSqlToUtil(date)
  cal.setTime(date)
  cal.add(Calendar.DATE, days) //minus number would decrement the days
  val sqlDate = dateUtilToSql(cal.getTime)
  return sqlDate
 }

        /*
         * Obtain the date before x days
         */
        def subtractDays(date:java.sql.Date , days:Int):java.sql.Date={
                val cal:Calendar = Calendar.getInstance
                val utilDate = dateSqlToUtil(date)
  cal.setTime(date)
  val negativeDays = -days
  cal.add(Calendar.DATE, negativeDays) //minus number would decrement the days
  val sqlDate = dateUtilToSql(cal.getTime)
  return sqlDate
 }

        def getDistinctSymbols(df: DataFrame): List[String]={
                //Da erro em modo YARN
                logger.warn("Get distinct symbols")
                val distinctSymbols = df.select("symbol")
                        .distinct
                        .map( r => r.getString(0) )
                        .collect
                        .toList
                        .sorted
                return distinctSymbols
        }

        def setCheckpointDf(df: DataFrame, b: Boolean): DataFrame={
                //logger.warn("Checkpoint")
                //spark.sparkContext.setCheckpointDir("/tmp")
                spark.sparkContext.setCheckpointDir("file:///home/osiris_wrath/Code/hdfs/checkpoint")
                //logger.warn(df.explain)
                var d = df.checkpoint(b)
                //logger.warn("Checkpoint ended")
                return d
        }

        def normalizeWeightsDf(df: DataFrame): DataFrame={
                //val rowList = sampleDf.collect().toList.map(e => e.toString())
                val soma = df.agg(sum("weight"))
                        .map( r => r(0).asInstanceOf[Double] )
                        .collect
                        .toList
                val normalizacao = col("temp")/lit(soma(0))
                var d = df.withColumnRenamed("weight", "temp")
                        .withColumn("weight", normalizacao)
                        .drop("temp")
                        //.cache
                        .toDF
                return d

        }
}
```

OptimizationModule.scala

```
package sparkgenetic.spark

import grizzled.slf4j.Logging
import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.expressions._
import java.util.concurrent.atomic.AtomicLong
import scala.util.Random
import org.apache.spark.sql.types.StructType
import scala.collection.mutable._
import org.apache.spark.sql.functions._
import java.sql.Date
import java.util.Calendar
import java.text.SimpleDateFormat
import scala.collection.mutable.ListBuffer
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.linalg.Vectors
import org.apache.spark.sql.Row
import org.apache.spark.mllib.linalg._
import org.apache.spark.mllib.stat.Statistics
import org.apache.spark.rdd.RDD
import java.nio.file.{Paths, Files}
import java.util.concurrent._


case class pairRow(i: Double, j: Double, corr: Double)

object OptimizationModule extends SparkSessionWrapper with Logging{
	import spark.implicits._
	val stocksDf = DataModule.stocksDf.withColumn("weight", lit(0.0))
		.cache
	val symbolsList = DataModule.symbolsList
	var numAssets = 20
	val floor = 0.01
	val ceiling = 0.25

	import org.apache.spark.sql.catalyst.ScalaReflection

	// EVALUATION (FITNESS) FUNCTIONS
	def evaluatePortfolioReturn(portfolioDf:DataFrame, windowInt:Int, dDate:Date):Double={
		//logger.warn("evaluatePortfolioReturn")
		val ret = PortfolioModule.getPortfolioReturn(portfolioDf, windowInt)
		//Max Return and Min Variance
		val fitness = ret
		//logger.warn(fitness)
		fitness
	}

	def evaluatePortfolioLinearCombination(portfolioDf:DataFrame, windowInt:Int, dDate:Date, returnWeight:Double,
riskWeight:Double):Double={
		//logger.warn("evaluatePortfolioLinearCombination " + returnWeight + " , " + riskWeight)
		val ret = PortfolioModule.getPortfolioReturn(portfolioDf, windowInt)
		val risk = PortfolioModule.getPortfolioVariance(portfolioDf, windowInt, dDate, numAssets)
		//Max Return and Min Variance
		val fitness = ret*returnWeight - (risk*riskWeight)
		//logger.warn(fitness)
		fitness
	}

	def initRandomPop(dDate:Date, windowInt:Int, popSize:Int):LinkedHashMap[DataFrame,Double]={
		var mapBuf = scala.collection.mutable.LinkedHashMap.empty[DataFrame, Double]
		//Random Initialization of 1st generation
		(0 until popSize).par.foreach{
			x =>
				//logger.warn("Random Individual "+ x)
				val p = createRandomPortfolio(dDate.toString, windowInt)
				mapBuf = this.synchronized{
					mapBuf += (p -> 0.0)
					mapBuf
				}
		}
		return mapBuf
	}
```

```
	def checkpointPop(map:LinkedHashMap[DataFrame,Double]):LinkedHashMap[DataFrame,Double]={
		var mapBuf = scala.collection.mutable.LinkedHashMap.empty[DataFrame, Double]
		var newMapBuf = scala.collection.mutable.LinkedHashMap.empty[DataFrame, Double]
		//logger.warn("checkpointing generation to cut RDD lineage")
		//logger.warn(mapBuf.size)
		mapBuf = map
		mapBuf.par.foreach{
			case(k:DataFrame,v:Double) =>
				//logger.warn(k.explain)
				val newK = DataModule.setCheckpointDf(k, true)
				//logger.warn(newK.explain)
				newMapBuf = this.synchronized{
					newMapBuf += (newK -> v)
					newMapBuf
				}
				//logger.warn(k.show)
		}
		mapBuf = newMapBuf
		//sort population low to high by value
		mapBuf = scala.collection.mutable.LinkedHashMap(mapBuf.toSeq.sortWith(_._2 > _._2):_*)
		mapBuf
	}

	def evaluatePop(fitnessType:String, windowInt:Int, dDate:Date,
map:LinkedHashMap[DataFrame,Double]):LinkedHashMap[DataFrame,Double]={
		var mapBuf = map
		fitnessType match{
			case "return" =>
				//logger.warn("evaluate return fitness begin")
				mapBuf.par.foreach{
					case(k:DataFrame,v:Double) =>
						val fitness = evaluatePortfolioReturn/*LinearCombination*/(k,
windowInt, dDate/*, 0.5, 0.5*/)
						mapBuf = this.synchronized{
							mapBuf.update(k, fitness)
							mapBuf
						}
						logger.warn(k + "  " + mapBuf.get(k))
						//logger.warn(k.show)
				}
				//sort population low to high by value
				mapBuf = scala.collection.mutable.LinkedHashMap(mapBuf.toSeq.sortWith(_._2 >
_._2):_*)
				mapBuf
			case "linear" =>
				logger.warn("evaluate linear fitness begin")
				mapBuf.par.foreach{
					case(k:DataFrame,v:Double) =>
						val fitness = evaluatePortfolioLinearCombination(k, windowInt,
dDate, 0.75, 0.25)
						mapBuf = this.synchronized{
							mapBuf.update(k, fitness)
							mapBuf
						}
				}
				//sort population low to high by value
				mapBuf = scala.collection.mutable.LinkedHashMap(mapBuf.toSeq.sortWith(_._2 >
_._2):_*)
				mapBuf
			case _ =>
				map
		}
	}

	def selectionParents(popSize:Int, map:LinkedHashMap[DataFrame,Double],
threshold:Double):LinkedHashMap[DataFrame,Double]={
		//logger.warn("Truncation Parent Selection using threshold")
		//parent truncation selection using threshold idx
		val selection:Int = popSize - (popSize*threshold).toInt
		//logger.warn("Selection "+selection)
		var mapBufKeep = map.dropRight(selection)
		var mapBufDrop = map.drop(selection)
		//logger.warn(mapBufKeep.size)
		//logger.warn(mapBufDrop.size)
		mapBufKeep
```

```
	}

	def selectionCrossover(popSize:Int, map:LinkedHashMap[DataFrame,Double], threshold:Double,
fitnessType:String):List[DataFrame]={
		//logger.warn("Roulette Wheel Crossover Selection")
		//parent truncation selection using threshold idx
		val selection:Int = popSize - (popSize*threshold).toInt
		val parentsPair = scala.collection.mutable.ListBuffer.empty[DataFrame]
		var r = scala.util.Random
		var values = ListBuffer.empty[Double]
		var mapBufKeep = map
		if(fitnessType == "linear"){
			var h = 0
			mapBufKeep.values.par.foreach{
				x =>
					values = this.synchronized{
						values +=
DataModule.changeNumberRange(mapBufKeep.values.toList(h), -10000, 10000, 1, 10000)
						h = h+1
						values
					}
			}
		}
		//Sum of all chromosomes fitness
		var valuesSum = map.values.sum
		if(fitnessType == "linear"){
			valuesSum = values.sum
		}
		//get parent pair for crossover
		var pA:DataFrame = spark.emptyDataFrame
		var pB:DataFrame = spark.emptyDataFrame

		//choose parent A
		var d = r.nextDouble()
		//change random double range from [0.0,1.0] to [0,sum of all fitness values]
		var crossoverPoint = DataModule.changeNumberRange(d, 0.0, 1.0, 0.0, valuesSum)
		//logger.warn("crossover point "+ crossoverPoint)
		var cumulative = 0.0
		var found = false
		(0 until mapBufKeep.values.size).foreach{
			x =>
				//logger.warn("cumulative point "+ cumulative)
				if(fitnessType == "linear")
					cumulative = cumulative + values.toList(x)
				else
					cumulative = cumulative + mapBufKeep.values.toList(x)

				if(cumulative >= crossoverPoint && found==false){
					logger.warn(x)
					found = true
					//logger.warn("found cumulative point "+ cumulative)
					pA = mapBufKeep.keys.toList(x)
				}
		}

		//choose parent B
		var same = false
		d = r.nextDouble()
		crossoverPoint = DataModule.changeNumberRange(d, 0.0, 1.0, 0.0, valuesSum)
		cumulative = 0.0
		found = false
		logger.warn("Double "+crossoverPoint)
		(0 until mapBufKeep.values.size).foreach{
			x =>
				if(fitnessType == "linear")
					cumulative = cumulative + values.toList(x)
				else
					cumulative = cumulative + mapBufKeep.values.toList(x)
				if(cumulative >= crossoverPoint && found==false){
					logger.warn(x)
					found = true
					//logger.warn("found cumulative point "+ cumulative)
					pB = mapBufKeep.keys.toList(x)
				}
		}
```

```
		if(pA == pB ){
			same = true
		}
		//If the same chromosome was chosen, repeat selection roulette selection until a new parent B is found
		while(same == true){
			//logger.warn("same "+same)
			d = r.nextDouble()
			crossoverPoint = DataModule.changeNumberRange(d, 0.0, 1.0, 0.0, valuesSum)
			cumulative = 0.0
			found = false
			(0 until mapBufKeep.values.size).foreach{
				x =>
					if(fitnessType == "linear")
					cumulative = cumulative + values.toList(x)
				else
					cumulative = cumulative + mapBufKeep.values.toList(x)
					if(cumulative >= crossoverPoint && found==false){
						found = true
						//logger.warn("found cumulative point "+ cumulative)
						pB = mapBufKeep.keys.toList(x)
					}
			}
			if(pA != pB){
				same = false
			}
		}
		//Return pair list
		parentsPair += pA
		parentsPair += pB
		parentsPair.toList
}

def operatorCrossover(numAssets:Int, parentsPair:List[DataFrame]):List[DataFrame]={
		//logger.warn("One Point Crossover Operator")
		//do crossover and get offspring pair
		val childrenPair = scala.collection.mutable.ListBuffer.empty[DataFrame]
		val pA = parentsPair(0)
		val pB = parentsPair(1)
		var r = scala.util.Random
		var p1:DataFrame = spark.emptyDataFrame
		var p2:DataFrame = spark.emptyDataFrame
		var duplicate = true
		//do loop until children don't have duplicate symbols
		while(duplicate == true){
			//Choose the partition point for One Point Crossover
			var rand = r.nextInt(numAssets+1)
			if(rand == 0)
			rand = 1
			//Get chromosomes heads and tails
			val pAHead = pA.sort(desc("symbol"))
				.limit(rand)
			val pATail = pA.sort(asc("symbol"))
				.limit(numAssets-rand)
			val pBHead = pB.sort(desc("symbol"))
				.limit(rand)
			val pBTail = pB.sort(asc("symbol"))
				.limit(numAssets-rand)
			//Switch heads and tails and re-normalize weights
			p1 = pAHead.union(pBTail)
			p2 = pBHead.union(pATail)

			val dup1 = p1.select("symbol")
				.distinct
				.count
			val dup2 = p2.select("symbol")
				.distinct
				.count
			if(dup1 != numAssets || dup2 != numAssets){
				duplicate = true
				}else{
					duplicate = false
				}
			}
			p1 = DataModule.normalizeWeightsDf(p1)
			p2 = DataModule.normalizeWeightsDf(p2)
			childrenPair += p1
```

```
                childrenPair += p2
                childrenPair.toList
        }

        def operatorMutation(portfolio:DataFrame, mutationRate:Double, sDate:String, windowInt:Int):DataFrame={
                //logger.warn("Mutate Individual by getting new weights")
                val t0 = System.nanoTime
                var r = scala.util.Random
                var p = portfolio
                var p3:DataFrame = spark.emptyDataFrame
                val m = r.nextInt(100).toDouble / 100.0
                if(m < mutationRate){
                        //decide which gene to mutate
                        val g = r.nextInt(numAssets)
                        val pList = p.sort(desc("weight"))
                                .limit(numAssets)
                                //.sort(asc("symbol"))
                                .select(col("symbol"))
                                .map( r => r.getString(0) )
                                .collect
                                .toList
                        var symb = pList(g)
                        var allowed = false
                        var p2:DataFrame = spark.emptyDataFrame
                        var aW = p.filter(col("symbol")===symb)
                                .select("weight")
                                .map( r => r(0).asInstanceOf[Double] )
                                .collect
                        var w = aW(0)
                        val limit = 20
                        var currentLimit = 0
                        while(allowed == false){
                                val t1 = System.nanoTime
                                val deltaT = t1 -t0
                                val mutationRuntime = TimeUnit.NANOSECONDS.toSeconds(deltaT).toDouble
                                if(mutationRuntime >= 15){
                                        p2 = createRandomPortfolio(sDate, windowInt)
                                        allowed = isPortfolioWeightValid(p2)
                                        logger.warn("mutation "+mutationRuntime + "  " + allowed)
                                }
                                currentLimit = currentLimit+1
                                if(currentLimit == limit){
                                        logger.warn("currentLimit")
                                        val g = r.nextInt(numAssets)
                                        symb = pList(g)
                                        //logger.warn(symb)
                                        aW = p.filter(col("symbol")===symb).select("weight")
                                                .map( r => r(0).asInstanceOf[Double] ).collect
                                        w = aW(0)
                                        currentLimit = 0
                                }
                                //mutate that gene - generate new weight
                                var xy = r.nextInt((ceiling*1000).toInt).toDouble
                                if(xy == 0.0)
                                        xy = 1.0
                                val newW:Double = xy/1000.0
                                //update on portfolio
                                val p1 = p.withColumn("weight", when(col("weight").equalTo(w),
newW).otherwise(col("weight")))
                                //normalize portfolio
                                p2 = DataModule.normalizeWeightsDf(p1)
                                allowed = isPortfolioWeightValid(p2)
                        }
                        //val p2 = p1
                        p3 = p2.sort(desc("weight"))
                        //logger.warn(p3.show)
                }else{
                        p3 = p
                }
                p3
        }

        // ALGORITHM

        def geneticAlgorithm(generations:Int, popSize:Int, mutationRate:Double, windowInt:Int, dDate:Date, cardinality:Int,
fitnessType:String):DataFrame={
```

```
logger.warn("geneticAlgorithm")

//statistics
val fitnessHistory = scala.collection.mutable.ListBuffer.empty[Double]
val iterationRuntimeHistory = scala.collection.mutable.ListBuffer.empty[Double]
var genRuntime:Double = 0.0
val gt0 = System.nanoTime()

numAssets = cardinality
val listBuf = scala.collection.mutable.ListBuffer.empty[DataFrame]
var mapBuf = scala.collection.mutable.LinkedHashMap.empty[DataFrame, Double]
var newMapBuf = scala.collection.mutable.LinkedHashMap.empty[DataFrame, Double]

//Population Model: Generational (pop changes every iteration)

(0 until generations).foreach{
        g =>
                //count time for each generation
                val t0 = System.nanoTime
                if(g == 0){
                        //Random Initialization of 1st generation
                        mapBuf = initRandomPop(dDate, windowInt, popSize)
                        //logger.warn("1st gen randomly init")
                        //End 1st generation random init
                }
                logger.warn("-----GENERATION "+g+"-----")

                //Checkpoint periodically to cut RDD lineage and avoid OutOfMemory exceptions
                if((g+1)%2 == 0){
                        mapBuf = checkpointPop(mapBuf)
                }
                //End of periodic checkpointing

                //Evaluate individuals in population using fitness function
                evaluatePop(fitnessType, windowInt, dDate, mapBuf)
                //End evaluation

                //Parent set selection
                val threshold = 0.5
                //index of the threshold selection point
                val selection:Int = popSize - (popSize*threshold).toInt
                val parents = selectionParents(popSize, mapBuf, threshold)
                var mapBufKeep = parents
                //End parent set selection

                //Use elitism for next Gen: keep best portfolio for next generation
                val maxFit = mapBuf.maxBy{ item => item._2 }
                //logger.warn("Best fitness in generation "+maxFit)
                fitnessHistory += maxFit._2
                val elitistP = maxFit._1
                //logger.warn("elitist "+maxFit._2)
                //logger.warn(maxFit._1.show)
                newMapBuf += (elitistP -> 0.0)
                //logger.warn("Elitism done")
                //End elitism

                //Need to generate popSize-1 new individuals

                //Do crossover selection with parents map
                //Roulette Wheel Selection for crossover
                var r = scala.util.Random
                //z: number of crossover iterations -> each iteration generates 2 offspring
                val z = ((selection-1)/2).toInt
                logger.warn("number of crossovers "+z+" : 1 crossover = 2 new individuals")
                for(i <- 0 to z){
                        logger.warn(i)
                        //do crossover selection and get parent pair
                        val parentsPair = selectionCrossover(popSize, parents, threshold,
fitnessType)
                        //do crossover and get offspring pair
                        val childrenPair = operatorCrossover(numAssets, parentsPair)
                        listBuf += childrenPair(0)
                        listBuf += childrenPair(1)
                }
                //logger.warn("roulette selection and crossovers end")
```

```
                                        //AFTER CROSSOVER, mutate the crossover children by using
                                        //weight variations with probability mutationRate
                                        //logger.warn("mutation start")
                                        //for each child chromosome, get final new generation mutated individual
                                        listBuf.par.foreach{
                                                p =>
                                                        val mutatedP = operatorMutation(p, mutationRate,
DataModule.dateToString(dDate), windowInt)
                                                        newMapBuf = this.synchronized{
                                                                newMapBuf += (mutatedP -> 0.0)
                                                                newMapBuf
                                                        }
                                        }
                                        listBuf.clear
                                        //logger.warn("mutation end")

                                        //Fill the remaining new population with random portfolios
                                        val diff = popSize - newMapBuf.size
                                        //logger.warn("Dif: "+diff)
                                        (0 until diff).par.foreach{
                                                x =>
                                                        //logger.warn("Pop iteration "+ x)
                                                        val p = createRandomPortfolio(dDate.toString, windowInt)
                                                        newMapBuf = this.synchronized{
                                                                newMapBuf += (p -> 0.0)
                                                                newMapBuf
                                                        }

                                        }
                                        mapBuf = newMapBuf
                                        logger.warn("newMapBuf size: "+newMapBuf.size)
                                        newMapBuf = scala.collection.mutable.LinkedHashMap.empty[DataFrame, Double]
                                        logger.warn("-----GENERATION END-----")

                                        //statistics
                                        val t1 = System.nanoTime
                                        val deltaT = t1 -t0
                                        val runtimeSec = TimeUnit.NANOSECONDS.toSeconds(deltaT).toDouble
                                        iterationRuntimeHistory += runtimeSec
                        }

                        logger.warn("Evaluate last generation")
                        mapBuf = evaluatePop("return", windowInt, dDate, mapBuf)
                        //sort population low to high by value
                        mapBuf = scala.collection.mutable.LinkedHashMap(mapBuf.toSeq.sortWith(_._2 > _._2):_*)

                        //Print and return optimized portfolio
                        val maxFit = mapBuf.maxBy{ item => item._2 }
                        //logger.warn(maxFit._1.show)
                        //logger.warn(PortfolioModule.getPortfolioReturn(maxFit._1, windowInt))

                        //statistics
                        val gt1 = System.nanoTime
                        val deltaT = gt1 -gt0
                        genRuntime = TimeUnit.NANOSECONDS.toSeconds(deltaT).toDouble

                        //Write optimized portfolio to file
                        logger.warn("Print the best portfolio to a new output file")
                        var y = 0
                        var z = 0
                        var sName = "portfolio"+y
                        var sPath = "file:///home/osiris_wrath/Code/hdfs/runs/"+sName+".csv"
                        var boolean = true
                        while(boolean == true){
                                sName="portfolio"+y
                                sPath = "file:///home/osiris_wrath/Code/hdfs/runs/"+sName+".csv"
                                try{
                                        maxFit._1.write.format("csv").option("header", "true").save(sPath)
                                        z = y
                                        logger.warn(z)
                                        boolean = false
                                }catch{
                                        case _: Throwable =>
                                                boolean = true
                                                y = y+1
                                }
```

```
		}
		//Write Statistics to File
		var listOfLists = scala.collection.mutable.ListBuffer.empty[List[Double]]
		listOfLists += fitnessHistory.toList
		listOfLists += iterationRuntimeHistory.toList
		listOfLists += List.fill(generations)(genRuntime)
		listOfLists.toList.foreach(println)

		sName="fitnessHistory"+y
		sPath = "file:///home/osiris_wrath/Code/hdfs/runs/"+sName+".csv"
		listOfLists(0).toDF.write.format("csv").save(sPath)
		sName="iterationRuntimeHistory"+y
		sPath = "file:///home/osiris_wrath/Code/hdfs/runs/"+sName+".csv"
		listOfLists(1).toDF.write.format("csv").save(sPath)
		sName="gaRuntime"+y
		sPath = "file:///home/osiris_wrath/Code/hdfs/runs/"+sName+".csv"
		listOfLists(2).toDF.write.format("csv").save(sPath)

		var tuples = fitnessHistory.zip(iterationRuntimeHistory)
			//.zip(List.fill(numAssets)(genRuntime))


		return maxFit._1

	}


	// Chromossome: stockDF.schema + weight

	/*
	 * Creates a random portfolio dataframe - representing a chromossome - and normalizes its weights
	 */
	def createRandomPortfolio(sDate: String, windowInt:Int): DataFrame={
		import spark.implicits._
		val string = s"mean_${windowInt}"
		val date = DataModule.stringToDate(sDate)
		val l = scala.collection.mutable.ListBuffer.fill[Double](numAssets)(0.0)
		var r = scala.util.Random
		var b = false
		var c = true
		var sampleDf = stocksDf.filter(col("date") === date)
			.filter(col(string).isNotNull)
		//while floor and ceiling constraints are being violated, apply death penalty and re-create a random
chromosome
		while(b == false){
			c = true
			(0 until numAssets).par.foreach{
				i =>
					val xy = r.nextInt(1000).toDouble
					if(xy == 0.0)
						1.0
					var k:Double = xy/1000.0
					l(i) = k
			}
			val sum = l.sum
			(0 until numAssets).par.foreach{
				i =>
					l(i) = l(i) / sum
					if(l(i) < floor || l(i) > ceiling)
						c = false
			}
			if(l.sum != 1.0)
				c = false
			b = c
		}
		sampleDf = sampleDf.drop("weight")
			.orderBy(rand)
			.limit(numAssets)
		val lSymbols = sampleDf.select("symbol")
			.map( r => r.getString(0) )
			.collect
			.toList
		val lTuples = l.zip(lSymbols)
		val lDf = lTuples.toDF("weight","symbol1")
		sampleDf = sampleDf.join(lDf, sampleDf.col("symbol").equalTo(lDf.col("symbol1")) )
			.drop("symbol1")
```

```
                sampleDf
        }

        //CONSTRAINT FUNCTIONS
        /*
         * Function that checks if the Portfolio weights are valid,
         * according to the problem contraints
         * Contraints:
         * min weight: .010
         * max weight: .250
         */
        def isPortfolioWeightValid(portfolioDf: DataFrame): Boolean={
                //logger.warn("isPortfolioWeightValid")
                val maxWeight = 0.25
                val minWeight = 0.01

                var list = portfolioDf.sort(desc("weight"))
                        .limit(numAssets)
                        .select(col("weight"))
                        .map( r => r.getDouble(0) )
                        .collect
                        .toList

                var b = true
                for(w <- list){
                        if(w > maxWeight || w < minWeight){
                                b = false
                        }
                }
                b
        }

}
```

PortfolioModule.scala

```
package sparkgenetic.spark

import org.apache.spark.SparkEnv
import grizzled.slf4j.Logging
import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.Dataset
import org.apache.spark.sql.expressions._
import scala.util.Random
import org.apache.spark.sql.types.StructType
import scala.collection.mutable._
import org.apache.spark.sql.functions._
import java.sql.Date
import java.util.Calendar
import java.text.SimpleDateFormat
import scala.collection.mutable.ListBuffer
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.linalg.Vectors
import org.apache.spark.sql.Row
import org.apache.spark.mllib.linalg._
import org.apache.spark.mllib.linalg.distributed._
import org.apache.spark.mllib.stat.Statistics
import org.apache.spark.rdd.RDD
import java.nio.file.{Paths, Files}

object PortfolioModule extends SparkSessionWrapper with Logging {
  import spark.implicits._
	val stocksDf = OptimizationModule.stocksDf
		.cache
	val symbolsList = DataModule.symbolsList

	// PORTFOLIOS
	def createUnconstrainedRandomPortfolio(sDate: String, windowInt:Int, numAssets:Int): DataFrame={
		val date = DataModule.stringToDate(sDate)
		val string = s"mean_${windowInt}"
		logger.warn("createUnconstrainedRandomPortfolio")
		var sampleDf = stocksDf
		sampleDf = stocksDf
			.filter(col("date") === date)
			.filter(col(string).isNotNull)
			.orderBy(rand)
			.sample(false, 0.25)
			.limit(numAssets)
			.withColumn("weight", rand)
			.sort(desc("weight"))
		sampleDf = DataModule.normalizeWeightsDf(sampleDf)
		sampleDf = DataModule.setCheckpointDf(sampleDf, true)
		sampleDf
	}


	def createBuyAndHoldPortfolio(sDate: String, windowInt: Int, numAssets:Int): DataFrame={
		logger.warn("createBuyAndHoldPortfolio")
		val date = DataModule.stringToDate(sDate)
		if(!DataModule.windowsList.contains(windowInt)){
			throw new IllegalArgumentException("Window value is not one of the allowed values: "+
DataModule.windowsList)
		}
		val string = s"mean_${windowInt}"
		var sampleDf = stocksDf.filter(col("date") === date)
			.filter(col(string).isNotNull)
			.orderBy(col(string).desc)
			.limit(numAssets)
			.withColumn("weight", lit(1.0/numAssets))
			//.cache
			.toDF
		sampleDf = DataModule.setCheckpointDf(sampleDf, true)
		return sampleDf
	}

	// Returns the portfolio with highest returns @ the specified date in the defined rolling window
	def createOptimizedPortfolio(sDate:String, windowInt:Int, numAssets:Int, generations:Int, popSize:Int,
mutationRate:Double, fitnessType:String):DataFrame={
```

```
		logger.warn("createOptimizedPortfolio")
		val dDate = DataModule.stringToDate(sDate)
		if(!DataModule.windowsList.contains(windowInt)){
			throw new IllegalArgumentException("Window value is not one of the allowed values: "+
DataModule.windowsList)
		}
		var portfolioDf = OptimizationModule.geneticAlgorithm(generations, popSize, mutationRate, windowInt,
dDate, numAssets, fitnessType:String)
		return portfolioDf
	}

	def getPortfolioReturn(portfolio:DataFrame, windowInt:Int): Double={
		//logger.warn("getPortfolioReturn")
		val stringCol = s"mean_${windowInt}"
		var portfolioDf = portfolio.withColumn("weightMeanRor", col(stringCol)*col("weight"))
		val df = portfolioDf.agg(sum("weightMeanRor"))
		val list = portfolioDf.agg(sum("weightMeanRor"))
			.map( r => r(0).asInstanceOf[Double] )
			.collect
			.toList
		list(0)
	}

	/*
	 * Average correlation of assets in a portfolio according to Tierens and Anadu (2004)
	 * Method (A) with full correlation matrix
	 */
	def getPortfolioVariance(portfolio:DataFrame, windowInt:Int, dDate:Date, numAssets:Int): Double={
		//logger.warn("getPortfolioVariance")
		val stringCol = s"mean_${windowInt}"
		var portfolioDf = portfolio
		var sampleDf = stocksDf
		val leftLimitDate = windowInt match{
			//week
			case 5 => DataModule.subtractDays(dDate, 8)
			//2 weeks
			case 10 => DataModule.subtractDays(dDate, 16)
			//month
			case 20 => DataModule.subtractDays(dDate, 31)
			//quarter
			case 60 => DataModule.subtractDays(dDate, 93)
			//half-year
			case 120 => DataModule.subtractDays(dDate, 183)
			//year
			case 252 => DataModule.subtractDays(dDate, 366)
			case _ => throw new IllegalArgumentException("Window value is not one of the allowed values:
"+ DataModule.windowsList)
		}
		val rightLimitDate = windowInt match{
			//week
			case 5 => DataModule.addDays(dDate, 1)
			//2 weeks
			case 10 => DataModule.addDays(dDate, 1)
			//month
			case 20 => DataModule.addDays(dDate, 1)
			//quarter
			case 60 => DataModule.addDays(dDate, 1)
			//half-year
			case 120 => DataModule.addDays(dDate, 1)
			//year
			case 252 => DataModule.addDays(dDate, 1)
			case _ => throw new IllegalArgumentException("Window value is not one of the allowed values:
"+ DataModule.windowsList)
		}
		//PREPARE DATA
		val pSymbolsList = portfolioDf.sort(desc("weight"))
			.limit(numAssets)
			.sort(asc("symbol"))
			.select(col("symbol"))
			.map( r => r.getString(0) )
			.collect
			.toList
		val pWeightsList = portfolioDf.sort(desc("weight"))
			.limit(numAssets)
			.sort(asc("symbol"))
			.select(col("weight"))
```

```
                        .map( r => r.getDouble(0) )
                        .collect
                        .toList

                sampleDf = sampleDf.filter( (col("date") <= rightLimitDate) && (col("date") >= leftLimitDate) )
                var tempDf = sampleDf.union(portfolioDf)
                        .distinct
                        .sort(col("date").asc)
                var pivotedRorSymbolDf = tempDf.filter($"symbol" isin (pSymbolsList:_*))
                        .groupBy("date")
                        .pivot("symbol")
                        .agg(first(s"ror_${windowInt}"))
                        .sort(asc("date"))
                        //.cache
                        .toDF
                val datesDf = pivotedRorSymbolDf.na.drop("any").select("date")
                var df = pivotedRorSymbolDf.na.drop("any").drop("date")
                df = DataModule.setCheckpointDf(df, true)
                val rows = new VectorAssembler().setInputCols(df.columns).setOutputCol("corr_features")
    .transform(df)
    .select("corr_features")
    .rdd
        val items_mllib_vector = rows.map(_.getAs[org.apache.spark.ml.linalg.Vector](0))
        .map(org.apache.spark.mllib.linalg.Vectors.fromML)
        val correlMatrix = new RowMatrix(items_mllib_vector).computeCovariance
        val pairwiseArr = new ListBuffer[Array[Double]]()
        //Get result into local array
                for( i <- 0 to correlMatrix.numRows-1){
                  for(j <- 0 to correlMatrix.numCols-1){
                    pairwiseArr += Array(i, j, correlMatrix.apply(i,j))
                  }
                }
                val pairwiseDf = pairwiseArr.map(x => pairRow(x(0), x(1), x(2))).toDF()
                var outerSumUp = 0.0
                var outerSumDown = 0.0
                var outerSum = 0.0
                (0 to numAssets-1).par.foreach{
                        i =>
                                var innerSum = 0.0
                                for(j <- 0 to numAssets-1){
                                        val cov= pairwiseDf.filter( (col("i") === i) && (col("j") === j) )
                                                .select("corr")
                                                .first
                                                .getDouble(0)
                                        innerSum = innerSum + (cov*pWeightsList(i)*pWeightsList(j))
                                }
                                outerSum = outerSum+innerSum
                }
                outerSum
        }

        def getPortfolioValue(portfolio:DataFrame, sDate:String, iBudget:Int, transaction:Int, numAssets:Int):Double={
                val portfolioShares = getPortfolioWithShares(portfolio, sDate, iBudget)
                val value = getPortfolioSharesValue(portfolioShares, sDate, numAssets)
                val v = value - transaction*numAssets
                return v
        }


        def getPortfolioROI(portfolio:DataFrame, sStartDate:String, sEndDate:String, iBudget:Int, transaction:Int,
numAssets:Int):Double={
                logger.warn("getPortfolioROI")
                val sharePortfolio = PortfolioModule.getPortfolioWithShares(portfolio, sStartDate, iBudget)
                val startValue = PortfolioModule.getPortfolioSharesValue(sharePortfolio, sStartDate, numAssets) -
transaction*numAssets
                //logger.warn(startValue)
                val endValue = PortfolioModule.getPortfolioSharesValue(sharePortfolio, sEndDate, numAssets)
                //logger.warn(endValue)
                val roi = (endValue - startValue)/startValue
                return roi
        }

        /*
         * Returns, from a portfolio with shares column,
         * its value in EUR at sDate
         */
```

```
	def getPortfolioSharesValue(portfolioShares:DataFrame, sDate:String, numAssets:Int):Double={
		val dDate = DataModule.stringToDate(sDate)
		//get conversion rate of EUR to USD in that date
		logger.warn("portfolio value @ date: "+sDate)
		val pSymbolsList = portfolioShares.sort(desc("weight"))
			.limit(numAssets)
			.select(col("symbol"))
			.map( r => r.getString(0) )
			.collect
			.toList
		val pWeightsList = portfolioShares.sort(desc("weight"))
			.limit(numAssets)
			.select(col("weight"))
			.map( r => r.getDouble(0) )
			.collect
			.toList
		val pCurrencyList = portfolioShares.sort(desc("weight"))
			.limit(numAssets)
			.select(col("currency"))
			.map( r => r.getString(0) )
			.collect
			.toList
		var sum = 0.0
		(0 until pSymbolsList.size).foreach{
			i =>
				var ddDate = dDate
				val symbol = pSymbolsList(i)
				val weight = pWeightsList(i)
				val currency = pCurrencyList(i)
				var symbolPrice = 0.0
				var nonExistant = true
				while(nonExistant == true){
					try{
					symbolPrice = stocksDf.filter($"date"===ddDate)
						.filter(col("symbol")===symbol)
						.select(col("adjClose"))
						.first
						.getDouble(0)
					nonExistant = false
					}catch{
						case _ : Throwable =>
							nonExistant = true
							ddDate = DataModule.subtractDays(ddDate, 1)
					}
				}
				val shares = portfolioShares
					.filter(col("symbol")===symbol)
					.select(col("shares"))
					.first
					.getInt(0)
				var xRate = DataModule.currencyDf
					.filter($"date"===DataModule.dateToString(ddDate))
					.select(col("adjClose"))
					.map( r => r.getDouble(0) )
					.collect
					.toList
				val rate = xRate(0)
				if(currency == "EUR"){
					sum = sum + (shares*symbolPrice)
				}else{
					sum = sum + ((symbolPrice*rate)*shares)
				}
		}
		return sum
	}

	def getPortfolioWithShares(portfolio:DataFrame, sDate:String, iBudget:Int):DataFrame={
		val date = DataModule.stringToDate(sDate)
		var portfolioDf = portfolio.withColumn("sharesTemp", (col("weight")*iBudget)/col("adjClose") )
			.withColumn("shares", col("sharesTemp").cast("int"))//.cast("IntegerType"))
			.drop("sharesTemp")
		portfolioDf = DataModule.setCheckpointDf(portfolioDf, true)
		return portfolioDf
	}
}
```