UNIVERSITY OF BIRMINGHAM

EntroPy

Portfolio Optimisation Package using Efficient Frontier and Monte Carlo to support Algorithmic Trading Strategies

Wasim Noordin
ID: 1805449
MSc in Computer Science

Supervisor: Dr Miqing Li
Inspector: Dr Shan He

School of Computer Science
Unviersity of Birmingham

02 October 2023

Abstract

In the domain of wealth management, there exists a significant need for comprehensive tools that can effectively manage, analyse, and optimise financial portfolios. The primary challenge is to provide a free, modular platform that seamlessly translates a profound understanding of fundamental financial theory into a practical implementation. This implementation should offer insights into stock returns, moving averages, and the intricate properties of portfolios. While numerous projects have ventured into this space, many are either behind paywalls, deprecated, or offer only a subset of essential functionalities. Furthermore, the rationale behind their design choices and documentation often remains obscured.

EntroPy addresses these gaps by presenting a versatile toolkit tailored for a wide audience, from novices in quantitative finance to seasoned traders. The system is structured with distinct modules, each dedicated to specific functionalities such as computing the Sharpe and Sortino ratios, assessing volatility, and analysing portfolio returns. Additionally, EntroPy boasts a suite of technical analysis tools, enabling users to delve into simple and exponential moving averages, Bollinger bands, and more. A notable feature is its capability to automatically pinpoint buy and sell signals, aligning with market trends.

For those looking to refine their portfolio's capital allocation, EntroPy offers two potent techniques: Monte Carlo Simulations and Mean Variance Optimisation via the Efficient Frontier. Depending on their risk appetite, users can either aim to maximise the Sharpe ratio, minimise volatility, or seek risk-adjusted efficient returns, all by pursuing a strategy of stable diversification. The toolkit also provides a deep dive into stock-specific metrics, including but not limited to Kurtosis, Skewness, and value at risk. Users can juxtapose their portfolio's performance against prevailing market index returns and, if needed, isolate and visualise metrics for individual stocks.

Engineered using Python and leveraging the power of libraries like Numpy and Pandas, EntroPy is designed for anyone with a foundational grasp of the Python language. The toolkit's architecture and the rationale behind each function is deliberate, enabling even beginners to programming to generate meaningful visuals with ease. To supplement this, the accompanying report is enriched with theoretical insights, shedding light on the design choices and the metrics employed. Lastly, the integrity of EntroPy is upheld by a rigorous unit testing suite, ensuring the precision of all computational functions.

Keywords: Wealth Management, Portfolio Optimisation, Efficient Frontier, Monte Carlo Simulations, Technical Analysis, Python.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my Mother, Sahr, and my Father, Aamir. Their unwavering support, love, and encouragement have been the pillars of strength throughout my journey. It is their wisdom, patience, and sacrifices that have illuminated my path, making it possible for me to pursue my passions and achieve my goals. Every word in this report carries a piece of their teachings and the values they instilled in me. I am eternally grateful for their guidance in inspiring me to be the best version of myself. To them, I dedicate this work, with all my love and respect.

I extend my heartfelt appreciation to my supervisor, Dr. Miqing Li, and my inspector, Dr. Shan He. Their invaluable advice, dedication, and mentorship have been instrumental in shaping this report. Through countless meetings, they have shown immense patience, always challenging me to grow, explore, and push the boundaries of my understanding. Their expertise and guidance have not only enriched my work but have also played a pivotal role in my personal and academic growth.

Table of Contents

List of Figures

List of Tables

1 Overview

1.1 Introduction

Portfolio Optimisation is a rapidly growing subsector of quantitative finance, standing at the forefront of a dynamic intersection with finance, probability, statistics, and computer science. A portfolio consists of combinations of assets from a possible set, forming an asset universe. Portfolio optimisation is the systematic selection of asset allocations in a portfolio to achieve the highest possible expected return for a given level of risk, or conversely, to minimize risk for a specified level of expected return. This method is an example of multi-objective optimisation and refers to the quantitative process derived from the Modern Portfolio Theory (MPT) postulated by Markowitz in 1952 (Behan, 2022).

In today's financial markets, portfolio optimisation is a cornerstone for facilitating successful investment strategies. It plays a pivotal role in the investment management industry, contributing significantly to the 22.7% of globally held assets by the top five asset management firms (Kennedy, 2023). Projections estimate that total Assets under Management (AuM) will increase from $84.9 trillion in 2016 to $145.4 trillion in 2025 (Ifedigbo, 2023). Various entities in the finance industry employ portfolio optimisation, including investment banks (e.g., Goldman Sachs, Citigroup), hedge funds (e.g., Renaissance Technologies, Citadel), asset management firms (e.g., Vanguard, Blackrock), proprietary shops (e.g., Jane Street, Optiver), and high-frequency trading firms (e.g., Jump, HRT) (MordorIntelligence, 2023).

While quantifying the exact influence of portfolio optimisation is speculative, evidence suggests its usage has surged since 1996 (Heller, 2023). The transient nature of proprietary strategies, coupled with technological advancements, has enabled many firms to increase their transaction volume and single-trade efficiency. Combined with algorithmic trading, benefits such as reduced trading costs, improved data visualisation, transparency, and optimised portfolio performance have been realised (Palmstierna, 2019).

Portfolio optimisation is a crucial element of algorithmic trading. This form of trading uses automated systems to execute strategies based on various market indicators, price movements, and other quantitative factors. Before deploying a trading strategy, traders must decide on capital allocation across financial instruments (like stocks, bonds, derivatives) or trading signals. Portfolio optimisation assists in determining the ideal weights for each asset or signal to maximise returns for a set risk level, or vice versa. This allocation considers historical data, expected returns, volatilities, and asset correlations.

Once portfolio weights are set, the trading strategy or algorithm tests against historical market data. This evaluation, known as backtesting, doesn't require any financial investment beyond developmental costs (Chan, 2013). Through backtesting, traders hope that observed historical performance can predict a strategy's future effectiveness. This method offers insights into potential profitability, risks, drawdowns, and other metrics. While beyond this report's scope, it's essential to understand where portfolio optimisation fits in the workflow of an algorithmic trading system that includes backtesting.

1.2 History and Rise of the Portfolio Concept

The origins of portfolio optimisation can be traced back to the early 1950s when Harry Markowitz introduced the Modern Portfolio Theory (MPT) in his seminal paper, "Portfolio Selection," published in 1952. Before it became the groundbreaking work that it is recognized as today, shifting the focus from individual asset analysis to the collective behaviour of assets in a portfolio, this work remained largely unnoticed for over a decade.

Markowitz's theory introduced two key concepts: expected return and variance (or volatility) as measures of portfolio performance. He proposed the Efficient Frontier, a curve that represents the set of optimal portfolios offering the highest expected return for a specified level of risk. This idea was revolutionary, providing a quantitative approach to balance risk and reward, challenging the traditional ad-hoc methods of asset selection. The subsequent rise of portfolio optimisation was further propelled by advancements in computational power and mathematical modelling, which allowed for the efficient processing and scaling of high-throughput data flows (Kolm et al., 2014).

The embrace of portfolio optimisation profoundly impacted the finance industry. Firstly, it spurred the growth of mutual funds and other investment vehicles that harnessed the principles of diversification (Fernando et al., 2003). Asset managers began offering products tailored to specific risk-return profiles, catering to a wider array of investor preferences. MPT empowered investors desiring bespoke risk/reward profiles, with more aggressive individuals willing to accept higher risk than their conservative counterparts (Beattie, 2021). These shifts also laid the foundation for the Capital Asset Pricing Model (CAPM), initially used to determine the appropriate required rate of return for an asset, taking into account the asset's sensitivity to non-diversifiable systematic risk, as well as the expected market return and a risk-free asset rate (Mossin, 1966).

Moreover, the emphasis on risk-adjusted returns led to the creation of numerous new financial metrics, such as the Sharpe Ratio. This metric evaluates the performance of an investment against a risk-free asset, adjusting for its associated risk. The principles of portfolio optimisation also significantly influenced the evolution of algorithmic trading and robo-advisors. Automated trading strategies often integrate portfolio optimisation techniques to adjust portfolios in real-time, ensuring they align with set risk-return goals (Yeo et al., 2023). Similarly, robo-advisors, providing automated investment advice, utilize portfolio optimisation algorithms to suggest asset allocations tailored to individual investor profiles (Morgan Stanley, 2023).

1.3 Significance of Portfolio Optimisation

Before deploying or even designing a system for autonomous live trading with real capital, it's imperative that users fully grasp the performance and behavior of their strategy, as well as their capital allocation methods. One of the most effective ways to achieve this understanding is through portfolio optimisation. It is indispensable in any investment strategy, be it systematic or otherwise. Some of the salient benefits of portfolio optimisation include:

1. Covariance and Correlation

   Portfolio optimisation heavily relies on the covariance matrix, which captures how different assets move in relation to each other. By understanding these relationships, which are a fundamental aspects of EntroPy, investors can combine assets in a way that reduces the overall portfolio variance, leading to more stable returns. This is technically achieved by combining assets that have negative or low correlations, thereby reducing the portfolio's susceptibility to individual asset volatility (Bartz et al., 2013).

2. Utility Maximisation

   From a technical standpoint, portfolio optimisation can be viewed as a utility maximisation problem. Investors derive utility from returns and disutility from risk. The optimisation process seeks to find the portfolio that maximizes this utility, given the investor's risk aversion. Philosophers Bentham and Mill hypothesised that, given consumers act rationally; they aim to derive the maximum advantage for their needs (CFI Team, 2021). Yet, factors like limited rationality and inherent biases can lead them to choose combinations that might not fully optimise their utility. Furthermore, a consumer's optimal utility package isn't static; it can evolve based on shifts in their personal preferences for goods, fluctuations in prices, and variations in their income (Read, 2004). Assuming a

consistent data quality, this is where EntroPy's reliance on mathematical and statistical methods inherently reduces subjective biases that might arise from human judgment.

3. Constrained Optimisation

   Real-world investing often comes with constraints, whether they are regulatory, liquidity-based, or self-imposed. Portfolio optimisation techniques, especially non-linear and quadratic programming, allow for the incorporation of these constraints, ensuring that the resulting portfolio is not only optimal but also feasible (Jin et al., 2016). As will be examined in Chapter 3, this precisely corresponds to the objective function seeking to simultaneously minimise volatility and maximise Sharpe ratio against some inequality constraints.

4. Sharpe Ratio and Risk-Adjusted Returns

   Portfolio optimisation extends beyond raw returns. Metrics like the Sharpe ratio, which measures the risk-adjusted return of an investment, play a crucial role. By optimizing for the Sharpe ratio, investors aim to achieve the highest return per unit of risk taken. The Sharpe ratio serves as a standardised metric, that is often maximised as a primary objective. Informed decision making is made possible by maximising the Sharpe ratio; for instance, two portfolios might offer identical expected returns, but if one boasts a superior Sharpe ratio, it suggests a return achieved with reduced risk—a critical factor in capital allocation.

5. Idiosyncratic Risk Deprecation

   Idiosyncratic risk refers to the non-systematic (hence unpredictable) risk associated with individual assets within a portfolio, arising from factors that are specific to that particular asset. For instance, this might include company-specific news, earnings reports, or changes in management (Goyal & Santa-Clara, 2003). This risk can be diversified away by holding a well-apportioned portfolio, meaning that more uncorrelated assets are added to a portfolio, the overall idiosyncratic risk decreases. Finding the optimal weights for each asset in a portfolio, optimisation ensures that the portfolio is not overly exposed to any single asset's idiosyncratic risk.

1.4 Methodology: Monte Carlo Simulations & The Efficient Frontier

Within the scope of this report, two optimisation approaches particularly stand out due to their significance and practicality: Monte Carlo simulations and the Efficient Frontier. Their exploration is central to the aims of this research. A detailed discussion on their selection and distinct attributes is provided in Chapter 2, while Chapter 3 delves into the rationale for their inclusion. To offer a brief overview and prepare the reader for subsequent discussions, both concepts are succinctly highlighted below, clarifying their roles and reasons for utilisation.

Monte Carlo simulation is a stochastic computational technique rooted in statistical sampling. It serves as a method to model the probability of different outcomes in a process that cannot easily be predicted due to the intervention of random variables. By simulating a large number of scenarios for asset returns, Monte Carlo methods provide insights into the expected returns and risks of different portfolio constructions. This technique is particularly valuable in contexts where the asset return distribution is complex or when exact analytical solutions are unattainable. Its strength lies in its ability to model and account for the non-linearities and random path dependencies in financial markets, which traditional deterministic models might overlook.

The Efficient Frontier is a cornerstone concept in Modern Portfolio Theory (MPT). It represents a set of optimal portfolios that offer the highest expected return for a specific level of risk. These

portfolios lie on a hyperbolic curve in the risk-return space, where the x-axis represents portfolio risk (often measured as standard deviation) and the y-axis denotes expected return. Portfolios that reside on this frontier are deemed "efficient" as they provide the maximum possible expected return for a given level of risk. The selection of a portfolio from the Efficient Frontier depends on an investor's risk appetite. The significance of the Efficient Frontier in portfolio optimisation is its ability to guide investors in constructing portfolios that align with their risk-return preferences, ensuring that for any given level of risk, the portfolio has the maximum expected return.

1.5 Target Audience & Research Question

Because of its methodological approach grounded in theoretical principles while emphasising practical solutions, the intended recipients encompass:

- Risk Analysts
- Quantitative Researchers and Traders
- Portfolio Managers
- Software Engineers and Architects
- Individuals who possess a fundamental familiarity with statistics and quantitative finance

The primary aim of this project is to bolster the ease of constructing portfolios that endeavor to optimise stock (equity) strategies, which can be modularly integrated into algorithmic trading.

Within the scope of this report, the following research questions will be explored:

1. Which methodologies can be employed to design and implement a low-compute-cost portfolio optimisation framework for evaluating the financial performance of equities?
2. How can EntroPy, both flexibly and pragmatically, leverage Monte Carlo simulations and Efficient Frontier sets, along with a range of associated financial metrics, to attain portfolio optimisation?
3. How does EntroPy ensure efficient management of diverse data sources, whether it's self-generated data, data extracted from files, or data retrieved from an API/library (such as yfinance), in a tractable manner?
4. Beyond financial metrics, how does EntroPy provide insights into the internal computational logic of portfolio structures, thus facilitating more straightforward strategy optimisations with minimal code?
5. In what ways does EntroPy present users with easily discernible input and output formats, like CSV and relevant visual representations (plots), ensuring clarity in interpretation?

In essence, the overarching goal of this project is to craft a dependable and efficient tool that streamlines the process of portfolio optimisation for equities. Instead of zeroing in on empirical outcomes and the profitability of strategies, this report emphasizes the software employed to derive those results. This is accomplished through a mix of Python-based object-oriented and procedural programming, details of which will be further elaborated upon in the context of its implementation in Chapter 5. In the grand scheme, this research aspires to pave the way for subsequent studies on portfolio optimisation within the realm of algorithmic trading, positioning itself as a precursor to other foundational elements such as backtesting, data procurement, and signal propagation.

In line with its motivations and objectives, EntroPy intends to develop an object that stores the stock prices of a given selected financial portfolio. It then proceeds to analyse this data and generate diverse plots showcasing performance from returns, and modelling moving averages using Bollinger Bands with buy/sell indicator integration. Moreover, EntroPy facilitates optimisation via the Efficient Frontier or a Monte Carlo simulation for a given financial portfolio, with the chief goal of minimising the amount of code required to achieve said optimisations.

1.6 Motivation

Many contemporary platforms offering portfolio optimisation products boast a wide array of functionalities. These span from gathering delayed or real-time financial data, implementing analytical tools for profitable stock allocation, and including features for clear data visualization to enhance readability. Furthermore, they support the creation of optimal asset distributions, facilitate live trading, and even offer backtesting environments. However, these platforms come with limitations. Many require users, such as private investors, to purchase licenses to utilize their software – for instance, mlfinlab – leading to additional costs (Hudson and Thames, 2020). Additionally, numerous platforms exclusively process and offer insights on specific asset classes, whether they be equities, cryptocurrencies, commodities, etc. A detailed comparison of related products, juxtaposed with the novelty of EntroPy, is available in Chapter 9's evaluation.

Consequently, the primary objective of this report is to conceptualize and implement a portfolio optimisation framework named EntroPy. This framework prioritizes insights into the efficacy of a given strategy while advocating for modularization across feature functionalities. The motivations driving this project are:

1. Effective Portfolio Construction via the Efficient Frontier:
    - Problem: Investors often struggle with determining the best combination of assets to maximise returns for a given risk level.
    - Solution: EntroPy allows users to construct portfolios from a set of stocks and compute key metrics. A module pertaining to an Efficient Frontier will visualise the risk-return trade-off by minimising an objective function via either volatility or Sharpe Ratio.

2. Risk Assessment using Monte Carlo Simulations:
    - Problem: Predicting future portfolio performance and understanding potential risks is challenging.
    - Solution: A module will offer Monte Carlo simulations, allowing users to simulate a range of possible portfolio outcomes based on historical data. This probabilistic approach gives a clearer picture of potential risks and returns.

3. Trend Analysis for Investment Decisions:
    - Problem: Investors and traders often need tools to discern market trends for timely investment decisions.
    - Solution: A moving averages module will offer functionalities to compute simple, exponential, and Bollinger bands of moving averages, popular indicators in technical analysis.

4. Data Retrieval and Pre-processing:
    - Problem: Fetching and pre-processing stock data for analysis can be tedious and time-consuming.
    - Solution: Two modules will hold stock and index data, reducing the need eliminating the need for manual data retrieval and pre-processing. An additional module pertaining to constructing the portfolio will ensure integrity by strictly managing input datatypes (i.e. only allowing data-frames or series, etc.).

5. Quantitative Analysis for Financial Research:
    - Problem: Financial analysts and researchers require a set of tools to perform quantitative analyses on stock data.

- Solution: A dedicated module will offer functions to compute returns, volatilities, value-at-risk, downside risk, Sharpe ratio, and annualise correlations, fostering rigorous financial research.

6. Optimal Decision Making:
    - Problem: Making investment decisions requires a balance between expected returns and associated risks.
    - Solution: By leveraging optimisation techniques, integrated from libraries like SciPy, EntroPy will aid in determining optimal portfolio weights. This optimisation will ensure that the portfolio aligns with the investor's risk tolerance and return expectations, which are also modularly managed in a separate file for clarity.

7. Educational Tool for Financial Concepts:
    - Problem: There is a need for hands-on tools that can help students, hobbyists and even enthusiasts learn about financial concepts in a practical manner.
    - Solution: The inclusion of scripts in Chapter 8 will provide practical demonstrations of portfolio optimisation, Monte Carlo simulations, and other financial concepts, serving as a pedagogical resource.

8. Bespoke Selection of Relevant Financial Metrics:
    - Problem: Existing software solutions employ varied metric combinations without elucidating their selection rationale. For instance, PyPortfolioOpt uses Black-Litterman and Efficient Frontier but lacks robust tools for technical analysis or Monte Carlo simulations (Martin, 2023).
    - Solution: EntroPy delivers a comprehensive, yet manageable, array of analytical tools, encompassing two types of optimisation, technical analysis, and stock-specific data presentations. A thorough examination of existing implementations, juxtaposed with EntroPy's open-source contribution rationale, is available in Chapter 9.

1.7 Report Structure

This report is structured into approximately 10 chapters:

Chapter 1 sets the foundation for the entire report. It introduces the concept of portfolio optimisation, provides a brief historical perspective, and underscores its significance in the financial sector. This chapter also spells out the primary motivations for this research, outlines the core research questions, and offers an overview of the employed methodology.

Chapter 2, serving as a foundation of theoretical knowledge, elucidates key financial terms pivotal to this research. It offers insights into the financial backdrop against which this research is set and touches upon the mathematical concepts that underpin the methodologies used.

Chapter 3 focusses on a review of relevant literature pertaining to Mean-Variance Optimisation via the Efficient Frontier and Monte Carlo Simulations, with a holistic evaluation of each advancement.

Chapter 4 provides an in-depth look at the system specification. It presents an overview, illustrated with UML diagrams, and delves into the features and deliverables of the product. It outlines both functional and non-functional requirements. It also sheds light on the design principles that guided the development process, and visualises and discusses the system’s architecture. Finally, it explores the rationale behind the choice of Python and other technical frameworks such as SciPy, NumPy, and Pandas.

Chapter 5 chronicles the journey from conceptualization to realization, detailing how the proposed solution was implemented in terms of data structures selection, algorithms used, and core features.

Chapter 6 is dedicated to the validation of the solution. It discusses the unit tests conducted to ensure the system's robustness and reliability.

Chapter 7 acts as a deep dive into the management aspect of the project, this section outlines actions, timelines, deliverables, and reflections. It provides a step-by-step breakdown of the project's lifecycle, from research to design and development phases, illustrated with a Gantt chart.

Chapter 8, a critical section, it presents the outcomes of the software, via some sample scripts that are used to construct a portfolio from a user’s local storage or from an API. There are also two further scripts, one that highlights how moving averages may be computed, and one that showcases an example optimisation using Monte Carlo simulation and Efficient Frontier.

Chapter 9 offers reflective insights on the accomplishments of this research, pinpointing areas of improvement and potential future avenues. This chapter evaluates EntroPy's final implementation against the initial requirements and contrasts it with existing software solutions. Furthermore, it proposes recommendations for subsequent ventures in this field.

Chapter 10 brings the report to its conclusion, encapsulating the pivotal findings, insights, and contributions of this research.

2 Foundations of Portfolio Theory

This chapter offers an insight into the principles underlying returns, their distribution patterns, and the fundamentals of Markowitz's mean-variance portfolio optimisation. It aims to elucidate the reader on some of the core principles surrounding key components within EntroPy, to more seamlessly ease them into the following practical interpretations. Furthermore, it explores the limitations of using volatility as a risk metric, and the challenges and benefits associated with Value-at-Risk.

The section begins by exploring the concepts of forecast return and volatility, accentuating the significance of the dispersion matrix. The discussion then extends to performance metrics like the Sharpe and Sortino ratios, and risk measures such as downside risk, the aforementioned value at risk, and the risk-free rate of return. In terms of assets, the characteristics and metrics associated with stocks and indices, including kurtosis, volatility, and the beta coefficient are examined. The chapter also explores moving averages, detailing the simple and exponential moving averages, Bollinger bands, and their respective bands. Lastly, the section briefly addresses the optimisation techniques employed in portfolio management, emphasizing the strategy of minimisation using the negative Sharpe ratio.

2.1 Assets

A financial asset is an intangible item indicating a right to future monetary inflows. An asset may commonly be termed as a financial instrument. Every financial instrument involves at least two entities. The issuer is the party committed to future cash disbursements, while the investor is the entity holding the financial instrument and consequently the entitlement to the payments from the issuer (Drake and Fabozzi, 2010). Typically, these assets are exchanged in financial marketplaces.

The main types of financial assets, alongside their examples, comprise (Ganti, 2023):

- Equities (i.e. common, preferred, or convertible stocks, and global depositary receipts)
- Cash / Liquid (i.e. certificates of deposit, commercial paper, or time deposits)
- Bonds (i.e. municipal, zero-coupon, or convertible bonds, and floating-rate notes)
- Funds (i.e. index, commodity, or closed-end funds, and unit / real-estate investment trusts)
- Derivatives (i.e. currency futures, oil & gold options, and stock / equity swaps)

As an initial step, EntroPy concentrates on equities as a proof of concept. This design is rooted the following 3 main strategic considerations:

1. Ubiquity and Popularity: Equities (stocks), representing ownership in a company, are among the most widely traded and recognized financial instruments globally. In 2023, their global equity market cap is estimated at $108.6 trillion, and their universal appeal makes them an ideal starting point (SIFMA, 2023).
2. Rich Data Availability: The equity market is characterised by a wealth of data, from price histories to company fundamentals. For instance, The London Stock Exchange (LSEG) offers a breadth of discretised data offerings for markets such as the FTSE 100 and AIM. Offering, a three-tiered delivery, LSEG provide off-book transactions that facilitate reporting for both on-Exchange and OTC trades, Level 1 access for real-time optimal prices, trade data, and security liquidity and Level 2 access tick-by-tick analysis (London Stock Exchange, n.d.). The New York Stock Exchange (NYSE) offers equally systematic services.
3. Diverse Range: Equities span across various sectors, industries, and geographies. This ranges from healthcare, to materials, real estate, technology, energy, utilities, and many more (Thune, 2023). This diversity offers a comprehensive testing ground for EntroPy,

ensuring that the tool is versatile and adaptable across different market conditions and segments.

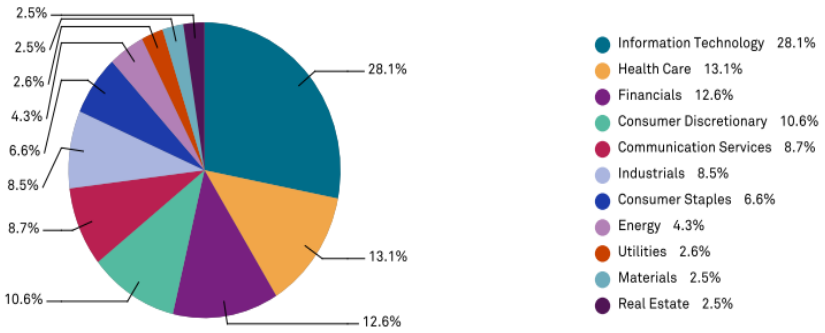


Figure 1: S&P 500 Equity Sector Breakdown 2023 (S&P Dow Jones Indices, n.d.)

2.2 Stocks and Indexes

This report focusses on two main types of related equity assets: stocks and indexes. Stocks have previously been outlined in the discussion surrounding equities. An index is a statistical measure that tracks the performance of a group of stocks. It provides a snapshot of the market or a specific segment of the market. Examples include the S&P 500, which tracks 500 large-cap U.S. stocks, or the Nikkei 225, which tracks 225 top-rated companies in the Tokyo Stock Exchange. Indexes are often used as benchmarks to compare investment performance. For example, the Dow Jones Industrial Average may be used as benchmarks to evaluate the performance of U.S. equities, using r-squared and / or beta coefficient measures (Young, 2023).

Specifically related to the asset, three primary statistical measures are employed: the beta coefficient, portfolio kurtosis, and skewness.

Beta (denoted as β) measures the volatility or systematic risk of a security or portfolio in comparison to the market as a whole. If $\beta = 1$, this indicates that the security's price moves with the market. If $\beta > 1$, this indicates that the security is more volatile than the market and will move with more momentum, while $\beta < 1$ means it is less volatile and comprises less momentum. It is defined as (Johansson and Petersson, 2017):

$$\beta = \frac{cov(R_n - R_f, R_M - R_f)}{var(R_M - R_f)}$$

Where:

- β: beta coefficient
- cov: covariance or dispersion, which describes the directional relationship between fluxes of a stock's returns against a market's return
- var: variance, which defines the spread of data points from the mean datum
- R_n: forecast return of an individual given asset n
- R_f: risk-free interest rate of return, describing the interest an investor would expect from an absolutely risk-free investment over a specified period of time
- R_M: forecast return of the overall market

Beta plays a crucial role in the Captial Asset Pricing Model (CAPM), which describes the relationship between the forecast return of an asset and its risk as measured by Beta (Bodie, Kane, and Marcus, 2012). According to CAPM, the formula for calculating the forecast return of a given asset, considering its risk, is:

$$R_n = R_f + \beta(R_M - R_f)$$

The concept of covariance has been introduced previously but can be expressed as:

$$cov = \sum \frac{(R_i - mean_i) \times (R_j - mean_j)}{n_{cov} - 1}$$

Where:

- R_i & R_j: daily returns for the stocks *i* and *j*
- $mean_i$ & $mean_j$: mean returns for the stocks *i* and *j* over the specified period
- n_{cov}: sample size of the period, days

Typically, a positive covariance equates to both stock being higher or lower concurrently and move in tandem. A negative covariance indicates an inverse relationship between two variables, where, when one underperforms, the other often excels. Positive covariance is visualised graphically via the data points trending upwards, whereas negative covariance sees points trending downwards (Hayes, 2023a).

2.3 Conditional Model for Dynamic Beta

It is often expected that investors strategize with a singular, homogenous time-frame in mind. This assumption implies that both the asset's beta and the market risk premium remain invariant, irrespective of the time – essentially, they are perceived as constants. However, in practice, investors typically forecast and strategize across multiple time horizons, indicating that both an asset's beta and the market risk premium can fluctuate based on specific temporal benchmarks (Ghysels, 1998). This distinction between "unconditional" and "conditional" interpretations of CAPM becomes evident when examining the expression for covariance, where F denotes “forecast”:

$$cov(x_1, x_2) = F(x_1 x_2) - F(x_1)F(x_2)$$

When substituting the beta for random variable x_1, and the excess forecast return for the overall market, R_M, for random variable x_2, the following equations are obtained:

$$F(x_1 x_2) = cov(x_1, x_2) + F(x_1)F(x_2)$$

$$\therefore F(\beta R_M) = F(\beta)F(R_M) + cov(\beta, R_M)$$

Where $F(\beta R_M)$ is the risk premium associated with a given asset. Under the assumptions of the unconditional model, both the expected beta $F(\beta)$, and the market risk premium $F(R_M)$, remain constant. Their covariance $cov(\beta, R_M)$, is zero, indicating no variability between them. However, if $F(\beta)$ and $F(r_M)$ are dynamic and exhibit positive covariance over time, the unconditional beta may inadvertently downplay the true market risk associated with the asset. Conversely, a negative covariance would lead to an overestimation of this market risk. This nuanced understanding of the relationship between beta and market risk premium is articulately explored in the insights of Barinov (Barinov, 2014).

While it is important to consider for underpinning portfolio allocations, EntroPy does not consider the conditional’s model inclusion of the dynamic nature of beta and the market risk premium, to ensure that the project remains simple and usable. There are significantly more data quality requirements including time-varying estimates of beta and market risk premium. Not all datasets or sources, as is seen in Chapter 5, provide this level of granularity, making the unconditional model more universally applicable (Corradi et al., 2011). EntroPy prioritises computational efficiency, and for tools aiming for real-time analysis or quick insights, the conditional model demands a much more extensive and robust software infrastructure, which is presently out-of-scope.

2.4 Returns

Within the context of portfolio theory, a return can be defined as the financial gain or loss of an asset or portfolio over a specific period. Essentially, a positive return indicates a gain, while a negative one signals a loss, typically expressed as a percentage. The foundational principle is that higher risks often lead to the potential for greater returns, but they also come with the possibility of larger losses. The return over a given time period, R_t, is:

$$R_t = \frac{V_f - V_i}{V_i}$$

Where:

- V_f: asset price at final (end) time, *f*
- V_i: asset price at initial time, *i*

Before diving into further metrics, it's assumed that the foundational Efficient Market Hypothesis (EMH) applies consistently. EMH suggests that financial markets function in a state of informational efficiency (Jones and Netter, 2023). Formally, this hypothesis posits that asset prices in financial markets integrate and reflect all available information. This makes it challenging for investors to consistently achieve abnormal returns on a risk-adjusted basis, given the current information set (Cochrane, 2014).

Additionally, EMH states that persistently generating Alpha is unattainable (Wall Street Prep, n.d.). Alpha is a term that describes the abnormal rate of return on a security or portfolio in excess of what is predicted by an equilibrium model like the Capital Asset Pricing Model (CAPM). The existence of a positive alpha challenges the tenets of the EMH, which asserts that all publicly available information is already incorporated into asset prices. Interestingly, the pursuit of alpha directly associates with achieving returns that are higher than those predicted by the risk taken. If markets are truly efficient, consistently achieving a positive alpha would be an anomaly. However, the very existence of active portfolio management and the continuous search for alpha imply that many practitioners believe in the possibility of outperforming the market, at least over specific periods (Malkiel, 2003).

There is a plethora of relevant return standards that are useful in visualising the performance of an equity or investment. One of the most significant is the forecast (expected) return, which defines the magnitude of the prospective profit or loss. The forecast return is given by the unbiased approximation, assuming the efficient market hypothesis applies, of $F(R_h)$, the historical mean returns (Random Services, n.d.):

$$F(R_h) = \frac{1}{n}\sum_{i=1}^{n} R_t$$

Where:

- R_t: the return over a given time period
- n: the number of time phases (i.e., annual trading days)

Following this, the forecast return on a given asset portfolio, $F(R_p)$, is calculated as a weighted mean of the returns of each individual asset in the portfolio's asset universe, over time t (Brealey et al., 2012). It is given by the equation:

$$F(R_p) = (w_1 \times R_h^1) + (w_2 \times R_h^2) + \cdots + (w_N \times R_h^N)$$

Where w_1, w_2, ..., w_N represents the total weighting or allocation of an asset weighting in the portfolio, and R_h^1, R_h^2, ..., R_h^N is the forecast return for each distinct portfolio asset. It follows that all

weights must sum to 1, and that each elementary weight w_i is comprised in the subset of the vectorised weight W, for the portfolio assets:

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

$$w_i \in W \qquad for\ all\ i = 1, 2, \dots, N$$

It is important to note that the expected return is not a guaranteed rate of return. Instead, it serves as a well-founded projection to gauge the future worth of portfolios. Moreover, it offers a metric to assess real returns.

In the scope of this report, 5 primary return types are paramount to evaluating an investment's performance. While their exact implementation is discussed in a subsequent chapter, a brief description of each, alongside their formulae, and significance are outlined below. These include:

1. Cumulative returns

   Cumulative returns give the aggregated return of an investment over a given time frame, taking into account any dividends. They are given by:

   $$R_{cumulative} = \frac{price_{t_i} - price_{t_0} + dividend}{price_{t_0}}$$

   Cumulative returns provide a comprehensive view of the total return on an investment from the starting point, factoring in dividends, which can be crucial for long-term investment analysis.

2. Daily returns

   Daily returns represent the day-to-day percentage changes in the value of an investment. They are computed via:

   $$R_{daily} = \frac{price_{t_i} - price_{t_{i-1}}}{price_{t_{i-1}}}$$

   Daily returns offer a granular perspective on the daily fluctuations of an asset, vital for understanding short-term volatility and performance.

3. Proportioned daily returns

   This metric calculates the daily mean returns adjusted by a set of predetermined allocations or weights for each asset in a portfolio. It is expressed as:

   $$R_{proportioned} = \sum_{i=1}^{n} R_{daily} \times allocation_i$$

   In a diversified portfolio containing multiple assets, each asset might not be equally significant. Proportioned daily returns account for the relative importance of each asset, giving a more accurate depiction of the portfolio's daily performance.

   Allocating weights to each asset also allows for adjustments to be made based on various strategies or perspectives. For instance, if certain assets are deemed riskier or more crucial, they might be assigned higher weights. Conversely, more conservative or less significant assets might have reduced weights.

Proportioned daily returns also aid in performance attribution, enabling investors to understand which assets (and in what proportion) contributed to the portfolio's performance on a given day.

4. Logarithmic daily returns

   Logarithmic daily returns leverage the natural logarithm to determine the rate of change in asset value from one day to the next. The logarithmic return is defined as:

$$R_{log} = \log\left(\frac{price_t}{price_0}\right)$$

   Or, with respect to the daily return, is simply:

$$R_{log} = \log(1 + R_{daily})$$

   Logarithmic returns, due to their time-additive property, simplify the aggregation of returns over time, making them crucial for time-series and continuous compounding analysis.

   The mathematical properties of logarithms can simplify certain types of financial analyses, such as in geometric Brownian motion models used in option pricing (van der Wijst, n.d.).

   Logarithmic returns often exhibit more stable statistical properties, and are more amenable to assumptions of constant volatility and constant correlation over time. They are also often found to be more normally distributed than simple returns, making them more suitable for subsequent methods that assume normality (Quantivity, 2011).

5. Historical average returns

   This metric provides the annualized average of daily returns, scaled by the number of regularised trading days in a year, typically 252 days (SwingTradeSystems, 2023). Historical average returns are given by the mean of daily returns, annualized:

$$R_{historical} = \bar{R}_{daily} \times regular\ trading\ days = \bar{R}_{daily} \times 252$$

   When combined with measures of risk, like standard deviation, the historical average return can be used to assess the risk-adjusted performance of an asset or portfolio. It forms the basis for metrics like the Sharpe Ratio, which will be shortly discussed.

   Furthermore, financial markets can be influenced by a multitude of short-term events that might cause daily returns to spike or dip temporarily. By focusing on the average return over a longer period, one can filter out some of this "noise" and get a clearer picture of the asset's underlying performance trend (Anagnostidis, 2018).

2.5 Characterising Return Distributions

2.5.1 Probability Density & Cumulative Distribution Functions

Two pivotal tools in the analysis of return distributions are the Probability Density Function (PDF) and the Cumulative Distribution Function (CDF). These prove useful in evaluating risk and expectations of a given investment's prices. Both PDF and CDF are instances of normal or symmetric distributions, where the probability of an event is equal at a given point above or below the maxima or minima. This is commonly represented as a bell-shaped curve, and is graphically

depicted below for differing mean (μ) and standard deviation (σ) values in the figures below (Kull, 2014):

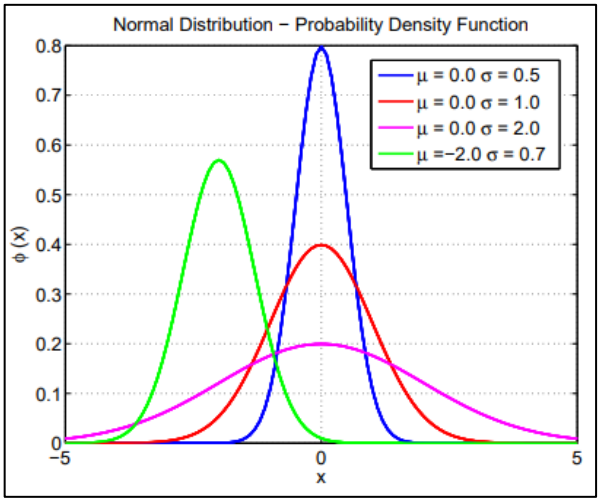


Figure 3: PDF with varying mean & standard deviations.

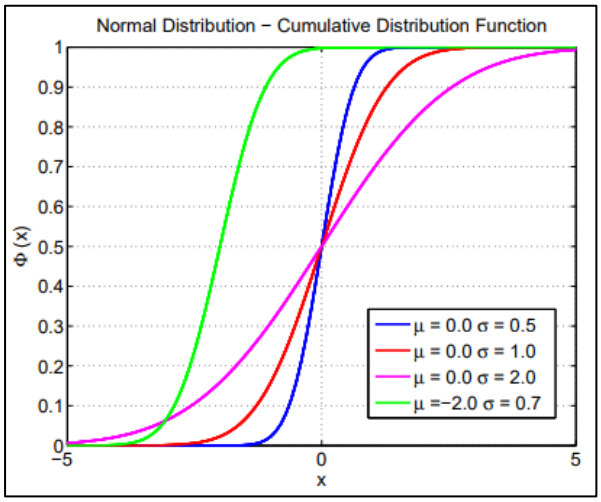


Figure 3: CDF with varying mean & standard deviations.

Graphically, the y-axis of both the PDF and CDF represents cumulative probabilities, while the x-axis displays potential returns or outcomes. The function always starts at 0 and increases monotonically, reaching 1 as x approaches infinity.

The PDF (Probability Density Function) of a normal distribution provides the probability of a random variable assuming a specific value. Essentially, it captures uncertainties associated with investment strategies, describing the likelihood of different return profiles for a portfolio. A narrow, peaked PDF might suggest a portfolio with more predictable returns, albeit potentially at a lower yield. Conversely, a broader, flatter PDF might indicate higher potential returns but with increased uncertainty and risk. It is given by the following equation, where μ represents the mean of the distribution, and σ is the standard deviation:

$$\phi(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-0.5\left(\frac{x-\mu}{\sigma}\right)^2}$$

While the PDF provides a snapshot of the likelihood of specific returns, the CDF cumulatively aggregates these probabilities, encompassing 100% of all potential outcomes, and offering a holistic view of the expected behaviour of a portfolio. Namely, the CDF provides the probability that a random variable will take a value less than or equal to a given forecast return. For a continuous random variable with PDF $\phi(x)$, the CDF $\Phi(x)$ may be defined as the integral of the probability density function as:

$$\Phi(x) = \int_{-\infty}^{x} \phi(x)\, dt = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}}\, dt$$

Nevertheless, there are several experiential studies that suggest that normally-distributed portfolios place infinitesimal probabilities on extreme outliers, which are more significant in investing than other perhaps more scientific applications, such as healthcare or energy (Fama-French, 2019). Therefore, EntroPy needs a way to more elegantly account for non-normality in its analyses of risk management. This leads to skewness and kurtosis.

2.5.2 Observed Phenomena: Skewness & Kurtosis

Practical data distributions, in finance or otherwise, largely experience and are characterised by a skewness and kurtosis that is either less than or greater than 0.

Skewness measures the degree of asymmetry or the “peakedness” in a data set. The greater the skewness, the lesser the symmetry. When visualized on a bell curve, a perfectly symmetrical distribution of data points around the median signifies no skewness. However, any deviation from this symmetry, either to the left or right, indicates the presence of skewness. While a normal distribution is characterized by zero skew, other distributions, such as the lognormal, can manifest varying degrees of skewness (Zucchi, 2021).

Skewness can either be positive, negative or zero. In a positively skewed set, the tail of the distribution stretches right, suggesting the mass distribution of data points lie to the left of the mean, with occasional high-value outliers on the right. A negative skewed set operates vice versa, where the tail extends more towards the left, indicating that most data points are situated to the right of the mean, with sporadic low-value outliers on the left. Zero skewness represents data that is symmetrically distributed, mirroring the normal distribution.

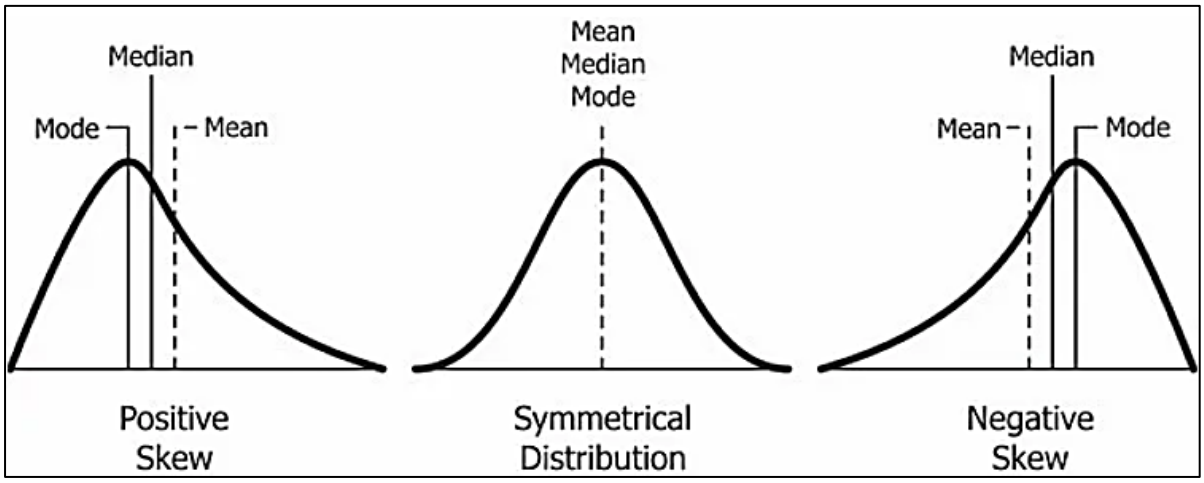


Figure 4: A comparative overview of mean & median positions in unimodal distributions with distinct skewness (Dugar, 2018).

The skewness, sk, is given by the following formula. This computes the sample skewness based on the third standardised moment of a variable X about the mean (Efremov, 2020):

$$skew(X) = \frac{1}{n-1}\sum_{i=1}^{n}\left(\frac{x_i - \mu}{\sigma}\right)^3$$

Where:

- $skew$: sample skewness
- n: number of data points in the sample
- x_i: individual data point
- μ: sample mean
- σ: sample standard deviation
- $\frac{1}{n-1}$: bias correction factor for small samples

Investors will tend to be “right-skew” leaning, in that they favour positive skews. A positively skewed distribution indicates that there are occasional high-value returns (outliers) that are significantly above the mean. This means that while most returns might be moderate or even below average, there is a chance for exceptionally high returns. Investors are often attracted to the possibility of these "jackpot" outcomes. The seminal “Prospect Theory” posited by Kahneman and Tversky suggests that investors are often driven by loss aversion, and the endowment effect (Kahneman and Tversky, 1979). In essence, their work suggests that investors' preferences for positively skewed investments might not solely be based on rational calculations of expected returns. Instead, psychological factors, such as the allure of high returns (even if improbable) and aversion to potential losses, play a crucial role in shaping investment choices (Zimpelmann, 2022).

Kurtosis is another statistical metric that is used in tandem with skewness, and it provides insights into the shape of a data distribution, particularly focusing on the “tailedness”. In a graphical representation, the tails are the data points that deviate the most from the mean and kurtosis essentially gauges the probability of extremeness of the outcomes relative to normally-distributed predictions (Croarkin et al., 2001). A distribution with high kurtosis has more data in its tails than a normal distribution, while one with low kurtosis has less.

As seen in the figure below, kurtosis can be classified into three main characterisations:

- Mesokurtic (Kurtosis = 3.0, Excess = 0): This type of distribution has kurtosis similar to a normal distribution, indicating a moderate risk level.
- Leptokurtic (Kurtosis > 3.0, Excess > 0): Leptokurtic distributions have extended thin tails, often resulting from outliers. Such distributions are indicative of higher risks but also the potential for higher returns. This profile results in the occurrence of more probable extreme events.
- Platykurtic (Kurtosis < 3.0, Excess < 0): These distributions have shorter thick tails, suggesting fewer outliers and a more stable, lower risk profile. This profile results in the occurrence of less probable extreme events.

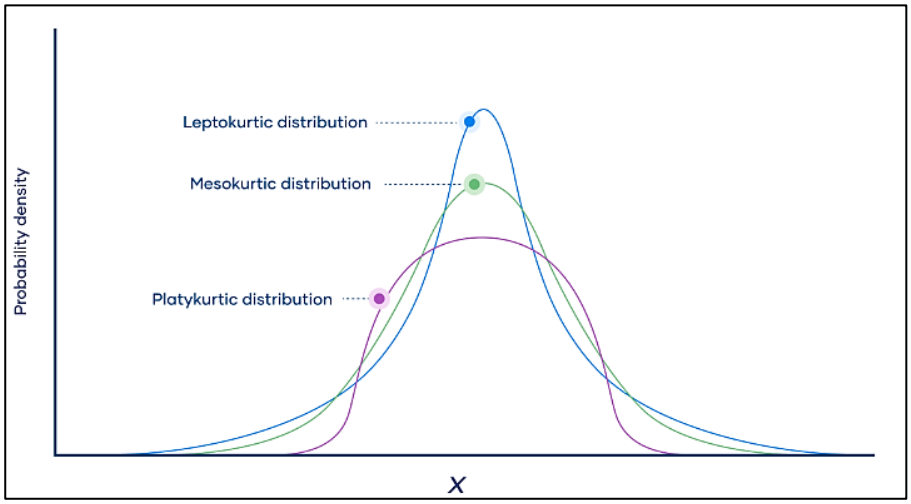


Figure 5: Kurtosis type distributions (Turney, 2022).

The kurtosis, $kurt$, is defined by the following formula. This is based on the fourth moment of a variable X about the mean, standardized by the standard deviation (Inani, 2016):

$$kurt(X) = \frac{1}{n-1}\sum_{i=1}^{n}\left(\frac{x_i - \mu}{\sigma}\right)^4$$

Where:

- $kurt$: sample skewness
- n: number of data points in the sample
- x_i: individual data point
- μ: sample mean
- σ: sample standard deviation
- $\frac{1}{n-1}$: bias correction factor for small samples

Kurtosis is crucial in financial data as it highlights "tail risk" or the likelihood of a loss due to a rare event based on a probability distribution. If such events are more frequent than predicted, the tails are described as "fat."

Investors and portfolio managers will often seek to minimise kurtosis (platykurtic) distributions due to the associated stability and predictability in returns. There are a myriad of advantages and literatures to support this:

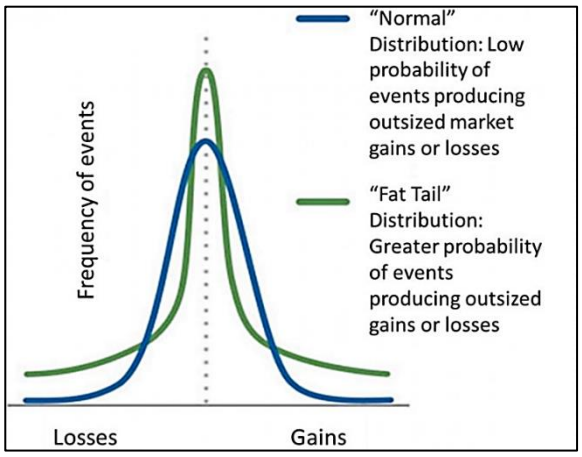


Figure 6: Fat tail distributions present more probability of extreme profit and loss occurrences (Damodaran, n.d.).

1. Reduced Tail Risk

   Platykurtic distributions have thinner tails, which translates to a reduced risk of significant losses, which is often a primary concern for investors. Taleb emphasises the concept of Black Swan events, which are unexpected and rare, have an extreme impact, and are retrospectively (and erroneously) predicted to have been predictable (Taleb, 2007). These significant losses are analogous to Black Swan events.

   One of the central themes of Taleb's work is the inadequacy of standard financial predictive models. He classifies these models as Extremistan, where variables are scalable, and outliers can dominate (e.g., wealth distribution or book sales), nullifying the application of normal distributions (Taleb, n.d.). Such models often rely on Gaussian distributions (or bell curves) and underestimate the probability of Black Swans, being blindsided by them.

2. Lower Volatility

   A lower kurtosis often indicates that the investment does not experience extreme fluctuations, making it less volatile. Lower volatility, which is a concept that will be explored shortly, is often associated with lower risk, which can be more attractive to risk-averse investors. Works by Ang et al. found that equities with higher idiosyncratic volatility (i.e., volatility that is independent of overall market movements) tend to have lower subsequent returns (Ang et al., 2006). This is contrary to the traditional finance notion that higher risk (volatility) should be compensated with higher expected returns, again supporting platykurtic distributions.

3. Easier Risk Management

   With fewer extreme values, it is often easier for portfolio managers to predict potential losses and manage risk effectively. Jorion introduces and elaborates on the concept of Value at Risk (VaR), which again will be subsequently discussed, and quantifies the maximum potential loss an investment portfolio could face over a specified period for a given confidence interval (Jorion, 2006). VaR, as a measure, is particularly sensitive to the tails of the distribution. If returns follow a leptokurtic distribution, a higher probability of extreme negative returns can lead to underestimation of risk if not properly accounted for.

2.6 Risk

Financial risk is the probability that the actual returns of an investment will deviate from its anticipated returns. This encompasses the potential for both gains and losses. A fundamental principle in finance is that with greater risk comes the potential for greater returns, and vice versa. However, higher risk does not guarantee higher returns, just the potential for them.

Engaging in asset transactions with unpredictable future results inherently brings about risks for the stakeholder. Addressing these risks is pivotal for the smooth functioning of financial systems. To illustrate (Elliott and Kopp, 1999):

- Singular investors often diversify their asset portfolios to shield themselves from abrupt shifts in specific stocks or market segments.
- Regulatory bodies in finance aim to reduce the frequency and repercussions of financial institution failures by setting boundaries on permissible trade types and magnitudes, including, for instance, constraints on short selling.
- Within investment entities, risk control specialists set limits on individual trader actions to prevent exposure levels that might be unsustainable during severe market fluctuations.

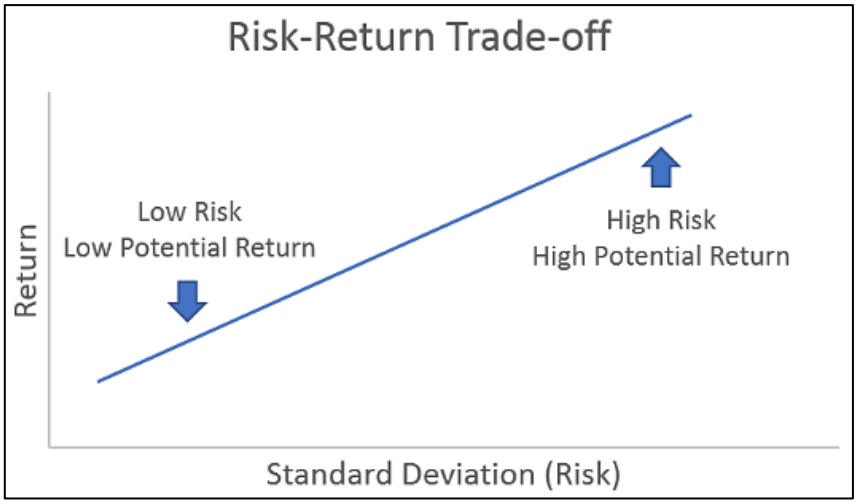


Figure 7: Generalised risk-return trade-off relationship (Model Investing, 2018.)

At its core, the standard deviation, denoted as σ, of an investment's return R offers insight into how R-values deviate from its average. This goal revolves around identifying a portfolio that amplifies expected gains while curbing risk. For instance, if portfolios P_1 and P_2 have mean returns of μ_1 and μ_2 and standard deviations of σ_1 and σ_2, respectively, P_1 would be more appealing if $\mu_1 \geq \mu_2$ and $\sigma_1 \leq \sigma_2$. In such a scenario, P_1 is deemed to overshadow P_2. An "efficient" portfolio is one that no other portfolio can surpass, forming the foundation of the efficient frontier.

Risk management's principal focus is on mitigating downside risk (Sortino and Satchell, 2001). This has led to the creation of risk metrics that zero in on the distribution's lower tail of the final position's random variable. Presently, the most prevalent risk exposure metric is Value at Risk (VaR). The inception and adoption of VaR were reactions to financial crises in the early 1990s, such as the Collapse of Barings Bank (Bhalla, 1995). The subsequent sections will delve deeper into relevant metrics including the volatility, VaR, downside risk, and risk-free rate of return. There will also be an exploration of the limitations of the metrics selected in the context of satisfying the axioms of coherent risk measures.

2.6.1 Volatility

Volatility gauges the variation in returns for a specific financial instrument or market index. Generally, a higher volatility indicates that the asset can have a broader range of potential values, making it riskier. This fluctuation is often calculated using metrics like standard deviation or variance of the returns. Volatility is the fundamental metric used in Markowitz' mean-variance optimisation, which is further evaluated in Chapter 3.

The volatility, σ, may be approximated by a biased sample mean, according to:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(R_t - R_h)^2}$$

Where:

- σ: standard deviation, an analogue for volatility.
- n: number data-points for returns.
- R_t: returns over a time period t
- μ: forecast return over a historical period h

Volatility's strength lies in its simplicity and ease of calculation. It has, however, several limitations, such as being unable to differentiate direction. Volatility measures the magnitude of price movements, both up and down, and does not discern assets that are experiencing rapid growth from ones that are crashing. More significantly, traditional volatility measures often assume returns are normally distributed. However, as is discussed previously, financial returns often exhibit fat tails (kurtosis) and can be skewed, which means extreme events can occur more frequently than a normal distribution would suggest. Therefore, the inherent symmetrical treatment of profits and losses means that volatility cannot accurately capture Black Swan events or other outliers (Artzner et al., 1999). The Global Financial Crisis of 2008, for instance, directly oppose this notation of normally-distributed markets (Reid, 2015).

Modern portfolio management strives to be more conservative and account for losses associated with risk. Measures such as Value at Risk and Downside Risk facilitate more precise control of markets where these fat-tailed distributions dominate (Cont et al., 2010).

2.6.2 Value at Risk

Value at Risk (VaR) provides an estimate of the potential loss that could arise from adverse market movements. Coined by JP Morgan Chase in 1994, it is the most significant potential loss (with the largest negative return) over a specified time frame, which will not surpass a predetermined confidence level (α). VaR is characterized by the minimal value γ, given a confidence level α within the range (0, 1), where the probability P of the loss L surpassing gamma is at most $(1 - \alpha)$. The general abstract definition is therefore (Artzner et al., 1999):

$$VaR_\alpha = \min\{\gamma \in \mathbb{R}: P(L > \gamma) \leq 1 - \alpha\}$$

Where:

- VaR_α: Value-at-Risk at confidence level of α
- γ: threshold value representing a potential loss
- P: probability of L exceeding the threshold γ
- L: actual loss
- α: confidence level / interval

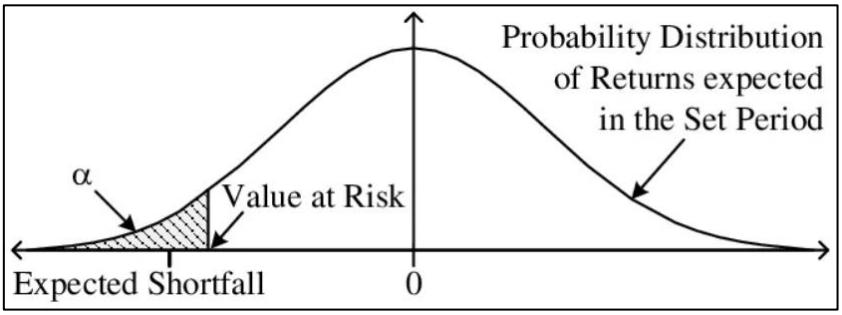


Figure 8: Visualisation of the quartile-nature of a Value-at-Risk metric for an arbitrary confidence level, α (Koors and Page, 2012).

In the context of EntroPy, a parametric (variance-covariance) method is employed, assuming returns are normally distributed for simplicity and speed, consistency with the CAPM, and widespread use throughout the banking industry (Mehta et al., 2012). It uses the mean (μ) and standard deviation (σ) of returns to compute VaR via:

$$VaR_\alpha = \mu - Z_\alpha \sigma$$

Where: Z_α is the z-score corresponding to the desired confidence level α.

2.6.3 Downside Risk & Risk-free Rate of Return

Downside risk, DsR, gauges the potential decrease in an asset's value due to unfavourable market conditions. It provides an estimate of the maximum loss an investor might face, focusing solely on the negative outcomes without considering potential gains. This risk measure is particularly useful as it offers a one-sided perspective, contrasting with other measures that consider both gains and losses symmetrically. Value-at-Risk is an example of a downside risk measure.

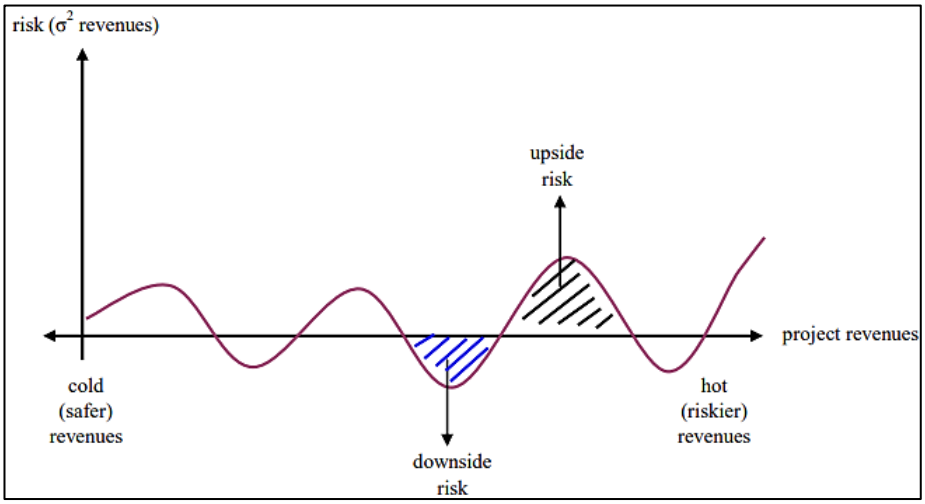


Figure 9: Relationship between returns and risk, highlighting an example of downside and upside risk (Moro-Visconti, 2016).

Two common metrics for assessing downside risk are downside beta and the downside semi-deviation. The latter, often referred to as target semi-deviation (TSV), has gained traction as an industry standard and is the choice of dispersion measure for EntroPy (Nawrocki, 1999). Downside semi-deviation measures the variability of returns below a certain threshold, often the risk-free rate of return or zero, and it focuses on the negative deviations from this threshold (Chong et al., 2013). It is computed via:

$$DsR = \sqrt{\frac{1}{n}\sum_{i=1}^{n} \min\{0, R_i - R_{rfr}\}^2}$$

Where:

- n: total number of trading days
- R_i: return on day i
- R_{rfr}: risk-free rate of return

The risk-free rate of return is a concept that represents the expected return on an investment that is considered free of any risk. It is essentially the return an investor would expect from an absolutely certain investment over a specified period. In determining the risk-free rate, especially in the context of a specific currency, financial experts often look at the yield to maturity of a government bond issued in that currency. For instance, in the U.S., the return on zero-coupon Treasury bonds (T-bills) is frequently used as a proxy for the risk-free rate in U.S. dollars (Bodie et al., 2017).

2.6.4 Axioms of Coherent Risk Measures

Coherent risk measures are a class of risk measures that satisfy a set of axiomatic properties designed to ensure consistency and rationality in risk assessment. These properties were introduced to address some of the limitations and inconsistencies observed with traditional risk measures. The four primary properties that a coherent risk measure must satisfy are (Artzner et al., 1999):

1. Monotonicity: If portfolio P_1 always results in equal or better outcomes than portfolio P_2, its risk, ρ, should not be higher.

$$if\ P_1 > P_2, \quad then \quad \rho(P_1) \leq \rho(P_2)$$

2. Sub-additivity: The combined risk, ρ, of two portfolios, P_1 and P_2, should not exceed the sum of their individual risks. This property ensures that diversification is always beneficial.

$$\rho(P_1 + P_2) \leq \rho(P_1) + \rho(P_2)$$

3. Positive Homogeneity: For any positions in a portfolio are scaled up by a positive factor, λ, the risk, ρ, should scale up by the same factor.

$$\forall\ \lambda \geq 0 \quad \rightarrow \quad \rho(\lambda P_1) = \rho(\lambda P_2)$$

4. Translation Invariance: Adding a risk-free amount, c, to a portfolio should decrease the risk by that amount.

$$\rho(P_1 + c) = \rho(P_1) - c$$

Typically, all 4 axioms should be satisfied in collectively by the suite of metrics used, if one wants to universally consider real-world complexity and diversification effects on their portfolio. While it is ideal for a single metric to satisfy all axioms and be deemed “coherent”, this is largely not the case in the overwhelming amount of practical applications; most metrics may be more suitable for specific contexts or objectives, as is observed in Chapter 5, even if they are not coherent by definition (Roberson, 2019). In the case of EntroPy, the following axioms are satisfied by the selected risk-measures:

- Volatility (Standard Deviation):
  - Positive Homogeneity: If a portfolio is scaled up (e.g., double the investment in every asset), its volatility scales up linearly.

- Value at Risk (VaR):
  - Monotonicity: If one portfolio always has a higher or equal value than another for all possible scenarios, its VaR will be less or equal.
  - Sub-additivity: Not always satisfied. VaR doesn't always account for the diversification benefits between assets. For two portfolios, the VaR of their combined portfolio can be greater than the sum of their individual VaRs.
  - Positive Homogeneity: If a portfolio is scaled up (e.g., double the investment in every asset), its VaR scales up linearly.
  - Translation Invariance: If a risk-free asset is added to a portfolio, the VaR will decrease by the amount of the risk-free asset.

- Downside Risk:
  - Monotonicity: If one portfolio always has a higher or equal value than another for all scenarios below a certain threshold, its downside risk will be less or equal.

  - Sub-additivity: Satisfied especially if downside risk is measured to account for diversification.
  - Positive Homogeneity: If a portfolio is scaled up (e.g., double the investment in every asset), its downside risk scales up linearly.
  - Translation Invariance: If a risk-free asset is added to a portfolio, the downside will decrease by the amount of the risk-free asset.

- Risk-Free Rate of Return:
  - This is not a risk measure, but rather a benchmark rate. As such, it doesn't directly pertain to the axioms of coherent risk measures.

The most notable shortcoming is observed with VaR's lack of sub-additivity, which it violates in scenarios where two individual assets have a lower VaR than a portfolio of both assets combined. This limitation has led to the development of other risk measures, like Conditional Value at Risk (CVaR), which aim to address these shortcomings and better adhere to the principles of coherence. However, within the scope of this report and software framework, solely VaR is implemented for the following reasons:

1. Standardization: Many regulatory bodies and financial institutions have standardized around VaR for reporting and compliance purposes. Implementing VaR ensures compatibility and alignment with industry standards and practices (Manganelli and Engle, 2001).

2. Computational Efficiency: VaR can be more computationally efficient to calculate than other risk measures, especially for large portfolios or when using historical simulation methods (Tamplin, 2023).

3. Flexibility: VaR can be computed using various methods, including the historical simulation, parametric (variance-covariance), and Monte Carlo simulation.

4. Foundation for Further Development: Implementing VaR provides a foundational risk metric upon which additional features and metrics, like Conditional Value at Risk (CVaR), can be added in future software updates, due to the system's modularised design. Starting with VaR allows for the establishment of a risk assessment framework that can be expanded upon as needed (Kidd, 2012).

2.7 Risk-Adjusted Performance Metrics

2.7.1 Sharpe Ratio

The Sharpe ratio, Sh_r, is a pivotal measure that evaluates the performance of an investment, such as a security or portfolio relative to a risk-free asset with an adjustment for its risk. Posited by William Sharpe in 1966, it essentially calculates the extra return an investor can expect for each unit of risk they take on (Sharpe, 1966). It can be defined as the ratio of excess return to standard deviation of a portfolio's return:

$$Sh_r = \frac{R_f - R_{rfr}}{\sigma_P}$$

Where:

- R_f: portfolio forecast return
- R_{rfr}: risk-free rate of return
- σ_P: portfolio standard deviation

In accordance with literature and industry-wide acceptance, this report uses the maximisation of the Sharpe ratio as one of the objective functions in construction an efficient portfolio (Chopra & Ziemba, 1993). The greater the Sharpe ratio, the larger the return per unit risk the investor attains.

The Sharpe ratio does, however, have some limitations. Firstly, it does not differentiate between upward and downward volatility in returns, as seen in the figure below.

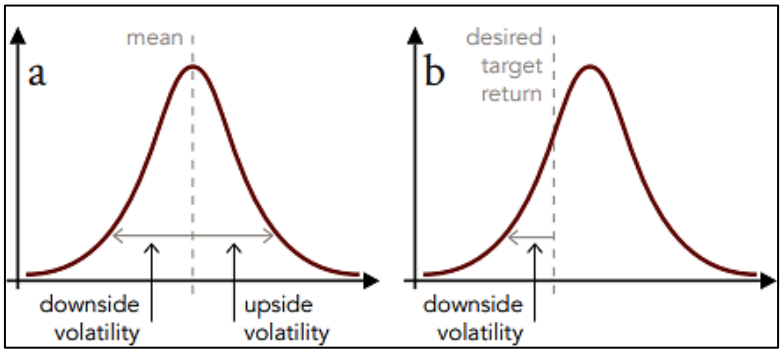


Figure 10: Sharpe ratio measures return divided by upside and downside volatility (Rollinger & Hoffman, 2012).

The Sharpe ratio becomes less reliable when returns do not follow a normal distribution. It is particularly insufficient when contrasting strategies with positive skews, like trend following, to those with negative skews, such as option selling, as shown in below.

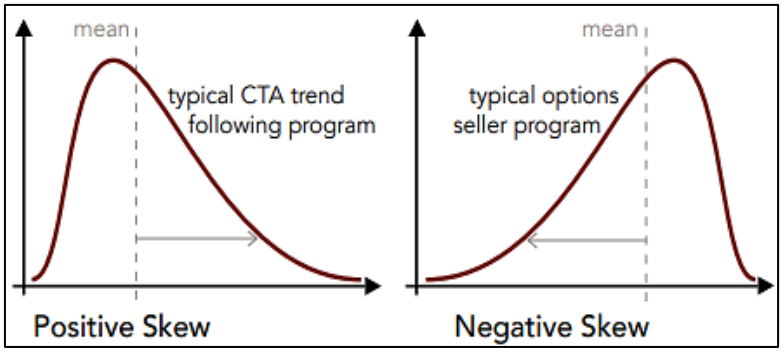


Figure 11: Effect of skewness of Sharpe and upside/downside risk (Rollinger & Hoffman, 2012).

For distributions with a positive skew, the actual risk taken to achieve performance is often less than what the Sharpe ratio indicates. On the other hand, for distributions with a negative skew, the standard deviation tends to downplay the risk, suggesting the strategy is riskier than the Sharpe ratio portrays.

2.7.2 Sortino Ratio

The Sortino ratio is a variation of the Sharpe ratio that evaluate the risk-adjusted return of an investment, portfolio, or strategy. Unlike the Sharpe ratio, which considers both positive and negative volatility, the Sortino ratio focuses solely on negative volatility. The Sortino ratio differentiates harmful volatility from total volatility by using the standard deviation of negative asset returns, called downside deviation. This distinction arises because the Sortino ratio only penalizes returns that fall below a predetermined target or required rate of return. It is given by the equation:

$$So_r = \frac{R_f - R_{rfr}}{DsR}$$

Where: DsR is the Downside Risk or Target Downside Deviation (TDD).

When the return distributions are symmetrical and the target return is close to the median of the distribution, the Sortino ratio and the Sharpe ratio may yield similar results. However, as the

skewness of the return distribution increases and the target deviates from the median, the two ratios can produce vastly different outcomes (Scandinavian Capital Markets, 2021).

In practice, the Sortino ratio is favoured by investors use a lower partial standard deviation (LPSD) instead of the standard deviation. This is because the Sortino ratio, by focusing only on negative volatility, provides a more accurate representation of the risk an investor is exposed to, especially when the primary concern is the potential for losses (Bodie et al., 2017).

2.8 Moving Averages

In finance, the concept of technical analysis employs mathematical methods to anticipate future price levels or trends of a security by examining its past prices and trading volumes. This projection is termed a technical indicator, a type of metric originating from general price movements in a stock or asset. Traders use this indicator as a cue for making investment decisions, with the predicted trend guiding whether to purchase, offload, or retain a security. However, these indicators aren't infallible and can provide inaccurate predictions, known as false signals (Zakamulin, 2015).

A frequently employed technique in technical analysis is the notion of moving averages. This strategy aims to discern trends by eliminating the noise caused by random price variations. It achieves this by calculating an average based on a predetermined set of historical price points. This set's duration is often termed the 'look-back period'. Suppose P_{xt} represents the price of a high-risk portfolio x after time t, w_{xt} is the corresponding weight of P_{xt}, and L represents the duration of the look-back period. The general formula for a weighted moving average for portfolio x, MA_x, is thus given by (Glendrange & Tveiten, 2016):

$$MA_x = \frac{w_{xt}P_{xt} + w_{xt-1}P_{xt-1} + \cdots + w_{xt-L}P_{xt-L}}{w_{xt} + w_{xt-1} + \cdots + w_{xt-L}} = \frac{\sum_{i=0}^{L} w_{xt-i}P_{xt-i}}{w_{xt-i}}$$

In the scope of EntroPy, several different types of moving averages (and an indicator) are integrated to provide a holistic overview of price movements:

- Simple Moving Average (SMA): offers a straightforward approach, averaging out prices over a specified period to smooth out short-term fluctuations and highlight longer-term trends.
- Exponential Moving Average (EMA), takes this a step further by giving more weight to recent prices, making it more reactive to new market information.
- Bollinger Bands (Indicator): incorporate volatility into the mix, providing dynamic overbought or oversold indicators that adapt to market conditions.

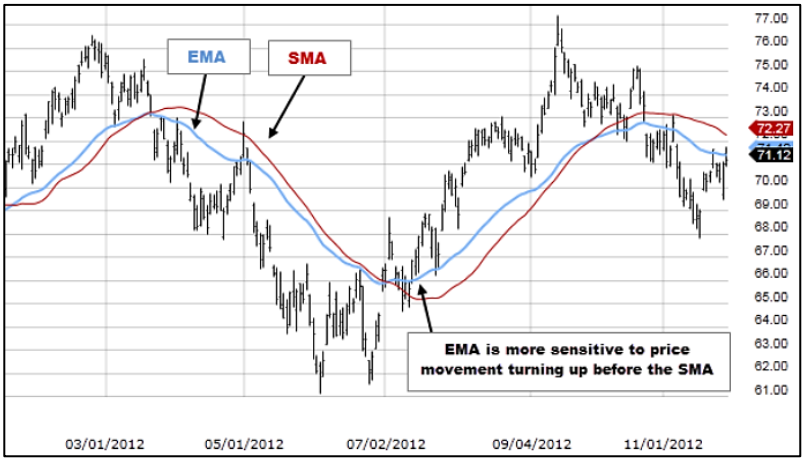


Figure 12: SMA vs EMA sensitivity (Fidelity, 2023).

2.8.1 Simple Moving Average

The Simple Moving Average (SMA) is a method used to determine the average price of a security over a specified number of periods. It's achieved by summing up the prices for a set number of periods and then dividing by that number. SMA is calculated via the following equation:

$$SMA = \frac{1}{L+1}\sum_{i=0}^{L} P_{xt-i}$$

Where: P_{xt} is the price of the asset x at time t, and L is the look-back period or number of periods over which SMA is computed.

SMA is a beneficial tool in trading, but it is essential to identify its limits. This indicator can be influenced by sudden price spikes, leading occasionally to misleading cues. By treating all prices uniformly, the SMA fails to differentiate between high and low volatility data, even though such distinctions can offer crucial insights into market directions (FXOpen, 2023). This characteristic makes the SMA less agile in detecting shifts in trends, positioning it more as a tool for affirming trends rather than forecasting them.

2.8.2 Exponential Moving Average

The Exponential Moving Average (EMA) emphasizes more on the latest data points. This makes the EMA more sensitive to recent price shifts compared to SMA. EMA is given by:

$$EMA = \frac{\sum_{i=0}^{L} \lambda^i P_{xt-i}}{\sum_{i=0}^{L} \lambda^i}$$

Where: λ^i is the weighting factor, which determines the degree of exponential decay. It is a value between 0 and 1. As λ^i approaches 1, the weight of recent prices will increase, and the faster the EMA will adapt to price changes. λ^i operates as a smoothing measure, and is adjustable to the investor's required sensitivity, making it preferable for investors focused on swift higher-frequency trading gains.

However, the very sensitivity of the EMA to price fluctuations can sometimes lead to predictions experiencing recency bias. This bias might cause investors to stay invested during a thriving bull market longer than advisable, overlooking signs of potential downturns. Similarly, during bear markets, their pessimism might deter them from seizing opportunities, doubting the market's potential to rebound (Richards, 2012). Given this potential for inaccuracies, it's always practical for traders to implement a stop-loss strategy when leveraging the EMA to map out their trading moves.

2.9 Bands of Moving Averages & Bollinger Bands

Merging two moving averages can lead to the formation of an envelope. This envelope is composed by positioning one moving average at the peak of the price range and the other at its trough. This configuration sets the upper and lower limits of a asset's typical trading range. The band around the security's price mirrors its theoretical trading scope. Much like how tracking error showcases prices relative to a specified benchmark, these bands depict volatility in relation to the moving average price.

These trading bands, delineated both above and below the price range, are designed to gauge the likelihood of breaching these boundaries. This concept parallels how VaR metrics assess the probability of a portfolio losing a specific percentage of its value within a certain confidence interval.

When a security touches the upper limit of its band, it indicates a potential price peak, hinting at an imminent turnaround. Conversely, if the security's price reaches the lower limit, it signals a

buying opportunity. In subsequent Chapters, the implementation of these bands for SMA and EMA are observed.

Bollinger Bands (BBands) are a type of statistical chart that represents the prices and volatility of a financial instrument or commodity over time. BBands are composed of three lines:

1. Middle BBand: SMA or EMA of closing prices, P_i, typically of $n = 20$ moving average periods:

$$midBB = \frac{\sum_{i=1}^{n} P_i}{n}$$

2. Upper BBand:

   This involves transforming the asset price's moving average by a standard deviation(s), usually $\sigma = 2$, and adding this transformed moving average from the middle BBand.

$$upperBB = midBB + \left[\sigma \times \sqrt{\frac{\sum_{i=1}^{n}(P_i - midBB)^2}{n}}\right]$$

3. Lower BBand:

   Similar to the upper BBand, this involves transforming the asset price's moving average by a standard deviation(s), usually $\sigma = 2$, and instead subtracting this transformed moving average from the middle BBand.

$$lowerBB = midBB - \left[\sigma \times \sqrt{\frac{\sum_{i=1}^{n}(P_i - midBB)^2}{n}}\right]$$

There is a plethora of indicators that may be used, and the choice of indicator here is rather arbitrary, as most quantitative traders will use a bespoke and sophisticated combination of many indicators to have a winning strategy (Salkar et al., 2021). Evidently this is not within the scope of EntroPy. Therefore, while the selection of BBands as the indicator of choice is somewhat stylistic, it is still important to consider their advantages and limitations (Belyaev, 2021):

Table 1: Evaluation of Bollinger Bands as a sole technical indicator.

Advantages	Limitations
Versatile, suitable for any asset and across various time frames.	Relying solely on BBands can be risky; it is advisable to concurrently use their signals with other technical indicators (MACD, RSI, etc.)
Signals generated by BBands not only pinpoint potential entry points, but also suggest areas for stop-loss and profit-taking.	While BBands can highlight extreme price levels, they are not optimal for this specific task.
Offer a clear and straightforward visual representation.	Tend to be more effective in range-bound markets than in strongly trending ones.
Serve a dual purpose: indicating volatility and acting as a momentum oscillator.	

3 Literature Review

This chapter will discuss three primary frameworks and sets of algorithms pertaining to portfolio optimisation, while encompassing all relevant metrics covered thus far. Primarily, this review will start by identifying by main objective functions for optimisation using the EntroPy framework. This is essential to elucidating why specific methods are employed to achieve a given optimisation criterion. These objective functions underpin the main principles of managing a user's portfolio.

Following this, the Literature Review will discuss the predominant methods of Mean-Variance Optimisation, Monte Carlo Simulations, and the Efficient Frontier in allocating portfolios. Mean-Variance Optimisation is provided as a conceptual foundation to the Efficient Frontier. A detailed exploration of the underlying workings of each implementation of these frameworks / algorithms can be found in Chapter 5.

3.1 Objective Functions for Optimisation

Irrespective of the technique of optimisation that is used within EntroPy, the principle of diversification remains a cornerstone in risk mitigation. Yet, its impact lessens as the number of assets in a portfolio grows. Generally, the optimal diversification benefit is observed in portfolios containing approximately 20 assets (Kull, 2014). This underscores that while diversification can curtail risk, its efficacy has limits.

Optimisation can be conducted in several ways, but each involves adjusting the weights $(w_1, w_2, \dots, w_n)$ of different stocks within a portfolio, P. A few optimisations, of which (3), (4), (5), and (6) are implemented in EntroPy in some capacity, include the following (Nguyen, 2019). Functions (1) and (2) are more general and apply to the overall goal of the system:

1) Maximising the forecast return

$$P = \underset{w:\ \sum_{i=1}^{n} w_i + constraints}{\operatorname{argmax}} \quad R_f$$

2) Maximising a general risk-adjusted return

$$P = \underset{w:\ \sum_{i=1}^{n} w_i + constraints}{\operatorname{argmax}} \quad R_f - \tau\sigma_P^2$$

3) Minimising the volatility

$$P = \underset{w:\ \sum_{i=1}^{n} w_i + constraints}{\operatorname{argmin}} \quad \sigma_P$$

4) Minimising the volatility for a specified objective return

$$P = \underset{w:\ \sum_{i=1}^{n} w_i + constraints}{\operatorname{argmax}} \quad \sigma_P$$

Subject to:

1. $W^T \hat{R}_f = R_{obj}$: ensures that the portfolio return is equal to the objective return.
2. $\sum_{i=1}^{n} w_i = 1$: ensures that the portfolio weights sum up to 1.
3. $w_i \geq 0 \forall i$: ensures non-negative weights, which means no short selling. This is optional.

5) Maximising the Sharpe ratio

$$P = \underset{w:\ \sum_{i=1}^{n} w_i + constraints}{\operatorname{argmax}} \quad \frac{R_f - R_{rfr}}{\sigma_P}$$

6) Maximising the Sharpe ratio for a specified objective volatility

$$P = \underset{w:\ \sum_{i=1}^{n} w_i + constraints}{\operatorname{argmax}} \frac{R_f - R_{rfr}}{\sigma_P}$$

Subject to:

1. $\sigma_P = \sigma_{obj}$: ensures that the portfolio volatility is equal to the objective volatility.
2. $\sum_{i=1}^{n} w_i = 1$: ensures that the portfolio weights sum up to 1.

Where:

- R_f: forecast return
- σ_P: portfolio volatility
- τ: risk-aversion coefficient
- $\hat{R}_f$: vector of forecast returns
- W: vector of asset weights
- R_{rfr}: risk-free rate of return

3.2 Mean-Variance Optimisation

Over six decades have elapsed since Markowitz introduced Mean-Variance Optimisation (MVO) for portfolio management. Foundational tenets of the mean-variance framework are elaborated upon in numerous academic sources, such as Fabozzi (Fabozzi et al., 2007). These tenets include:

- The foundational belief of the MPT mean-variance model posits that an investor's inclinations are encapsulated by a utility function, which considers two moments of portfolio returns: the forecast return and the variance of portfolio.
- Central to MVO is the idea that, given a specific expected return, a logical investor will opt for a portfolio with minimised variance from the feasible portfolio options.

Although the methodology has gained considerable traction among financial professionals, it has not been without its detractors in academia (Tu and Zhou, 2011). Many argue that the framework often falls short of reflecting real-world scenarios.

The initial critique to be discussed centres on the use of variance as a stand-in for risk. While this overview might be seen as a simplified rendition of their detailed example, Hult et al. highlight that both location (mean) and dispersion (variance) serve as apt indicators of potential reward and associated risk when returns closely follow a normal distribution (Hult et al., 2012). However, they emphasize that since variance encompasses a broad spectrum of potential deviations from the mean, it might not appropriately represent the risk factor if, for instance, the return has an asymmetrical distribution. This can be visualised in the Figure below (Marakbi, 2016):

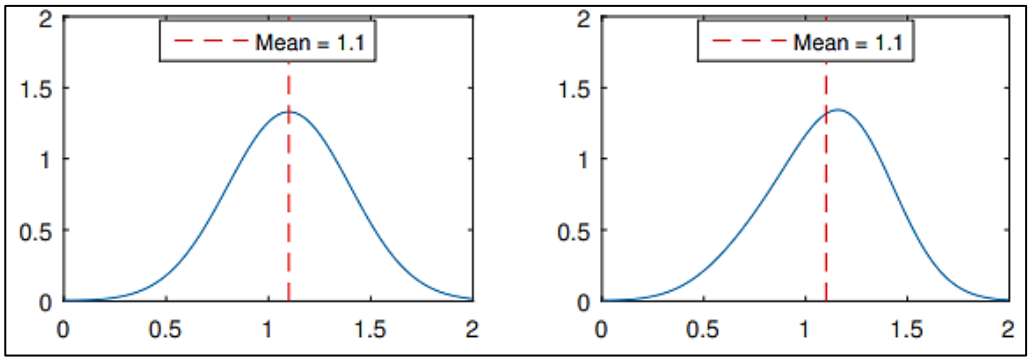


Figure 13: Density functions for a random portfolio P, with Forecast Return, Rf = 1.1 and Variance, Var(P) = 0.32. LHS portfolio is N-(1.1, 0.32)-distributed, RHS portfolio is distributed via a 2-point fusion of normal distributions.

From a mean-variance viewpoint, both profiles in the figure above appear identical due to their matching mean and variance. Yet, when considering downside risk, the two profiles differ

significantly because of the skewed nature of the density function on the right. Since variance uniformly measures deviations from the average, it does not effectively differentiate between various return distributions.

The other critique of MVO pertains to the issue of estimation error. To effectively utilize MVO, it is essential to estimate asset returns' means and covariances since these moments are not inherently known. These approximations are then used to derive a solution for the investor's optimisation challenge (Grootveld and Hallerbach, 1999). These studies highlight a significant limitation of the mean-variance method: the potential error in these estimated moments. This issue stems from the optimiser's assumption that the inputs are precise values rather than statistical estimates.

In the process of estimating asset means and return covariances, the traditional method involves analysing historical return data to compute sample estimates. This method operates on the belief that past data can offer insights into future performance. Yet, the literature has consistently highlighted the pitfalls of using sample estimates in portfolio contexts. For example, Frankfurter, et al. contend that portfolios optimised using MVO with sample estimates do not necessarily surpass a portfolio with equal weightings, often termed the "naïve portfolio" (Frankfurter et al., 1971). Furthermore, Best and Grauer demonstrated that the estimation error from parameter estimates is carried over to the portfolio weights derived from optimisation (Best and Grauer, 1991). As a result, the estimated optimal weights will likely differ from the truly optimal ones.

Ledoit and Wolf suggest that the subpar outcomes of MVO portfolios arise from error-burdened sample estimates (Ledoit and Wolf, 2003). Here, the most extreme sample coefficients often exhibit extreme values, not necessarily reflecting reality but rather the inherent error. They believe that MVO tends to gravitate towards these extreme coefficients, assigning them the most significant portfolio weights. This observation forms the basis for Michaud's critique, where he labelled the portfolio optimisers introduced by Markowitz as “estimation error maximisers, terming this issue "Markowitz’s optimisation enigma” (Michaud, 1989). Michaud's critique suggests that when MVO incorporates imprecise estimates, asset managers might not fully showcase their genuine asset-selection skills, resulting in less-than-ideal portfolios.

Despite the limitations of MVO highlighted above, its inclusion in the software remains justified for being a systematic approach to portfolio construction, offering a foundation upon which more complex models can be built. Its principles have been widely adopted and recognized in the financial industry, making it a familiar tool for many professionals (Rigamonti, 2020). By incorporating MVO, users are granted the flexibility to use it as a starting point, and then apply modifications or combine it with other methodologies to address its limitations.

3.3 The Markowitz Efficient Frontier

The Markowitz Efficient Frontier (MEF) is a product of MVO. MEF represents a curve that displays the optimal portfolios in terms of risk and return. It signifies the portfolio that offers the highest expected return for a specific level of risk, or conversely, the one that has the least risk for a set expected return. Investors typically gravitate towards these optimal portfolios. When an investor has a particular risk tolerance in mind, it is only rational to seek the maximum potential return for that risk level.

A general MEF is depicted in the figure below:

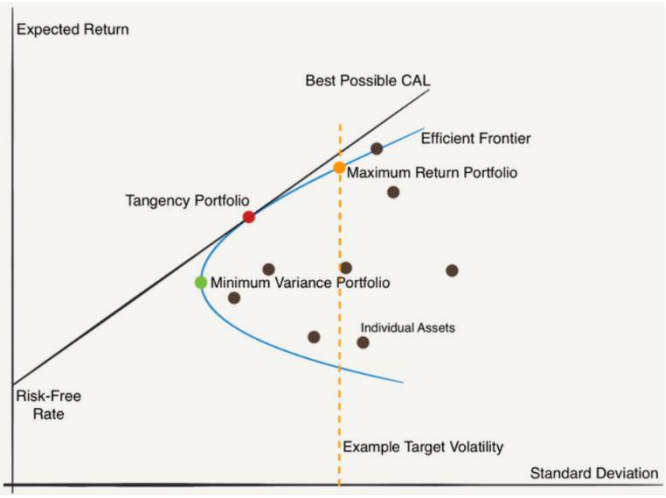


Figure 14: A generalised efficient frontier (Quantpedia, 2019).

The “Efficient Frontier is a curve that illustrates the various combinations of assets leading to the most efficient portfolios. These portfolios either have the least risk for a given return or the highest return for a given risk. Risk is plotted on the x-axis, while return takes the y-axis. The area enclosed by the efficient frontier, but not directly on it, showcases either individual assets or their combinations that are considered sub-optimal.

The “Tangency Portfolio” signifies the portfolio with the peak Sharpe ratio. Moving away from this point in either direction along the frontier results in a diminished Sharpe ratio, meaning a reduced return-to-risk ratio. The inflexion point on the hyperbola, where it shifts from being convex to concave, is the location of the minimum variance portfolio. For any given risk level, there's also what's termed the “Maximum Return Portfolio”. As the name implies, this portfolio offers the highest return for its level of risk.

The point where the “Capital Market Line” (CML) intersects the vertical axis represents the return of a risk-free asset, such as government bonds. The CML touches the efficient frontier precisely at the Maximum Sharpe portfolio point. This tangency CML symbolizes various compositions of a risk-free asset and the tangency portfolio, often referred to as the maximum Sharpe portfolio or simply the "optimal portfolio."

To formalise an MEF, consider a canonical portfolio problem, where an investor is aiming to invest a portfolio with n assets. First, the elementary metrics must be defined, Given that:

- σ_i: volatility for asset i
- w_i: weight of asset i
- R_{fi}: forecast return of asset i

The asset weights, W, the forecast returns, R_f, and the covariance matrix, Σ , are given in vectorised notation as:

$$W = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}, \quad R_f = \begin{bmatrix} R_{f1} \\ \vdots \\ R_{fn} \end{bmatrix}, \quad \Sigma = \begin{bmatrix} \sigma_{11} & \cdots & \sigma_{1n} \\ \vdots & \ddots & \vdots \\ \sigma_{n1} & \cdots & \sigma_{nn} \end{bmatrix}$$

The portfolio volatility, σ_P, and portfolio return, R_P, can therefore be defined as:

$$\sigma_P = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} \sigma_{11} & \cdots & \sigma_{1n} \\ \vdots & \ddots & \vdots \\ \sigma_{n1} & \cdots & \sigma_{nn} \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}, \quad R_P = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} R_{f1} \\ \vdots \\ R_{fn} \end{bmatrix}$$

The MEF can be defined by two distinct criteria:

1. Minimisation of the portfolio variance (volatility) spanning across lowest to highest returns
2. Maximisation of the investor utility function based on a tolerance factor (τ)

EntroPy directly computes MEF using both of these approaches, as defined in the previous "Optimisation" section. The first approach is tied to minimising volatility for a specified objective return. The second objective of maximising the Sharpe ratio is a special case of maximising a utility function, as both methods consider an investor's preference for return relative to risk.

As it is not strictly relevant to this report, and for the sake of brevity, the derivation of lengthy equations using Lagrangians for both of these approaches is omitted, but follows a similar logic to works by Jiao (Jiao, 2003). In essence, Jiao's relationship for the portfolio variance is given by:

$$W^T \Sigma W = \frac{\alpha}{\alpha\gamma - \beta^2}\pi^2 - \frac{2\beta}{\alpha\gamma - \beta^2}\pi + \frac{\gamma}{\alpha\gamma - \beta^2}$$

Where α, β, and γ are constants given by:

$$\alpha = I^T \Sigma^{-1} I, \quad \beta = R_f \Sigma^{-1} I, \quad \gamma = R_f^T \Sigma^{-1} R_f$$

And where:

- I: column vector of ones. Length = number of portfolio assets
- Σ^{-1}: inverse of the covariance matrix

Therefore, the portfolio comprising a minimal risk will have co-ordinates $\left(\frac{1}{a}, \frac{b}{a}\right)$. Similarly, the maximisation of the utility function can be expressed by (Jiao, 2003):

$$W_{opt} = \Sigma^{-1}\alpha^T(\alpha\Sigma^{-1}\alpha^T)^{-1}\beta + \tau\Sigma^{-1}\left(R_f - \alpha^T(\alpha\Sigma^{-1}\alpha^T)^{-1}\alpha\Sigma^{-1}R_f\right)$$

Where:

- A: m × n matrix (no. equality constraints × no. of assets).
- β: m × 1 vector of limits.
- τ: coefficient for risk aversion (optional)

The return of the portfolio, σ_P, can be expressed as:

$$R_P = W_{opt}^T R_f$$

The volatility, σ_P, (standard deviation) of the portfolio can be expressed as:

$$\sigma_P = \sqrt{W_{opt}^T \Sigma W_{opt}}$$

Recall that the Sharpe Ratio, i.e. the variable to be maximise in the objective utility function, is given by:

$$Sh_r = \frac{R_P - R_f}{\sigma_P}$$

Substituting the above portfolio return and volatility equations into the Sharpe ratio formula:

$$Sh_r = \frac{W_{opt}^T R_P - R_f}{\sqrt{W_{opt}^T \Sigma W_{opt}}}$$

This equation gives the Sharpe ratio in terms of the optimal weights W_{opt} from the utility function, the covariance matrix Σ, and the vector of expected returns R_f.

3.4 Monte Carlo Simulation

Monte Carlo Simulations (MCS) can help address the estimation error problem inherent in MVO. By running thousands (or even millions) of simulations based on a range of input values (e.g., expected returns, volatilities, and correlations), Monte Carlo provides a distribution of potential outcomes rather than a single point estimate. Furthermore, one of the critiques of MVO is its reliance on the assumption of normally distributed returns. Monte Carlo simulations can be designed to account for non-normal return distributions, such as those with fat tails or skewness, providing a more realistic representation of potential portfolio outcomes.

Monte Carlo Simulations have been outlined previously, but are essentially a wide categorisation of computational algorithms that utilise repeated-random sampling to solve deterministic problems, or any problem that has a probabilistic interpretation. Hence, it is very useful in cases where the probability distribution cannot be analytically derived, either due to complexity, or due to the dynamic behaviour of stochastic various derived from the models (Pedersen, 2014).

A MCS varies in its execution, but it generally aligns with the following 6 steps (Leonelli, n.d.):

1. Define a model

   The model is implicitly defined by the relationships between asset returns, their volatilities, and other relevant metrics. For instance:

   $$R_P = f(W, R_f, \Sigma_c)$$

   Where:

   - R_P: portfolio return
   - W: vector of portfolio weights (allocations)
   - R_f: average asset returns vector
   - Σ_c: covariance matrix of asset returns

2. Specify input distributions

   The input return distributions of a given asset are implicitly assumed to be based on the historical asset revenue data:

   $$R_{dist} \sim Historical\ Data$$

3. Generate random inputs

Random samples are drawn from the input distributions to generate portfolio allocations. This is usually conducted using a normal or a uniform distribution:

$$f_{norm}(x|R_f, \sigma_v^2) = \frac{1}{\sqrt{2\pi\sigma_v^2}} e^{-\frac{(x-R_f)^2}{2\sigma_v^2}}$$

$$f_{uni}(x|a,b) = \frac{1}{b-a} \qquad for\ a \le x \le b$$

With subsequent normalisation of the weight values, this is expressed as:

$$W_i = \frac{random\ weight}{\|random\ weight\|_1}$$

Where:

- σ_v: portfolio volatility
- W_i: normalized weight (or proportion) of the i^{th} asset in the portfolio
- $random\ weight$: i^{th} randomly generated value from the uniform distribution for the i^{th} asset
- $\|random\ weight\|_1$: $L1$ norm (or Manhattan norm) of the random weight vector, which is simply the sum of the absolute values of its components

4. Perform deterministic computation

   For each set of random inputs (portfolio asset weights), the portfolio's relevant annualised return and volatility metrics, and Sharpe ratio are computed. I.e., for the aforementioned:

$$R_i = W_i \times R_f$$

$$V_i = \sqrt{W_i^T \times \Sigma_{cov} \times W_i}$$

$$S_i = \frac{R_i \times R_{rfr}}{\sigma_i}$$

   Where:
   - R_i: annualised return for the i^{th} MCS iteration
   - V_i: volatility for the i^{th} MCS iteration
   - W_i^T: transposed portfolio weight vector for the i^{th} MCS iteration
   - S_i: Sharpe ratio for the i^{th} MCS iteration
   - R_{rfr}: risk-free rate of return

5. Repeat steps 3 – 4 for a specified number of iterations

6. Analyse results

   The results of the MCS are analysed to identify portfolios with specific characteristics. For instance:

$$i_{\text{min\_vol}} = \text{argmin}(V)$$

$$i_{\text{max\_sharpe}} = \text{argmax}(S)$$

   Where:

   - $i_{\text{min\_vol}}$: index of the portfolio with the minimum volatility
   - $i_{\text{max\_sharpe}}$: index of the portfolio with the maximum Sharpe ratio

With this understanding, the focus will now pivot to a comprehensive evaluation of MCS in the broader academic literature (Cheng et al., 2023). This will allow contextualization of its use, understanding of its evolution, and an appreciation of its significance in modern financial modelling.

Alrabadi and Aljarayesh were among the first to explore the potential of Monte Carlo simulations in forecasting stock returns on the Amman Stock Exchange (Alrabadi & Aljarayesh, 2015). Their study, spanning from 2003 to 2012, juxtaposed the predictive effectiveness of Monte Carlo simulations with traditional and exponential moving average methods. They employed four distinct metrics to gauge prediction precision: root mean square error (RMSE), mean absolute error (MAE), mean absolute percentage error (MAPE), and Theil's inequality coefficient. Their findings underscored the superior accuracy of Monte Carlo simulations over the other techniques evaluated.

Tan subsequently delved into the repercussions of the COVID-19 outbreak on financial markets and portfolio strategy (Tan, 2021). By analysing data from six tech stocks in the US market, he created the ideal portfolio and delineated the efficient frontier through the Markowitz framework, employing >1000 MCS cycles. This approach facilitated a comprehensive analysis of the portfolio's optimal condition pre and post-pandemic. The study illuminated the pandemic's influence on investment decisions and offered insights for prospective portfolio design and management.

Within portfolio risk management, discerning the risks tied to various stock market assets stands as a paramount concern. A prominent method to gauge, forecast, and handle such risks involves the use of Value-at-Risk (VaR), as discussed previously. This approach leverages diverse statistical methods to quantify the associated risk. Osei, et al. juxtaposed the historical simulation method with the Monte Carlo technique to ascertain VaR (Osei, et al., 2018). They utilized these methods on six unique shares from the Ghana stock market, considering confidence intervals of 95% and 99%. Their findings suggested that the Monte Carlo approach stood out as the superior method for VaR estimation. While this method boasts adaptability as a primary strength, its implementation demands time and relies on advanced hardware infrastructure.

Pasieczna leveraged MCS to gauge VaR (Pasieczna, 2019). Moreover, MCS was utilized to determine model parameters by aligning actual historical data across various durations to specific probability distributions. The analysis was conducted for value at risk with confidence levels of 95% and 99% across six projected periods ranging from 1 – 250 trading days. The findings underscored that while the method's effectiveness is predisposed to the selection of historical durations and prior estimations, it remains a dependable approach for VaR quantification.

Finally, Lee devised strategies to optimise the interest rate for stock market investments (Lee, 2019). This research introduced a fusion of the Markowitz framework and MCS, which was subsequently benchmarked against the prevailing interest rate. The suggested MCS approach showed promise in motivating investments by improving interest in share issuances.

4 Project Specification and Design

4.1 Project Overview

As previously explored, portfolio management is not merely about selecting a group of assets. It additionally involves understanding the intricate relationships between these assets, predicting their future performance based on historical data, and making informed decisions that align with an investor's risk appetite and financial goals. The challenge intensifies when considering the vast array of financial instruments available, each with its unique risk and return profile.

EntroPy emerges as a comprehensive solution to these challenges, offering a suite of tools tailored for portfolio optimisation and as a precursor for algorithmic trading practitioners. At its core, the system leverages two pivotal techniques:

1. Efficient Frontier (Markowitz Model):

Traditional portfolio theories, such as the Markowitz Efficient Frontier, form the bedrock of EntroPy. This model, renowned for its emphasis on the risk-return trade-off, guides investors in determining the optimal portfolio allocation. However, while foundational, relying solely on this model can be limiting.

2. Monte Carlo Simulation:

To address the limitations of traditional models, EntroPy integrates Monte Carlo simulations. This computational technique, by simulating a vast number of potential market scenarios, offers a more holistic view of possible investment outcomes. It provides insights into the range of potential returns and their associated probabilities, enabling investors to make decisions grounded in robust statistical analysis.

In essence, EntroPy will allow the user will be able to generate an object that will retain stock prices for a collection of investments with initial weights, i.e. a portfolio. The package will analyse this data and generate various plots, including moving averages with buy/sell indicators, Bollinger bands, and various return profiles (daily, historical, logarithmic, etc.). Additionally, using Efficient Frontier Analysis and/or a Monte Carlo simulation of the portfolio, the user will be able to view the allocation at which, for example, they maximise return for a given risk. This was detailed in the previously discussed objective functions.

This section aims to give an overview of the functional and non-functional requirements for EntroPy. It also utilises UML techniques to construct key diagrams that outline core functionality and features, including an overview of the system architecture, a class diagram, and a use case diagram.

4.2 Requirements

Due to the extensive scope of EntroPy, the comprehensive list of functional and non-functional requirements for each planned module are detailed in Appendix A. The functional requirements typically describe the services delivered by the system, in response to particular inputs, ultimately to accomplish tasks (AltexSoft, 2021). The functional requirements pertain to the following intended modules:

- Measures (Common, Ratio, and Risk)
- Minimisation
- Efficient Frontier Analysis
- Monte Carlo Simulation
- Technical Analysis (Moving Average and Bollinger Bands)
- Performance (i.e. Returns)

- Investment (Stock and Index)
- Portfolio Optimisation
- Portfolio Composition

Conversely, the non-functional requirements place focus on the behaviour of the system, underpinned by performance, security, reliability, portability, scalability, and usability, among others (Box UK, n.d.). All requirements are outlined using MoSCoW prioritisation, to reach a common understanding with stakeholders on the importance of each requirement (Clegg & Barker, 1994). Specifically, these are categorised as:

- Must have (M): essential requirements that are needed for the minimum viable product
- Should have (S): imperative requirements that are recommended for the system, but not obligatory.
- Could have (C): desirable requirements that may bolster novelty or innovation, but may be excluded.
- Won't have (W): beneficial requirements, but will ultimately not be implemented in the current product version.

4.3 Technology Justifications

4.3.1 Python

Python is a widely used scripting high-level language in quantitative finance, typically for prototyping models in hedge funds and quant divisions within banks (Protasiewicz, 2023a). Scripting languages like Python also greatly ease debugging, and have more flexibility in the types of strategies that can be tested. This idea is centred around Python being a REPL language, that is "Read-Evaluate-Print-Loop" (Chan, 2013). REPL languages allows programmers to type in a mathematical function, which comprises the majority of EntroPy, and have a program directly evaluate it while printing the result. The program then can await the next expression or input. In essence, Python best behaves as a handheld calculator, except it allows a much greater deal of automation and sequential execution than other languages.

Furthermore, Python is free and open-source, and is cross-platform. It also boasts a rich ecosystem of libraries tailored for quantitative finance. Libraries such as NumPy, Pandas, and SciPy provide essential tools for mathematical computations, data manipulation, and statistical analysis. These libraries are optimised for performance, ensuring efficient computations even with large datasets.

Python is used in a variety of difference sectors in the financial services industry, and some example products include (Protasiewicz, 2023b):

- Analytics tools (big data, complex visualisation): Iwoca, Holvi.
- Banking software (payment platforms, online-banking): Venmo, Stripe, Zopa.
- Crypocurrency (insights, predictions): Dash, Enigma, ZeroNet.
- Stock Algotrading (market microstructure, strategies): Quantopian, Zipline, Backtrader.

Compared to traditional languages used to build algorithmic trading systems, such as C++ or OCAML, Python is far less performant, but has a more intuitive syntax and it more readable (Minsky, n.d.). Python is also more flexible in terms of which variables it can operate; arrays, scalars, and strings are essentially managed using a comparable syntax and are passed to functions identically (Wang, 2021). In essence, C++ is superior is execution platforms, while Python is superior in backtesting and non-real-time systems.

4.3.2 NumPy

One library that is extensively used in numerical computing, and thus quantitative finance, is NumPy. NumPy arrays are more memory-efficient than Python lists (Pierre, 2023). In quantitative finance, where datasets can be extremely large, this efficient memory usage ensures that operations are faster and require less storage. With NumPy's vectorised operations, code can be executed element-wise on arrays without the need for explicit loops, leading to faster computations. For instance, calculating the daily returns for a stock can be done in a single line using NumPy arrays.

Furthermore, many quantitative finance algorithms, especially in portfolio optimisation and risk management, require extensive linear algebra operations. While EntroPy does not currently include any explicitly, NumPy provides a comprehensive suite of functions for matrix operations, eigenvalue problems, and singular value decompositions, which are foundational in these areas (QuantStart, 2023).

NumPy offers a wide range of statistical functions, which are crucial in finance for tasks like constructing confidence intervals. For instance, calculating the mean, variance, and standard deviation of returns is straightforward with NumPy, with dedicated functions being used to calculate said measures. As discussed, Monte Carlo simulations are widely used in quantitative finance for option pricing, risk management, and capital allocation, and NumPy provides functions to generate random numbers from various distributions, facilitating these simulations.

4.3.3 Pandas

Pandas is the other primary library used in EntroPy. It is excellent at performing time series analyses, handling data with its DatetimeIndex, facilitating operations like resampling, time zone conversions, and date shifting. The DataFrame structure is also a powerful tool; it is a two-dimensional, size-mutable, and heterogeneous tabular data structure that can hold data with labeled axes (rows and columns). This makes it ideal for financial data tables, such as historical price series. Pandas also automatically aligns data by labels when performing operations between DataFrames or Series.

Furthermore, financial datasets often have missing values, and Pandas provides robust tools for filling missing data, forward filling, backward filling, or using interpolation methods, ensuring continuity in financial calculations. Pandas also offers seamless for merging and concatenating datasets. For calculating moving averages, Pandas allows developers to use rolling windows as an in-built function, which is more efficient and less-error prone that manually coding the calculations (Ranjan, 2023). Finally, Pandas is built on top of NumPy, inheriting its performance benefits.

4.3.4 SciPy

In quantitative finance, optimisation is key. To reduce the probability of error, and to simplify programming overhead, SciPy is used in the project. Specifically, the `optimise` module within SciPy is used extensively for minimisations, such as of the negative Sharpe ratio or the portfolio variance subject to target return constraints. The function responsible for this, `minimize`, allows for both equality and inequality constraints, making it versatile for these scenarios (Saturn Cloud, 2023). This function is also designed for efficiency over a manual implementation. Should future modules require it, `minimize` also has the option of switching between various optimisation algorithms, from gradient-based methods like BFGS or conjugate gradient to trust-region methods. Depending on the nature of the objective function (e.g. convexity, differentiability), quants may choose the most appropriate algorithm.

4.3.5 Matplotlib

Matplotlib is selected as the visualisation tool for this project. It is used primarily due to its comprehensive suite of plotting capabilities tailored for financial data visualisation. It allows users to interpret stock prices, returns, volatilities, and other financial metrics through a range of customizable charts, from time series line plots to distribution histograms (Matplotlib, 2023). The library's seamless integration with Pandas streamlines the visualisation process, enabling direct plotting from DataFrames and Series. Additionally, if required, specialised extensions like mplfinance offer financial-specific plots, such as candlestick charts, vital for technical analysis (Goldfarb, 2023). Beyond its core plotting features, Matplotlib allows users to construct multi-panel plots for side-by-side comparisons and annotate to highlight specific market events.

4.3.6 yFinance

yFinance is specifically designed to fetch financial data from Yahoo Finance, which is one of the most comprehensive free sources of historical stock prices, financial statistics, and other market data (Bland, 2021). Again, yFinance returns data in the form of Pandas DataFrames, which is the standard for data manipulation in Python. This seamless integration allows users to immediately apply statistical analyses, data transformations, or visualisations without the need for additional data wrangling (McKinney, 2020). yfinance offers flexibility in fetching data, and users can retrieve data for specific date ranges, at different intervals (daily, weekly, monthly), and for multiple tickers simultaneously.

4.4 UML Diagrams

The choice of diagrams and models to include in documentation or design phases is heavily influenced by the nature and requirements of the project. For a project, such as EntroPy, that is purely “backend-focused”, certain diagrams become more relevant, while others may not be as applicable. All UML diagrams produced in this report were done so by using Visual Paradigm Online (Visual Paradigm, 2023).

As discussed, three main UML diagrams are incorporated in this report. Their justifications are:

1. System architecture diagram

   For a backend project, understanding how different modules communicate and depend on each other is crucial. It gives stakeholders and other developers a clear picture of the system's structure and how data flows within it.

2. Class diagram

   Given that the project is backend-focused, the class diagram becomes essential. It offers a static view of the system, showcasing the classes, their attributes, methods, and relationships.

3. Use case diagram

   Understanding the system's functionalities and how external entities (like other APIs) interact with it is vital. A use case diagram provides a high-level view of these functionalities and interactions.

The following UML diagrams and visual representations were considered but ultimately not implemented:

1. Sequence diagram

   For systems such as EntroPy, which comprise numerous asynchronous processes and heavy mathematical computations, modelling these interactions can be extremely

challenging. Capturing interactions in a sequence diagram can lead to an overwhelming and cluttered representation.

2. Wireframes and UI omission

   EntroPy operates as a library or package, meaning it does not have a user interface (UI). Wireframes are visual representations of a UI, making them irrelevant and superfluous for backend-only projects.

4.4.1 System Overview / High-Level Architecture

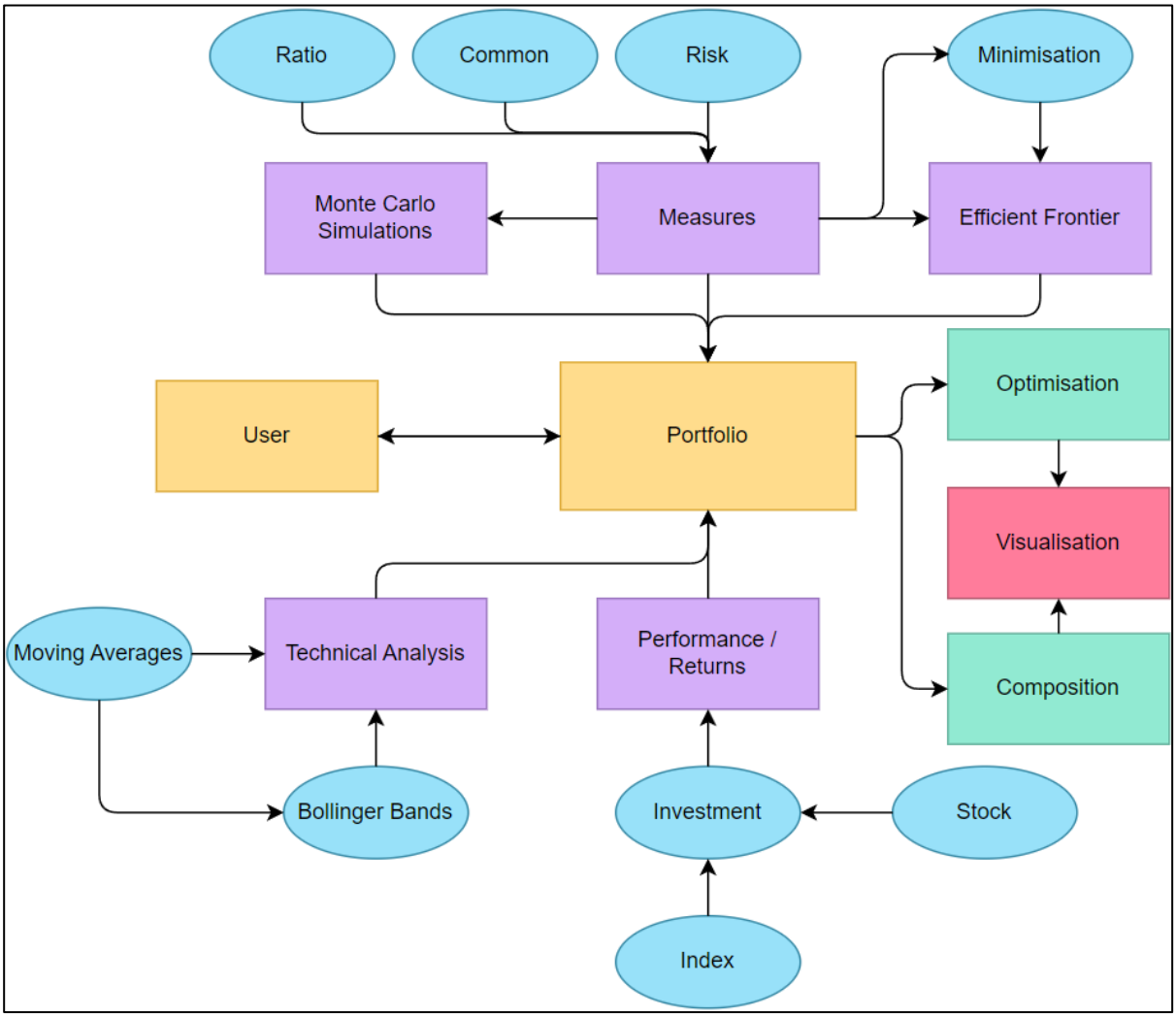


Figure 15: EntroPy system architecture diagram.

EntroPy essentially strives to be a programmatic implementation of all of the theoretical concepts discussed in Chapter 2. The detailed implementation of these will be discussed in Chapter 5. In this section, a brief overview of functionality and dependencies is discussed.

Starting with the “Measures” module, one can see that this is divided into three sub-components, each of which provided utility functions relating to a financial portfolio. The “Common” module includes metrics such as annualised return and volatility, “Risk” defines value-at-risk and downside risk, and “Ratio” outlines functions relating to the Sharpe and Sortino ratios. Annualising metrics is commonplace in finance and may be done to allow for a standardised time frame to compare different investments or portfolios. Annualising also takes into account the compounding effect. For instance, if an investment returns 1% per month, the annualised return is not 12% but higher due to compounding at ~12.68%. Finally, annualising volatility allows investors to understand the

typical year-to-year fluctuations they might expect in a portfolio, which is crucial for benchmarking and ensuring that the investor's investment strategy aligns with their risk tolerance.

As seen for the system architecture diagram, the "Measures" module is utilised by a variety of other modules for their own computations. To begin, it is used in the "Monte Carlo Simulations" (MCS) module, which is performs the MCS to find portfolios with minimum volatility and maximum Sharpe Ratio. A function will be needed in MCS to calculate uniformly distributed random allocations for portfolio stocks and their respective annualised returns, volatilities, and Sharpe Ratios. These annualised metrics will be imported from "Measures".

A "Minimisation" module will also require annualised metrics from "Measures". The main goal of this module, aside from handling tuples for annualised volatility and return, is to determine the inverse Sharpe ratio, which is often used as the objective function in optimisation problems where the goal is to maximise the Sharpe ratio. It will need the annualised Sharpe ratio to return its negative component.

The "Efficient Frontier Analysis" module will depend on functions from both "Measures" and "Minimisation". This module is used to represent a set of portfolios that give the highest expected return for a defined level of risk. Aside from validation and visualisation of results, the "Efficient Frontier Analysis" module will function by optimising portfolios based on 4 criteria:

1. Minimising portfolio volatility
2. Maximising the Sharpe ratio.
3. Achieves a specified target return while minimising volatility.
4. Achieves a specified target volatility while maximising the Sharpe ratio.

Once completed, the module will evaluate the Markowitz Efficient Frontier (MEF) based on provided or generated targets, and plots the optimal MEF points on a graph (i.e. for targets (3) and (4). Furthermore, it will plot optimal portfolios for minimum volatility and maximum Sharpe ratio (i.e. targets (1) and (2)). The annualised metrics from "Measures" are applied to the average return and dispersion matrix for MEF. Similarly, the inverse Sharpe from "Minimisation" is minimised when calculating the maximum Sharpe, and the annualised volatility and return tuples are used for the targeted volatility and return objectives.

Now consider another foundational module, "Performance", which must contain functions to calculate various types of returns based on stock prices. These functions will allow users to analyse stock price data in different ways, such as calculating cumulative returns, daily returns, proportioned daily returns, logarithmic daily returns, and historical average returns. The performance is dependent on directly by the classes that practically characterise a user's assets, i.e. via the "Investment" module. The "Investment" is a generic financial instrument, and will provide methods to compute various financial metrics based on its historical price data, such as daily returns, forecasted returns, volatility, skewness, and kurtosis. Therefore, calculating daily and historical average returns will need to be used from "Performance" to compute daily returns and forecasted returns.

The "Stock" and "Index" modules are a type of asset, and therefore will be implemented as children classes to the "Investment" parent. "Stock" is distinguished from its parent and sibling classes by providing methods to compute the beta coefficient, a measure of the stock's volatility in relation to the overall market. Moreover, "Index" represents a financial index, which is a composite of multiple financial instruments, in this case stocks. It is unique against "Stock" in its need for a specific function that calculates daily returns for the "Index". Indices represent a broader market or sector, and their daily returns provide insights into the overall market's movement. Therefore, it is more relevant to emphasise daily returns in the context of an index rather than an individual stock. Both "Stock" and "Index" will inherit all attributes from "Investment".

The “Technical Analysis” is divided into two modules, one for “Moving Averages” and one for “Bollinger Bands”. “Moving Averages” will focus on simple and exponential moving averages and their standard deviations. It will also contain a main function that orchestrates these, determines buy/sell signals based on crossovers, and visualises the results. “Bollinger Bands” are a technical analysis tool defined by a set of lines plotted two standard deviations (positively and negatively) away from a simple moving average of the asset’s price. It can help determine overbought or oversold conditions in a traded security, and depends on the simple and exponential moving averages from “Moving Averages” to smooth out past price data to create a single flowing line. The area between the upper and lower bands will be shaded to represent the volatility range of stock prices.

The “Portfolio” module will be separated into two modules, “Optimisation” and “Composition”. The “Optimisation” module will essentially import all the aforementioned modules, with a core class to leverage these imported modules to offer a wide range of portfolio optimisation and analysis functionalities. The class is designed to be flexible, allowing users to set various attributes, incorporate stocks, and extract stock details, and it provides methods to calculate various metrics, such as risk, return, and performance ratios. The overall class structure will require an initialisation of various DataFrames, metrics, constants, and instances with respect to the portfolio that will be allotted corresponding functions from the imported modules. These will facilitate portfolio optimisation using Monte Carlo Simulation and Efficient Frontier Analysis.

The “Composition” module must satisfy the following four aims:

1. Fetch stock data for given symbols and a date range (using the yfinance API).
2. Constructs a portfolio using an API, by fetching “Stock” and “Index” modules, and determines the final allocation for the stocks.
3. Constructs a portfolio using provided DataFrames. It prepares by using the “Stock” module, sets the financial index using the “Index” module for the portfolio if provided, and adds each stock to the portfolio.
4. Formulates the final portfolio based on provided arguments. This includes argument validation, assembly, and attribute evaluation.

The “User” ultimately interacts with the “Portfolio” suite of modules by writing scripts to define what they want to achieve using the optimisation techniques within the package. The “Visualisation” module is not a separate component per se, but is integrated into the main “Optimisation” and “Composition” modules. These scripts and visualisations are better illustrated through examples detailed in Chapter 8.

4.4.2 Use Case Diagram

Overleaf, the use case diagram represents an example script, where a user first needs to define the data and initial stock allocations they want to use in two example operations: (1) optimise and visualise a portfolio using the Efficient Frontier and (2) display specific stock attributes for a given stock over a defined time period. These chosen use cases can serve as entry points for users. Once they understand and become comfortable with these primary operations, they can delve deeper into the package's documentation or “scripts” section in the repository to explore the full range of functionalities.

The use case diagram is not intended to capture every possible interaction but rather to give a snapshot that is easy to understand at a glance, adhering to the principle of “progressive disclosure” (Spillers, 2023). In the context of this package for portfolio optimisation, there are a plethora potential operations. Including every single one of them in the use case diagram would not only clutter the visual representation but also make it challenging for users to grasp the core functionalities quickly.

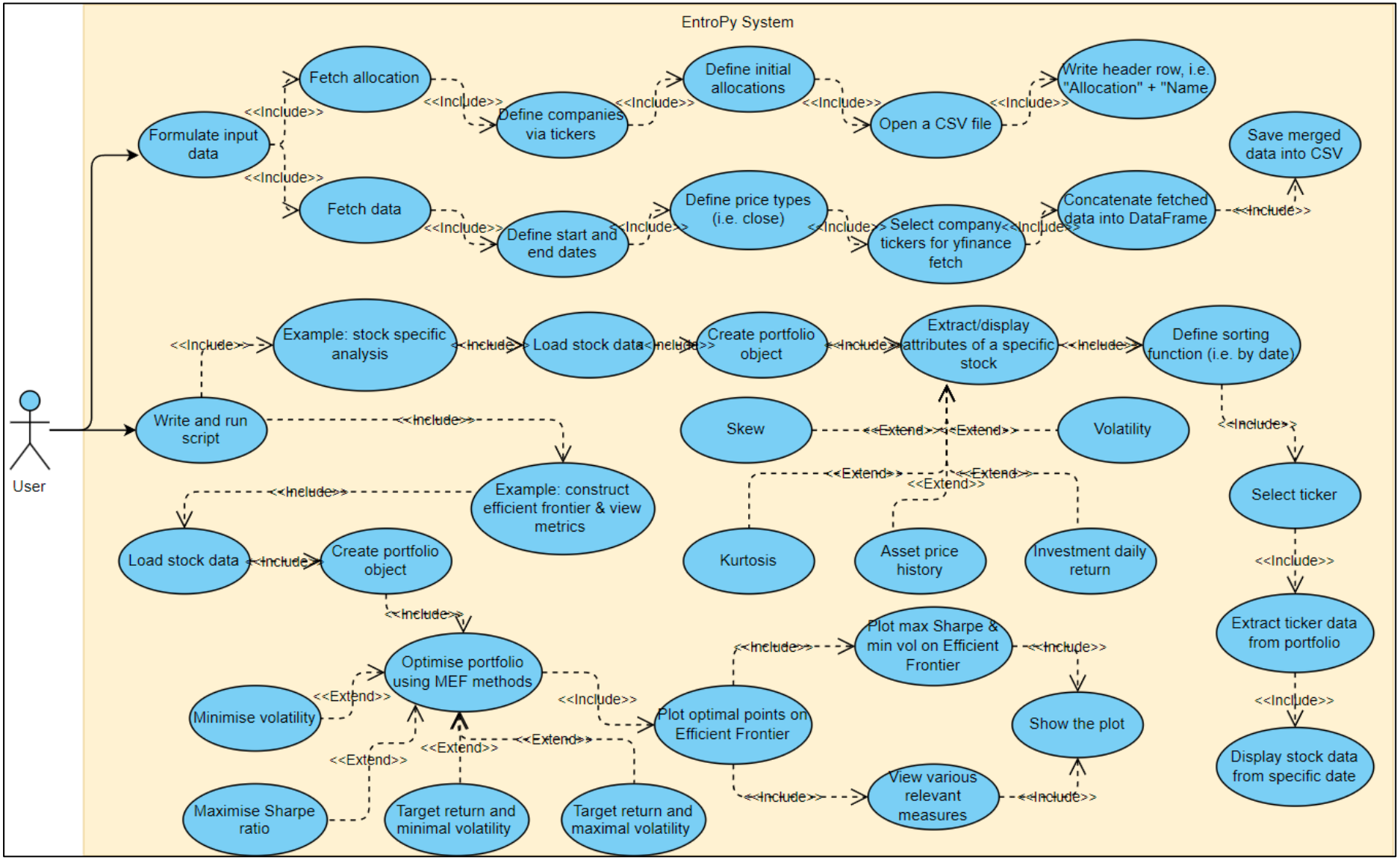


Figure 16: Use case diagram for extracting stock allocations and data to be used in scripts that: (1) optimise and visualise the Markowitz Efficient Frontier, and (2) display various attributes of a specific stock for a given data range.

4.4.3 Class Diagram

Following the system architecture diagram, a class diagram can be constructed. Note that for the Portfolio_Optimised_Functions class, many of the attributes and methods are omitted due to their repetitive and extensive nature. Also, consider the inheritance presents for the MonteCarloSimulation and MonteCarloMethodology classes, and the Investment and Stock/Index classes respectively, where each child extends the parent class. Most multiplicities are one-to-one as specific component (i.e. a Sharpe Ratio) is associated with one specific portfolio. Similarly, a specific Monte Carlo or Efficient Frontier model is tied to one particular portfolio. The only exceptions to this are the Stock/Index and Portfolio_Optmised_Functions classes, which are many-to-one, as a single portfolio can consist of multiple stocks or indices, but each individual stock or index belongs to one and only one portfolio in this context.

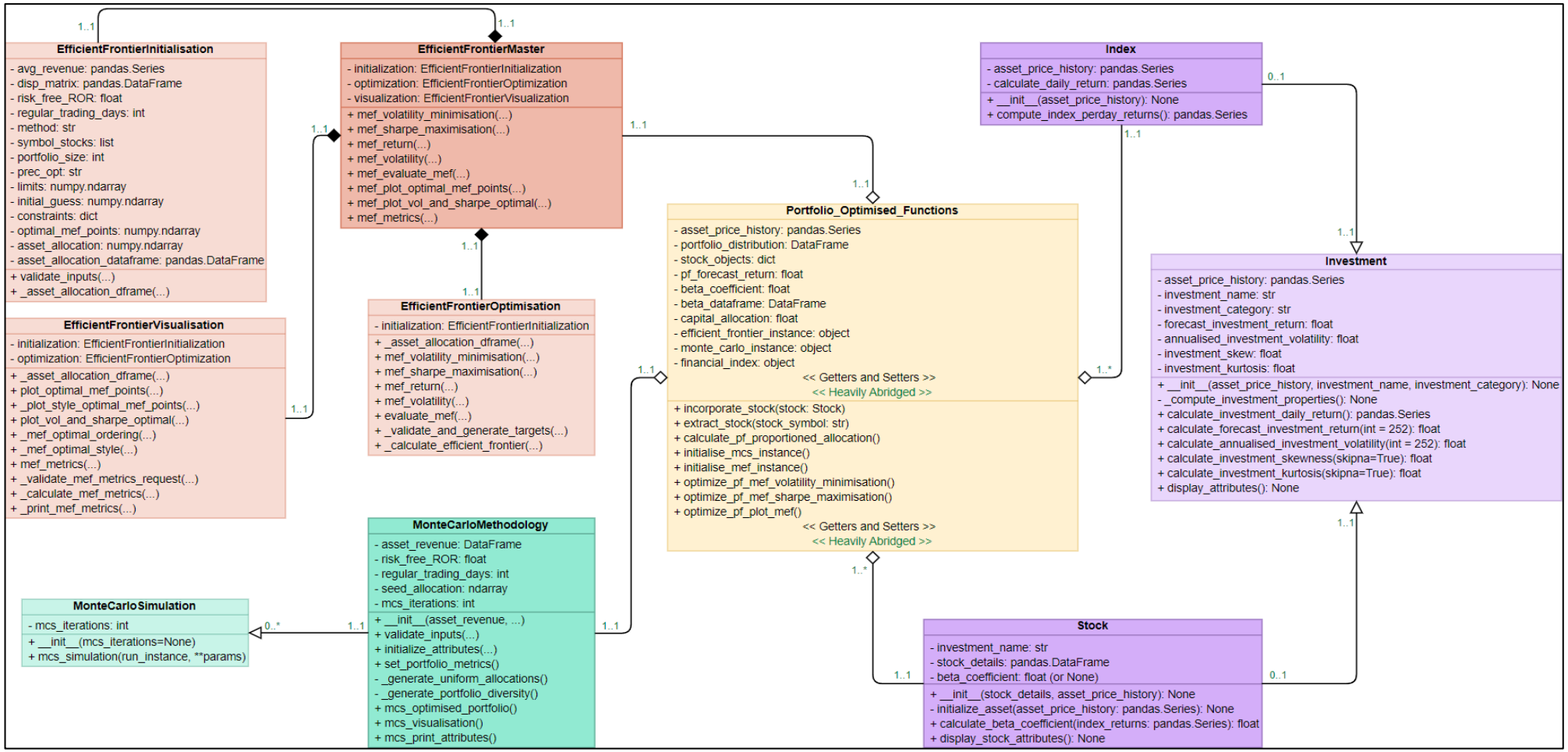


Figure 17: Class diagram for EntroPy.

5 Solution Implementation

In the subsequent implementation discussion, the primary emphasis is placed on the core logic, algorithms and functions underpinning the calculations. This concentration is both intentional and strategic. While validations and visualisations are undeniably essential components of any comprehensive analysis, they often serve as auxiliary tools that enhance or support the main computational logic. For readers who are keen on exploring the intricacies of the validations and visualisations, they are encouraged to refer to the well-documented and commented Git repository, where each module is elaborated in detail. In Chapter 8, there will be a complement to this theoretical exposition with practical scripts that not only run the source code but also showcase the visualizations and insights derived from the core logic.

5.1 Measures

The “measures” module provides reusable functions to calculate various metrics related to portfolio performance. They have been split into 3 files to maintain clean and DRY (Don’t Repeat Yourself) code to be used in subsequent computations to construct a portfolio. The primary data structures used throughout all files are NumPy arrays and Pandas series, which are efficient for mathematical operations and data manipulation.

The `measures_common` file includes stratified averages, portfolio volatility, and annualized measures such as return, volatility, and Sharpe Ratio. One of the foundational functions in this module is `calculate_stratified_average(...).` This function computes the expected returns of a given portfolio. The core of its computation is the line:

```
st_average = numpy.sum(avg_asset_value * proportion_of_investment)
```

Here, an element-wise multiplication is performed between the average asset values and their respective proportions in the portfolio using the `sum` algorithm. The resulting product is then aggregated to yield the stratified average. To ensure data integrity, the function, like most others in this module, incorporates a for-loop to validate that the inputs are either NumPy arrays or Pandas series, safeguarding against potential data inconsistencies.

Building on this foundation, the module introduces the `calculate_portfolio_volatility(...)` function. Its primary objective is to compute the portfolio's volatility. The volatility computation is rooted in the following segment:

```
portfolio_variance = coefficients @ disp_matrix @ coefficients.T
portfolio_volatility = numpy.sqrt(portfolio_variance)
```

Initially, the portfolio variance is determined using matrix multiplication, assigning the dispersion matrix `(disp_matrix)` with the coefficients, which represent the proportion of investments. Subsequently, the volatility is derived as the square root of this variance.

Lastly, the module culminates with the `calculate_annualisation_of_measures(...)` function, a comprehensive utility that furnishes the annualized return, volatility, and Sharpe Ratio of a portfolio. The annualized return is ascertained through:

```
return_factor = regular_trading_days
annualised_return = return_factor * average_return
```

Where the `return_factor` is typically the number of regular trading days in a year, which is 252 days on average (Samuelsson, 2023). For the `annualised_return`, the value of `average_return` is taken from the `calculate_stratified_average` function. The annualized

return is calculated by multiplying the average return by the `return_factor`. This essentially scales the average return to an annual basis.

```
volatility_factor = numpy.sqrt(regular_trading_days)
annualised_volatility = pfolio_volatility * volatility_factor
```

The `volatility_factor` is the square root of the number of `regular_trading_days`. This factor is used to annualize the volatility, as volatility scales with the square root of time. The `calculate_portfolio_volatility` function computes the `pfolio_volatility` based on the dispersion matrix and the proportion of investments. The `annualized_volatility` is derived by scaling the `pfolio_volatility` by the `volatility_factor`.

The final metric, the Sharpe Ratio is calculated in an external function from the measures_ratios module. It is initialised as None and subsequently validated to ensure it is not-zero.

```
sharpe_ratio = None
try:
  sharpe_ratio = calculate_sharpe_ratio(annualised_return,
  annualised_volatility, risk_free_ROR)
except ZeroDivisionError:
    print("Warning: Division by zero… setting Sharpe Ratio to infinity.")
    sharpe_ratio = float('inf')
```

A Sharpe Ratio of zero would suggest that the expected return on the investment equates to the risk-free rate. In the context of evaluating investments, a Sharpe Ratio of zero is not particularly informative. It indicates that the investor is not receiving any additional return for the risk they are assuming compared to merely investing in a risk-free asset. The decision to set the annualized Sharpe Ratio to infinity in such cases aims to signify that the investment offers an exceptionally high risk-adjusted return. In essence, if there is no risk (zero volatility) and the investment yields a positive return, then the risk-adjusted return is infinitely superior to the risk-free rate. However, in practical scenarios, a zero volatility is highly improbable for real-world investments. Should this arise, it might be attributed to inadequate or erroneous data, necessitating further scrutiny.

Finally, the function packages the computed annualized return, annualized volatility, and Sharpe Ratio into a tuple and returns it:

```
annual_measures = annualised_return, annualised_volatility, sharpe_ratio
```

A tuple is selected here for maintaining order and immutability. By returning values in a tuple, the order of the values is preserved, allowing the caller to reliably unpack or access the values based on their position, which is highlighted in later modules. Immutability is advantageous in situations where the user may want to ensure that the returned values remain unchanged.

The next module, `measures_ratios`, is specifically used to compute the Sharpe and Sortino Ratios. The Sharpe Ratio is calculated as the ratio of the excess return of an investment (over the risk-free rate) to its volatility. This is illustrated in the lines:

```
    excess_return = portfolio_return - risk_free_ROR
    sharpe_ratio = excess_return / portfolio_volatility
```

The Sharpe Ratio takes the risk free rate, or risk_free_ROR, as an input argument. For the sake of consistency, this has a default value of 0.005427 (or 5.427%), but may be modified as per the user's preferences (MarketWatch, 2023).

The Sortino Ratio is calculated in a similar way, but instead of taking the portfolio_volatility as the risk benchmark, it focusses more on the downside_risk:

```
excess_return = forecast_revenue - risk_free_ROR
sortino_ratio = excess_return / downside_risk
```

In the Sortino ratio calculation, the 'zip' function is used as a data structure to combine multiple lists into a single iterable. This is a clear use of lists to group related data:

```
for value, name in zip([forecast_revenue, downside_risk,
risk_free_ROR], ["exp_return", "downside_risk", "risk_free_ROR"]):
_validate_sortino_input(value, name)
```

The 'zip' function is excellent at pairing numeric inputs with descriptive string names. Furthermore, instead of writing separate validation checks for each input, zip allows one to write a single loop that can handle multiple inputs. This reduces redundancy in the code and makes it more maintainable. Zip's inherent iterative validation also means that if a value does not meet the expected criteria (i.e. being an integer or float), an error is raised with a descriptive message that includes the name of the variable that caused the error. This makes it easier to identify which specific input was problematic.

Both functions for the Sharpe and Sortino Ratios comprise sub-functions to validate inputs for both native Python numeric types (int and float) and NumPy numeric types (NumPy.number, NumPy.integer, and NumPy.floating). This ensures compatibility with a wide range of numeric inputs.

The final "measures" module is `measures_risk`. This module deals with calculating the value-at-risk (VaR) and downside risk. The VaR is a measure used to quantify the level of financial risk within a firm or investment portfolio over a specific time frame, and is computed in eponymously-titled function, with the first step being:

```
inverse_cdf_value_at_risk = scipy.stats.norm.ppf(1 - confidence_interval)
```

This line calculates the inverse CDF for a standard normal distribution at the given confidence level. The function, `scipy.stats.norm.ppf()`, is the Percent-Point Function (PPF) or the inverse of the CDF, returning the z-score for a given percentile. The argument `1 - confidence_interval` is used because VaR is typically concerned with the left tail (worst losses) of the distribution (Tollander, 2020). For instance, if the confidence interval is 95%, the most interesting region comprises the worst 5% of possible outcomes. This is a common approach in the parametric method of VaR calculation, and assumes that returns are normally distributed for simplicity.

```
diff_value_at_risk = asset_average -
                     asset_volatility * inverse_cdf_value_at_risk
```

Here, the code calculates the difference between the asset's average return and a value derived from its volatility multiplied by the previously calculated z-score. This difference represents the potential loss at the specified confidence level, expressed as a percentage of the asset's value.

```
asset_value_at_risk = asset_total * diff_value_at_risk
```

The final VaR is obtained by multiplying the previously calculated difference with the total value of the asset. This gives the absolute monetary value at risk over the specified time frame at the given confidence level. Multiplying the percentage loss by the total asset value converts the relative risk into an absolute monetary value, which is more tangible and easier to interpret for decision-makers.

The downside risk provides a measure of the potential downside volatility of a portfolio. It focuses only on returns that fall below a certain threshold (in this case, the risk-free rate). The first logic step within the downside risk function is:

```
differences = wtd_daily_mean - risk_free_ROR
```

This line calculates the differences between the weighted daily mean returns of the portfolio and the risk-free rate of return. The result represents how much each daily return exceeds or falls short of the risk-free rate.

```
negative_returns = numpy.minimum(0, differences)
```

Here, the code identifies returns that are below the risk-free rate. The `NumPy.minimum()` function is used to compare each value in the differences array with 0. If a value in differences is positive (above the risk-free rate), the function will return 0 for that value. If it's negative (below the risk-free rate), it will return that negative value. Investors are typically more concerned about the potential for loss than the potential for gain. As previously discussed, there is an asymmetric perception of risk in that the discomfort of losing money is more severe than the desire of gaining the same amount (Schmidt & Zank, 2005). This phenomenon, known as loss aversion, is the reason that computations that focus on downside risk will highlight the negative values.

```
squared_negative_returns = negative_returns ** 2
mean_squared_negative_returns = numpy.mean(squared_negative_returns)
```

Squaring the negative returns ensures that all values are positive and emphasizes larger deviations from the risk-free rate. The code then calculates the mean (average) of the squared negative returns. This value represents the average squared deviation of returns that fall below the risk-free rate.

```
downside_risk = numpy.sqrt(mean_squared_negative_returns)
```

The downside risk is calculated by taking the square root of the mean of the squared negative returns. This value represents the standard deviation of the negative returns and is a measure of the portfolio's downside volatility.

5.2 Minimisation

Minimisation is a precursor to finding optimal values in subsequent optimisation algorithms such as those that manage constructing the Efficient Frontier and allocating investments in the portfolio. This module contains a set of functions that primarily utilise `scipy.optimise.minimize` in subsequent modules to calculate the annualised volatility and return, and the inverse Sharpe Ratio.

The first aspect to consider before using the annualised volatility and return in further modules is normalising the allocations of the different assets within a portfolio to ensure that they add to 1. This is crucial because the proportions represent the fraction of the total portfolio invested in each asset:

```
proportion_of_investment = numpy.array(proportion_of_investment)
proportion_of_investment /= numpy.sum(proportion_of_investment)
```

After normalization, the code calls the `calculate_annualisation_of_measures` function from `measures_common` to compute the annualised metrics:

```
annualised_return, annualised_volatility, sharpe_ratio =
calculate_annualisation_of_measures(proportion_of_investment, avg_revenue,
disp_matrix)
```

The main output of the annualised volatility and return, and inverse Sharpe Ratio functions is:

```
return annualised_volatility, annualised_return, -(min_sharpe_ratio)
```

While the code is simple here, it is important to understand why the following metrics have been selected in minimisation. While the annualised return itself is typically maximised (since investors want the highest return), in some contexts, one might minimise the annualised volatility. For instance, this may be the case in portfolios selected based on their expected returns for a given level of risk, which will be seen in the implementation of the "Markowitz Efficient Frontier" module. Investors often want to minimize risk (volatility) for a given level of return. By minimizing volatility, one aims to find a portfolio that offers the desired returns with the least amount of risk. For annualised volatility, investors often want to minimize risk for a given level of return or maximize return for a given level of risk, which will be discussed in implementations of the Efficient Frontier.

The inverse (or negative) Sharpe ratio is used in minimisation algorithms to effectively maximize the actual Sharpe ratio. This is useful given the design of many algorithms in SciPy focussing on minimisation rather than maximisation to optimise (SciPy, n.d.a). The preference for minimisation over maximisation, aside from historical conventions, is due to its mathematical simplicity. Many optimisation techniques are based on solving for points where the derivative is zero and the second derivative is positive, which is straightforward. Another reason may stem from the concept of duality, where the primal problem is minimisation and the dual problem is maximisation. In linear programming problems, such as ones found in EntroPy, minimisation simplifies the treatment of dual problems (MIT, n.d.).

5.3 Markowtiz' Efficient Frontier

The Efficient Frontier is a foundational concept in Modern Portfolio Theory (MPT), introduced by Harry Markowitz in the 1950s. It represents a set of optimal portfolios that offer the highest expected return for a specified level of risk. Graphically, the Efficient Frontier is a curve on a plot where the x-axis represents portfolio risk (usually measured as standard deviation or volatility) and the y-axis represents portfolio return.

Any point on this curve indicates a portfolio that has the maximum possible return for its level of risk. Conversely, for a given level of return, a portfolio on the Efficient Frontier has the least risk. Portfolios that lie below the Efficient Frontier are sub-optimal because they do not provide enough return for the level of risk they carry. Those that lie to the right of the frontier take on unnecessary risk for the level of return they offer.

Due to its complexity, the implementation of the Efficient Frontier is split into 4 classes, each of which handle different functionality. This greatly improves modularity and allows more segmented testing, as will be seen in Chapter 6. The four classes are located in the file `markowitz_efficient_frontier`, and are briefly highlighted below:

1. `EfficientFrontierInitialization`:

   This class initializes the efficient frontier with various parameters like average revenue, dispersion matrix, risk-free rate of return, trading days, and optimisation method. It validates the provided inputs and sets up initial conditions for portfolio optimisation, such as constraints and initial guesses for asset allocations.

2. `EfficientFrontierOptimisation`:

   This class handles the optimisation of the portfolio based on different criteria:

   - Minimizing volatility.
   - Maximizing the Sharpe ratio.
   - Achieving a specified target return while minimizing volatility.

- Achieving a specified target volatility while maximizing the Sharpe ratio.

It uses the scipy.optimise.minimize function to perform the optimisation based on the provided criteria.

3. `EfficientFrontierVisualization`:

   This class is responsible for visualizing the results of the portfolio optimisation. It can plot the efficient frontier, the optimal portfolios for minimum volatility and maximum Sharpe ratio, and display metrics related to the optimised portfolio.

4. `EfficientFrontierMaster`:

   This is a master class that integrates the initialization, optimisation, and visualization of the Efficient Frontier. It provides a unified interface to work with the efficient frontier, making it easier for users to perform portfolio optimisation and visualize the results without dealing with the individual components separately.

For the sake of brevity, this discussion on the implementation of the Efficient Frontier will focus on calculations, functionality and overarching design choices. However, it is important to note that each function has a robust error handling associated with it, which can be viewed on the associated GitLab repository in more detail.

5.3.1 Efficient Frontier – Initialisation

The three main functions to focus on in initialising the Efficient Frontier are:

1. `__init__`
2. `validate_inputs`
3. `_asset_allocation_dframe`

The `__init__` method serves as the constructor method for the class, which is called when an instance of the class is created. It expects the following input parameters:

- `avg_revenue`: Expected average returns for the assets.
- `disp_matrix`: A matrix representing the dispersion of returns between assets.
- `risk_free_ROR`: The risk-free rate of return (default is 0.005427).
- `regular_trading_days`: The number of trading days in a year (default is 252, which is standard for equity markets).
- `method`: The optimisation method to be used (default is "SLSQP").

The instance variables are the initialised using the provided data, with 3 important metrics to highlight:

```
self.symbol_stocks = list(avg_revenue.index)
self.portfolio_size = len(self.symbol_stocks)
self.prec_opt = ""
```

The `symbol_stocks` extracts the asset symbols or names from the index of the `avg_revenue` Pandas Series list. This is then used to calculate the number of assets in the portfolio in the `portfolio_size`. Finally, `prec_opt`, or preceding optimisation, initializes an empty string, which is used later to store data relating to optimisations that have performed.

The optimisation now needs to be setup. This is achieved by:

```
limit = (0, 1)
self.limits = numpy.full((self.portfolio_size, 2), (0, 1))
```

```
        self.initial_guess = numpy.full(self.portfolio_size, 1.0 /
        self.portfolio_size)
```

In the first line, limit defines a tuple (0, 1) which represents the bounds for each asset's weight in the portfolio. A 2-dimensional NumPy array is created in the second line, filled with the values (0, 1) for each asset. This sets the bounds for the optimisation problem. The initial guess initializes an array where each asset has an equal weight in the portfolio.

A constraint function now is to be defined to ensure that the sum of portfolio weights equals 1:

```
        def constraint_func(x):
            return numpy.sum(x) - 1
        self.constraints = {"type": "eq", "fun": constraint_func}
```

The setter in the following line sets up the constraint for the optimisation problem using the previously defined function. In the subsequent lines, placeholders for optimised values and allocations are initialised, which will be filled once an optimisation pass is completed:

```
        self.optimal_mef_points = None
        self.asset_allocation = None
        self.asset_allocation_dataframe = None
```

Optimisation by `scipy.minimize` is only supported for certain solvers. These are defined in the `validate_inputs` methods aptly as `supported_solvers`, as per the documentation (SciPy, n.d.b):

```
        supported_solvers = [
            "BFGS", "CG", "COBYLA", "dogleg", "L-BFGS-B",
            "Nelder-Mead", "Newton-CG", "Powell", "SLSQP", "TNC",
            "trust-constr", "trust-exact", "trust-krylov", "trust-ncg"]
```

The only remaining aspect to setup before optimisation is the creation of an empty Pandas DataFrame, that represents the allocation of assets:

```
def _asset_allocation_dframe(self, allocation):
      ...
      return Pandas.DataFrame(allocation, index=self.symbol_stocks,
      columns=["Allocation"])
```

As seen, the function creates and returns a Pandas DataFrame with the allocation values, indexed by the asset symbols, and with a column named "`Allocation`".

5.3.2 Efficient Frontier – Optimisation

The `EfficientFrontierOptimisation` class performs various optimisations on the portfolio. It begins by initialising the `EfficientFrontierOptimisation` object by storing the provided initialization instance.

```
    def __init__(self, initialization):
        self.initialization = initialization
```

The first optimisation is a simple one: minimise the portfolio volatility. Firstly, the data and parameters necessary for optimisation are extracted:

```
        # Extract data
        avg_revenue = self.initialization.avg_revenue.values
        disp_matrix = self.initialization.disp_matrix.values

        # Extract paratemers
        initial_guess = self.initialization.initial_guess
        method = self.initialization.method
        limits = self.initialization.limits
        constraints = self.initialization.constraints
```

Following this, parameters for the optimisation function itself need to be defined:

```
        param_opt = (avg_revenue, disp_matrix)
```

This creates a tuple of parameters that will be passed to the optimisation function. Now the optimisation itself is performed:

```
        output = optimise.minimize(
            mef_minimisation.calculate_annualized_volatility,
            args=param_opt, x0=initial_guess, method=method,
            bounds=limits, constraints=constraints,)
```

The function uses `scipy.minimize` to perform the optimisation. The objective in this case is to minimise the annualised volatility of the portfolio. The result, stored in the output variable, contains the optimal asset allocations that achieve the minimum volatility.

The function now updates the `prec_opt` attribute of the initialisation object to indicate that the last optimisation type was "`Minimum Volatility`":

```
       self.initialization.prec_opt = "Minimum Volatility"
```

A Boolean is used to indicate whether to record an optimised allocation. This is passed as an argument in the validation step below:

```
        if not isinstance(record_optimised_allocation, bool):
             raise TypeError("The variable record_optimised_allocation should
             be a boolean value.")
```

The updating of the allocation depends on the value of `record_optimised_allocation`. If true, the function saves the optimised allocations in the initialisation object and then returns a DataFrame representation of these allocations. If false, the function directly returns the optimal allocations without saving them.

The second optimisation, `mef_sharpe_maximisation`, is also simple in that it aims to maximise the Sharpe Ratio. To avoid repeating the steps, since they are similar, only differences between `mef_volatility_minimisation` and `mef_sharpe_maximisation` will be discussed.

In addition to the previous data and parameter extractions, the Sharpe Ratio also requires the `risk_free_ROR`:

```
        risk_free_ROR = self.initialization.risk_free_ROR
```

And the parameters and optimisation similarly reflect this:

```
        # Parameters for optimisation function
```

```
param_opt = (avg_revenue, disp_matrix, risk_free_ROR)

# Optimisation using SciPy's minimize
output = optimise.minimize(
    mef_minimisation.calculate_inverse_sharpe_ratio,
    ...)
```

The third optimisation, `mef_return`, differs at it aims to optimise the portfolio to achieve a specified target return while minimising volatility. All data and parameters are the same as the `mef_volatility_minimisation` function, with the key difference being in the optimisation itself, in the definition of the constraints:

```
def return_constraint(x):

    return mef_minimisation.calculate_annualised_return(x,
    avg_revenue, disp_matrix) - target_return

    constraints = [{"type": "eq", "fun": constraint_func}, {"type":
    "eq", "fun": return_constraint}]
```

The `mef_return` function has two constraints:

1. Ensuring the sum of the portfolio weights equals 1 (`constraint_func`).
2. Ensuring the portfolio's return matches the target_return (`return_constraint`).

This second constraint, `return_constraint`, is unique to the `mef_return` function and ensures that the optimised portfolio meets the desired return while still minimizing volatility.

The fourth optimisation, `mef_volatility`, is similar to `mef_return`, but instead its aim to is achieve a specified target volatility while maximizing the Sharpe ratio. It also has the same data and parameter extractions as `mef_sharpe_maximisation`. Rather than having a return constraint, it encompasses a volatility constraint:

```
def volatility_constraint(vx):
    return mef_minimisation.calculate_annualized_volatility(vx,
    avg_revenue, disp_matrix) - target_volatility

     constraints = [{"type": "eq", "fun": constraint_func}, {"type":
    "eq", "fun": volatility_constraint}]
```

Therefore, its two constraints are:

1. Ensuring the sum of the portfolio weights equals 1 (`constraint_func`).
2. Ensuring the portfolio's volatility matches the target_volatility (`volatility_constraint`).

While it shares the first constraint with `mef_sharpe_maximisation` and `mef_return`, the second constraint is again unique to it.

Now that all relevant optimisations are completed, a way is needed to extract portfolios for a series of target returns. This is achieved by the `evaluate_mef` function and associated helper-functions. These targets can either be provided by the user or are generated:

```
def _validate_and_generate_targets(self, targets):
  if targets is None:
      min_return = self.initialization.avg_revenue.min() *
      self.initialization.regular_trading_days
```

```
            max_return = self.initialization.avg_revenue.max() *
            self.initialization.regular_trading_days
            targets = numpy.linspace(round(min_return, 2), round(max_return,
            2), 500)
        return targets
```

If no targets are provided, the function generates a set of 500 targets. This is achieved by calculating the minimum and maximum possible returns based on the average revenue and the number of regular trading days. The function then creates a linear space of targets between these two values.

The Efficient Frontier is calculated via the helper function `_calculate_efficient_frontier`. The main logics to discuss are encapsulated in:

```
optimal_mef_points = [] # Optimised Markowitz Efficient Frontier points

for target in targets:
      asset_allocation = self.mef_return(target,
      record_optimised_allocation=False)

      annualized_return, annualized_volatility, sharpe_ratio =
      calculate_annualisation_of_measures(...)

      # Append to the efficient frontier list
      optimal_mef_points.append([annualized_volatility, target])
```

Ultimately, the code above, for each target in the provided targets list, is calculating the efficient return. At the start of the for loop, a call is made to the `mef_return` method to get the asset allocation for the given target return. Then the annualised return, volatility, and Sharpe ratio is calculated for the given asset allocation using the `calculate_annualisation_of_measures` function. Finally, the calculated annualised volatility and the target return is appended to the `optimal_mef_points` list.

5.3.3 Efficient Frontier – Visualisation & Master

The `EfficientFrontierVisualisation` and `EfficientFrontierMaster` classes play more of an ancillary role than the former two that have been discussed. Furthermore, these components are more straightforward in their functionality, and will so be discussed briefly for closure. They will not be explored extensively to maintain focus on the core mechanics of the module.

Then `EfficientFrontierVisualisation` class is designed to visualise the Efficient Frontier by using the `matplotlib` library, specifically `pylab`. The class is initialized in the `__init__` method with two objects: initialization and optimisation. These objects store the data and optimisation results, respectively. A function, `plot_optimal_mef_points` acts as the core of the visualisation, checking if the efficient frontier has been calculated and, if not, triggering its calculation. It then extracts the volatility and return values and plots them, providing a visual representation of the Markowitz Efficient Frontier:

```
def plot_optimal_mef_points(self):
   if self.initialization.optimal_mef_points is None:
         self.optimisation.evaluate_mef()

   annualised_volatility = self.initialization.optimal_mef_points[:, 0]
   annualized_return = self.initialization.optimal_mef_points[:, 1]
```

```
        self._plot_style_optimal_mef_points(annualised_volatility,
    annualized_return)
```

The other significant method is `plot_vol_and_sharpe_optimal`, which plots the optimal portfolios for minimum volatility and maximum Sharpe ratio. It calculates the allocations and magnitudes for both portfolios and then visualizes them using specific styles. The final method is `mef_metrics`, which calculates and displays metrics for the Efficient Frontier that include the number of trading days, the risk-free rate, the predicted annualised return, the Sharpe Ratio, and most importantly – the most favourable allocation of assets. An illustration of these visuals is provided and discussed in depth in Chapter 8.

The `EfficientFrontierMaster` class serves as an integrative layer that seamlessly combines the initialisation, optimisation, and visualisation components of the Markowitz Efficient Frontier. This master class is designed to provide a unified interface to the user, making it easier to perform various operations related to portfolio optimisation without diving deep into the individual components. The master class delegates to other classes, allowing the following methods to act as wrappers to allow internal functions to perform their previously defined tasks:

1. Properties delegating to initialisation:
    - `avg_revenue`
    - `disp_matrix`
    - `risk_free_ROR`
    - `regular_trading_days`
    - `method`
    - `symbol_stocks`
    - `portfolio_size`
    - `asset_allocation`
    - `asset_allocation_dframe`
    - `optimal_mef_points`

2. Methods delegating to optimisation:
    - mef_volatility_minimisation
    - mef_sharpe_maximisation
    - mef_return
    - mef_volatility
    - mef_evaluate_mef

3. Methods delegating to visualisation:
    - mef_plot_optimal_mef_points
    - ef_plot_vol_and_sharpe_optimal
    - mef_metrics

5.4 Monte Carlo Simulation

Monte Carlo Simulation (MCS) is used to model the probability of different outcomes in a process that cannot easily be predicted due to the intervention of random variables. MCS can be used to simulate the potential future prices of an asset, given its historical volatility and return, and provides a range of possible outcomes and the probabilities they will occur for any choice of action. While it has a different objective to Markowitz' Efficient Frontier (MEF), it must be noted that MCS runs utilise a set of random input values, and the method involves running multiple scenarios of a model to understand the variability of an outcome. The expected output should be a probability distribution of potential future asset prices.

The `MonteCarloSimulation` class within the `monte_carlo_simulation` file is to be used for the actual execution of the MCS process. It again begins with an initialisation of the relevant parameters:

```
    def __init__(self, mcs_iterations=None):
```

```
        self.mcs_iterations = mcs_iterations if mcs_iterations is not
        None else 5000
```

The constructor initializes the class with an optional parameter `mcs_iterations`, which specifies the number of iterations for the Monte Carlo simulation. If the user does not provide a value for `mcs_iterations`, it defaults to `5000` and will run `5000` times, which is considered sufficient by other literature (Intaver, n.d.). The core of this class is the `mcs_simulation` method:

```
def mcs_simulation(self, run_instance, **params):
        mcs_run_output = list(map(lambda _: run_instance(**params),
        range(self.mcs_iterations)))
        return numpy.asarray(mcs_run_output, dtype=object)
```

As seen, the `mcs_simulation` method takes in 2 primary parameters. The first, `run_instance`, is a user-defined function that will be executed during each iteration of the Monte Carlo simulation. Essentially, this function represents the process or system a user wants to simulate. The second is `**params`, which acts as a way to pass a variable number of keyword arguments to the `run_instance` function. It allows the user to provide any additional parameters required by the `run_instance` function without explicitly defining them in the `mcs_simulation` method.

Within the method itself, a list comprehension combined with the map data structure is used to call the `run_instance` function for the specified number of iterations (`self.mcs_iterations`). For each iteration, the `run_instance` function is executed with the provided parameters (`**params`), and its result is stored in the list `mcs_run_output`. The list of results, `mcs_run_output`, is then converted to a NumPy array with a specified data type of object. This conversion facilitates more efficient data manipulation and analysis (Stack Overflow, 2014).

The other class in this module is the `MonteCarloOptimisation` class. This extends the `MonteCarloSimulation` class, implying that it inherits all its attributes and methods:

```
class MonteCarloMethodology(MonteCarloSimulation)
```

There are three noteworthy functions relating to optimisation in this class. The first function is the `_generate_uniform_allocations`:

```
def _generate_uniform_allocations(self):
     random_proportion = numpy.random.uniform(size=self.asset_count)
```

The `random_proportion` here generates a random allocation for each asset in the portfolio. The `numpy.random.uniform` function returns random floats in the half-open interval [0.0, 1.0). The justification for `numpy.random.uniform` stems from the need to ensure a uniform distribution in MCS. Each possible allocation value between 0 and 1 has an equal chance of being selected. This is crucial when exploring a wide range of possible portfolio allocations without any inherent bias towards any specific allocation. It also provides flexibility in terms of specifying the size of the output. A user may generate a single number, a 1D array, or even multi-dimensional arrays, all with a single function call (NumPy, n.d.a). Furthermore, the size parameter, set to `self.asset_count`, ensures that the number of random values generated matches the number of assets in the portfolio. Now, the system needs to normalise `proportion_of_investment` is normalised, by ensuring that the sum of all random allocations equals 1:

```
# Normalize the allocations so they sum to 1
proportion_of_investment = random_proportion /
numpy.linalg.norm(random_proportion, ord=1)
```

The normalization is done by dividing each value in `random_proportion` by the Manhattan or L1 norm (sum of absolute values) of the entire `random_proportion` array. `Linalg` is required to

compute the norm of a vector, and for a vector, the L1 norm is the sum of the absolute values of its elements. The function sum is avoided here as using `linalg.norm` is a more general approach that can handle both positive and negative values, by taking their absolute values (NumPy, n.d.b).

```
# Compute portfolio values using the external function
pfl_holdings = calculate_annualisation_of_measures(...)
return (proportion_of_investment, numpy.array(pfl_holdings))
```

Following this, again `calculate_annualisation_of_measures` is used to compute the portfolio values (annualised returns, volatilities, and Sharpe Ratios) for the given allocation (`proportion_of_investment`). A tuple is returned, and the first element is the normalised allocation for each asset. The second element is the portfolio values (annualised returns, volatilities, and Sharpe Ratios) converted into a NumPy array.

Now that portfolios are uniformly allocated, a function needs to take responsibility for diversifying them and running MCS. This is achieved through the `_generate_portfolio_diversity` function, using the nested function `_generate_uniform_allocations` as an argument. The lines below illustrate an execution of the `mcs_simulation` method:

```
def execute_instance():
    return self._generate_uniform_allocations()

mcs_simulation_output = self.mcs_simulation(execute_instance)
```

Essentially, `_generate_uniform_allocations` will repeatedly call `execute_instance`, which in turn calls `_generate_uniform_allocations`, generating a list of portfolio allocations and their associated metrics. The `mcs_simulation_output` tuple now needs to be unpacked into two separate lists to create DataFrames in which values can be stored:

```
proportion_list = [item[0] for item in mcs_simulation_output]
product_list = [item[1] for item in mcs_simulation_output]
```

The `mcs_simulation_output` is a list of tuples. Each tuple contains two items: the portfolio proportions and the associated metrics (returns, volatility, Sharpe ratio). These two lines of code extract the portfolio proportions and metrics into separate lists, `proportion_list` and `product_list`, respectively. Now that there are two lists, the DataFrame Columns can be defined:

```
proportion_col = list(self.asset_revenue.columns)
product_col = ["Annualised Return", "Volatility", "Sharpe Ratio"]
```

Here, the column names for the two DataFrames that will be created are defined. `proportion_col` gets the column names from the `asset_revenue` attribute, which represents the assets in the portfolio. Conversely, `product_col` is a predefined list of metric names. Finally, the DataFrames can are created:

```
proportion_data_frame = pandas.DataFrame(data=proportion_list,
columns=proportion_col)
product_data_frame = pandas.DataFrame(data=product_list,
columns=product_col)
```

Two Pandas DataFrames are created: `proportion_data_frame` and `product_data_frame`. The former contains the portfolio proportions, and the latter contains the associated metrics for each portfolio. These DataFrames are essential in plotting MCS using matplotlib and are a useful way of coherently and orderly showing attributes such as the annualised return, volatility, minimum

volatility, maximum Sharpe Ratio, and optimised product DataFrame. Again, discussions pertaining to MCS visualisations are omitted due to their relatively straightforward implementation and subsequent showcasing in Chapter 8. For more details, please refer to documented code within the `monte_carlo_simulation` file within the GitLab repository.

The final function to consider is the `mcs_optimised_portfolio`. This is a vital method that practically performs portfolio optimisation by executing the Monte Carlo method. The function begins by calling the `_generate_portfolio_diversity` function:

```
def mcs_optimised_portfolio(self):
      proportion_data_frame, product_data_frame =
      self._generate_portfolio_diversity()
```

As discussed, `_generate_portfolio_diversity` runs a MCS to generate random portfolios and their corresponding metrics. The results are stored in two DataFrames: `proportion_data_frame` (asset allocations for each portfolio) and `product_data_frame` (metrics like return, volatility, and Sharpe ratio for each portfolio). The function finds optimal portfolios via:

```
      volatility_minimisation =
      product_data_frame["Volatility"].values.argmin()

      sharpe_ratio_maximisation = product_data_frame["Sharpe
      Ratio"].values.argmax()
```

These lines identify the indices of the portfolios with the minimum volatility and maximum Sharpe ratio, respectively. The `argmin()` and `argmax()` functions return the indices of the minimum and maximum values in an array. These are used due to their efficiency; `argmin` and `argmax` are optimised and can quickly identify the indices of the minimum and maximum values without having to sort the entire array or list (Brownlee, 2020).

Following this, a dictionary is used to store the indices of optimal portfolio indices, as it allows for an effective reference:

```
      optimal_indices = {
          "Minimum Volatility": volatility_minimisation,
          "Maximum Sharpe Ratio": sharpe_ratio_maximisation }
```

After storing the indices, the system extracts optimal portfolio allocation and metrics:

```
      # Extract optimal proportions & retain in a DataFrame
      optimised_proportion = pandas.DataFrame(
            {key: proportion_data_frame.iloc[val] for key, val in `
            optimal_indices.items()}).T

      # Extract optimal products (return, volatility, Sharpe) retain in a DF
      optimised_product = pandas.DataFrame(
            {key: product_data_frame.iloc[val] for key, val in
            optimal_indices.items()}).T
```

This first extraction creates a DataFrame (`optimised_proportion`) that contains the asset allocations for the optimal portfolios. The Pandas indexing `iloc` method is used to extract rows from the `proportion_data_frame` based on the indices stored in `optimal_indices`. It is used due to a similar reason to `argmax` and `argmin`, in that it is performant (uses integer-positions) and

is explicit (does not rely on labels, uses positions) (Yıldırım, 2021). Similarly, the second extraction line creates a DataFrame (`optimised_product`) that contains the metrics (return, volatility, Sharpe ratio) for the optimal portfolios.

Once extracted, the corresponding class attributes are updates with the new (improved) data:

```
self.proportion_data_frame = proportion_data_frame
self.product_data_frame = product_data_frame
self.optimised_proportion = optimised_proportion
self.optimised_product = optimised_product
```

These lines update the class attributes with the DataFrames created during this method's execution. This ensures that the results of the optimisation are stored within the class instance and can be accessed or used by other methods. Finally, the method returns the DataFrames containing the asset allocations and metrics for the optimal portfolios:

```
return optimised_proportion, optimised_product
```

5.5 Portfolio Analysis, Moving Averages, and Bollinger Bands

The `mov_avg` module is designed to calculate and visualise moving averages for stock prices. Moving averages are a simple and visually clear means of performing financial analyses to smooth out short-term fluctuations and highlight longer-term trends or cycles. Before calculating specific types of moving averages, an overall orchestrator function is required to, among other functionality, primarily manages the calculated moving averages from subsequent functions and determines buy/sell signals for said averages:

```
def moving_average_calculator(...):
      # Compute moving averages for the given window sizes
      moving_avg_prices = compute_moving_averages(input_stock_prices,
      moving_avg_function, window_sizes)

      # Determine buy/sell signals based on moving average crossovers
      bs_signals = determine_buy_sell_signals(moving_avg_prices,
      min(window_sizes), max(window_sizes))
```

For modularisation and clarity, the main `moving_average_calculator` method uses the helper functions to realise computations. The first helper is `compute_moving_averages` to calculate a moving average for a given window size:

```
def compute_moving_averages(data, moving_avg_function, window_sizes)

    moving_avg_prices = data.copy(deep=True)

    for window in window_sizes:
      indicator = f"{window}d"
      moving_avg_prices[indicator] = moving_avg_function(data,
      window_size=window).iloc[:, 0]
    return moving_avg_prices
```

Before making any modifications, a `deep copy` of the input data is created, as seen in the second line, ensuring that the original dataset remains unchanged. The following `for-loop` then iterates over each window size provided in the `window_sizes` list. For each window size, it calculates the

moving average using the specified `moving_avg_function`, which will be defined following this implementation. The result of this calculation is then added as a new column to the `moving_avg_prices DataFrame`. The column name is determined by the indicator variable, which is a string representation of the window size (e.g., "50d" for a 50-day moving average). Finally, the `.iloc[:, 0]` part is used to extract the first column of the result.

The second helper function manages the generation of buy/sell signals based on crossovers. A crossover occurs when two different moving averages cross each other. Short-term or "fast" moving average are typically 50-day moving averages, while long-term or "slow" moving average are often 200-day moving averages. When the short-term moving average crosses above the long-term moving average, it is considered a bullish (buy) signal. This suggests that the asset's price is gaining momentum, and it will likely continue to rise. Conversely, when the short-term moving average crosses below the long-term moving average, it is regarded as a bearish (sell) signal. This proposes that the asset's price is losing momentum, and may continue to fall (Chen, 2022).

```
def determine_buy_sell_signals(...):
    # Deviation based on crossover of short & long-term moving avg
    bs_signals['deviation'] =
    numpy.where(moving_avg_prices[f"{min_window_size}d"] >
    moving_avg_prices[f"{max_window_size}d"], 1.0, 0.0)

    # Determine buy/sell signals based on changes in the deviation
    bs_signals['signal'] = bs_signals['deviation'].diff()
    return bs_signals
```

The logic for the first part, the deviation, is as follows. A new column, deviation, is added to the `bs_signals DataFrame`, and then the `numpy.where` function checks for each data point if the short-term moving average (`min_window_size`) is greater than the long-term moving average (`max_window_size`). If the condition is true, it indicates a potential buy signal, and the value `1.0` is assigned. Otherwise, `0.0` is assigned. Again, `numpy.where` is used due to it being a vectorised operation meaning it operates on entire arrays rather than individual elements, making it faster than traditional Python loops (Sekan, 2021). Instead of creating new arrays or lists, `numpy.where` also operates in-place, which is be more memory-efficient. Furthermore, Pandas, which is used extensively in this project, is built on top of NumPy. Therefore, many NumPy functions, such as `numpy.where`, can be used directly on Pandas Series and DataFrames, ensuring seamless integration and operation.

The second part, the signals, aptly manages the determination of the buy/sell signals. Another new column, signal, is added to the `bs_signals DataFrame`. The `diff` method then calculates the difference between consecutive items in the deviation column. A positive difference (from 0 to 1) indicates a buy signal, while a negative difference (from 1 to 0) indicates a sell signal. No change (0 to 0, or 1 to 1) means no action is performed.

The `moving_average_calculator` function is engaging 4 main types of moving averages, as well as functionality for Bollinger Bands. For now, the moving averages include:

1. Simple moving averages (SMA)
2. Exponential moving averages (EMA)
3. Simple moving average standard deviations (SMASD)
4. Exponential moving average standard deviations (EMASD)

The SMA is calculated through the `simple_moving_average_mean` function, and is useful in to smoothing out short-term fluctuations and highlighting longer-term trends or cycles:

```
def simple_moving_average_mean(...)
```

```
        # Create a rolling window object for the input stock prices
        simple_moving_window = input_stock_prices.rolling(
          window=window_size, min_periods=min_periods, center=window_index,
          win_type=win_type, on=on, axis=axis, closed=closed)
        # Compute the mean for each window to get the simple moving average
        simple_moving_average = simple_moving_window.mean()
        return simple_moving_average
```

The first part, which uses the Pandas `rolling` method, creates a rolling window object for the input stock prices. The `rolling` window helps to act as a view on the data that moves by a specified step size and has a specified window size. It is useful used to compute statistics, such as mean and standard deviation, over a moving window. The calculation of the SMA occurs in the second part. For each window in the `rolling` view, the mean of values within that window is computed.

The design choice of including `rolling` stems from its proficiency in smoothing data. For instance, time series data, especially in finance, can be noisy with short-term oscillations. `Rolling` operations, like calculating a moving average, help smooth out these fluctuations to reveal underlying trends. `Rolling` also has other benefits such as giving users flexibility in specifying the window size, which determines how many data points are considered in each local view. Missing data is handled exceptionally with the `min_periods` parameter, which allows users to specify the minimum number of observations required to have a value.

The second moving average type is the EMA, which gives more weight to the most recent prices, making it more responsive to recent price changes compared to the SMA:

```
    def exponential_moving_average_mean(...)
        # Create exp moving window object for the input stock prices
        exponential_moving_window = input_stock_prices.ewm(
            span=window_size, com=com, halflife=halflife, alpha=alpha, ...)
        # Compute the mean for the exponential moving window
        exponential_moving_average = exponential_moving_window.mean()
        return exponential_moving_average
```

For EMA, the Pandas `ewm` method is used instead of rolling, to create an EMA object. Again, `ewm` gives users similar flexibility to modify parameters such as the `window_size`, which relates to the `span` argument. Alternatively, users could modify the centre of mass or `com` argument to control the decay of recency bias. The `alpha` parameter can even be changes to modify the smoothing factor in the EMA formula (Pandas, n.d.a). The `ewm` method has similar efficiency advantages in vectorised computations. Again, by taking the mean of the EMA object in the second part, the EMA for the input stock prices in obtained.

Both the SMASD and EMASD follow the same design principles as their respective SMA and EMA counterparts. That is to say, that the main differences in implementation between the `simple_moving_average_standard_deviation` against `simple_moving_average_mean`, and `exponential_moving_average_standard_deviation` against `exponential_moving_average_mean` are as follows:

```
    def simple_moving_average_standard_deviation(...
          ddof: int = 1
          ...):
        # Compute the strd deviation for each window in the rolling window
```

```
        simple_ma_standard_deviation = simple_ma_rolling_window.std(
    ddof=ddof)
        return simple_ma_standard_deviation
```

As is the case with SMA, the SMASD also sets up the `rolling` window. However, the difference is that the function calculates the standard deviation for each window in the `rolling` window using the `std` method. Another parameter is also provided, in the NumPy `std` method – the delta degrees of freedom, or `ddof`. This parameter adjusts the divisor used in the calculation, providing the user a means to use either the population or sample standard deviation formula.

For the `exponential_moving_average_standard_deviation`:

```
    def exponential_moving_average_standard_deviation(...):
        # Compute the standard deviation for the exponential moving window
        exponential_ma_standard_deviation = exponential_moving_window.std()
        return exponential_ma_standard_deviation
```

The only difference here, between EMA and EMASD is the use of the `std` method instead of the mean method, as both functions still use `ewm` to create the EMA object. Unlike SMASD, EMASD does not allow the user to modify `ddof`. This is because the `ewm` function in EMASD provides an exponentially weighted moving window. The weights in `ewm` decrease exponentially, so each data point has a different weight. Given this weighting scheme, the concept of degrees of freedom as used in the simple rolling window does not directly apply (Pandas, n.d.b).

Bollinger bands are computed in a separate module, `bbands`, for clarity, but comprise dependencies from functions within the `mov_avg` module, such as `moving_average_calculator`. Here, the focus is placed on the logic behind the computations of the Bollinger Bands, though the module has the facility to perform input validations and visualisations (as these will be shown in Chapter 8). The `compute_bollinger_bands` function operates by utilising three helper functions, including `_calculate_moving_average`, `_calculate_standard_deviation`, and `_calculate_bollinger_bands`:

```
  def compute_bollinger_bands(...):
    moving_averages, feature_column = _calculate_moving_average(...)
    std_deviation = _calculate_standard_deviation(...)
    bollinger_bands = _calculate_bollinger_bands(...)

    return bollinger_bands, feature_column
```

The moving average is calculated via `_calculate_moving_average` by calling `moving_average_calculator` to compute the moving averages for the stock prices:

```
  def _calculate_moving_average(input_stock_prices, ...):
    bband_moving_averages = moving_average_calculator(...)
    feature_column = input_stock_prices.columns.values[0]
    bband_moving_averages = bband_moving_averages.rename(columns=
    {str(window_size) + 'd': str(window_size) + '-day'})
    return bband_moving_averages, feature_column
```

The `feature_column` line extracts the name of the first column from the `input_stock_prices` DataFrame. This is the name of the column containing the stock prices. After calculating the moving average, the resulting DataFrame (`bband_moving_averages`) will have a column named with the pattern `<window_size>d`. For example, if the window size is 50, the column would be named "`50d`".

This subsequent line renames that column to a more descriptive name, such as "`50-day`". The function returns two values: (1) `bband_moving_averages`: DataFrame containing the moving averages, and (2) `feature_column`: name of the column containing the stock prices in the original `input_stock_prices` DataFrame. This in useful to provide context to the caller about which column in the original DataFrame was used for the moving average calculation.

```
def _calculate_standard_deviation(input_stock_prices, moving_avg_function,
window_size, feature_column):
  # Determine the standard deviation function based on the moving avg type
  standard_deviation_function = simple_moving_average_standard_deviation
  if moving_avg_function == simple_moving_average_mean else
  exponential_moving_average_standard_deviation

  # Compute the standard deviation
  bband_standard_deviation =
  standard_deviation_function(input_stock_prices[[feature_column]],
  window_size=window_size)
  return bband_standard_deviation
```

The first part determines which standard deviation function to use based on the type of moving average function provided. It is straightforward and essentially, it ensures that the correct type of standard deviation (simple or exponential) is chosen based on the moving average type.

In the second part, where the standard deviation is computed, the function is provided with the stock prices from the `feature_column` of the `input_stock_prices` DataFrame. The `window_size` parameter is also passed to specify the size of the window for the standard deviation calculation. The result is stored in the `bband_standard_deviation` variable. Essentially, this module uses the functionality previously outlined in the `mov_avg` modules to variably calculate the standard deviation.

The final helper function, `_calculate_bollinger_bands`, is responsible for the designation of values to the lower and upper bounds of the Bollinger Bands:

```
def _calculate_bollinger_bands(...):
  window_label = str(window_size) + "-day"

  # Extract the appropriate column from bband_standard_deviation
  std_deviation_series = bband_standard_deviation.iloc[:, 0]

  # Compute the Bollinger Bands
  upper_bollinger=moving_averages[window_label]+(std_deviation_series*2)
  lower_bollinger=moving_averages[window_label]-(std_deviation_series*2)
  bollinger_bands = moving_averages.assign(
        **{"Upper BB": upper_bollinger, "Lower BB": lower_bollinger})
  return bollinger_bands
```

The first part creates a string label for the window size, which will be used to reference the appropriate column in the `moving_averages` DataFrame. This function then extracts the first column of the `bband_standard_deviation` DataFrame and stores the result in `std_deviation_series`. This column contains the standard deviations for the stock prices.

The second part, relating to computing the Bollinger Bands, shows the upper and lower Bands. The upper Bollinger Band is calculated by adding two times the standard deviation

(`std_deviation_series * 2`) to the moving average. The lower Bollinger Band is calculated by subtracting two times the standard deviation from the moving average. These calculations are based on the typical definition of Bollinger Bands (Dillikar, 2021).

Finally, the Pandas assign method is used to add two new columns to the DataFrame: "`Upper BB`" and "`Lower BB`". "`Upper BB`" will contain the values of the upper Bollinger Band, and "`Lower BB`" will contain the values of the lower Bollinger Band. The `**` syntax is used to unpack the dictionary and pass its key-value pairs as keyword arguments to the assign method. The assign method is used due to its immutability and chainability, as well as the facility to use Python's unpacking (**) to dynamically create column names (Harrison, 2022).

5.6 Performance

Evidently, a module needs to be dedicated to computing different types of returns from stock price data. Returns are fundamental metrics in finance, providing insights into the performance of an asset or portfolio over time. By understanding returns, investors can make informed decisions about buying, selling, or holding assets. In this module, performance, there are 5 main functions used to quantify various asset returns:

1. Cumulative return
2. Daily return
3. Daily proportional return
4. Daily logarithmic return
5. Historical average return

The cumulative return is computed via the function `calculate_cumulative_return`:

```
def calculate_cumulative_return(input_stock_prices, ...):
    # Filter out rows with any NaN values
    cleaned_data = input_stock_prices.dropna(...)
    # Get the initial prices for comparison
    initial_prices = cleaned_data.iloc[0]

    dividend_adjusted_prices = cleaned_data + cumulative_dividend
    price_ratios = dividend_adjusted_prices / initial_prices
    cumulative_return = price_ratios - 1.0
    return cumulative_return
```

The first step involves removing rows with missing values (NaNs) from the financial data, which is very common. This is achieved via the `cleaned_data` variable. The `initial_prices` (from the first row of the cleaned data) are extracted. These will be used as a reference to calculate returns over time. The `dividend_adjusted_prices` reflect dividends, which are a form of profit-sharing given to stockholders. Adding them to the stock prices gives a more accurate representation of total returns. Following this, the `price_ratios` are calculated, which represent how much the stock prices have grown (or shrunk) relative to the initial prices. The `cumulative_return` is calculated as the total return over the period. By subtracting 1 from the price ratios, the percentage return is obtained.

The Pandas `dropna` method is used for its simplicity in removing rows or columns with missing values. Unlike alternatives such as `fillna`, which fills missing values, no artificial data points are introduced, so market behaviour is more accurately represented. `fillna` also relies on interpolation, which can be more complex and unsuitable for financial time-series data (Kundu, 2020).

Daily returns are evaluated through the `calculate_daily_return` function:

```
def calculate_daily_return(input_stock_prices):
    daily_return = input_stock_prices.pct_change(periods=1, ...)
    # Remove rows with NaN values
    daily_return.dropna()
    # Replace infinite values with NaN
    daily_return = daily_return.replace([numpy.inf, -numpy.inf],
numpy.nan)
    return daily_return
```

This function starts by using the `pct_change` method to calculate the percentage change between the current and a prior element in the DataFrame, effectively computing the daily returns. The `periods=1` argument ensures that the percentage change is calculated between consecutive days. Following this, the first row will contain a NaN value as there is no prior day to compare with. This step removes any rows with NaN values, as done for the cumulative return. However, there may also be some rare cases in which he daily return might result in infinite values (i.e., if the stock price for a day was 0 and then increased). This step replaces any infinite values (`numpy.inf or -numpy.inf`) with NaN for consistency.

Additionally, the Pandas `pct.change` method is used for reasons of computational efficiency and the ability to handle multiple data columns. The `pct_change` method also provides parameters like periods, which allows users to compute returns over different intervals (e.g., weekly or monthly returns) if needed by the user.

The daily proportional return refers to the daily return of a portfolio, given the individual stock prices and their respective allocations (weights) in the portfolio. It is computed by the `calculate_daily_return_proportioned` function:

```
def calculate_daily_return_proportioned(input_stock_prices, alocation):
    daily_return = calculate_daily_return(input_stock_prices)
    daily_return_weighted = (daily_return*allocation).sum(skipna=True)
    return daily_return_weighted
```

As seen, the cumulative return makes use of the `calculate_daily_return` function to compute the daily returns for each stock in the portfolio. In the following line the `daily_return` is multiplied by allocation to give the contribution of each stock to the portfolio's overall daily return. The `sum` function is used to aggregate the weighted daily returns across all stocks for each day to get the portfolio's overall daily return. The argument `skipna=True` ensures that NaN values are skipped.

The daily logarithmic return for stock prices is calculated using `calculate_daily_return_logarithmic`:

```
def calculate_daily_return_logarithmic(input_stock_prices):
    percentage_change = calculate_daily_return(input_stock_prices)
    calculate_log = lambda x: numpy.log(1 + x)
    daily_return_logarithmic = percentage_change.apply(calculate_log)
    # Remove rows with NaN values
    daily_return_logarithmic = daily_return_logarithmic.dropna(...)
    return daily_return_logarithmic
```

The function begins by calling the previously defined `calculate_daily_return` function to compute the daily returns for the stock prices. The result is a percentage change from one day to the next. Now, a lambda function, `calculate_log`, is defined. This function takes a value `x` (which

represents the percentage change) and calculates the natural logarithm of (`1 + x`). This formula is used to compute the logarithmic return. The apply function is next used to apply the `calculate_log` function to each element in the `percentage_change` DataFrame. This results in a new DataFrame, `daily_return_logarithmic`, which contains the logarithmic daily returns. As before, the rows containing NaN are removed with the `dropna` function.

The lambda function is selected due to is subsequent use in the apply function. As apply is a higher-order function that takes the lambda function, it allows for seamless application to each element in the `percentage_change` DataFrame. The combination of apply and lambda is a common pattern in Pandas for element-wise operations on DataFrames or Series. The apply method is selected as it facilitates element-wise operations, but it may be less performant for larger datasets, in comparison with other vectorised Pandas methods (Harris, 2021).

Finally, the historical average return is computed by `calculate_historical_avg_return`. This function ingests stock data, computes the average daily return, and then scales it up to an annualized return based on the number of regular trading days in a year:

```
def calculate_historical_avg_return(stock_data, reg_trading_days=252):
    daily_returns = calculate_daily_return(stock_data)
    avg_return = pandas.DataFrame.mean(daily_returns, skipna=True)
    historical_avg_return = avg_return * regular_trading_days
    return historical_avg_return
```

The function begins by calling the previously defined `calculate_daily_return` function to compute the daily returns for the given `stock_data`. The average of the daily returns is simply calculated using the mean method, passing the `daily_returns` as an argument. The argument `skipna=True` ensures that NaN values are ignored when calculating the mean. The following line calculates the historical average return by multiplying the average daily return by the number of regular trading days in a year. The underlying assumption is that the average daily return, when scaled up by the number of trading days in a year, gives an estimate of the annual return.

5.7 Investment

The investment module is designed to represent a generic financial instrument, such as a stock or a financial index. It provides a suite of methods to compute various financial metrics based on the historical price data of the instrument. The most pertinent aspects of the Investment class are computations tied to properties such as the daily return, forecast return, annualised volatility, skewness, and kurtosis, all with respect to the investment:

```
class Investment:
                                    ...
    def _compute_investment_properties(self) -> None:
        self.forecast_investment_return =
        self.calculate_forecast_investment_return()
        self.annualised_investment_volatility =
        self.calculate_annualised_investment_volatility()
        self.investment_skew = self.calculate_investment_skewness()
        self.investment_kurtosis = self.calculate_investment_kurtosis()
                                    ...
```

As seen, the computations are accomplished through the 4 helper functions above. Before the helper functions are analysed, a function must be implemented to compute the daily returns for an investment using its historical price data:

```
def calculate_investment_daily_return(self) -> pandas.Series:
      try:
            investment_daily_return =
            calculate_daily_return(self.asset_price_history)
      except Exception as e:
            raise RuntimeError(f"An error occurred while calculating
            daily returns: {str(e)}")
      return investment_daily_return
```

The `calculate_investment_daily_return` function calculates the daily return using the imported `calculate_daily_return` function from the performance module. It passes the `asset_price_history` to this function to compute the daily returns. If any exception occurs during this calculation (e.g., due to data inconsistencies or other unexpected issues), it is caught, and a descriptive RuntimeError is raised.

The `calculate_forecast_investment_return` function is similar with the only key difference as utilising the imported `calculate_historical_avg_return` from the investment module:

```
def calculate_forecast_investment_return(..., trading_days: int=252):
                                    ...
      forecast_investment_return = calculate_historical_avg_return(...)
                                    ...
```

The annualised volatility for an investment, using its historical price data and a given number of regular trading days, is calculated in `calculate_annualised_investment_volatility`:

```
def calculate_annualised_investment_volatility(self,
regular_trading_days: int=252):
                                    ...
      daily_investment_returns=self.calculate_investment_daily_return()
      # Calculate annualised volatility using std dev of daily returns
      annualised_investment_volatility =
      numpy.sqrt(regular_trading_days) * daily_investment_returns.std()
                                    ...
      return annualised_investment_volatility
```

The method attempts to calculate the annualised volatility in two main steps. The first, as seen is to use the previously defined `calculate_investment_daily_return` method to compute the daily returns for the investment. The second step is to calculate the annualised volatility by multiplying the square root of the `regular_trading_days` using `numpy.sqrt`, by the standard deviation of daily returns, using `std`. The square root of time to scale volatility comes from the properties of a random walk. If price changes are assumed to be normally distributed and independent over time, which is a simplifying assumption often made in quantitative finance, then the variance of the returns scales linearly with time, but the standard deviation (volatility) scales with the square root of time. This is widely known in the context of the Black-Scholes model, for instance (Haugh, 2016).

The remaining two helper functions relate to calculating the skewness and kurtosis of an investment, of which the theoretical concepts have been discussed previously. Skewness and kurtosis are respectively calculated by the following functions:

```
def calculate_investment_skewness(self, skipna=True):
      return self.asset_price_history.skew(skipna=skipna)
```

```
def calculate_investment_kurtosis(self, skipna=True):
      return self.asset_price_history.kurt(skipna=skipna)
```

For skewness, the Pandas method `skew` is used to calculate the skewness of the values in the object. A negative value means the distribution is skewed left, a positive value means it's skewed right, and a value close to 0 indicates a relatively symmetrical distribution.

For kurtosis, the Pandas method `kurt` used to compute the kurtosis of a Series or DataFrame. Kurtosis measures the "tailedness" of a distribution. A high kurtosis indicates a distribution with heavy tails and sharp peaks, while a low kurtosis indicates a distribution with light tails and a flat peak.

For both skewness and kurtotis, the argument `skipna` is passed to their respective methods. Given that `skipna` is `True`, the method will ignore NaN values when calculating skewness and kurtosis. This is a primary reason, aside from being optimised for working with Pandas DataFrames, for using both `skew` and `kurt` instead of manually computing using something like `scipy.stats`. It is naturally more convenient, reduces the risk of error through manual implementation, and allows for consistency with the other Pandas methods used prior.

5.8 Stock and Index

The stock and index modules respectively contain the `Stock` and `Index` classes, which are subclasses (children) of the Investment class (parent), and are specifically tailored to represent stocks and indexes as financial instruments. This design choice of inheritance is made for two reasons:

1. Enhanced functionality: by inheriting from the Investment class, the `Stock` and `Index` classes can leverage all the functionalities of its parent class, such as calculating daily returns, while also segmenting their unique functions. This promotes reusability and a clear hierarchical structure.
2. Flexibility for future investment types: The modular design of having a separate `Stock` and `Index` class allows for the ability to add more specific metrics, without affecting the generic Investment class. Furthermore, there is the possibility of adding other asset types such as securities, bonds, commodities, and so on.

The `Stock` and `Index` classes both have their super-classes initialised in a similar way. For the `Stock`:

```
def initialize_asset(self, asset_price_history: pandas.Series):
      super().__init__(asset_price_history, self.investment_name,
      investment_category="Stock")
```

The `super()` function is used to call a method from the parent (`Investment`) of the current class. This line is initializing the parent class with the provided `asset_price_history` and the `investment_name` attribute of the current instance. The `investment_category` is explicitly set to "`Stock`" because this method belongs to the `Stock` class, which is a specific type of Investment.

The `Stock` class is distinctive through its calculation of the beta coefficient. The first part is:

```
def calculate_beta_coefficient(self, index_returns):
      # Compute daily returns for the stock
      stock_daily_returns = self.calculate_investment_daily_return()
      # Convert market daily returns to DataFrame
```

```
index_returns_dataframe =
index_returns.to_frame()[index_returns.name]
# Compute dispersion matrix between stock/index daily returns
disp_matrix = numpy.cov(stock_daily_returns,
index_returns_dataframe)
```

The first block within the `calculate_beta_coefficient` function calculates the daily returns for the stock using the inherited `calculate_investment_daily_return` method. The result is a pandas Series of daily returns. Following this, the `index_returns` are converted into a DataFrame to ensure compatibility with the upcoming dispersion calculation. The `disp_matrix` is calculated using the NumPy `cov` method to determine the covariance between the stock's daily returns and the index's daily returns. Again, `cov` is used as it is optimised for efficiency, being done so using lower-level languages such as C and Fortran (Vishwajit, 2022). The second part of the `calculate_beta_coefficient` is:

```
# Extract dispersion between stock & index, & variance of market
dispersion = disp_matrix[0, 1]
market_variance = disp_matrix[1, 1]
# Compute the Beta parameter
beta_coefficient = dispersion / market_variance
# Store the Beta value in the object's attribute
self.beta_coefficient = beta_coefficient
return beta_coefficient
```

In this function, the dispersion (aka covariance) matrix is a 2x2 matrix that contains:

- Variance of the stock's returns.
- Variance of the market's returns.
- Dispersion between the stock's returns and the market's returns.

In the second block, from the covariance matrix, the dispersion and `market_variance` are extracted. The dispersion is the covariance between the stock's returns and the market's returns, and the `market_variance` is the variance of the market's returns. Now, the `beta_coefficient` is evaluated by dividing the dispersion by the `market_variance`. This aligns with the theoretical methodology explored in an earlier section. A beta coefficient greater than 1 indicates that the stock is more volatile than the market, while a beta coefficient less than 1 indicates that the stock is less volatile. The calculated beta coefficient is finally stored in the `beta_coefficient` attribute of the `Stock` object. This allows for easy retrieval later without any recalculation.

In the case of the Index, for superclass initialisation, the `investment_name` is set to the name of the `asset_price_history` Series, and the `investment_category` is explicitly set to "`Financial Index`":

```
super().__init__(asset_price_history, investment_name =
asset_price_history.name, investment_category="Financial Index")
# Compute index returns and store them
self.calculate_daily_return = self.compute_index_perday_returns()
```

After initializing the superclass, the Index class computes the daily returns for the index using its own method `compute_index_perday_returns()`. This helper method is defined as follows:

```
def compute_index_perday_returns(self):
```

```
        index_perday_returns =
        calculate_daily_return(self.asset_price_history)
        return index_perday_returns
```

It utilizes the `calculate_daily_return` function, imported from the performance module, to compute the daily returns for the index based on its historical price data.

5.9 Portfolio Optimisation

The module `portfolio_optimisation` defines a class `Portfolio_Optimised_Functions` that is designed to manage and optimise a financial portfolio. This class contains and computes all previously discussed financial measures relating to a portfolio, which can be defined (in this case) as an assembly of `Stock` class instances. Due to the extensive scale of the module, only the key aspects of its implementation will be discussed further. The notable functions can be divided into the following sections, which should appear familiar based on previous module definitions:

1. Stock Management
2. Portfolio Methods
    a. Risk (VaR, downside risk)
    b. Return (daily, cumulative, logarithmic, historical average, forecast)
    c. Beta and dispersion
    d. Shape (skewness, kurtosis)
    e. Volatility (stock volatility, portfolio volatility)
    f. Capital allocation
    g. Performance (Sharpe & Sortino ratios)
3. Portfolio Optimisations
    a. Monte Carlo Simulation (initialisation, optimisation, visuals, printing attributes)
    b. Markowitz Efficient Frontier (initialisation, minimise volatility, maximise Sharpe, optimise for target return, optimise for target volatility, visuals and plots)
4. Visualisations & Reporting (stock visualisation, printing portfolio attributes)

While metrics such as VaR, cumulative return, beta coefficient, skewness, volatility, and Sharpe Ratio are essential, they are well-documented explorations of previous modules. For the sake of brevity and to avoid redundancy, this dissertation will prioritise the `portfolio_optimisation` module's distinct capabilities. Visualisations are also relatively straightforward and transparent, focussing more on stylistic choice rather than underlying logic. Therefore, discussion will focus on point (1), stock management, as this uniquely highlights how assets are added, tracked, and adjusted within a portfolio.

The stock management can be split into two main functions; one function manages stock integration, and the other manages updating portfolio metrics that are computed by methods in point (2). The primary method to integrate a new stock into the portfolio is manager by the master function:

```
    def incorporate_stock(self, stock: Stock, suspend_changes=False):
            self._store_stock_object(stock)
            self._append_stock_details_to_portfolio(stock)
            self._add_stock_data_to_history(stock)

            if not suspend_changes:
                self._cascade_changes()
```

The helper function, `_store_stock_object`, stores the stock object in a dictionary:

```
    def _store_stock_object(self, stock: Stock):
```

```
            self.stock_objects.update({stock.investment_name: stock})
```

The `_append_stock_details_to_portfolio` helper adds the details of the provided stocks to the portfolio's distribution, i.e. the allocations of different assets in the overall portfolio:

```
  def _append_stock_details_to_portfolio(self, stock: Stock):
        self.portfolio_distribution = pandas.concat(
            [self.portfolio_distribution,  stock.stock_details.to_frame().T])
        self.portfolio_distribution.name = "Allocation of stocks"
```

The `to_frame` method is used to convert the stock's details into a DataFrame and transpose it using `.T`. This ensures the stock's details are in the correct format to be appended. The `concat` method is selected to join the current portfolio distribution with the new stock's details. The `concat` method is used instead of append as it can handle DataFrames with different sets of columns. For example, if, the `portfolio_distribution` DataFrame and the new stock's details had different columns, `concat` would introduce NaN values where necessary. Additionally, `concat` maintains the original data in the `self.portfolio_distribution`, where other methods may overwrite or require more complex operations to append new rows (Ponraj, 2020).

The final helper function is also modularised for clarity and impartiality, and is responsible for adding stock data to the `asset_price_history`:

```
    def _add_stock_data_to_history(self, stock: Stock):
        self._insert_stock_data(stock)
        self._format_asset_history_index(stock)
        self._compute_and_store_beta(stock)
```

The first helper function, `_insert_stock_data`, is used to add a new column to the `asset_price_history` DataFrame:

```
      def _insert_stock_data(self, stock: Stock):
            self.asset_price_history.insert(loc=len(self.asset_price_history.
            columns), column=stock.investment_name,...)
```

The `self.asset_price_history` is a DataFrame that holds the historical price data of all assets (stocks) in the portfolio. The insert method is used to add a new column (representing a stock's historical prices) to this DataFrame. The argument using `loc=len` sets the index of where the new column (`investment_name`) should be inserted by calculating the number of existing columns in the DataFrame. In this case, the `investment_name` will be added as the last column in the DataFrame.

The `_format_asset_history_index` function updates the index of the index of the `asset_price_history` DataFrame, setting the index to match the index of the provided stock's historical data:

```
    def _format_asset_history_index(self, stock: Stock):
      self.asset_price_history.set_index(stock.asset_price_history.index.valu
      es, inplace=True)
      self.asset_price_history.index.rename("Date", inplace=True)
```

As seen, the method `set_index` sets a new index for the DataFrame. `stock.asset_price_history.index.values` extracts the index values from the `asset_price_history` attribute of the provided stock object. This proposes that each stock has

its own historical data with an associated "Date" index. The second part using the rename method to give the DataFrame index the name of "Date".

The final helper function to manage the addition of stocks is _compute_and_store_beta:

```
def _compute_and_store_beta(self, stock: Stock):
  beta_stock = stock.calculate_beta_coefficient(
  self.financial_index.calculate_daily_return)
```

The argument passed to this method is self.financial_index.calculate_daily_return. The method calculate_daily_return is called from the performance module on financial_index, which returns the daily return of the financial index. The beta coefficient for the stock is then calculated relative to this daily return.

The second part of the stock management aspect of the portfolio optimisation module focusses on updating various metrics of the portfolio based on different categories. First, a function is written to check if the portfolio is in a state where it is ready for updating:

```
def _is_portfolio_ready_for_update(self) -> bool:
  return not (self.portfolio_distribution.empty or not self.stock_objects
  or self.asset_price_history.empty)
```

The return logic:

- Checks if the portfolio_distribution DataFrame is empty
- Checks if the stock_objects dictionary is empty or None
- Checks if the asset_price_history DataFrame is empty

The method returns True if none of the above conditions are met, meaning the portfolio is ready for an update. Otherwise, it returns False. The main method to manage updating changes to portfolio metrics is _cascade_changes:

```
def _cascade_changes(self):
    if self._is_portfolio_ready_for_update():
        self._cascade_elementary_metrics()
        self._cascade_risk_metrics()
        self._cascade_performance_metrics()
        self._cascade_statistical_metrics()
        self._cascade_beta_metrics()
```

Each helper function is rudimentary and essentially updates all metrics outlined in point (2), so will not be further discussed. The _cascade_changes function is called in several of the setters, including regular_trading_days, risk_free_ROR, & confidence_interval_value_at_risk. Most importantly, it is used in the incorporate_stock function once a new stock is added to the portfolio.

5.10 Portfolio Composition

The final module deals with the construction and validation of a financial portfolio using various data sources, primarily the yfinance API. Again, this module is sizeable, so the discussion will only focus on the most pertinent and foundational functions. The first function required is designed to fetch stock data for given symbols and a specified date range using the yfinance API:

```
  def yfinance_api_invocation(...):
                                    ...
```

```
        # Use the helper function to fetch stock data
        stock_data = _fetch_yfinance_data(...)
                                    ...
```

Which is operated by the helper function:

```
    def _fetch_yfinance_data(stock_symbols, from_date, to_date):
        try:
            import yfinance
            return yfinance.download(stock_symbols, start=from_date,
    end=to_date)
        except ImportError:
            raise ImportError( "Please ensure that the package YFinance is
          installed, as it is prerequisited.")
        except Exception as download_error:
          # Raise a general error for other issues during the data fetch
          raise Exception("An error occurred while fetching stock data from
          Yahoo Finance via yfinance.")
```

The function attempts to import the yfinance module. If the import is successful, it then tries to download stock data using the `yfinance.download` method with the provided symbols and date range. If the download is successful, the function immediately returns the downloaded data.

An example of the validation and error handling is also shown. If there is an ImportError, where yfinance is not installed, the function raises a custom error message. If any other exception occurs during the data fetch, the function raises a general error message.

There are two ways in which EntroPy manages the construction of a portfolio. One is through using an API, which currently only supports yfinance, and the other is by reading from a csv file, which involves returning a portfolio using a DataFrame as the input. The first way is illustrated by the function `_portfolio_assembly_api`:

```
    def _portfolio_assembly_api(stock_symbols, apportionment=None,
    start=None, end=None, api_type="yfinance", financial_index: str=None):

        pf_api_construct = Portfolio_Optimised_Functions()
        index_data = pandas.DataFrame()

        # Fetch stock and index data from the specified API
        stock_data = _fetch_stock_data_from_api(...)
        index_data = _fetch_index_data(...)
        # Determine the final allocation for the stocks
        final_allocation = _get_portfolio_allocation(...)

        pf_api_construct = _portfolio_assembly_df(...)
        return pf_api_construct
```

The `pf_api_construct` is assigned a new instance of the `Portfolio_Optimised_Functions` class from the `portfolio_optimisation` module. The `Portfolio_Optimised_Functions` class encompasses all code within points (1) – (4) discussed previously. Following this, an empty Pandas DataFrame is initialized for `index_data`. The helper functions `_fetch_stock_data_from_api` and `_fetch_index_data` simply fetch stock and index data from

the specified API using `yfinance_api_invocation`. There are relatively basic so will not be discussed further. Similarly, `_get_portfolio_allocation` composes the stock apportionment for the portfolio using the API, by generating a balanced allocation and extracting columns names from `stock_df` and validating them. This extraction is noteworthy and computed by the following:

```
def _extract_and_validate_names_from_data(stock_df):
    # Extract column names
    column_titles = stock_df.columns
    # Split column names by '-' and strip whitespace
      column_prefixes = [title.split("-")[0].strip() for title in
      column_titles]
    # Check for conflicting column names
    for x, prefix in enumerate(column_prefixes):
        conflict_prefix = [compar_prefix for position, compar_prefix in
enumerate(column_prefixes) if position != x]
        if prefix in conflict_prefix:
            raise ValueError(generate_error_message(prefix))
    return column_titles
```

For each column title in `column_titles`, the title is split at the first occurrence of the "-" character. The first part (or prefix) of the split title is then stripped of any leading or trailing whitespace. This results in a list of cleaned column prefixes, which is stored in the `column_prefixes` variable. The function iterates over each prefix in `column_prefixes` using its index (x) and value (`prefix`). For each prefix, it creates a list (`conflict_prefix`) of all other prefixes in `column_prefixes` (excluding the current prefix). If the current prefix (`prefix`) is found in the `conflict_prefix` list, it means there is a conflicting column name. In this case, the function raises a ValueError with an error message generated by the `generate_error_message(prefix)` function, which can be found in the repository.

The `split` and `strip` methods are used for its simplicity and efficacy is removing singular characters, such as “-”. For stock symbols from yfinance, the system expects a consistent format in any case, such as AAPL or GOOGL, and so these are suitable over regex or string methods (find, index) due to their directness in processing (Bland, 2021).

The remaining function, `_portfolio_assembly_df`, is crucial component to assembling a portfolio based on the provided stock data, apportionment, column ID tags, and index data.

```
def _portfolio_assembly_df(
    stock_data, apportionment: pandas.DataFrame = None,
    column_id_tags: List[str] = None, index_data = None):
                                    ...
    # Prepare the stock data
    stock_data = _prepare_data(...)
    # Initialize a portfolio object
    portfolio_df = Portfolio_Optimised_Functions()
                                    ...
    # Add each stock to the portfolio
    for i in range(len(apportionment)):
        _add_stock_to_portfolio(...)
    # Cascade any changes made to the portfolio
    portfolio_df._cascade_changes()
    return portfolio_df
```

The function begins by formulating the `stock_data` DataFrame for further processing using the helper function `_prepare_data`, by fetching the required columns based on stock symbols. The function then initialises a new `Portfolio_Optimised_Functions` object, which will represent the assembled portfolio. At the start of the `for-loop`, the function iterates over each row in the apportionment DataFrame. For each row, it adds the corresponding stock to the `portfolio_df` object using the helper function `_add_stock_to_portfolio`. Finally, the function calls the `_cascade_changes` function from the `portfolio_optimisation` module of the `portfolio_df` object. This method updates and recalculates the various aforementioned metrics of the portfolio based on the stocks that were added.

The `_add_stock_to_portfolio` is an extension of the `incorporate_stock` function from the `portfolio_optimisation` module, with the goal of extracting the relevant data series for a given stock, creating a `Stock` object with that data, and then adds that stock to the provided portfolio:

```
def _add_stock_to_portfolio(...):
    # Extract the data series for the given stock name
      stock_series = stock_data.loc[:,
      stock_name].copy(deep=True).squeeze()
    # Create a stock instance with provided apportionment & data series
      stock_instance = Stock(apportionment_row,
      asset_price_history=stock_series)
    # Incorporate the stock into the portfolio
    portfolio.incorporate_stock(stock_instance, suspend_changes=True)
```

In the first comment's code, using the `loc` method of the DataFrame, the function extracts the data series corresponding to the given `stock_name`. The `copy(deep=True)` ensures that the extracted series is a `deep` copy, so changes to it will not affect the original `stock_data` DataFrame. Furthermore, the NumPy `squeeze()` method, which removes dimensions of size 1 from the shape of the DataFrame or Series, is excellent for converting the extracted DataFrame slice into a Pandas Series if it is a single column (NumPy, n.d.c).

For the second comment's code, the function creates an instance of a `Stock` object using the `apportionment_row` and the extracted `stock_series`, which represents the individual stock with its historical price data and apportionment (allocation) details. The final part calls the `incorporate_stock` method of the portfolio object to add the created `stock_instance` to the portfolio. The argument `suspend_changes=True` is specified to ensure that any recalculations or updates to the portfolio should be temporarily suspended while the stock is being added.

The final aspect, portfolio construction, is something that the system needs to manage through a suite of robust error handling and `kwargs` validations. Prior to any validations, three utility functions, which help detect the presence of elements across sets (i.e. to prevent duplication, or calculate the difference), are defined:

```
1. def _contains_all(set1, set2):
      return set(set1).issubset(set2)

2. def _contains_at_least_one(set1, set2):
      return bool(set(set1) & set(set2))

3. def _complement_of_sets(set1, set2):
      return list(set(set2) - set(set1))
```

The first function converts `set1` into a set and then checks if it is a subset of `set2`. The `issubset()` method returns True if all elements of the calling set are present in the set passed as an argument, otherwise it returns False.

The second function converts both `set1` and `set2` into sets. Then, it uses the & operator to return the intersection of the two sets. The `bool()` function is used to check if the resulting set is non-empty, and if the intersection is non-empty, then at least one element from `set1` is present in `set2`.

The third function converts both `set1` and `set2` into sets. Then, it returns the difference between the two sets (i.e., elements that are present in `set2` but not in `set1`), and the result is then converted back into a list.

Following the utility functions, an example of the types of validation function used is `validate_kwargs`, which ensures that the keyword arguments provided to the final construction function are both present and valid.

```
def validate_kwargs(kwargs, provided_args):
    documentation_ref = (
        "Please refer to the report examples for guidance on
formatting.")
    # Check if any arguments were provided
    if not kwargs:
        raise ValueError(f"finalise_portfolio() requires input
arguments.\n{documentation_ref}")
    # Check if any unsupported arguments were provided
    if not _contains_all(kwargs.keys(), provided_args):
        forbidden_arg_ref = _complement_of_sets(provided_args,
kwargs.keys())
        raise ValueError(
            f"Unsupported argument detected {forbidden_arg_ref}. Valid
arguments include: : {provided_args}. {documentation_ref}")
```

The `documentation_ref` initialises a reference string that points users to some documentation for guidance. This string will be used in error messages to guide users on how to format their input correctly. In the context of this project, the “documentation” can be considered as this report, the associated comments within each module’s code, and the scripts to generate visuals from the code. This first `if not kwargs` conditional checks if the `kwargs` dictionary is empty (i.e., the user provided no keyword arguments). If `kwargs` is empty, the function raises a ValueError indicating that input arguments are required for the final portfolio construction.

The next part uses `contains_all` to checks if all keys in the `kwargs` dictionary are present in the `provided_args` list. The forbidden arguments are detailed by `forbidden_arg_ref`, which identifies unsupported `kwargs` using the `_complement_of_sets` function. This returns a list of elements that are present in the second set (`kwargs.keys()`) but not in the first set (`provided_args`). Finally, if unsupported arguments are detected, the function raises a ValueError indicating which arguments are unsupported and provides a list of valid arguments. The other validation functions work in a similar way, instead using different set utility functions.

The last significant function in the portfolio’s construction is:

```
def validate_final_portfolio(fin_portfolio):
    necessary_attributes = [
```

```
            (fin_portfolio.portfolio_distribution.empty,
            'portfolio_distribution'), ...]
        # Check for any invalid attributes in the portfolio
            invalid_attrs = [attr_name for is_invalid, attr_name in
            necessary_attributes if is_invalid]
            if invalid_attrs:
              raise ValueError(f"Error creating Portfolio instance. Invalid
              attributes: {invalid_attrs}. {documentation_ref}")
```

The `necessary_attributes` is a list of two-element tuples: the first element is Boolean expression that checks the validity of an attribute of the `fin_portfolio` object and a string representing the name of the attribute being checked. The list contains checks for various attributes of the `fin_portfolio` object, such as whether certain attributes are `empty` or `None`, including metrics such as the portfolio distribution, asset price history, volatility, skewness and kurtosis. The `invalid_attrs` is a list comprehension that iterates over each tuple in `necessary_attributes`.

The final function, `formulate_final_portfolio` orchestrates the `validate_kwargs` and `validate_final_portfolio` helper functions to assemble a portfolio:

```
def formulate_final_portfolio(**kwargs):
      provided_args = ["apportionment", "stock_symbols", "start", "end",
      "stock_data", "api_type", "financial_index",]
      validate_kwargs(kwargs, provided_args)
      # Assemble the portfolio based on the provided arguments
      fin_portfolio = handle_portfolio_assembly(kwargs, provided_args)
      validate_final_portfolio(fin_portfolio)
      return fin_portfolio
```

The `provided_args` is a list of strings that represent the expected keyword arguments that the function can accept. These are the valid arguments that users should provide when calling this function. The `handle_portfolio_assembly`, which has not been discussed, essentially determines whether to assemble the portfolio using an API (yfinance) or a DataFrame, as per the functions `_portfolio_assembly_api` and `_portfolio_assembly_df`. Finally, the function returns the `fin_portfolio` object, which represents the assembled and validated portfolio.

6 Testing

Testing is a pivotal phase that ensures the reliability and functionality of EntroPy. For this project, which revolves around the management, analysis, and optimisation of financial portfolios, the testing approach has been streamlined to focus predominantly on white-box unit testing. Given the intricate nature of financial computations and the potential ramifications of even minor errors, it is paramount to ensure that each individual component of the software operates flawlessly. Unit testing, in this context, offers several advantages (Tran, 2022):

- Granularity: Unit tests target specific functions or methods, allowing for a detailed examination of each component's behaviour.
- Isolation: By testing functions in isolation, it becomes easier to pinpoint the root cause of any issues, ensuring that errors in one module don't cascade into others.
- Efficiency: With the project's structure, as seen in the modularity of the src folder in the repository, it is evident that all functions are designed to perform specific, well-defined tasks. Unit testing these functions ensures that they perform their designated tasks efficiently and accurately.
- Continuous Integration: Given the active development process observed in the repository, unit tests were integrated into the development workflow after the completion of each corresponding source code function, ensuring that any new changes or additions do not inadvertently introduce errors.

A voluminous table enumerating the >100 test case suite will not be included in an appendix. Including such an extensive table would significantly increase the length of this report, as many tests are repetitive and comprise many steps, potentially making it cumbersome for readers and detracting from the primary content and findings.

The subsequent section will delve into the specifics of the most significant unit tests conducted for each key module and function within the project. The major modules to be discussed:

- Technical Analysis (Moving Averages & Bollinger Bands)
- Monte Carlo Simulations
- Efficient Frontier Analysis
- Portfolio Composition & Optimisation

The modules selected for discussion inherently showcase the robustness of the entire system. Modules like "measures_common" "investment," and "mef_minimisation" (and their dependent modules) serve as precursors to the ones highlighted, in that they are imported. By discussing the primary modules, the precursor modules are indirectly validated. Given that the main modules, which are dependent on the precursors, pass all tests, it stands to reason that the foundational modules are functioning correctly.

However, it is crucial to emphasise that thorough testing has been conducted on **every** module with a 100% pass rate to ensure the reliability and robustness of EntroPy. For readers or evaluators interested, the specifics of each test case, complete with comprehensive documentation, can be found in the accompanying repository submitted alongside this report. Specifically, they are located under the "tests" folder. Note that all tests are carried out using the `pytest` framework, due to its rich plugin architecture, built-in support for assert statements to provide clear failure reports, and ability to parallelise tests (Gezer, 2022). The aim of this discussion is to showcase unique tests where possible and give the reader a flavour of the types of tests conducted, as many are repetitive.

6.1 Tests: Technical Analysis

This section will highlight several key unit tests relating to the mov_avg and bbands modules. The first test validates the correctness of the Simple Moving Average (SMA) mean calculation, and satisfies functional requirements 5.1, 5.3 and 5.7:

```
def test_simple_moving_average_mean():
    data = [1, 2, 3, 4, 5]
    sma_dataframe = pandas.DataFrame({"values": data})
    expected_sma = [1.5, 2.5, 3.5, 4.5]

    sma_result = simple_moving_average_mean(sma_dataframe,
window_size=2).dropna().values.flatten().tolist()
    assert expected_sma == sma_result
```

This test starts by initialising a list of data points that will be used to calculate SMA. The list is then converted into a pandas DataFrame, which is the expected input format for the `simple_moving_average_mean` function. The expected SMA values for the given data with a window size of 2 are them manually calculated in `expected_sma`. Following this, the test calls the `simple_moving_average_mean` function to compute the SMA for the given data. Any NaN values are dropped and then the result is flattened and converted to a list for easy comparison. The `assert` statement at the end verifies that the computed SMA values match the manually calculated expected values.

The next test relates validating the form and structure of the Exponential Moving Average (EMA) mean result, and satisfies functional requirements 5.8 and 5.20:

```
def test_exponential_moving_average_mean_form():
    x1 = range(10)
    x2 = [i**2 for i in range(10)]
    ema_dataframe = pandas.DataFrame({"0": x1, "1": x2})

    # Compute EMA and drop NaN values
    res = exponential_moving_average_mean(ema_dataframe,
window_size=2).dropna()

    assert len(res) == len(ema_dataframe) - 1
    assert not res.isnull().any().any()
```

The test first initializes a list `x1` with numbers from 0 to 9 and a list `x2` with the squares of numbers from 0 to 9. The lists are then converted and sorted as `ema_dataframe` into columns of a pandas DataFrame. This DataFrame is required as the input for the `exponential_moving_average_mean` function. The next block calls the `exponential_moving_average_mean` function to compute the EMA for the given data with a window size of 2.

The first assert statement asserts that the length of the computed EMA result (`res`) matches the length of the input DataFrame, `ema_dataframe`, minus the window size. This ensures that the EMA calculation is correctly considering the window size. The second assert statement validates that there are no NaN values in the computed EMA result, confirming that the EMA calculation is correctly handling potential edge cases and not producing any invalid values.

The final test validates the functionality of the `moving_average_calculator` function, ensuring it correctly computes moving averages and maintains the appropriate structure in its output. This satisfies functional requirements 5.1, 5.4, 5.8, and 5.14:

```
def test_moving_average_calculator():
    x = numpy.sin(numpy.linspace(1, 10, 100))
    ma_calc_dataframe = pandas.DataFrame({"Stock": x})

    # Compute moving averages using the function
    ma = moving_average_calculator(ma_calc_dataframe,
exponential_moving_average_mean, window_sizes=[10, 30],
visualise=False)

    assert ma.shape[1] == ma_calc_dataframe.shape[1] + 2
    assert not ma.iloc[30:].isnull().any().any()
    assert len(ma) == len(ma_calc_dataframe)
```

The first line generates an array x of 100 values, representing the sine of numbers linearly spaced between 1 and 10. This array will serve as the dataset for which moving averages are calculated. `ma_calc_dataframe` refers to the variable from converting the array x into a column of a pandas DataFrame named "`Stock`", which is used as input for the `moving_average_calculator` function.

The next block calls the `moving_average_calculator` function to compute the moving averages for the given data using the `exponential_moving_average_mean` function, with two window sizes, 10 and 30 being specified.

The first assertion verifies that the number of columns in the computed moving average result (`ma`) matches the number of columns in the input DataFrame (`ma_calc_dataframe`) plus the two specified window sizes. This ensures that the moving average calculator is correctly producing separate columns for each window size. The second assertion checks that there are no NaN values in the computed moving average result after the 30th row, which certifies that the moving average calculator is correctly handling potential edge cases related to the largest window size and not producing any invalid values in the latter part of the result. The final assertion confirms that the length (number of rows) of the computed moving average result (`ma`) matches the length of the input DataFrame (`ma_calc_dataframe`). This makes sure that the moving average calculator is maintaining the same number of data points in its output as in its input.

For the bbands module, the following test validates the functionality of the `bollinger_bands` function, ensuring it correctly computes and plots the Bollinger Bands for a given dataset. It satisfies functional requirements 5.1, 5.7, 5.12, 5.13, 5.16, and 5.17:

```
def test_bollinger_bands():
    x = numpy.sin(numpy.linspace(1, 10, 100))
    df = pandas.DataFrame({"Stock": x}, index=numpy.linspace(1, 10,
100))
    df.index.name = "Days"

    # Calculate & plot Bollinger Bands using a SMA with window size=15
    bollinger_bands(df, simple_moving_average_mean, window_size=15)
    ax = pyplot.gcf().axes[0]
    assert len(ax.lines) == 2
```

The first line generates an array x of 100 values, representing the sine of numbers linearly spaced between 1 and 10, which will serve as the dataset for which Bollinger Bands are calculated. The following line converts the array x into a column of a pandas DataFrame named "`Stock`". The index for this DataFrame is set to numbers linearly spaced between 1 and 10, representing "`Days`".

The next block calls the `bollinger_bands` function to compute and plot the Bollinger Bands for the given data using the `simple_moving_average_mean` function with a window size of 15. The variable `ax` is the result from retrieving the current figure's axis for further inspection. This allows examination of the plotted elements on the graph.

The assert statement verifies that that there are two lines plotted on the axis. These two lines represent the upper and lower Bollinger Bands. Therefore, this check ensures that the `bollinger_bands` function is correctly plotting both the upper and lower bands.

6.2 Tests: Monte Carlo Simulation

The first Monte Carlo Simulation (MCS) test validates the functionality of the `_generate_uniform_allocations` method, ensuring it correctly generates uniform allocations for assets. It satisfies functional requirement 4.10:

```
def test_generate_uniform_allocations():
    mcm = MonteCarloMethodology(asset_revenue=sample_asset_revenue)
    proportions, results = mcm._generate_uniform_allocations()

    assert proportions.shape[0] == 2
    assert numpy.isclose(numpy.sum(proportions), 1.0)
    assert len(results) == 3
```

The first line initialises an instance of the `MonteCarloMethodology` class using a sample asset revenue dataset (`sample_asset_revenue`). The next tuple unpacking line calls the `_generate_uniform_allocations` method on the mcm instance. This method is expected to return two outputs: proportions (representing the uniform allocations for assets) and results (containing metrics related to the allocations).

The first assertion tests that the proportions output has two rows, ensuring that the method is generating allocations for two assets. The second assertion validates that the sum of the proportions is close to 1.0, in that the generated allocations are uniform and collectively represent 100% of the portfolio. The final assertion tests that the results output contains three key metrics: Annualised Return, Volatility, and Sharpe Ratio.

Another test is written that validates the functionality of the `mcs_visualisation` method, ensuring it correctly visualises the results of the Monte Carlo simulation without raising any exceptions. This indirectly satisfies functional requirements 4.1, 4.5, 4.12 and directly satisfies 4.13:

```
def test_mcs_visualisation():
    seed_alloc = numpy.array([0.5, 0.5])
    mcm = MonteCarloMethodology(asset_revenue=sample_asset_revenue,
seed_allocation=seed_alloc)
    mcm.mcs_optimised_portfolio()

    try:
        mcm.mcs_visualisation()
        assert True
    except Exception as e:
        pytest.fail(f"Unexpected error raised: {e}")
```

The test starts by initialising an array `seed_alloc` with two values, both set to 0.5. This represents an initial allocation for assets, where each asset has an equal weight of 50%. The following line, defining mcm, initialises an instance of the `MonteCarloMethodology` class using a sample asset

revenue dataset (`sample_asset_revenue`) and the previously defined seed allocation (`seed_alloc`). The test then calls the `mcs_optimised_portfolio` method on the `mcm` instance. This method optimises the portfolio based on the Monte Carlo simulation results. It is a necessary step before visualisation to ensure that there are results to visualise, inadvertently satisfying optimisation functional requirements. The `try` block then attempts to run the `mcs_visualisation` method on the mcm instance, with the goal of checking if the visualisation method runs without raising any exceptions. If the `mcs_visualisation` method runs without any issues, the code will reach the `True` assertion line. The `except` block is reached if any exception is raised during the execution of the `mcs_visualisation` method. If an exception is caught, the test will be marked as failed using the `pytest.fail` method. And the error message will include details about the raised error.

6.3 Tests: Markowitz Efficient Frontier

To test this module, a mock_data function is first defined, which generates dummy data to simulate average revenue and a dispersion matrix for testing purposes:

```
def mock_data():
    avg_revenue = pandas.Series([0.05, 0.03, 0.04], index=["stockA",
"stockB", "stockC"])
    disp_matrix = pandas.DataFrame({
        "stockA": [0.1, 0.02, 0.03],
        "stockB": [0.02, 0.1, 0.04],
        "stockC": [0.03, 0.04, 0.1]
    }, index=["stockA", "stockB", "stockC"])
    return avg_revenue, disp_matrix
```

The avg_revenue variable represents a pandas Series of the average revenue for three stocks: stockA, stockB, and stockC. disp_matrix is a pandas DataFrame representing the dispersion matrix for the three stocks, providing the variances and covariances between the stocks. Finally, the function returns the created average revenue and dispersion matrix.

Now a function that checks correct initialisation of the EfficientFrontierInitialization class initialises its attributes when provided with valid input data. It directly satisfies functional requirements 3.2, 3.3, and 3.6, and indirectly satisfies 3.1, 3.5, and 3.18:

```
def test_initialization_valid_input():
    avg_revenue, disp_matrix = mock_data()
    ef_init = EfficientFrontierInitialization(avg_revenue, disp_matrix)
    assert ef_init.avg_revenue.equals(avg_revenue)
    assert ef_init.disp_matrix.equals(disp_matrix)
    assert ef_init.risk_free_ROR == 0.005427
    assert ef_init.regular_trading_days == 252
    assert ef_init.method == "SLSQP"
    assert ef_init.symbol_stocks == ["stockA", "stockB", "stockC"]
    assert ef_init.portfolio_size == 3
```

The first line calls the mock_data function to generate mock average revenue and a dispersion matrix. The variable ef_init represents an initialisation of an instance of the EfficientFrontierInitialization class using the mock data. The assertions achieve the following:

- avg_revenue.equals(avg_revenue): Asserts that the avg_revenue attribute of the initialised object matches the mock average revenue.

- disp_matrix.equals(disp_matrix): Asserts that the disp_matrix attribute of the initialised object matches the mock dispersion matrix.
- risk_free_ROR == 0.005427: Asserts that the risk-free rate of return attribute of the initialised object is set to the default value of 0.005427.
- regular_trading_days == 252: Asserts that the number of regular trading days attribute of the initialised object is set to the default value of 252.
- method == "SLSQP": Asserts that the optimisation method attribute of the initialised object is set to the default value of "SLSQP".
- symbol_stocks == ["stockA", "stockB", "stockC"]: Asserts that the stock symbols attribute of the initialised object matches the stock symbols from the mock data.
- portfolio_size == 3: Asserts that the portfolio size attribute of the initialised object is correctly set to 3, which is the number of stocks in the mock data.

The next test validates the functionality of the mef_return method of the EfficientFrontierOptimisation class, checking if the method behaves as expected when provided with a specific target return value. It directly satisfies functional requirements 3.11 and 3.15, and indirectly satisfies 3.1, 3.2, 3.8, 3.13, 3.20, 3.23, 3.29, and 3.31:

```
def test_mef_return(optimisation_fixture):
    target_return_value = 0.05
    allocation = optimisation_fixture.mef_return(target_return_value)
    assert optimisation_fixture.initialization.prec_opt == "Efficient Return"
    assert isinstance(allocation, pandas.DataFrame)
```

The test begins by initialising a variable target_return_value and setting its value to 0.05. This represents the desired return value that will be used as an input to the mef_return method. The following line calls the mef_return method on the optimisation_fixture instance. The method is then provided with the previously defined target_return_value as its argument, and the result of this, which is be the optimised allocation for the given target return, is stored in the allocation variable.

The first assertion checks if the prec_opt attribute of the initialisation object within the optimisation_fixture instance has been updated to the string "Efficient Return". This ensures that the method correctly updates the internal state to reflect that the optimisation was done based on efficient return criteria. The second assertion confirms that the allocation variable is an instance of a pandas DataFrame, ensuring ensures that the method returns the allocation results in the expected format.

6.4 Tests: Portfolio Composition & Optimisation

Testing the portfolio composition is much more involved than the other tests, as it requires all subcomponents and imports to be represent in the testing schema. The first step involves defining and importing data paths:

```
allocation_data_file = "/home/wasim/Desktop/EntroPy/data/MAANG_portfolio.csv"
price_data_file = "/home/wasim/Desktop/EntroPy/data/MAANG_stock_data.csv"
```

The first path (allocation_data_file) points to a CSV file that contains portfolio allocation data, i.e. with two columns: (1) the proportion allocation of capital, and (2) the company ticker. The second path (price_data_file) points to another CSV file that contains stock price data, with n columns: (1) the date and (n) the company ticker. Following this the allocation data and stock price data is loaded from yfinance:

```
    yf_apportionment_file = pandas.read_csv(allocation_data_file)
    yf_apportionment = yf_apportionment_file.copy()
```

```
yf_price_data_file = pandas.read_csv(price_data_file, index_col="Date",
parse_dates=True)
```

The stock names then need to be extracted and stored in a list, called names_yf. These are the testing variables:

```
names_yf = yf_apportionment.Name.values.tolist()
```

Next equal allocations (weights) and a date range for each stock must be calculated:

```
equal_allocations = [1.0 / len(names_yf) for i in range(len(names_yf))]
yf_allocations = pandas.DataFrame({"Allocation": equal_allocations,
"Name": names_yf})
start = datetime.datetime(2018, 1, 1)
end = "2023-01-01"
```

Equal allocations for each stock are calculated and stored in the equal_allocations list. These allocations are then used to create a new DataFrame (yf_allocations) that maps each stock name to its equal allocation. Following this a start date and an end date is defined for use in portfolio configurations.

A base configuration dictionary, base_config, is next defined. This specifies the stock symbols and the type of API (currently only yfinance):

```
base_config = {
    "stock_symbols": names_yf,
    "api_type": "yfinance"}
```

A helper function, create_config, is defined to create new configurations based on the base configuration. This function accepts any number of keyword arguments (overrides) that can be used to modify or extend the base configuration and is passed to the formulate_final_portfolio function:

```
def create_config(**overrides):
    config = base_config.copy()
    config.update(overrides)
    return config
```

These configurations are defined in the portfolio_configs function below:

```
portfolio_configs = [
    create_config(apportionment=yf_apportionment),
    create_config(),
    create_config(start=start, end=end),
    {"stock_data": yf_price_data_file},
    {"stock_data": yf_price_data_file, "apportionment":
yf_apportionment_file}]
```

1. The first configuration, detailing apportionment, is based on the base_config but has an additional key-value pair for apportionment. It also specifies the portfolio allocation data (yf_apportionment) to be used.
2. The second configuration is a direct copy of the base_config, containing only the stock symbols and the API type ("yfinance").

3. The third configuration is based on the base_config but includes additional key-value pairs for start and end dates. It specifies the date range for which the portfolio data should be considered.
4. The fourth configuration is not based on the base_config. Instead, it is defined separately and only contains a key-value pair for stock_data, specifying the yf_price_data_file to be used.
5. The fifth and final configuration is similar to the fourth in that it is also not based on the base_config. It contains key-value pairs for both stock_data and apportionment, but specifies the stock price data (yf_price_data_file) and also the portfolio allocation data (yf_apportionment_file) to be used.

Now that the test set-up and structure has been established, the unit tests can be defined. While there are unit tests written to assess each configuration, only one will be detailed below for brevity and due to their similarity. The following test satisfies the functional requirements 2.4, 8.12, 8.33, 9.1, 9.7, 9.8, 9.14, 9.15, 9.17, and 9.18 directly:

```
def test_portfolio_construction_with_apportionment():
    config = portfolio_configs[0]
    pf = formulate_final_portfolio(**config)

    # Check if portfolio and stock instances are of the correct type
    assert isinstance(pf, Portfolio_Optimised_Functions)
    assert isinstance(pf.extract_stock(names_yf[0]), Stock)

    # Check if pf asset prices and distribution are of type DataFrame
    assert isinstance(pf.asset_price_history, pandas.DataFrame)
    assert isinstance(pf.portfolio_distribution, pandas.DataFrame)
                                    ...
```

The config in the first line retrieves the first configuration from the portfolio_configs list, which is the configuration with apportionment data. In the second line, with the variable pf, the retrieved configuration, the formulate_final_portfolio function is called to create a portfolio instance. The **config syntax is used to unpack the configuration dictionary as keyword arguments.

The first and second assertions validate that the constructed portfolio (pf) is an instance of the Portfolio_Optimised_Functions class and that the first stock in the portfolio is an instance of the Stock class. The third and fourth assertions ensure that both the asset price history and the portfolio distribution of the constructed portfolio are stored as pandas DataFrames. Carrying on:

```
    assert len(pf.stock_objects) == len(pf.asset_price_history.columns)
    assert pf.asset_price_history.columns.tolist() == names_yf
    assert pf.asset_price_history.index.name == "Date"
    assert (pf.portfolio_distribution == yf_apportionment).all().all()

    allocations = pf.calculate_pf_proportioned_allocation()
    # Ensure each weight is between 0 and 1
    for allocation in allocations:
        assert 0 <= allocation <= 1
    # Ensure the sum of all weights is approximately 1
    assert abs(sum(allocations) - 1) <= 1e-9
```

The fifth, sixth and seventh assertions validate the structure of the asset_price_history DataFrame, ensuring that:

- The number of stocks in the portfolio matches the number of columns in the asset price history.
- The column names of the asset price history match the stock names.
- The index name of the asset price history is "Date".

The either assertion checks that the portfolio's distribution matches the predefined apportionment data (yf_apportionment).

Moving onto the final block, the calculate_pf_proportioned_allocation method is called to compute the proportioned allocation of assets in the portfolio. Each weight (i.e. allocation) in the resulting list is ensured to be between 0 and 1, inclusive. The final assertion verifies that the sum of all weights is checked to be approximately 1, with a small tolerance (1e-9) to account for potential floating-point inaccuracies.

The Portfolio Composition arrangement extends to tests concerning Portfolio Optimisation. Recall, tests have been written for Efficient Frontier in the same vein as the Monte Carlo test below, but are omitted from discussions due to their verbosity. The following test satisfies the functional requirements 1.7, 4.1, 4.2, 4.4, 4.10, 4.12, 6.4, 6.5, 8.22, 8.24, and 8.25:

```
def test_pf_mcs_optimised_portfolio():
    config = portfolio_configs[4]
    final_portfolio = formulate_final_portfolio(**config)
    numpy.random.seed(seed=0)
    individual_forecast_returns =
calculate_historical_avg_return(final_portfolio.asset_price_history)

    # Generate a few random portfolio weight combinations
    num_portfolios = 100
    random_weights =
numpy.array([numpy.random.dirichlet(numpy.ones(len(final_portfolio.stoc
k_objects)), size=1) for _ in range(num_portfolios)])
```

Again the config first line sets-up the portfolio, using the fifth configuration from the portfolio_configs list. The formulate_final_portfolio function is then called using this configuration to create a portfolio instance. A random.seed is also used to ensure that any random operations performed during the test are reproducible. Following this, the historical average return for each individual stock in the portfolio is calculated using calculate_historical_avg_return.

In the next block, 100 random portfolio weight combinations are generated, and the use of numpy.random.dirichlet ensures that the generated weights for each portfolio sum up to 1. Continuing with the unit test:

```
    # Manually compute expected returns
    expected_returns = [numpy.dot(weights[0],
individual_forecast_returns) for weights in random_weights]
    volatilities = [numpy.sqrt(numpy.dot(weights[0].T,
numpy.dot(final_portfolio.asset_price_history.cov(), weights[0]))) for
weights in random_weights]

    # Compute Sharpe Ratios
    risk_free_rate = final_portfolio.risk_free_ROR
```

```
        sharpe_ratios = (numpy.array(expected_returns) - risk_free_rate) /
    numpy.array(volatilities)
```

For each set of random weights, the expected return and volatility are computed manually. The expected return is computed by the dot product of the weights and the individual stock returns and the volatility is computed using the covariance matrix of the asset price history. The Sharpe ratio for each set of random weights is also computed using the basic formula:

$$Sharpe = \frac{Expected\ Return - Risk\ Free\ Rate}{Volatility}$$

One manual computations are complete, the Monte Carlo optimisation is run:

```
        # Run the Monte Carlo optimisation
        opt_w, opt_res =
    final_portfolio.pf_mcs_optimised_portfolio(mcs_iterations=500)
```

This is run on the portfolio for 500 iterations. The results, including the optimal weights (opt_w) and other metrics (opt_res), are retrieved.

```
        assert opt_res.loc["Minimum Volatility", "Volatility"] <=
    max(volatilities)
        assert opt_res.loc["Maximum Sharpe Ratio", "Sharpe Ratio"] >=
    min(sharpe_ratios)
```

The final part contains two assertions. The first assertion checks that the volatility of the portfolio identified as having "Minimum Volatility" from the Monte Carlo optimisation is less than or equal to the maximum volatility computed manually from the random weights. The second assertion confirms that the Sharpe ratio of the portfolio identified as having the "Maximum Sharpe Ratio" from the Monte Carlo optimisation is greater than or equal to the minimum Sharpe ratio computed manually from the random weights.

7 Project Management

7.1 Development Strategy

Defining requirements for EntroPy has been a challenging process, due to it originally being dependent on extensive theoretical research and scarce existing implementations. Despite the bulk of the research being done prior to any development, an iterative development process using incremental techniques was still employed, as there were continually gaps being identified in what information would be useful to show users. Adopting a waterfall model to development would have been too restrictive, as each module would have needed to be completed in its entirety, prior to starting the following module (Pal, 2023). This rigidity is not suitable for EntroPy; an example of why is represented in the “measures_ratios” module, where the Sortino ratio was in fact added after the development of most other modules, including the core “portfolio_optimisation” module. Similarly, a purely agile methodology is not suitable as requirements and deadlines are pre-defined, and a feedback loop between stakeholders and developers is not possible (Atlassian, 2023).

Iterative development comprised a repeating loop of work: (a) specification, (b) design, (c) implementation, and (d) test (Kumar, 2022). Each increment was intended to provide another feature or function that delivered a more effective product with each loop. An approximate division of the entire workflow of the whole project is as follows:

1. Research:
   - Understand the need for financial portfolio management and optimisation tools.
   - Examine existing financial theories and tools.
   - Research related work in the domain of financial portfolio management.
   - Explore computational methods, technologies, and best practices in financial data processing.

2. Requirements Specification:
   - Identify the core functionalities needed for financial portfolio management.
   - Document desired features such as computing returns, moving averages, optimisation with Monte Carlo and Efficient Frontier, etc.
   - Categorize and prioritize the requirements based on importance and feasibility.

3. Planning & Task Definition:
   - Create an overall project roadmap.
   - Break down the requirements into smaller tasks.
   - Estimate the effort required for each task and set milestones.

4. Design:
   - Plan the overall architecture of the system.
   - Prototype portfolio building, returns visualisation, and optimisation methods.

5. Development Phases:
   - Phase 1: Core Development
     - Implement the basic structure of the program.
     - Implement methods to compute returns, measures, moving averages, and represent financial instruments (stocks and indxes).
   - Phase 2: Optimisation Features
     - Introduce portfolio optimisation techniques (Efficient Frontier & Monte Carlo).
     - Develop visualisation tools for all modules, highlighting important metrics.
   - Phase 3: Portfolio Construction

    - Integrate all previous modules into one easily manipulated class.
    - Implement yfinance data extraction functionality, with robust structure criteria.
- Phase 4: Testing & Example Scripts
    - Continuously test each functionality.
    - Refine features based on testing results and feedback.
    - Create examples to help users with getting started and for documentation.

Throughout the project, evaluations between the development team and stakeholders were held to consider risks, monitor progress, and adjust tasks when necessary. Time was also dedicated to drafting reports, formulating the project demonstration, and periodically deploying the application. In addition to project plan via the Gantt chart below, each phase's features were segmented into smaller tasks, which were added to a phase-specific, using Jira, backlog in order of priority.

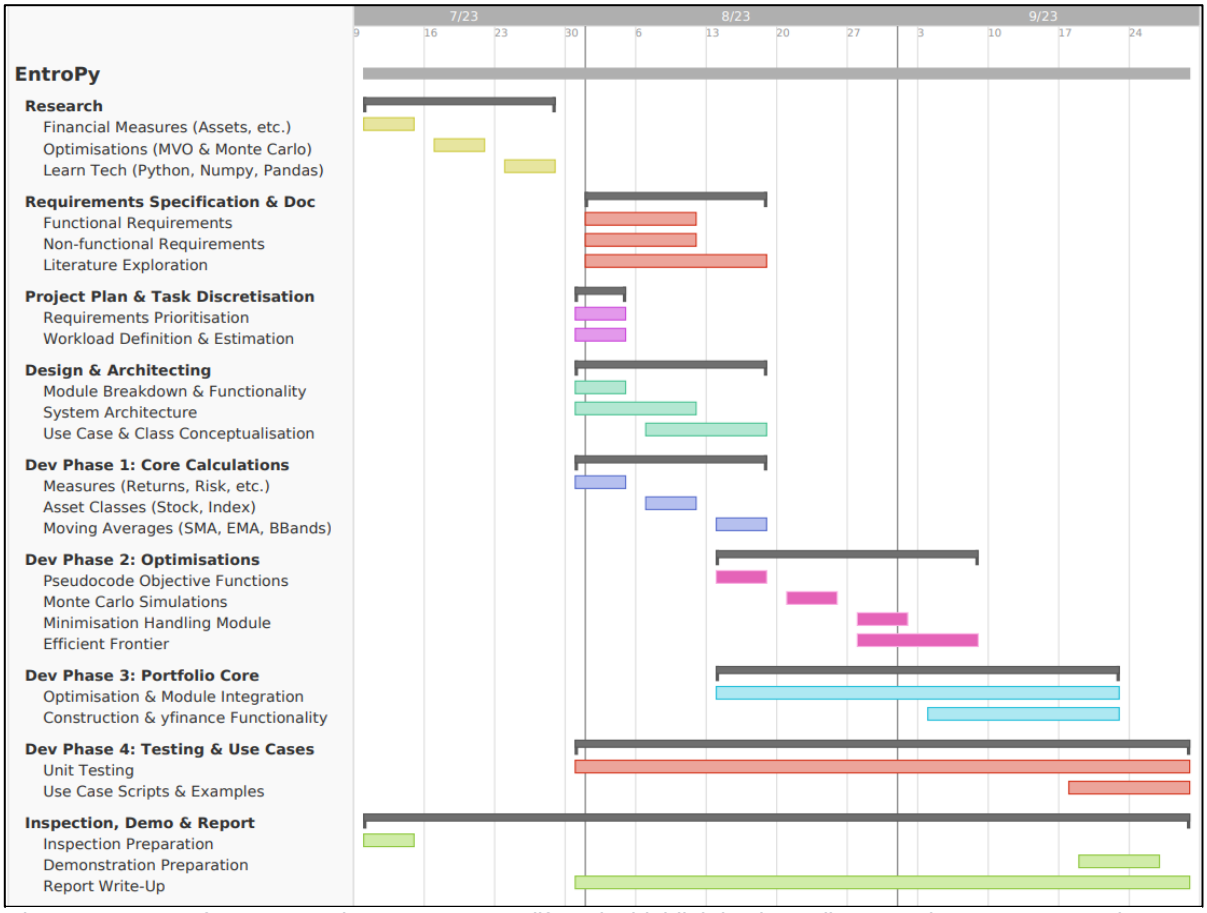


Figure 18: Gantt of EntroPy project management lifecycle, highlighting key milestones (TeamGantt, 2023).

7.2 Version Control

Version control systems act as an archive of code iterations and aid developers in seamlessly testing new feature implementations, while retaining the ability to roll-back previous system versions. They are especially useful in protecting against data losses if a storage device fails, and are useful in reducing the requirement of locally storing files (Onejohi, 2020).

During the development, testing and evaluative stages of this project, code contributions were regularly pushed to a private GitHub repository from a local VirtualBox-hosted Xubuntu desktop. These updates were usually completed daily or after any substantial session of programming. Acting as a version control mechanism, these contributions were fundamental in cases where reverting to older (functional) versions of the system was required. This was especially useful after any unsuccessful feature implementations. After development had concluded, the system was committed to the University of Birmingham's GitLab repository.

8 Results & Discussion

The essence of any software development project lies not just in the implementation but in the tangible outcomes it produces and the real-world challenges it addresses. This section delves into the results obtained from EntroPy, presenting them in various forms, from runnable scripts to illustrative software outputs. All example scripts can be found in the GitLab repository under the folder “scripts”. Any visualisation not discussed in this section is included in Appendix B alongside each associated script, for completeness.

8.1 Fetching & Handling Data

Prior to any visualisations, the CSV data must be defined to be used in following scripts for portfolio_composition. The two initial scripts responsible for this are script_fetch_allocation and script_fetch_data. The former script is used to define the initial allocations that the investor may have for a given number of companies, and the most important aspect is:

```
def main():
    companies = ["META", "AAPL", "AMZN", "NFLX", "GOOG"]
    allocations = [20.0, 30.0, 25.0, 15.0, 10.0]  # Example allocations
    create_portfolio_csv("MAANG_portfolio.csv", allocations, companies)
```

For the purposes of these tests, the companies selected are 5 prominent American technology companies: Meta (META, formerly Facebook), Apple (AAPL), Amazon (AMZN), Netflix (NFLX), and Google (GOOG). These collective form the adapted acronym MAANG (Caroline, 2021). Now that the initial allocation of these is defined, once the script is run, a CSV file named "MAANG_portfolio.csv" will be created in the current directory.

Following this, close price (in this example) stock data for each of these companies must be fetched using script_fetch_data:

```
def fetch_stock_data(ticker, start_date, end_date):
    stock_data = yfinance.download(ticker, start=start_date,
end=end_date)
    return stock_data['Close']
```

The function takes three input arguments. ticker: The first, ticker, is the stock ticker symbol (e.g., "AAPL" for Apple). The start_date and end_date is the starting and ending date for fetching the stock data. For these examples, a 5-year period from 2018 to 2023 is selected to highlight the robustness of the system to handling large amounts of data, and to provide some interesting insights into the effects of COVID-19 on the tech giants:

```
def main():
        start_date = "2018-01-01"
        end_date = "2023-01-01"
        companies = ["META", "AAPL", "AMZN", "NFLX", "GOOG"]
                                        ...
        merged_data.to_csv("MAANG_stock_data.csv")
```

Again, once the script is run, the closing stock prices for the specified companies over the date range from "2018-01-01" to "2023-01-01" will be fetched and saved to a CSV file named "MAANG_stock_data.csv" in the current directory.

8.2 Optimisation using Monte Carlo & The Efficient Frontier

Now that the data is available, the attention can be turned to the first runnable script, script_mcs_and_mef. The main non-boilerplate lines to focus on are:

```
final_portfolio = formulate_final_portfolio(stock_data=price_data)
opt_w, opt_res =
final_portfolio.pf_mcs_optimised_portfolio(mcs_iterations=5000)
final_portfolio.pf_mcs_visualisation()
final_portfolio.pf_stock_visualisation()
```

The script calls the formulate_final_portfolio function with the stock data to create a portfolio object. The pf_mcs_optimised_portfolio method of the portfolio object is called to calculate the optimised portfolio weights and results using Monte Carlo simulation. In this case, the number of iterations for the simulation is set to 5000. The results of the Monte Carlo simulation are visualised using the pf_mcs_visualisation method of the portfolio object. The stock data in the portfolio is visualised using the pf_stock_visualisation method. The resultant plot is as follows:

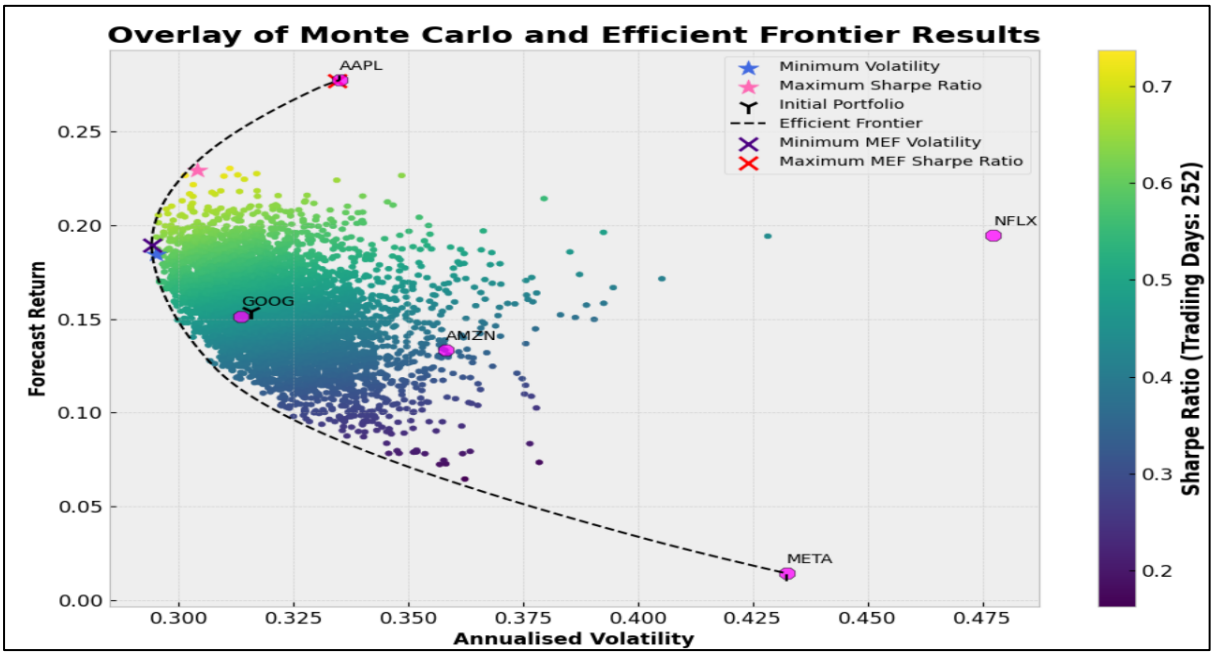


Figure 19: Output from script_mcs_and_mef: optimisation using Monte Carlo Simulation & the Efficient Frontier.

While this section will not cover the meaning behind every result produced by each script, there are some interesting insights that can be gleaned from this plot by relating it to real-world events. First, one can see that the initial portfolio average, denoted by the black symbol next to the GOOG stock, is considerably different from what the system recommends as the optimal asset distribution. AAPL has the strongest performance in terms of being exactly aligned with the maximum MEF Sharpe ratio, that is the maximum Sharpe ratio for a specified target volatility. This optimality of investing in AAPL can be supported by a 365.9% closing price increase from 2018 to 2023 (StatMuse, 2023). AAPL is also continually ranked as one of the most profitable companies currently, largely due to its bleeding-edge innovation, large market cap, low operating expenses and high profit margins (Eassa, 2017). From the launch of the iPhone 14 and now 15, the iPhone segment alone comprises 52.1% of its total revenue, and yearly launches continue to be a profitable endeavour (Johnston, 2023).

One can examine similar real-word behaviour to explain the poor performance of META, which comprises the lowest return to volatility ratio out of any other MAANG stock. Meta's "year of efficiency" could be one reason for its low desirability, where restricting plans announced the

termination of >10,000 employees and the cancellation of many upcoming projects such as NFTs on Instagram (Singh & Sawers, 2023). Perhaps the biggest indicator to support the plot is Meta's cancellation of the Metaverse project with a sunk cost of $36 billion (Olson, 2023). All things considered, the valuation of META had fallen by $700 billion from 2022 to 2023, some of which may also be attributed to competitors such as TikTok (IronFX, 2023). These factors support the plot indicator of META being a highly volatile and low-return stock, over the past 5 years.

The final real-world example to support the efficacy of this plot can be linked to NFLX, which boasts the highest volatility out of the MAANG stocks. Despite having a respectable forecast return, Netflix's volatility is likely attributed to its recent policy which restricted password-sharing amongst users. This has resulted in a dramatic reduction in growth rate from 2020 from 27.5% to 2.8% (Samrhitha, 2023). The erratic volatility seen on the plot is also supported by Netflix's inflated valuation in 2021 of $692 per share versus its valuation in September 2023 at $397 per share; this spike was largely due to the popularity of the platform throughout the pandemic (Forbes, 2023). The final reason to emergence of competitors, such as Disney+ and HBO Max, which saw NFLX stock prices drop and market share decrease to 44.21% in Q1 2023, against 49.77% in Q1 of 2022 (Pandey, 2023). This results in elevated volatility, which further supports the plot.

Some other notable features that can be garnered from this plot includes the minimum volatility and maximum Sharpe ratios, which are denoted by stars on the plot. These ratios are visualised from the Monte Carlo simulation part of the code, as previously discussed in the Chapter 5 section. The minimum MEF volatility, that is the minimum volatility for a specified objective return, is similar to the actual minimum volatility in this example.

8.3 Optimisation to Satisfy Objective Functions

The user can also generate more apparent data about their metrics for an efficient portfolio, via script_mef. This is achieved primarily through the following lines:

```
# Minimize volatility
final_portfolio.optimise_pf_mef_volatility_minimisation(show_details=Tr
ue)
# Maximize Sharpe ratio
final_portfolio.optimise_pf_mef_sharpe_maximisation(show_details=True)
# Target a specific return
final_portfolio.optimise_pf_mef_return(0.20, show_details=True)
# Target a specific volatility
final_portfolio.optimise_pf_mef_volatility(0.25,
show_details=True)
```

Calling this optimisation methods, which were previously defined in the Portfolio_Optimised_Functions class, the user here has defined four objective functions that are tailored to their specific risk appetite. In this instance, a target return of 20% and a target volatility of 25% are set, but these can be seamlessly changed based on the user's preferences. Running the script will produce a plot simply showcasing the Efficient Frontier line showcasing the annualised volatility against the forecasted return for the MAANG tickers previously defined. What is more convenient is the information displayed in the terminal, which is facilitated by the input argument `show_details=True`:

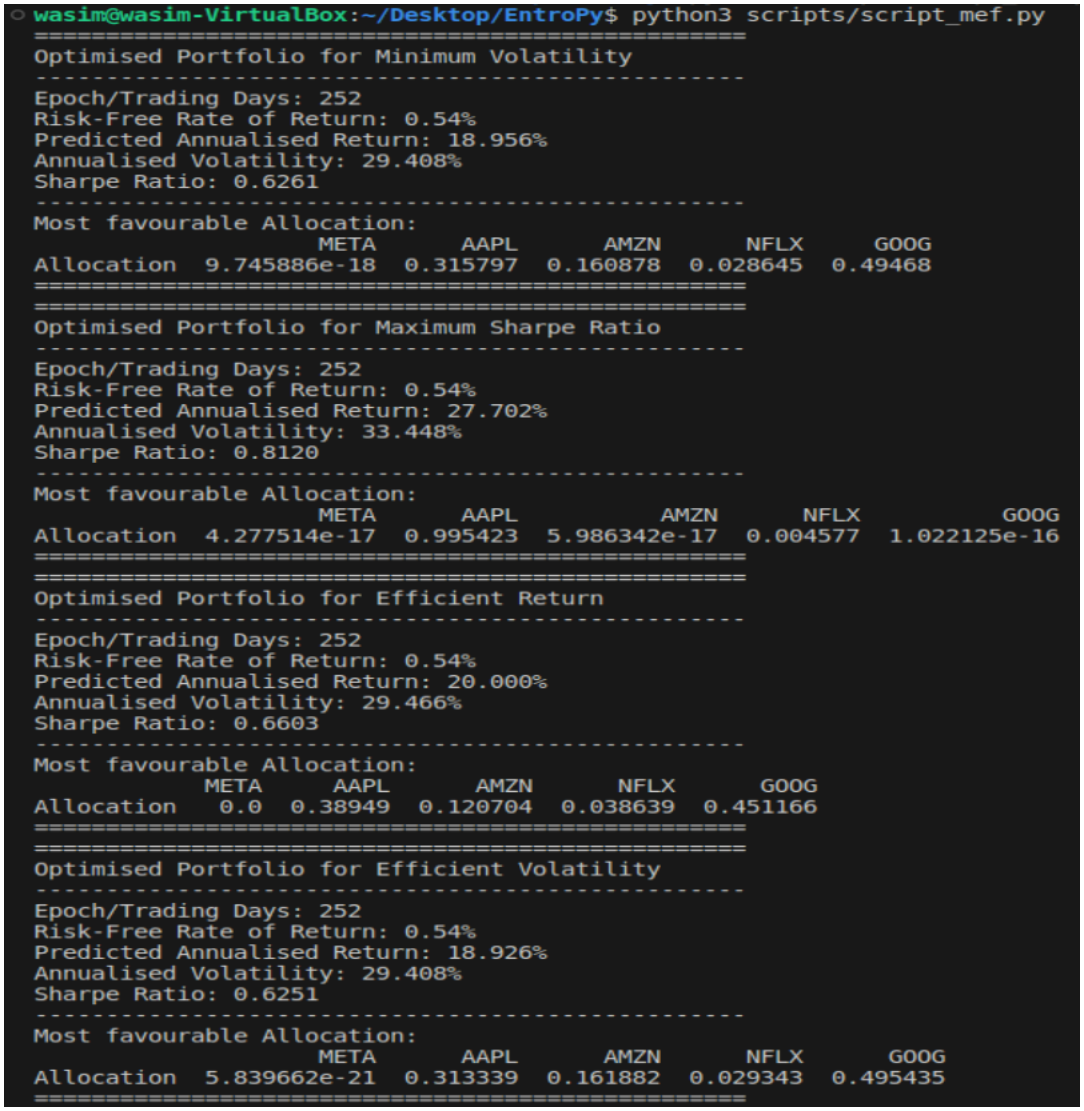


Figure 20: Output from script_mef: various optimal portfolio suggestions based on investor risk-apetite.

As seen from the output, four different allocations are proposed by the system, based upon the user's requirements. The user has a clear view on the trading window size, the predicted return for each allocation set and its associated volatility, and the Sharpe ratio as a metric of risk-adjusted returns.

If the user simply wanted to maximise their forecast return, clearly they would select the allocations of an optimised portfolio for the maximum Sharpe ratio, which boasts an annualised return of 27.7% on their capital investment. This allocation sees AAPL comprising the majority of the portfolio at 99.5%, with the remainder being far less desirable. However, the user may also be more risk-averse, and maximising the Sharpe ratio alone has the highest annualised volatility at 33.4%. This makes sense as the portfolio is poorly diversified and, therefore, is prone to experiencing much a much greater dispersion of returns from larger fluctuations around its mean price.

If the user seeks to minimise risk, in this example, they may decide to optimise for the minimum volatility or efficient volatility, as both present similar performances. As seen for both of these optimisations, the distribution of stocks is more equal, but AAPL and GOOG still comprise the majority of the portfolio at ~31% and ~49% respectively. The previous insights from the poor performance of META is seen in every optimisation profile, with a virtually non-existent allocation. Looking over all allocations and averaging for each stock, the overall desirability of the tickers is largely attributable to AAPL and GOOG, followed by AMZN and NFLX, and then META firmly last.

8.4 Moving Average Visualisations

Aside from optimisation, the user has a plethora of tools to create visualisations relating to technical analysis. The script, script_mov_avg, creates a simple and exponential moving average plot for the MAANG stocks over the 5-year period previously defined. While these plots are useful, what is perhaps more pertinent to discuss is the moving average analysis of a specific stock with buy-and-sell signal generation. In the script, is it performed using an exponential moving average for NFLX by:

```
# Plot the band of moving averages for a specific stock
stock_symbol = "NFLX"
stock_data =
final_portfolio.extract_stock(stock_symbol).asset_price_history.copy(de
ep=True)
window_sizes = [10, 50, 100, 150, 200]

# Compute and visualize the moving averages
moving_average_calculator(stock_data, exponential_moving_average_mean,
window_sizes, visualise=True)
                                        ...
pyplot.show()
```

Here, the script specifies that it wants to work with the stock symbol "NFLX". The `final_portfolio.extract_stock(stock_symbol)` method is used to extract the data related to the "NFLX" stock from the final_portfolio object. The `asset_price_history` attribute then fetches the historical price data for "NFLX". The `copy(deep=True)` ensures that a deep copy of the data is made, meaning any changes to `stock_data` will not affect the original data in `final_portfolio`. The `window_sizes` defines a list of day spans that will be used to compute moving averages. Each value represents the days over which the moving average is calculated, which is within the range of 10 – 200 days in this case to cover all short and long term oscillations. The `moving_average_calculator` function is called to compute and visualise the exponential moving averages for the `stock_data` using the specified window sizes. The output is:

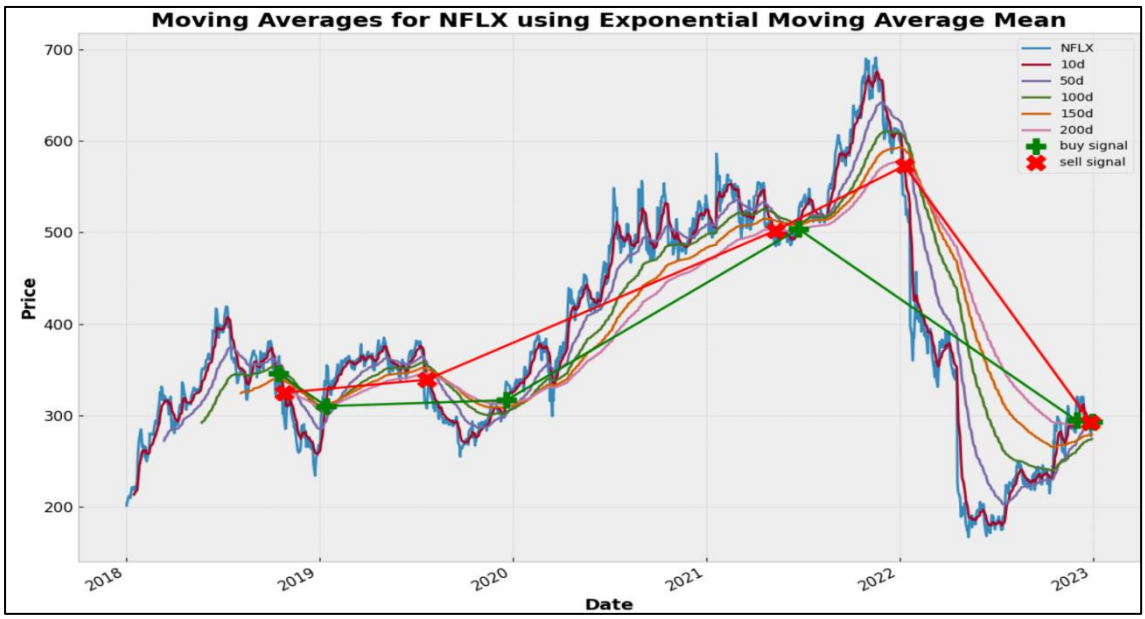


Figure 21: Output from script_mov_avg: Netflix EMA for various window sizes with buy/sell signals.

This plot gives a clear overview of the changes in NFLX stock price against varying exponential moving average window sizes. The buy and sell signals, which are calculated using moving average crossovers as previously discussed, are particularly useful for the user as they represent the market sentiment. A buy signal is generated when a short-term moving average (i.e. 50d or less) crosses above a long-term moving average (i.e. 100d or more). This crossover is seen as a bullish sign to the user and suggests to them that the stock's price might continue to rise, so they may want to buy. Conversely, a sell signal is generated when a short-term moving average crosses below a long-term moving average. This crossover is seen as a bearish sign, indicating that the stock's price might continue to fall and that the user may want to sell.

Using real-world examples, one can see that the period of early to late 2022 comprises a bearish market, which aligns with the heightened interest rates, (ongoing) conflict in Ukraine, and holdover from COVID-19 (Vega, 2022). Similarly, the bullish market from 2020 to 2021 soared after the initial downturn in March 2020 due to COVID-19. Despite the world grappling with job losses, business closures, and social unrest, the stock market remained resilient due to the Federal Reserve's support, congressional relief, and low bond yields, which supports the buy signal at the start of 2020 (Stewart, 2021).

8.5 Bollinger Bands using SMA Smoothing

Another technical analysis tool that users may utilise relates to constructing Bollinger Band plots using either the simple or exponential moving average. In `script_bbands`, this is done for AAPL, and is worthwhile to visualise the stock's volatility and potential buy/sell signals using 2 bands that sit ±1 standard deviations above and below the moving average. It is achieved by the following code:

```
stock_symbol = "AAPL"
stock_data =
final_portfolio.extract_stock(stock_symbol).asset_price_history.copy(de
ep=True)
span = 20 # Medium-term strategy: 2σ (short: 10d/1.5σ; long: 50d/2.5σ)
bollinger_bands(stock_data, simple_moving_average_mean, span)
pyplot.show()
```

The steps are simple and intuitive for the user. The stock symbol for Apple (AAPL) is first specified. The final_portfolio object (which contains data for multiple stocks) is used to extract the historical price data for Apple. The data is deep copied to ensure that any modifications to stock_data do not affect the original data in final_portfolio. The span or window size for the moving average is then set to 20 days, which is typical for a medium-term strategy.

If the user wants, they may later this for different strategies, as per the comment. In this instance, the bollinger_bands function is called with the stock data, the simple moving average function, and the specified span to plot the Bollinger Bands based on SMA.

One run, the plot will appear as:

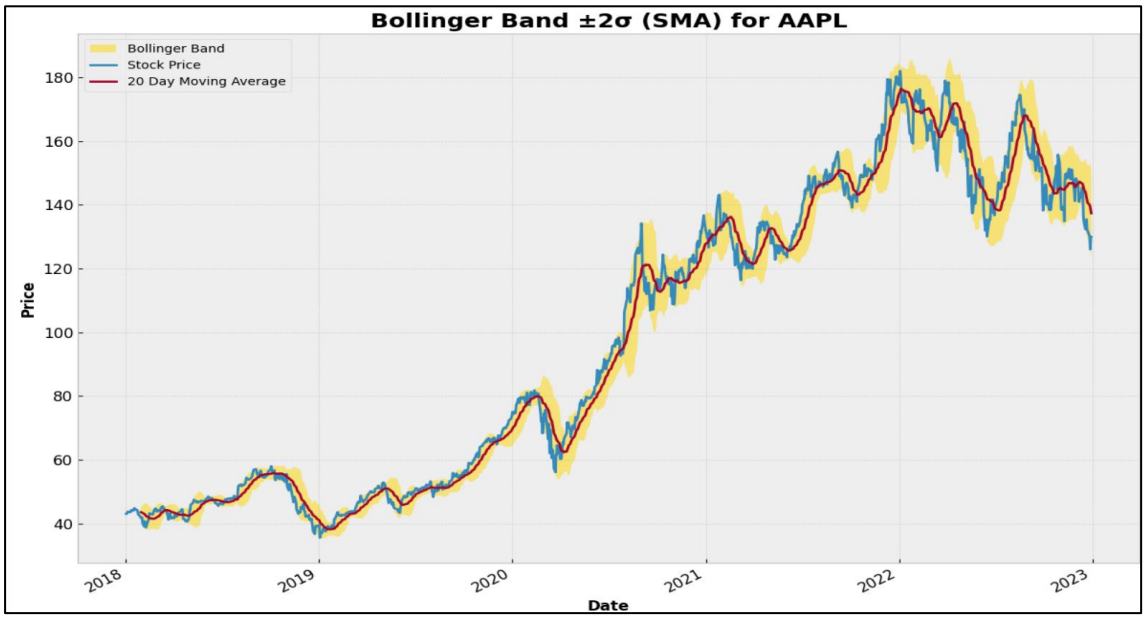


Figure 22: Output from script_bbands: Bollinger Bands for Apple using SMA smoothing.

As highlighted earlier in the report, most trading firms will have a bespoke amalgamation of technical indicators to have a profitable strategy, and simply using the Bollinger Bands is not sufficient nor advisable. Bollinger Bands is more as a proof of concept in the context of EntroPy, and provides some realisations:

(a) Volatility Indicator

When the bands are wide apart, such as between 2022 and 2023, it indicates that the stock is experiencing high volatility. Conversely, when the bands are close together, such as between early-mid 2018, it suggests low volatility.

(b) Overbought/Oversold Conditions:

If the stock price touches or crosses the upper band, it might be considered "overbought," suggesting it might be a good time to sell (i.e. mid-late 2021). If the stock price touches or crosses the lower band, it might be considered "oversold," suggesting it might be a good time to buy (i.e. early-mid 2020).

(c) Squeeze and Breakout Patterns:

A "Bollinger Squeeze" occurs when the bands come close together, indicating that the stock is consolidating and a breakout (sharp price movement) is likely to happen soon. This can be seen in numerous points on the plot. A breakout above the upper band indicates an upward price surge, while a breakout below the lower band indicates a downward price plunge.

8.6 Portfolio Construction with yfinance & DataFrames

Recall that there are two ways in which a portfolio can be constructed and analysed using EntroPy: (1) using an API (yfinance), and (2) using a DataFrame (as a CSV). This discussion will focus on the first approach, but the second is equally simple and similar. A script can be constructed to instead pull the allocation and price data from a csv file, instead of using yfinance as an API, which is detailed in script_pf_composition_csv. For the first approach, the user may refer to `script_pf_composition_api` as a reference. Constructing the portfolio using an API:

```
start_date = datetime.datetime(2018, 1, 1)
end_date = "2023-1-1"
financial_index = "^GSPC"
allocation_df = pd.DataFrame(list(stock_allocation.items()),
columns=["Name", "Allocation"])

# Construct portfolio based on defined stock allocation and date range
portfolio = formulate_final_portfolio(
    stock_symbols=allocation_df["Name"].tolist(),
    apportionment=allocation_df,
    start=start_date,
    end=end_date,
    api_type="yfinance",  # Specify the data source as 'yfinance'
    financial_index=financial_index,)

print(allocation_df)
print(portfolio.portfolio_distribution)
print(portfolio.asset_price_history.head(5))
portfolio.pf_print_portfolio_attributes()
```

The date range for which stock data is to be retrieved is first defined using `start_date` and `end_date`. A financial index ^GSPC, which represents the S&P 500, is chosen in this case. This index will be used to compare the performance of the portfolio against other indexes. The `formulate_final_portfolio` function is called to construct the portfolio. It takes the parameters:

- `stock_symbols`: List of stock symbols derived from the `allocation_df`.
- `apportionment`: The DataFrame containing stock names and their respective allocations.
- `start` and `end`: The date range for which stock data is to be retrieved.
- `api_type`: Specifies the data source as yfinance.
- `financial_index`: The index against which the portfolio's performance will be compared.

The constructed portfolio is stored in the `portfolio` variable, and the script will display:

1. The allocation percentages for each stock.
2. The distribution of stocks in the portfolio.
3. The first 5 rows (modifiable) of historical price data for the assets in the portfolio.
4. A summary of the portfolio's attributes and characteristics.

For the initial allocation defined in MAANG_portoflio.csv, the output of the script is as follows:

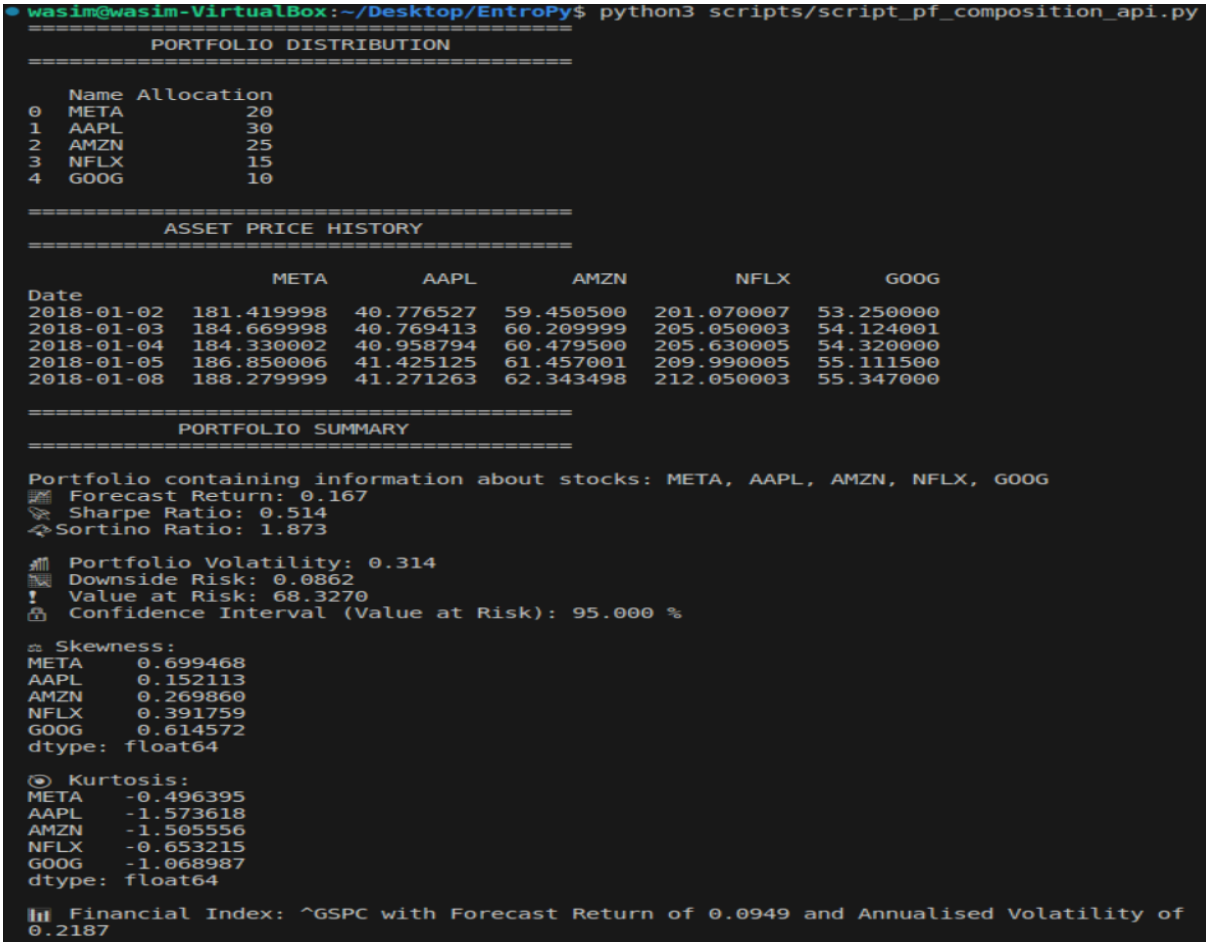


Figure 23: Output from script_pf_composition_api: analyse key portfolio metrics by pulling data from yfinance.

As seen, the system prints out the asset price history of each of the stocks for the first 5 date points. What is most useful to the user is the “Portfolio Summary” section, which gives a breakdown of various crucial metrics for their current portfolio distribution. For instance, over the 5-year period, even with a non-optimised portfolio, the user can expect to have a 16.7% return on their capital, with a middling 0.514 Sharpe ratio. The value of 1.873 for the Sortino ratio indicates that the portfolio is still profitable, and that the investor is being rewarded modestly for taking on a low level of risk. A Sortino ratio >2 is still preferred and considered ideal by the Corporate Finance Institute, whereas a negative ratio suggests that the investor will not be adequately rewarded given the risk-to-reward ratio (CFI Team, 2020). It may be surprising that the portfolio is still quite performant, even without investment, but it should be stated that each of these MAANG stocks are general outliers to most other (tech) companies due to their historical financial performance, vast user-base, and overwhelming market capitalisation (Q.ai, 2023).

Despite this, one can see that this performance is marred by a relatively high volatility of 31.4%; insights can be collected from the high value at risk (VaR) of 68.2% and the lower downside risk of 8.6%. The VaR means that, with 95% confidence, the maximum loss the investor expects to see on their MAANG stocks over a 5-year period is 68.3%. In other words, there is a 5% chance that the stocks will lose more than 68.3% of their value over the next 5 years. This is a significant potential loss, indicating a high level of risk. Conversely, the downside measures the expected loss in the worst-case scenarios. An 8.6% downside risk suggests that, when the market deteriorates, the investor can expect an average loss of 8.6%, which is a more moderate figure compared to the VaR, and it provides a different perspective on the risk of the portfolio.

Given this interpretation the significant difference between VaR and downside risk suggests that while there is a small chance of a very large loss, the average "bad" outcome is much less severe. The high VaR suggests that there are some extreme negative scenarios in the distribution of potential outcomes for MAANG stocks. This is likely due to the inherent volatility of these stocks, and external market factors (Navellier, 2022). The more moderate downside risk suggests that, on average, the losses in the "bad" scenarios are not as extreme as the worst 5% of outcomes. If the user is risk-averse, they now appreciate that they need to optimise and reallocate their capital within their portfolio.

The final metrics to consider are the skewness and kurtosis of each stock. Recall that skewness measures the asymmetry of a distribution around its mean. For the different stocks:

- META, GOOG: Moderately positively skewed. This suggests that their stock returns have had more frequent small losses and occasional large gains.
- AAPL, AMZN: Slightly positively skewed, so stock returns have been somewhat more balanced, but with a slight tendency towards more small losses and occasional large gains.
- NFLX: Positively skewed. Netflix's stock returns have had more frequent small losses and occasional large gains.

Kurtosis measures the "tailedness" of a distribution. From this portfolio:

- META, NFLX: Slightly platykurtic (negative kurtosis), so stock returns have a slightly lower probability of extreme values compared to a normal distribution.
- AAPL, AMZN, GOOG: Platykurtic, so stock returns have a lower probability of extreme values.

Therefore, if an investor prioritises potential for large gains (even if it comes with frequent small losses), then Meta, with the highest positive skewness, may be considered ideal. In practice, this is only one metrics out of many that should influence an investment decision, and one can see from previous optimisations that META is non-optimal. If the investor prioritises stability and want to minimise the probability of extreme returns (either extremely high or extremely low), then Apple or Amazon, with the most negative kurtosis values, may be considered ideal. However, it is paramount to consider other factors, such as fundamental analysis, growth potential, industry trends, and overall risk tolerance. Additionally, past performance is not indicative of future results, so it is crucial to consider a holistic view of each stock.

The final line in the output gives an insight into the performance of the wider index in comparison to individual stocks:

```
Financial Index: ^GSPC with forecast return of 0.0949 and Annualised
Volatility of 0.2187
```

This output provided gives information about the financial index ^GSPC, which is the ticker symbol for the S&P 500 Index. The S&P 500 is a stock market index that measures the stock performance of 500 large companies listed on stock exchanges in the United States. It is often used as a benchmark for the overall U.S. stock market and a gauge of the U.S. economy's health. It is suggested that the forecast return for the S&P 500 over a 5-year period from 2018 – 2023 is 9.49%, and the annualised volatility is 21.87%. Investors can use this information to compare the performance and risk of their portfolio or individual stocks against this benchmark or other index benchmarks (S&P 100, Dow-Jones, NASDAQ composite, etc.).

8.7 Stock Specific Analysis

There is also an example script to highlight properties specific to a singular stock, should the user want a more granular breakdown. The script responsible for this, script_stock_specific_analysis, outputs the following to the terminal:

```
wasim@wasim-VirtualBox:~/Desktop/EntroPy$ python3 scripts/script_stock_specific_analysis.py

Amazon Stock Attributes:
| Property                              | Value
|
+---------------------------------------+-----------------------------------------
---------------+
| Investment Name/Category              | Stock: AMZN
|
| Forecast Return                       | 0.1333
|
| Annualised Volatility                 | 0.3580
|
| Investment Skew                       | 0.2699
|
| Investment Kurtosis/Tailedness        | -1.5056
|
+---------------------------------------+-----------------------------------------
---------------+

Amazon stock data on 2018-01-02 00:00:00:
59.45050048828125

Amazon stock data after 2018-01-01:
Date
2018-01-03    60.209999
2018-01-04    60.479500
2018-01-05    61.457001
2018-01-08    62.343498
2018-01-09    62.634998
                ...
2022-12-23    85.250000
2022-12-27    83.040001
2022-12-28    81.820000
2022-12-29    84.180000
2022-12-30    84.000000
Name: AMZN, Length: 1258, dtype: float64

Amazon stock data for the year 2022:
Date
2022-01-03    170.404495
2022-01-04    167.522003
2022-01-05    164.356995
2022-01-06    163.253998
2022-01-07    162.554001
                ...
```

Figure 24: Output from script_stock_specific_analysis: key metrics and price displays by date for Amazon.

The key measures (such as volatility, kurtosis and skew) for AMZN, over the 5-year period as displayed more clearly in the second block, which is designed to emulate a table. As seen, the user is also able to display AMZN's stock data for:

- A specific date (2018-01-02)
- Dates after a specific date (2018-01-01)
- A specific year (2022)

8.8 Portfolio Analysis Plots

Some of the final plots generated by the scripts are arguably the simplest, but provide an unobtrusive and transparent view of various types of returns for the different portfolio stocks. The key functions, found within script_portfolio_analysis, include:

```
# Construct the portfolio using the loaded stock data
final_portfolio = formulate_final_portfolio(stock_data=stock_info)
ax1 = final_portfolio.asset_price_history.plot()
pyplot.show()
```

```
cumulative_returns = final_portfolio.calculate_pf_cumulative_return()
ax2 = cumulative_returns.plot()
pyplot.show()
```

...

The execution is intuitive in what each function is plotting, the overall script will produce plots for:

1. Portfolio Asset Price History
2. Portfolio Cumulative Returns
3. Daily Percentage Changes of Returns
4. Daily Logarithmic Returns (see repo)
5. Cumulative Logarithmic Returns (see repo)

For the sake of brevity, the subsequent discussion focusses on plots (1) – (3):

1. Portfolio Asset Price History

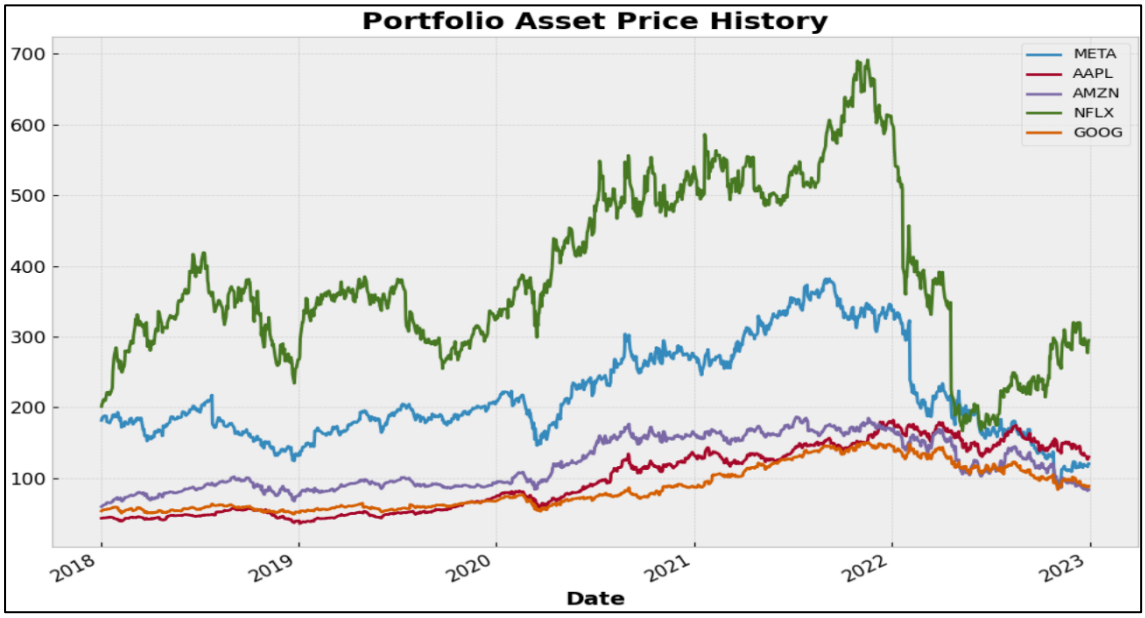


Figure 25: Output from script_portfolio_analysis: asset price history of MAANG portfolio from 2018 to 2023.

Traders and investors who use technical analysis rely heavily on asset price history, and use it to identify patterns, such as head and shoulders or double tops for instance. A head and shoulders pattern identifies potential trend reversals in stock price movements (Hayes, 2023b). It is one of the most reliable trend reversal patterns and is used to predict a shift from a bullish to a bearish trend or vice versa. Here one can see a head and shoulders pattern for NFLX in late 2021. A double top pattern, on the other hand, signals a potential reversal from a preceding uptrend to a new downtrend, and is considered a bearish reversal pattern (Potters, 2023). A double top pattern can be seen, for instance, with META in mid-2020 and mid-late 2021.

Significant spikes or drops in the asset price history can often be correlated with external events, such as earnings announcements, geopolitical events, or changes in economic indicators, which has been discussed in other visualisations. This helps investors understand how susceptible the asset is to external factors.

2. Portfolio Cumulative Returns

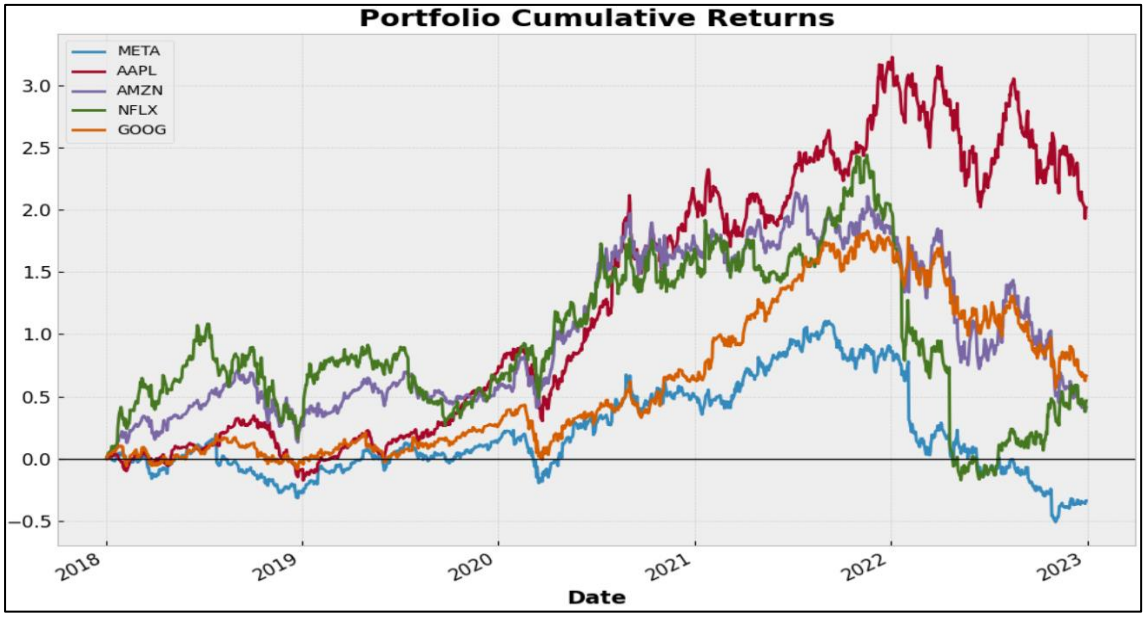


Figure 26: Output from script_portfolio_analysis: cumulative returns of MAANG stocks from 2018 to 2023.

By analysing the cumulative returns of the stocks within their portfolio, investors can see which assets are the biggest contributors to growth. In this case AAPL is firmly in the lead, followed by GOOG, and then by AMZN/NFLX. This can inform decisions about portfolio rebalancing and diversification. Cumulative returns are especially useful for long-term investors as they provide a snapshot of how an investment has performed over extended periods, which in this case is five years.

This can be validated by comparison against another widely-used visualisation tool, FinanceCharts, which displays a consistent trend over the 5-year period for META (FinanceCharts, 2023):

Figure 27: Comparison with FinanceCharts of META's cumulative returns. Other data is similarly validated.

Take META (the blue line in the cumulative returns figure above) as an example, and the user can see how both plots are identical.

3. Daily Percentage Changes of Returns

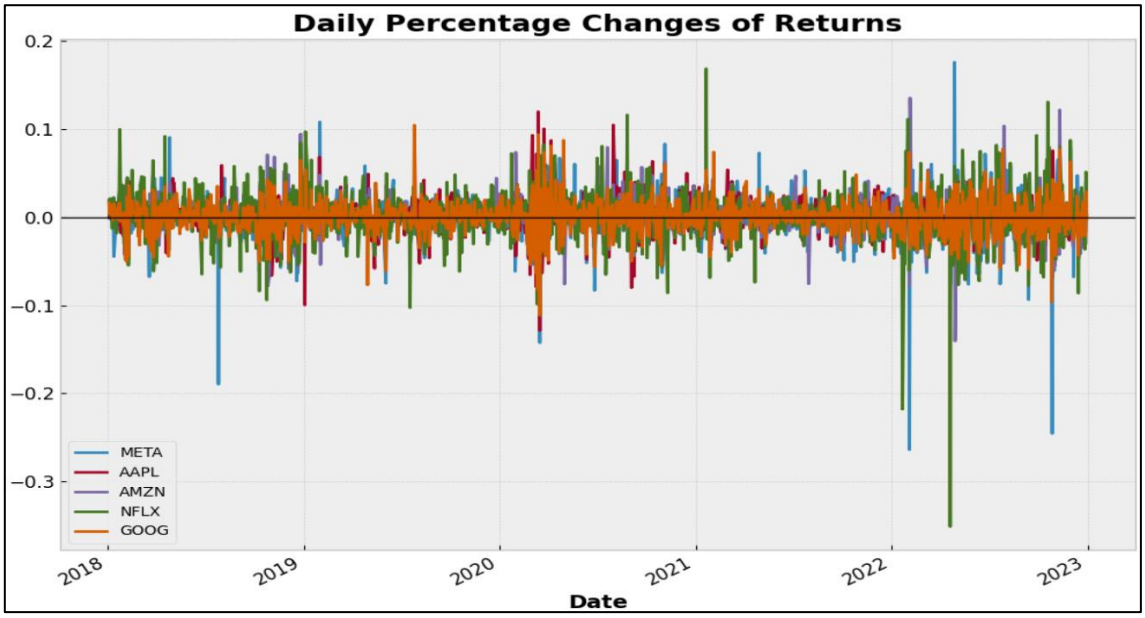


Figure 28: Output from script_portfolio_analysis: daily changes for MAANG portfolio from 2018 to 2023.

The daily percentage changes of returns represent the day-to-day fluctuations in a stock's price or a portfolio's value. Daily returns give an immediate sense of how much a stock or portfolio's value has changed from one day to the next in percentage terms. If a user wishes to compare and contrast the effect of real-world events on stock prices on a short-scale, this plot helps them do so. The variability in daily returns is also a measure of volatility. High fluctuations in daily returns indicate higher volatility, suggesting the stock or portfolio is riskier.

This is supported by other data, as NFLX and META have the largest volatility from this plot, and also the largest volatilities on the MCS/MEF optimised plot discussed at the start of this Chapter. For users who are trading on a daily basis with short-term strategies, by observing this plot, they can gauge the short-term performance of a stock. Daily returns can help investors understand how external events, such as earnings announcements, regulatory changes, or macroeconomic news, impact a stock's price on a day-to-day basis.

9 Evaluation

In Chapter 8, there was a comprehensive demonstration of the capabilities of the portfolio optimisation package. Through screenshots and concise code snippets, the report illustrated how potential users could utilize the package's robust features with minimal coding. While the primary focus of that section was to showcase functionality and user-friendliness, this evaluation will shift towards comparing the delivered features with the original requirements specification, investigating the uniqueness of features relative to existing market solutions, and identifying the inherent limitations of the package. Furthermore, it is essential to highlight that, unlike more general full-stack applications, user-based evaluations for this package presented specific challenges. Being specifically designed for an audience invested in quantitative finance, the package, despite being promoted in suitable communities on platforms such as Discord and Reddit, faced difficulties in gathering extensive user feedback due to its specialised nature.

9.1 Objective-Centric Assessment

Before examining the more granular requirements specification, EntroPy is evaluated against the original design objectives that were outlined in the Introduction in Chapter 1. This gives a broad-brush overview of EntroPy strengths and areas of improvement (AOI) in meeting its overall goals:

1. **Effective Portfolio Construction via the Efficient Frontier**

☑ Strength – Impactful Visualisations: EntroPy excels in generating compelling visual representations of the Efficient Frontier and offers 4 optimised portfolio options for different risk appetites. These visualisations enable users to grasp the trade-offs between risk and return effectively.

☒ AOI – Information Density: While detail provided is invaluable for those familiar with quantitative finance, newcomers who may not want to delve into Chapter 2 or 3's theory might find the wealth of data overwhelming. This could have been improved by implementing interactive features, such as tooltips or zoom functions, which may help users delve deeper into specific areas of interest on the plot. A guided walkthrough or more examples in the documentation may have also proved elucidative.

2. **Risk Assessment using Monte Carlo Simulations**

☑ Strength – Detailed Analysis & Customisability: EntroPy effectively leverages Monte Carlo simulations to provide users with insights into minimum volatility and maximum Sharpe ratios. EntroPy is flexible in number of MCS run iterations, allowing for both quick analyses and more in-depth simulations based on user preferences.

☒ AOI – Computational Demands: With an increase in the number of iterations, there are concerns regarding the computational time and resources required. Future versions of EntroPy could focus on parallelisation to enhance speed and efficiency. A progress feedback bar could also be implemented to make the waiting process more informative.

3. **Trend Analysis for Investment Decisions**

☑ Strengths – Diverse Tools & Actionable Insights: A rich set of tools is offered for trend analysis, encompassing both simple and exponential moving averages with window sizes of 10 – 200 days. This range caters to different investment short and long-term strategies. The integration of buy/sell signals provides actionable recommendations, to help users make informed investment decisions and determine the degree of bullishness and

bearishness in the market. Bollinger Bands also offers users insights into potential overbought or oversold conditions in the market.

☒ AOI – Visual Clutter: Some of the plots, particularly those showcasing simple and exponential moving averages, can be noisy and slightly cluttered. This might make it challenging for users to quickly interpret the data and discern clear patterns, especially as the time-span increases. Simplifying plots by offering options to toggle specific data series on or off, could enhance the user experience.

☒ AOI – Limited Technical Indicators: Bollinger Bands, in the current iteration of EntroPy, is the only technical indicator. For users aiming to construct a more comprehensive and potentially profitable strategy, ability to add more indicators would be beneficial. Future iterations of EntroPy should consider integrating MACD, RSI, or Stochastic Oscillator.

4. Data Retrieval and Pre-processing

☑ Strength – Streamlined Data Extraction & yfinance Integration: With just two scripts totalling approximately <25 lines of code, users can easily retrieve the necessary data, making the setup process straightforward. Support for yfinance ensures that users have access to a reliable and extensive data source.

☒ AOI – Single API Dependency: Relying solely on yfinance for data extraction means EntroPy is vulnerable to any outages, changes, or limitations associated with this particular API. This could potentially disrupt or limit data access for users. Future version may seek to diversify data sources by integrating additional APIs, such as Alpha Vantage or fredapi.

☒ AOI – User Experience in Data Retrieval: While the scripts provided are concise and effective, they may pose a barrier for users unfamiliar with scripting or those seeking a more interactive way to access data. Implementing a GUI could make the process of data retrieval more intuitive and user-friendly.

5. Quantitative Analysis for Financial Research

☑ Strength – Versatile Returns & History: EntroPy provides options to view daily, logarithmic, log-daily, cumulative, and historical returns for custom periods, allowing users to understand asset performance across different dimensions. Users can also quickly glean insights from the asset price history plots.

☑ Strength – Distinct Graphs: The choice to keep each type of analysis in separate graphs enhances readability and user focus. This decision aids in reducing cognitive load and allows users to concentrate on specific areas of interest without being overwhelmed by information.

☒ AOI – Integration of Advanced Quantitative Models: While the current suite of tools is robust for basic and intermediate analysis, there may be a demand for more advanced quantitative models, like factor analysis, cointegration tests, Black-Litterman models, GARCH or Copula models, for users looking for in-depth financial research. Models such as cointegration tests are useful for pairs trading strategies and understanding long-term relationships between assets (Paleologo, 2021). Moreover, GARCH (Generalised Autoregressive Conditional Heteroskedasticity) models are valuable in risk management and option pricing due to their ability to predict volatility.

☒ AOI – Interactivity and Drill-Down Features: While the distinct graphs are clear and effective, providing options for users to interact with the data or drill down for more detailed insights might further enhance the analytical experience. Implementing features such as zoom, pan, or data point tooltips can make the graphs more interactive and informative.

6. Optimal Decision Making

☒ Strength – Balanced Investment Approach: Libraries like SciPy provide users with a scientifically rigorous method for portfolio construction against the objective functions outlined in Chapter 3.

☒ Strength – Modularised Project Structure: EntroPy ensures clarity and ease of customisation by segregating optimisation processes, technical analysis and the management of investor's risk tolerance and return expectations into distinct modules. This enhances transparency and allows for more focused adjustments by users.

☒ AOI – Scenario Analysis: To further enhance the decision-making capabilities, introducing tools that allow users to model and visualise outcomes under different hypnotical market scenarios may be beneficial to understanding effects of these fluctuations on an investment (Murry, 2022). For instance, a suite of tools that modelled each of the following example may be insightful:

- **Interest Rate Hike**: How would each asset in the portfolio react to a 2% increase in interest rates? Historically, how have similar rate hikes impacted these assets?
- **Global Recession**: What would be the anticipated drop in stock values if a recession similar to the one in 2008 were to occur? How would bonds or other defensive assets fare?
- **Tech Surge**: If there is a significant technological breakthrough, how might tech stocks in the portfolio be affected? Would they likely see a 20% increase in value?

☒ AOI – Interactive Risk-Return Adjustments: While risk tolerance and return expectations are modularly managed, an interactive interface allowing users to easily adjust and visualise these parameters could enhance user experience. If this were to be implemented, a real-time environment where risk and return measures could be modified without preparation would make decision-making more instantaneous and effectual.

☒ AOI – Guidance on Optimisation Outcomes: Providing more explicit insights or explanations on the outcomes of the optimisation process can help users better understand and trust the recommendations. Adding tooltips, guides, or detailed breakdowns of optimisation results, outside of a simple terminal display, can ensure that users fully understand and are confident in the decisions they make using EntroPy.

7. Educational Tool for Financial Concepts

☑ Strength – Practical Demonstrations: Tangible demonstrations of financial concepts are seen in Chapter 8, enabling learners to see the real-world applications of theoretical principles.

☑ Strength – Comprehensive Theoretical Foundation: The detailed theoretical content of Chapter 2 and 3 ensures users have a robust grounding in foundational principles, enhancing their understanding when they delve into practical aspects.

☑ Strength – Real-World Analysis of a MAANG Portfolio: By incorporating real-world examples and case studies, specifically an analysis of the MAANG stock portfolio, EntroPy provides context, making the learning experience more relatable.

☒ AOI – Diverse Portfolio Examples: While the MAANG stock portfolio is valuable, including analyses of portfolios from different less-volatile sectors, such as energy or healthcare, or different regions outside of the US might offer a broader perspective on global market dynamics.

☒ AOI – Comparative Analysis Tools: Enabling users to compare different portfolio compositions side-by-side could deepen their understanding of trade-offs and financial decision-making. This may include the implementation of multiple portfolio inputs, plot synchronisation, and dynamic adjustments (i.e. time period “on-the-fly”).

8. Bespoke Selection of Relevant Financial Metrics

A richer comparison between EntroPy and existing implementations is found in the subsequent section. Below is an overview of achievements and limitations:

☑ Strength – Comprehensive Toolkit: EntroPy offers a broad suite of analytical tools, ensuring users have access to both fundamental and technical analysis methods. As discussed, this includes two types of optimisation, technical analysis indicators, and stock-specific data visualisations, providing a well-rounded analytical experience.

☑ Strength – Justified Metric Selection: Unlike many existing solutions that often incorporate metrics without clear rationale, EntroPy's selection of metrics is deliberate. By explaining the choice of metrics, as seen in Chapters 2 and 3, users can better understand the reasoning behind the tools at their disposal, leading to more informed analysis.

☒ AOI – Metric Customization: While EntroPy provides a bespoke selection of financial metrics, allowing users to customise or add their own metrics might cater to more advanced users or specific use cases. This would prove very challenging to do, however, as the current structure of portfolio core modules would potentially require modification to accommodate new metrics.

☒ AOI – Continuous Benchmarking: As new analytical methods emerge, continuously benchmarking EntroPy against other tools can ensure it remains at the forefront of portfolio optimisation solutions. Currently, EntroPy does not utilise any machine learning, which is becoming increasingly popular in other tools via Long Short-Term Memory networks (LTSMs) to predict time series financial data (Sen et al., 2021). Other tools may also incorporate real-time stress testing; for instance, instead of static, historical-based stress tests, real-time backtesting uses live market data and instantaneously applies various adverse scenarios to assess potential portfolio vulnerabilities (Cavestany, 2020). Such a feat is, however, clearly out-of-scope for the time frame given for this project.

9.2 Feature Novelty & Existing Applications

EntroPy was not designed merely as an addition to the existing array of tools but as a direct response to specific gaps and shortcomings observed in the current landscape. This section delves deep into a comparative analysis, juxtaposing EntroPy's unique offerings against popular portfolio optimisation tools available today, on websites such as Github. Through this exploration, the rationale behind EntroPy's conception becomes evident, illuminating its distinct contributions to the open-source community and the broader world of financial technology.

9.2.1 Pyfolio

Pyfolio is the first tool used as inspiration for EntroPy (Quantopian, 2020). Aside from being no longer maintained, due to Quantopian's (the creator) dissolution, Pyfolio is much larger in scope. It contains methods that construct a set of plots, called a tear sheet, for stocks. It also uses yfinance, but includes other metrics such as Calmar ratio, maximum drawdown and Omega ratio. It relied on one of Quantopian's other defunct projects Zipline, for a variety of complex and bespoke algorithms to facilitate actual trading strategy development. Furthermore, Pyfolio assisted in modelling a strategy's "round trip analysis", which was the frequency, duration, and profitability of independent trades.

Pyfolio focussed more on the integration between its backtesting library Zipline with performant and live risk analyses of trading algorithms. EntroPy, on the other hand, provides broader set of tools, including portfolio management, moving averages, and portfolio optimisation. EntroPy does not require any integration with existing technologies, and stands independent of the ecosystem and of other platforms. Pyfolio only offers Bayesian methods to estimate the posterior distribution of certain performance metrics, providing a probabilistic view of potential outcomes. This was the initial inspiration behind research and eventual implementation of Monte Carlo Simulations (MCS), which are better suited to simulating large numbers of random portfolio weights and computing the portfolio returns and volatilities for each to identify the optimal portfolio. Furthermore, Pyfolio does not facilitate any computations or visualisation relating to moving averages, and has no technical indicator analysis, such as EntroPy with its SMA/EMA and Bollinger Bands.

9.2.2 PyPortfolioOpt

PyPortfolioOpt is a library that primarily uses Mean-Variance Optimisation (MVO), Black-Litterman allocation and more novel techniques, such as Hierarchical Risk Parity, to optimise a portfolio (Martin, 2023). It is more direct competitor to EntroPy in that it also provides several objective functions to create a tangency portfolio (maximise Sharpe), to minimise the volatility, to obtain the efficient return (minimise risk for a given return), to obtain the efficient risk (maximise the Sharpe for a given risk), and the option of maximising the user's own quadratic utility based on their risk-aversion.

While a component may be similar, much is still unrelated. For instance, PyPortfolioOpt does not use MCS and provides limited facility for technical analysis. It only focusses on the exponential moving average, and not the simple, or any indicators, or buy/sell signals.

9.2.3 Mlfinlab

Mlfinlab is a machine-learning driven toolbox tailored for financial research (Hudson & Thames, 2021). Its relevant features include backtesting overfitting tools and measures to determine co-dependence in financial data. Mlfinlab was initially examined for its potential in utilising feature engineering for portfolio optimisation, that is to create new features based on historical data (Dwivedi, 2019). Labelling for supervised learning also has great promise in predicting future returns or identifying market anomalies. A large component of Mlfinlab is clustering for aspects such Hierarchical Risk Parity, which is an alternative portfolio optimisation method that seeks to construct portfolios based on the hierarchical structure of asset returns. Clustering was intended to be used in EntroPy to group assets based on similar metrics (such as returns or risk), but this proved too difficult given the time allowed.

After further exploration, Mlfinlab posed a number of limitations that EntroPy aimed to correct in its implementation. As previously discussed, the covariance matrix is a crucial component in portfolio optimisation and risk management. It represents the degree to which returns on two assets move in tandem. EntroPy includes this covariance (or dispersion) matrix, where Mlfinlab does not. The covariance matrix is required for a number of reasons, including its use in portfolio diversification and the Efficient Frontier. In particular, it is used in the Stock and Index classes, to compute the

covariance between returns for the two asset types. For the sake of brevity, no more uses will be discussed in detail, as these have already been elucidated in Chapter 5; they include calculations relation to the variance, beta coefficient, and allocations of capital in the Efficient Frontier.

Several auxiliary issues with MLfinlab have also been remedied in EntroPy's implementation. A serious concern by one user suggests that there is an error with Mlfinlab's implementation of one of the yfinance tickers, where it showed an unrealistic 80% drop in value, followed by a full recovery the next day (Hudson and Thames, 2020). The user further elaborates on the discrepancies in the returns, noting instances where the models returns are over 100%, which does not align with the actual asset performance. Aside from this, Mlfinlab is a paid library, whereas is one of the motivations for creating EntroPy is to allow every user to learn from and access wealth management software. All code is also visible, which promotes transparency, and allows users to inspect the codebase and verify the algorithms and methods implemented.

9.2.4 Riskfolio-Lib

Riskfolio-Lib is similar to PyPortfolioOpt in that it facilitates the creation of investment portfolios based on mathematically complex models with limited effort. It is fundamentally different, however, as it contains a plethora of functionality related to the Kelly Criterion, Hierarchical Risk Parity, and Hierarchical Equal Risk Contribution, in addition to the Black-Litterman model. Like PyPortfolioOpt, Riskfolio-Lib has been invaluable to learn theory from, but extremely limited in implementation due to it being built in cvxpy (C++) (CVXPY, n.d.). Its primary focus is on risk, comprising well over 60 convex metrics (Cajas, 2023).

Unlike EntroPy, it does not provide any technical analysis tools for moving averages or Bollinger Bands, and does not optimise using Monte Carlo Simulations, which ultimately helps to model the "worst-case" randomly sampled outcomes. Despite lacking these features, Riskfolio-Lib can be extremely overwhelming for beginners to quantitative finance, due to its overabundance of optimisation techniques and models. This can lead to decision paralysis, where users are unsure of which options to select or how to proceed. This can also lead to potential misuse, without a clear understanding of each risk measure or optimisation method, which could lead to suboptimal or incorrect results. Looking through documentation, one can see that for Black-Litterman optimisation alone, a user has 3 options of standard, Bayesian and Augmented BM. Overall, there are ~20 optimisation techniques, which can leave users overwhelmed.

Furthermore, the project's maintainer charges fees for all consultancies not related to errors in the source code. This means that users who need guidance or face issues with their implementations might incur additional costs, which could be a deterrent for some. Again, Riskfolio-Lib seems to enforce a barrier to entry, as Mlfinlab did. For novices or those new to the field, the potential of incurring fees for seeking help might discourage them from using the library or delving deeper into quantitative finance. There is also the issue of transparency and clarity; while the fee structure is clear, users might be uncertain about what constitutes a "simple error" versus a "complex error," leading to potential confusion about the costs they might incur.

9.2.5 Quant-trading & QuantPy

The final two software implementations pertain to Monte Carlo Simulations, which is a rarity in portfolio optimisation packages. The first, Quant-trading, is more of a collection of trading scripts using a variety of technical analysis tools, such as moving averages, Bollinger Bands, MACD, and statistical arbitrage (TM, 2022). It was used as the basis of constructing modules relating to technical analysis in EntroPy, but was also used as a cautionary tale in using Monte Carlo Simulations. The author egins by highlighting the misconception that Monte Carlo simulation can predict stock prices notes its excellent performance in studying stochastic processes, but then highlights a common argument and misconception, in that stock prices can be seen as a Wiener Process, implying that Monte Carlo simulation can predict stock prices (TM, 2018). However, this assumption is flawed as stock prices are not continuous in time; markets close, and overnight

volatility exists. The author tested the Monte Carlo simulation on General Electric's stock, which had a significant price drop in 2018. The simulation failed to predict the scale of the downslide. Another test on Nvidia's stock from 2006 to 2009, which saw a 75.6% drop in 2008, also showed that the simulation could not predict the scale of the decline.

Therefore, instead of using Monte Carlo to predict stock prices, like many projects had attempted previously, it is instead used to get an overall idea of distribution trends. Monte Carlo simulations can be used to assess the risk of a portfolio under various scenarios. By simulating thousands or even millions of different possible economic scenarios, one can gauge how a portfolio might perform under adverse conditions. Value at Risk (VaR) is also a widely used risk management tool, and Monte Carlo simulations can be employed to estimate VaR for a portfolio, providing a probabilistic view of the potential downside risk. Monte Carlo is also excellent in estimating capital budgeting, and by simulating various scenarios for project returns, investors can make an informed choice if their selected portfolio weight aligns with the randomly generates distribution of Monte Carlo. In the optimisation carried out for the MAANG stocks in Chapter 8, for example, one can see that NFLX and META do not fall within the randomly-distributed results, which suggests that these stocks present a non-probable (and highly volatile) return profile, which is true.

QuantPy proved instrumental, alongside Stack Overflow, in learning how Monte Carlo Simulations could be implemented to optimise a portfolio. The process of computing the mean returns and voariance, defining weights for the portfolio, and sampling uncorrelated variables (in this case the portfolio allocation) was the foundation of generating the Monte Carlo module (QuantPy, 2021).

9.3 Requirements Fulfilment

Beyond the initial analysis, a more data-driven method is implemented to assess the system. Below, the system's performance is compared against both functional and non-functional requirements listed in the report's appendix. Each requirement was categorised as either passed or failed. For the functional requirements:

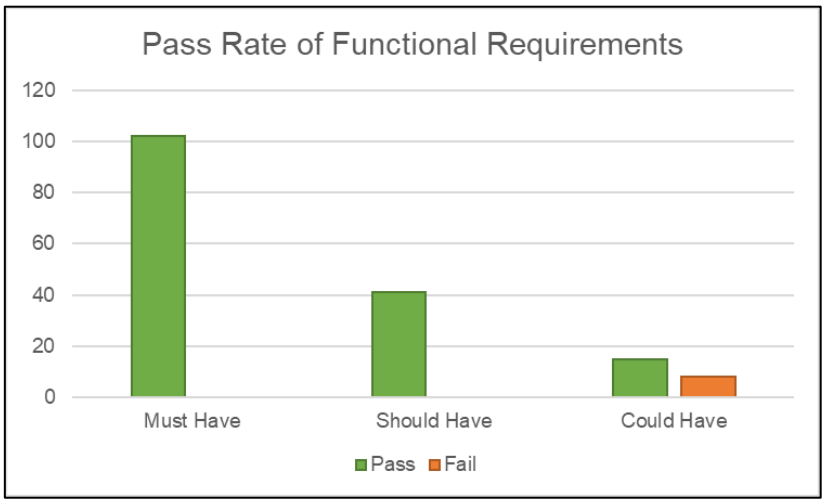


Figure 29: Pass rate of must, should, and could have functional requirements.

The overwhelming majority of functional requirements were implemented successfully. Out of the 166 “non-Won’t Have” requirements, 158 were satisfied, which comprised of 102 “Must Have”, 41 “Should Have”, and 15 “Could Have” requirements with a success rate of ~95%. This can be attributed to the robust research phase that was conducted prior to any programming or requirements specification, which ensured that only relevant and achievable functionalities were implemented. Nonetheless, 8 out of 23 “Could Have” requirements failed with a failure rate of ~34%, as there was insufficient time to implement these.

These requirements can be grouped according to their Table in Appendix A:

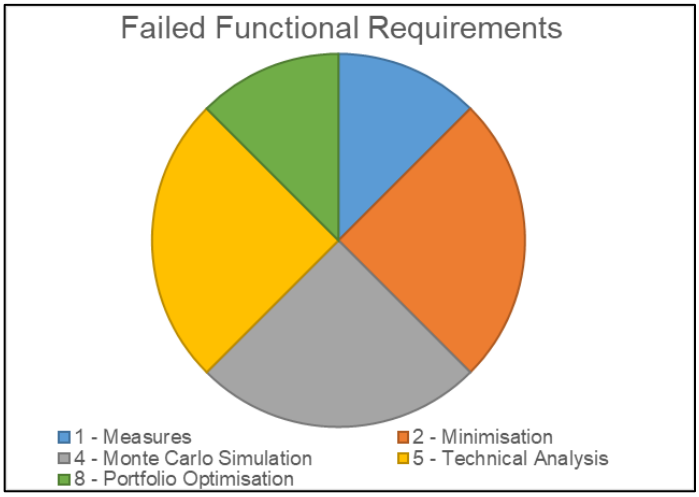


Figure 30: Granular breakdown of failed functional requirements.

Due to the small number of failures, a succinct analysis can be done of each area:

- 1 – Measures
  - Fail: 1.19. Logging was not implemented due to prioritisation of other development functionalities, such as those pertaining to financial computations. Despite this, every module has an extensive suite of standard Pythonic error-handling mechanisms that display immediately error messages to users.
- 2 – Minimisation
  - Fail: 2.10. Addition of additional parameters is permissible for different risk-free rates, but not current tested for entirely new financial metrics. To add these, the user would need to ensure that they are utilised downstream in modules relating to the Efficient Frontier and Portfolio Optimisation.
  - Fail: 2.11. Similar to Measures logging – see requirement 1.19.
- 4 – Monte Carlo Simulation
  - Fail: 4.6. Parallelisation for high numbers of iterations for MCS was not implemented due to time constraints. As a result, especially seen through running the tests, a large number of runs will result in moderate slowdown.
  - Fail: 4.7. Implementing convergence and subsequent termination was attempted in earlier iterations, but was ultimately unsuccessful due to difficulties implementing the programming logic.
- 5 – Technical Analysis
  - Fail: 5.9. The support for additional moving average types (triangular, weighted), was omitted due to the effectiveness of the current SMA and EMA being sufficient to demonstrate the technical analysis component as a proof of concept. Time limitations prevent any further implementations too.
  - Fail: 5.10. As discussed earlier, the lack of zooming and UI/UX elements that promote usability was a low priority compared to the complex program logic. The plots created are serviceable, but could be improved if more time was available.
- 8 – Portfolio Optimisation
  - Fail: 8.35. Ultimately, implementing stress testing, backtesting and scenario analysis is a major undertaking, requiring sophisticated techniques such as backpropagation, bootstrapping, and drawdowns. This was more set as a demanding stretch goal for EntroPy. These, alongside other models such as the Kelly Criterion, Black-Litterman Model, and Hierarchical Risk Parity, would require multiple developers with more time and specialised domain knowledge.

The non-functional requirements are more difficult to underpin and quantify, due to their representation of large concepts such as “scalability”, or “reliability”, as well as the challenge of testing such ideas. Nonetheless, their pass/fail rate may be represented below:

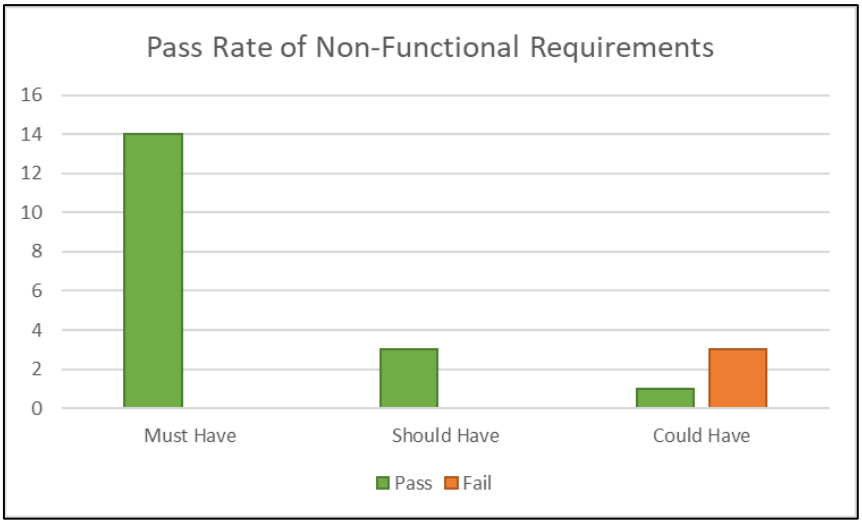


Figure 31: Pass rate of must, should, and could have non-functional requirements.

Out of 21 non-functional requirements, 18 passed with a pass rate of 86%. The failed requirements are delineated by:

- 1.6 – Flexibility and Configurability
  - Fail: 1.6.1. Again, no “friendly” UI/UX could have been implemented in this project due to time constraints, but consciously, to allow users to have more freedom in the data and insights they want to display.
- 2.6 – Portability
  - Fail: 2.6.2. EntroPy was not operable on Windows. There were attempts to remedy this, but the dependencies used had compatibility issues with the VSCode Windows environment. Given more time, this issue would be resolved. Nonetheless, Linux (XBuntu) was prioritised, due to most quantitative developers programming professionally on this operating system (QuantStart, 2022).
  - Fail: 2.6.3. EntroPy was not tested on MacOS. The development team did not have access to MacOS.

9.4 Limitations and Future Work

EntroPy is a portfolio optimisation package that has demonstrated its strength and versatility throughout its developmental stages. Yet, it is not without its shortcomings and areas that need further polishing.

One of the challenges faced by EntroPy is its niche target audience. It is designed specifically for those immersed in quantitative finance. This narrow focus has made it difficult to obtain widespread user feedback, even after efforts to promote it within pertinent communities. For those well-acquainted with quantitative finance, EntroPy is a valuable of information. However, for novices, the sheer volume of details can be daunting. The inclusion of interactive elements, such as tooltips or zoom capabilities, may have made this data more user-friendly.

Another concern is the computational strain. As users increase the iterations in Monte Carlo Simulations, there is a growing worry about the computational time and resources it demands. Visually, some of the plots, particularly those displaying moving averages, can appear congested, complicating quick data interpretation. In its current form, EntroPy only offers Bollinger Bands as a technical indicator. It also exclusively depends on yfinance for data extraction, which exposes it to potential risks if there are any disruptions or modifications to this API. The data retrieval scripts

provided might pose challenges for those not well-versed in scripting. There is also a noticeable demand for more sophisticated quantitative models, such as factor analysis, cointegration tests, and GARCH models. While allowing users to modify or introduce their metrics could appeal to a more advanced audience, it is a formidable task considering EntroPy's existing framework. Additionally, its compatibility has not been tested on Windows or MacOS, restricting its adaptability across various operating systems.

Looking ahead, there are several avenues for improvement. Parallelisation could be beneficial in boosting the speed and efficiency of Monte Carlo Simulations. A more intuitive user experience could be achieved by introducing a GUI for data retrieval, interactive risk-return modifications, and clearer insights into optimization results. Expanding the portfolio examples to encompass various sectors or regions could provide users with a more comprehensive view of global market trends. There is also room for integrating more technical indicators, such as MACD, RSI, or the Stochastic Oscillator. To minimise reliance on a single API, future versions might consider incorporating other APIs like Alpha Vantage or fredapi.

Advanced quantitative models, including cointegration tests, Black-Litterman models, GARCH, and Copula models, could be integrated to cater to those keen on thorough financial research. Tools that enable users to model and visualise outcomes under diverse market conditions could significantly enhance their decision-making prowess. Continuous benchmarking against emerging analytical methods is required to ensure that EntroPy remains at the forefront of its field. Ensuring compatibility across various platforms, including Windows and MacOS, should be a priority in future versions.

10 Conclusion

To conclude, the development of EntroPy has been successful in providing a suite of tools for users to optimise their portfolio within, through MVO in the Efficient Frontier and/or via Monte Carlo Simulations. EntroPy has met all previous set requirements in the conception stages of the project. Many complex financial concepts have been abstracted into a powerful and low-code package for users to employ for optimisations and technical analyses.

EntroPy is advantageous it effective visualization and analysis, and the package excels in generating intuitive visual representations of complex financial concepts like the Efficient Frontier. Another of EntroPy's core strengths lies in its adaptability, allowing users to tailor their analyses to specific needs, whether they desire quick insights from the asset price history or deep dives into data, such as the risk-adjusted returns of a stock. Furthermore, with its blend of theoretical foundations and practical demonstrations, EntroPy serves as both an analytical tool and an educational platform. The analysis of real-world portfolios, like the MAANG stock portfolio, offers users contextual insights, bridging the gap between theory and application.

EntroPy has been one of the most challenging pieces of work undertaken by the development team, requiring extensive research into the mathematics and finance that underpins many of the implementation choices in the software. The use of Python has eased this due to its relative flexibility and ability to utilise powerful libraries, such as Numpy and Pandas, to handle many of the other complex programming tasks through their collection of efficient and performant functions, such as skew, squeeze, kurt, and so on.

In comparison with existing contributions, this is all done within one framework and is offered as a free, modular and pedagogical product, intended to be built upon by more experienced developers in the quantitative finance space. Drawing parallels with existing tools, EntroPy has successfully filled several gaps in the current landscape. While platforms like Pyfolio, PyPortfolioOpt, and Mlfinlab have their merits, EntroPy distinguishes itself with its holistic approach, addressing both fundamental and technical analysis needs. Its independence, transparency, and focus on user empowerment are notable.

In terms of requirements fulfilment, EntroPy has a commendable success rate, achieving approximately 95% of its functional requirements. This speaks volumes about the robust research and development phases, ensuring the implementation of pertinent functionalities. However, some aspirations, especially in the "Could Have" category, remain unfulfilled, providing clear directives for future work.

The comprehensive testing approach employed in the development of EntroPy underscores its reliability and precision. Characterized by meticulous white-box unit tests, testing not only ensured each module's functional correctness but also guaranteed the system's overall robustness, with 100% pass rate in over 100 unit tests. While the project's vastness resulted in an extensive test suite, the focus on key modules provided a clear snapshot of the system's integrity. Testing was not just an afterthought; it was an integrated part of the development process. This instils confidence in EntroPy's capabilities to handle financial portfolio management tasks with unparalleled accuracy.

Looking forward, as with any software project of this magnitude, there are challenges and areas of improvement. EntroPy, in its current form, is dense with information, potentially overwhelming for those new to quantitative finance. Moreover, with scalability as a concern, especially regarding the Monte Carlo Simulations, future versions must investigate methods to optimize computational efficiency. With the aid of more experienced developers, advanced computation techniques such as GARCH and Black-Litterman models are on the horizon, aiming to satisfy to those with a passion for quantitative finance.

11 Bibliography

Alrabadi, D.W.H. & Aljarayesh, N.I.A. (2015). *Forecasting Stock Market Returns Via Monte Carlo Simulation: The Case of Amman Stock Exchange*. Jordan J. Bus. Adm., 11, pp. 745–756.

AltexSoft (2021). *Functional and Non-functional Requirements: Specification and Types*. AltexSoft. Available at: https://www.altexsoft.com/blog/business/functional-and-non-functional-requirements-specification-and-types/ [Accessed: 25 September 2023].

Anagnostidis, P. (2018). *Testing the white noise hypothesis of stock returns*. Economic Modelling. [Online] Available at: https://www.sciencedirect.com/science/article/abs/pii/S0264999318306849 [Accessed 24 September 2023].

Ang, A., Hodrick, R. J., Xing, Y., & Zhang, X. (2006). *'The Cross-Section of Volatility and Expected Returns*'. Journal of Finance, 61(1), pp. 259-299.

Artzner, P., Delbaen, F., Eber, J.-M., & Heath, D. (1999). *Coherent Measures of Risk*. Mathematical Finance, 9(3), 203–228.

Atlassian. (2023). *Agile*. [Accessed 29 September 2023]. Available at: https://www.atlassian.com/agile.

Barinov, A. (2014). *Trading Strategies and Financial Models*. Cumming: Conley Smith ePublishing, LLC.

Bartz, D., Hatrick, K., Hesse, C.W., Müller, K-R. and Lemm, S. (2013). *Directional Variance Adjustment: Bias Reduction in Covariance Matrices Based on Factor Analysis with an Application to Portfolio Optimization*. PLoS ONE, 8(7), p.e67503. Available at: https://doi.org/10.1371/journal.pone.0067503.

Beattie, A. (2021) Understanding the history of the modern portfolio, Investopedia. Available at: https://www.investopedia.com/articles/07/portfolio-history.asp (Accessed: 23 September 2023).

Behan, A. (2022). *Harry Markowitz*. Available at: https://www.investopedia.com/terms/h/harrymarkowitz.asp [Accessed 26 Sep. 2023].

Belyaev, E. (2021). *Advantages and Disadvantages of the Famous Bollinger Bands Indicator.* MQL5. Available at: https://www.mql5.com/en/blogs/post/745543 [Accessed 25 Sept. 2023].

Best, M.J. & Grauer, R.R. (1991). *Sensitivity analysis for mean-variance portfolio problems*. Management Science, 37(8), pp. 980–989.

Bhalla, A. S. (1995). *Collapse of Barings Bank: Case of Market Failure*. Economic and Political Weekly, 30(13), pp. 658–662. Available at: http://www.jstor.org/stable/4402560 [Accessed 25 September 2023].

Bland, G. (2021) *Yahoo Finance API Guide*, *AlgoTrading101*. Available at: https://algotrading101.com/learn/yahoo-finance-api-guide/ [Accessed: 25 September 2023].

Bodie, Z., Kane, A. & Marcus, A.J. (2012). *Investments*. 10th ed. Berkshire: McGraw-Hill Education.

Bodie, Z., Kane, A., & Marcus, A.J. (2017). *Investments* (11th ed.). McGraw Hill. p. 9. ISBN 9781259277177.

Box UK (n.d.) *Guide to Non-Functional Requirements: Types and Examples*. Box UK. Available at: https://www.boxuk.com/insight/guide-to-non-functional-requirements-types-and-examples/ [Accessed: 25 September 2023].

Brealey, R. A., Myers, S. C., and Franklin, A. (2012). *Principles of Corporate Finance*. New York: McGrawHill.

Brownlee, J. (2020). *How to Use argmax for Machine Learning*. Machine Learning Mastery. Available at: https://machinelearningmastery.com/argmax-in-machine-learning/ [Accessed 26 Sep. 2023].

Cajas, D. (2023). *Riskfolio-Lib*. GitHub. [Accessed 29 September 2023]. Available at: https://github.com/dcajasn/Riskfolio-Lib.

Caroline, J. (2021). *What Are MAANG Stocks?* [Online]. Available at: https://www.coinspeaker.com/guides/what-are-maang-stocks/ [Accessed 29 Sep. 2023].

Cavestany, R. (2020). *Operational Risk Capital Models* (2nd ed.). Risk Books. [Accessed 24 September 2023].

CFI Team. (2020). *Sortino Ratio*. Corporate Finance Institute. https://corporatefinanceinstitute.com/resources/wealth-management/sortino-ratio-2/

CFI Team. (2021). *Utility Maximization. Corporate Finance Institute*. [Online] Available at: https://corporatefinanceinstitute.com/resources/economics/utility-maximization/ [Accessed 24 September 2023].

Chan, E. (2013). Algorithmic Trading - Winning Strategies and their Rationale. New York: John Wiley & Sons.

Chen, J. (2022). *Crossover Definition*. Investopedia. Available at: https://www.investopedia.com/terms/c/crossover.asp [Accessed 26 Sep. 2023].

Cheng, L., Shadabfar, M. and Sioofy Khoojine, A. (2023). *A State-of-the-Art Review of Probabilistic Portfolio Management for Future Stock Markets*. Mathematics, 11, 1148. Available at: https://doi.org/10.3390/math11051148.

Chong, J., Jin, Y., & Phillips, G.M. (2013). *The Entrepreneur's Cost of Capital: Incorporating Downside Risk in the Buildup Method*. MacroRisk Analytics. Available at: [https://www.macrorisk.com/wp-content/uploads/2013/04/MRA-WP-2013-e.pdf] [Accessed 25 September 2023].

Chopra, V. & Ziemba, W. (1993). *The effects of errors in means, variances and covariances on optimal portfolio choice*. Journal of Portfolio Management, 19, 6-12.

Clegg, D. & Barker, R. (1994). *Case Method Fast-Track: A RAD Approach*. Addison-Wesley. ISBN 978-0-201-62432-8.

Cochrane, J.H. (2014). *Eugene F. Fama, Efficient Markets, and the Nobel Prize.* The University of Chicago Booth School of Business. [Online] Available at: https://www.chicagobooth.edu/review/eugene-fama-efficient-markets-and-the-nobel-prize [Accessed 24 September 2023].

Cont, R., Deguest, R., & Scandolo, G. (2010). *Robustness and sensitivity analysis of risk measurement procedures*. Quantitative Finance, 10(6), 593-606. DOI: 10.1080/14697681003685597.

Corradi, V., Distaso, W., & Fernandes, M. (2011). *Conditional alphas and realized betas*. University of Warwick, Imperial College London, Queen Mary University of London. Available at: https://www.birmingham.ac.uk/Documents/college-social-sciences/business/economics/events/conditional-alphas.pdf [Accessed 29 September 2023].

Croarkin, C., Tobias, P., and Zey, C. (2001). *Engineering Statistics Handbook*. National Institute of Standards and Technology.

CVXPY. (n.d.). *Introduction*. [Accessed 29 September 2023]. Available at: https://www.cvxpy.org/tutorial/intro/index.html.

Damodaran, A. (n.d.). *Session 4: Descriptive Statistics*. Stern School of Business, NYU. [Online] Available at: https://pages.stern.nyu.edu/~adamodar/pdfiles/Statistics101/Slides/Session4.pdf [Accessed 24 September 2023].

Dillikar, S. (2021). *How to Calculate Bollinger Bands of a Stock with Python*. Medium. Available at: https://medium.com/codex/how-to-calculate-bollinger-bands-of-a-stock-with-python-f9f7d1184fc3 [Accessed 26 Sep. 2023].

Drake, P. P. and Fabozzi, F. J. (2010). The Basics of Finance: An Introduction to Financial Markets, Business Finance, and Portfolio Management. New Jersey: John Wiley & Sons.

Dugar, D. (2018). *Skew and Kurtosis: 2 Important Statistics terms you need to know in Data Science*. Codeburst. [Online] Available at: https://codeburst.io/2-important-statistics-terms-you-need-to-know-in-data-science-skewness-and-kurtosis-388fef94eeaa [Accessed 24 September 2023].

Dwivedi, N. (2019). *Predicting Stable Portfolios Using Machine Learning*. Medium. [Accessed 29 September 2023]. Available at: https://medium.com/sfu-cspmp/predicting-stable-portfolios-using-machine-learning-f2e27d6dbbec.

Eassa, A. (2017). *Why Apple Inc. Is So Profitable*. The Motley Fool. Available at: https://www.fool.com/investing/2017/03/23/why-apple-inc-is-so-profitable.aspx [Accessed 26 Sept. 2023].

Efremov, A. (2020). *Lecture 6: Moments, Skewness, Kurtosis, Median, Quantiles, Mode*. Tomsk Polytechnic University. Available at: https://portal.tpu.ru/SHARED/a/ALEXYEFREMOV/eng/teaching/ProbTheory2020-Lecture6.pdf. [Accessed 24 Sept. 2023].

Elliott, R.J. and Kopp, P.E. (1999). *Mathematics of Financial Markets*. Springer Finance. [Online] Available at: http://www.untag-smd.ac.id/files/Perpustakaan_Digital_1/FINANCE%20Mathematics%20of%20financial%20markets%202nd%20ed.pdf [Accessed 25 September 2023].

Fabozzi, F.J., Kolm, P.N., Pachamanova, D., & Focardi, S.M. (2007). *Robust Portfolio Optimisation and Management*. New Jersey: John Wiley & Sons.

Fama-French. (2019). *Are Stock Returns Normally Distributed?* Dimensional. [Online] Available at: https://famafrench.dimensional.com/questions-answers/qa-are-stock-returns-normallydistributed.aspx [Accessed 24 September 2023].

Fernando, D. et al. (2003) *The global growth of mutual funds - World Bank.* Available at: https://documents1.worldbank.org/curated/en/342341468781751457/108508322_20041117143517/additional/multi0page.pdf (Accessed: 23 September 2023).

Fidelity. (2023). *Exponential Moving Average (EMA)*. Available at: https://www.fidelity.com/learning-center/trading-investing/technical-analysis/technical-indicator-guide/ema [Accessed 24 Sept. 2023].

FinanceCharts.com. (2023). *Meta Platforms (META) Total Return: 114.65% (TTM)*. Available at: https://www.financecharts.com/stocks/META/performance/total-return [Accessed 24 Sep. 2023].

Forbes (2023). Will Netflix Stock Return To Pre-Inflation Shock Highs Of Over $650? Forbes. Available at: https://www.forbes.com/sites/greatspeculations/2023/09/19/will-netflix-stock-return-to-pre-inflation-shock-highs-of-over-650/?sh=63695c555ff1 [Accessed 24 September 2023].

Frankfurter, G.M., Phillips, H.E., & Seagle, J.P. (1971). *Portfolio selection: the effects of uncertain means, variances, and covariances*. Journal of Financial and Quantitative Analysis, 6(05), pp. 1251–1262.

FXOpen. (2023). *Simple Moving Average (SMA): Definition and Examples*. Available at: https://fxopen.com/blog/en/simple-moving-average-sma-definition-and-examples/ [Accessed 25 Sept. 2023].

Ganti, A. (2023) *Asset Classes Definition. Investopedia*. Available at: https://www.investopedia.com/terms/a/assetclasses.asp [Accessed 29 September 2023].

Gezer, B. (2022). *Why pytest?*. Medium. Available at: https://medium.com/beyn-technology/why-pytest-e7f04145155f [Accessed 26 Sep. 2023].

Ghysels, E. (1998). *On Stable Factor Structures in the Pricing of Risk: Do Time-Varying Betas Help or Hurt?* Journal of Finance, 53(2), pp. 549-573.

Glendrange, G. & Tveiten, S. (2016). *Testing the Performance of Simple Moving Average With the Extension of Short Selling*. University of Agder, School of Business and Law, Department of Economics and Finance. Available at: https://core.ac.uk/download/pdf/225892536.pdf [Accessed 25 Sept. 2023].

Goldfarb, D. (2023). *mplfinance*. Available at: https://github.com/matplotlib/mplfinance [Accessed 26 Sep. 2023].

Goyal, A. & Santa-Clara, P. (2003) *Idiosyncratic Risk Matters*! The Journal of Finance, 58(3), pp. 975–1007. Available at: http://www.jstor.org/stable/3094569 [Accessed 29 September 2023].

Grootveld, H. & Hallerbach, W. (1999). Variance vs downside risk: Is there really that much difference? European Journal of Operational Research, 114(2), pp. 304–319.

Harris, S. (2021). *Avoiding apply()ing yourself in Pandas*. Towards Data Science. Available at: https://towardsdatascience.com/avoiding-apply-ing-yourself-in-Pandas-a6ade4569b7f [Accessed 26 Sep. 2023].

Harrison, M. (2022). *Professional Pandas: the Pandas assign method and chaining*. Ponder.io. Available at: https://ponder.io/professional-Pandas-the-Pandas-assign-method-and-chaining/ [Accessed 26 Sep. 2023].

Haugh, M. (2016) *The Black-Scholes Model*, IEOR E4706: Foundations of Financial Engineering. Available at: http://www.columbia.edu/~mh2078/FoundationsFE/BlackScholes.pdf (Accessed: 26 September 2023).

Hayes, A. (2023a). *Covariance: Formula, Definition, Types, and Examples*. Investopedia. [Online] Available at: https://www.investopedia.com/terms/c/covariance.asp [Accessed 24 September 2023].

Hayes, A. (2023b). *Head And Shoulders Pattern*. Investopedia. Available at: https://www.investopedia.com/terms/h/head-shoulders.asp [Accessed 24 Sep. 2023].

Heller, P. (2023) *Product portfolio optimisation enables growth, Sopheon*. Available at: https://www.sopheon.com/blog/product-portfolio-optimisation-to-enable-growth-at-speed (Accessed: 23 September 2023).

Hudson & Thames. (2021). *mlfinlab*. GitHub. [Accessed 24 September 2023]. Available at: https://github.com/hudson-and-thames/mlfinlab

Hudson and Thames. (2020) *Issue #441*. GitHub. Available at: https://github.com/hudson-and-thames/mlfinlab/issues/441 [Accessed 29 September 2023].

Hult, H., Lindskog, F., Hammarlid, O., & Rehn, C.J. (2012). *Risk and Portfolio Analysis: Principles and Methods*. Stockholm: Springer Science & Business Media.

Ifedigbo, S. (2023) *Global assets under management set to rise to $145.4 trillion by 2025, PwC*. Available at: https://www.pwc.com/ng/en/press-room/global-assets-under-management-set-to-rise.html (Accessed: 23 September 2023).

Inani, S.K. (2016). *Four moments of distribution: Mean, Variance, Skewness, and Kurtosis. Learning Econometrics*. [Online] Available at: http://learningeconometrics.blogspot.com/2016/09/four-moments-of-distribution-mean.html [Accessed 24 September 2023].

Intaver. (n.d.). *How Many Monte Carlo Simulations Are Required?* Available at: https://intaver.com/blog-project-management-project-risk-analysis/how-many-monte-carlo-simulations-are-required/ [Accessed 26 Sep. 2023].

IronFX. (2023). *Meta has lost $700 billion in market value*. Available at: https://www.ironfx.com/en/meta-has-lost-700-billion-in-market-value/ [Accessed 25 Sep. 2023].

Jiao, W. (2003). *Portfolio Resampling and Efficiency Issues*. Master's Thesis, Humboldt-Universität zu Berlin, School of Business and Economics. Available at: https://edoc.hu-berlin.de/bitstream/handle/18452/14691/jiao.pdf [Accessed 26 September 2023].

Jin, Y., Qu, R. & Atkin, J. (2016) C*onstrained Portfolio Optimisation: the state-of-the-art Markowitz Models*. ASAP Group, School of Computer Science, The University of Nottingham. Available at: https://www.cs.nott.ac.uk/~pszrq/files/ICORES16mv.pdf [Accessed 24 September 2023].

Johansson, A. and Petersson, R. (2017). *Beta Based Portfolio Construction: Stock Selection Based on Upside and Downside Market Risk*. Örebro University, School of Business, Economics, Master Thesis. [Online] Available at: https://www.diva-portal.org/smash/get/diva2:1189707/FULLTEXT01.pdf [Accessed 24 September 2023].

Johnston, M. (2023). *How Apple Makes Money*. Investopedia. Available at: https://www.investopedia.com/how-apple-makes-money-4798689 [Accessed 26 Sept. 2023].

Jones, S.L. and Netter, J.M. (2023). *i* Econlib. [Online] Available at: https://www.econlib.org/library/Enc/EfficientCapitalMarkets.html [Accessed 24 September 2023].

Jorion, P. (2006). *Value at Risk: The New Benchmark for Managing Financial Risk.*

Kahneman, D. and Tversky, A. (1979). *Prospect Theory: An Analysis of Decision under Risk.* Econometrica, 47(2), pp. 263–291. Available at: https://doi.org/10.2307/1914185 [Accessed 24 September 2023].

Kennedy, L. (2023) *Top 400 asset managers 2018: 10 years of asset growth, IPE*. Available at: https://www.ipe.com/top-400-asset-managers-2018-10-years-of-asset-growth/10025004.article (Accessed: 23 September 2023).

Kidd, D., CFA (2012). *Value at Risk and Conditional Value at Risk: A Comparison.* Available at: https://deborahkidd.com/wp-content/uploads/Value-at-Risk-and-Conditional-Value-at-Risk-A-Comparison-1.pdf [Accessed 25 Sept. 2023].

Kolm, P.N., Tütüncü, R. & Fabozzi, F.J. (2014). *60 Years of portfolio optimisation: Practical challenges and current trends*. European Journal of Operational Research, 234(2), pp. 356-371. [Online]. Available at: https://www.sciencedirect.com/science/article/pii/S0377221713008898 [Accessed 26 Sep. 2023]. doi: 10.1016/j.ejor.2013.10.060.

Koors, A. and Page, B. (2012) *Transfer and Generalisation of Financial Risk Metrics to Discrete Event Simulation*. WAMS 2012 - The International Workshop on Applied Modelling and Simulation. Available at: https://www.researchgate.net/publication/322701819_Transfer_and_Generalisation_of_Financial_Risk_Metrics_to_Discrete_Event_Simulation [Accessed 25 Sept. 2023].

Kull, M. (2014). *Portfolio Optimisation for Constrained Shortfall Risk: Implementation and IT Architecture Considerations*. ETH Zurich, MTEC. [Online] Available at: https://ethz.ch/content/dam/ethz/special-interest/mtec/chair-of-entrepreneurial-risks-dam/documents/dissertation/master%20thesis/Thesis_Matthias_Kull_2014.pdf [Accessed 24 September 2023].

Kumar, S. (2022). *Iterative Model in Software Engineering*. [Accessed 29 September 2023]. Available at: https://www.scaler.com/topics/software-engineering/iterative-model-in-software-engineering/.

Kundu, S. (2020). *The Pandas fillna and dropna methods*. Medium.com. Available at: https://medium.com/@sagnikkundu25/the-Pandas-fillna-and-dropna-methods-1fecad724aa9 [Accessed 26 Sep. 2023].

Ledoit, O. & Wolf, M. (2003). *Honey, I shrunk the sample covariance matrix*. UPF Economics and Business Working Paper, 691.

Lee, C. (2019). *Financing method for real estate and infrastructure development using Markowitz's portfolio selection model and the Monte Carlo simulation*. Eng. Constr., 26, pp. 2008–2022.

Leonelli, M. (n.d.). *5.3 Steps of Monte Carlo simulation | Simulation and Modelling to Understand Change*. Available at: https://bookdown.org/manuele_leonelli/SimBook/steps-of-monte-carlo-simulation.html [Accessed 25 Sept. 2023].

London Stock Exchange. (n.d.). *Real Time Data.* [Online] Available at: https://www.londonstockexchange.com/equities-trading/market-data/real-time-data [Accessed 24 September 2023].

Malkiel, B.G. (2003). *The Efficient Market Hypothesis and Its Critics.* Princeton University, CEPS Working Paper No. 91. [Online] Available at: https://www.princeton.edu/~ceps/workingpapers/91malkiel.pdf [Accessed 24 September 2023].

Manganelli, S. and Engle, R.F. (2001). *Value at Risk Models in Finance. European Central Bank Working Paper No. 75*. [Online] Available at: https://www.ecb.europa.eu/pub/pdf/scpwps/ecbwp075.pdf [Accessed 25 September 2023].

Marakbi, Z. (2016). *Mean-Variance Portfolio Optimisation: Challenging the role of traditional covariance estimation*. Master of Science Thesis, KTH Industrial Engineering and Management, Industrial Management, Stockholm.

MarketWatch. (2023). *TMUBMUSD03M | U.S. 3 Month Treasury Bill Overview*. Available at: https://www.marketwatch.com/investing/bond/tmubmusd03m?countrycode=bx [Accessed 26 Sep. 2023].

Martin, R. (2023). *PyPortfolioOpt*. GitHub repository. Available at: https://github.com/robertmartin8/PyPortfolioOpt [Accessed 24 Sep. 2023].

Matplotlib (2023). *Gallery*. Available at: https://matplotlib.org/stable/gallery/index.html [Accessed 26 Sep. 2023].

McKinney, T. (2020). *How to deal with multi-level column names downloaded with yfinance*. Stack Overflow. Available at: https://stackoverflow.com/questions/63107594/how-to-deal-with-multi-level-column-names-downloaded-with-yfinance/63107801#63107801 [Accessed 26 Sep. 2023].

Mehta, A., Neukirchen, M., Pfetsch, S. & Poppensieker, T. (2012). *Managing market risk: Today and tomorrow*. McKinsey Working Papers on Risk, Number 32. [Online]. Available at: https://www.mckinsey.com/~/media/McKinsey/dotcom/client_service/Risk/Working%20papers/Working_Papers_on_Risk_32.ashx [Accessed: 25 September 2023].

Michaud, R.O. (1989). *The Markowitz optimisation enigma: is 'optimised' optimal*? Financial Analysts Journal, 45(1), pp. 31–42.

Minsky, Y. (n.d.). *Jane and the Compiler*. Jane Street. Available at: https://www.janestreet.com/tech-talks/jane-and-compiler/ [Accessed 26 Sept. 2023].

MIT. (n.d.). *Chapter 04: Linear Programming*. Available at: https://web.mit.edu/15.053/www/AMP-Chapter-04.pdf [Accessed 26 Sep. 2023].

Model Investing (2018). *The Risk-Return Trade-Off.* [Online] Available at: https://modelinvesting.com/articles/the-risk-return-trade-off/ [Accessed 25 September 2023].

MordorIntelligence (2023) *Market research company - mordor intelligenceTM*. Mordor Intelligence. Available at: https://www.mordorintelligence.com/ (Accessed: 23 September 2023).

Morgan Stanley. (2023). *What's a Robo-Advisor?* [Online]. Available at: https://www.morganstanley.com/articles/whats-a-robo-advisor [Accessed 26 Sep. 2023].

Moro-Visconti, R. (2016). *The Risk Matrix of Project Finance in the Healthcare Sector*. SSRN Electronic Journal. Available at: https://www.researchgate.net/publication/228231157_The_Risk_Matrix_of_Project_Finance_in_the_Healthcare_Sector [Accessed 25 Sept. 2023].

Mossin, J. (1966) *'Equilibrium in a capital asset market'*, Econometrica, 34(4), p. 768. doi:10.2307/1910098.

Murry, C. (2022). *Scenario Analysis*. Investopedia. Available at: https://www.investopedia.com/terms/s/scenario_analysis.asp [Accessed 29 September 2023].

Navellier, L. (2022). *Do FAANG Stocks Still Have Their Bite?* Nasdaq. https://www.nasdaq.com/articles/do-faang-stocks-still-have-their-bite [Accessed 24 Sep. 2023].

Nawrocki, D.N. (1999). *A Brief History of Downside Risk Measures*. The Journal of Investing, 8(3), 9-25. Available at: [https://www.pm-research.com/content/iijinvest/8/3/9] [Accessed 25 September 2023].

Nguyen, H. (2019). *Portfolio Optimisation Methods: The Mean-Variance Approach and the Bayesian Approach*. Barksdale Honors College Thesis, The University of Mississippi. Available at: https://thesis.honors.olemiss.edu/1398/1/Hoang%27s%20Thesis%20-%20final.pdf [Accessed 25 Sept. 2023].

NumPy. (n.d.a). *numpy.random.uniform*. Available at: https://NumPy.org/doc/stable/reference/random/generated/NumPy.random.uniform.html [Accessed 26 Sep. 2023].

NumPy. (n.d.b). *numpy.linalg.norm*. Available at: https://NumPy.org/doc/stable/reference/generated/NumPy.linalg.norm.html [Accessed 26 Sep. 2023].

NumPy. (n.d.c) *numpy.squeeze — NumPy v1.22 Manual, NumPy*. Available at: https://NumPy.org/doc/stable/reference/generated/NumPy.squeeze.html [Accessed: 25 September 2023].

Olson, P. (2023). *Mark Zuckerberg, the Metaverse, and the Sunk Cost Fallacy*. Bloomberg. Available at: https://www.bloomberg.com/opinion/articles/2023-04-11/mark-zuckerberg-the-metaverse-and-the-sunk-cost-fallacy [Accessed 26 Sept. 2023].

Onejohi. (2020). *What is version control?* Medium. [Accessed 24 September 2023]. Available at: https://medium.com/@onejohi/what-is-version-control-94ef1b6defcf.

Osei, J., Sarpong, P. & Amoako, S. (2018). *Comparing Historical Simulation and Monte Carlo Simulation in Calculating VaR*. Quant. Financ., 3, pp. 22–35.

Pal, S (2023). *Software Engineering | Classical Waterfall Model*. GeeksforGeeks. [Accessed 29 September 2023]. Available at: https://www.geeksforgeeks.org/software-engineering-classical-waterfall-model/.

Paleologo, G. (2021). *Advanced Portfolio Management* (1st ed.). Wiley. Available at: https://www.perlego.com/book/2814666/advanced-portfolio-management-a-quants-guide-for-fundamental-investors-pdf [Accessed 29 September 2023].

Palmstierna, J. (2019) *The benefits of algorithmic trading of cryptocurrency, GSR Markets.* Available at: https://www.gsr.io/reports/the-benefits-of-algorithmic-trading-of-digital-assets/ (Accessed: 23 September 2023).

Pandas. (n.d.a). *pandas.DataFrame.ewm*. Pandas. Available at: https://Pandas.pydata.org/docs/reference/api/Pandas.DataFrame.ewm.htm [Accessed 26 Sep. 2023].

Pandas. (n.d.b). *pandas.DataFrame.std*. Pandas. Available at: https://Pandas.pydata.org/docs/reference/api/Pandas.DataFrame.std.html [Accessed 26 Sep. 2023].

Pandey, S. (2023). *Netflix's Market Share Decline Continues In 2023: Analysis Of Leading Streaming Platforms*. [online] Similarweb. Available at: https://www.similarweb.com/blog/insights/media-entertainment-news/streaming-q1-2023/ [Accessed 24 Sep. 2023].

Pasieczna, A.H. (2019). *Monte Carlo simulation approach to calculate value at risk: application to WIG20 and MWIG40*. Financ. Sci., 24, pp. 61–75.

Pedersen, M. (2014). *Portfolio Optimisation and Monte Carlo Simulation*. SSRN Electronic Journal. doi: 10.2139/ssrn.2438121. Available at: https://www.researchgate.net/publication/272217402_Portfolio_Optimisation_and_Monte_Carlo_Simulation [Accessed 25 Sept. 2023].

Pierre, S. (2023). *How to Create a List and Array in Python*. Built In. Available at: https://builtin.com/data-science/how-to-create-list-array-python [Accessed 26 Sept. 2023].

Ponraj, A. (2020) *A Tip A Day—Python Tip #5—Pandas Concat & Append | Dev Skrol*, Medium. Available at: https://medium.com/analytics-vidhya/a-tip-a-day-python-tip-5-pandas-concat-append-dev-skrol-18e4950cc8cc [Accessed: 25 September 2023].

Potters, C. (2023). *Double Top*. Investopedia. Available at: https://www.investopedia.com/terms/d/doubletop.asp [Accessed 26 Sep. 2023].

Protasiewicz, J. (2023a). *Python in Finance*. Available at: https://www.netguru.com/blog/python-in-finance [Accessed 26 Sept. 2023].

Protasiewicz, J. (2023b). *How Python is Used in Finance and Fintech*. Netguru. Available at: https://www.netguru.com/blog/python-in-finance [Accessed 26 Sept. 2023].

Q.ai. (2023). *FAANG Stocks Update: Do The Top Tech Stocks Still Have Bite In 2023?* Forbes. [Online] Available at: https://www.forbes.com/sites/qai/2023/02/22/faang-stocks-update-do-the-top-tech-stocks-still-have-bite-in-2023/?sh=60ca02577ddd [Accessed 24 Sep. 2023].

Quantivity. (2011). *Why Log Returns.* [Online] Available at: https://quantivity.wordpress.com/2011/02/21/why-log-returns/ [Accessed 24 September 2023].

Quantopian. (2020). *pyfolio*. GitHub. Available at: https://github.com/quantopian/pyfolio [Accessed 24 September 2023].

Quantpedia. (2021). *Markowitz Model*. Available at: https://quantpedia.com/markowitz-model/ [Accessed 26 September 2023].

QuantPy (2021). *Monte Carlo Simulation of a Stock Portfolio with Python*. YouTube. Available at: https://www.youtube.com/watch?v=6-dhdMDiYWQ [Accessed 29 September 2023].

QuantStart. (2022) *Best Operating System for Quant Trading*. Available at: https://www.quantstart.com/articles/best-operating-system-for-quant-trading/ [Accessed 29 September 2023].

QuantStart. (2023). *QR Decomposition with Python and NumPy*. Available at: https://www.quantstart.com/articles/QR-Decomposition-with-Python-and-NumPy/ [Accessed 26 Sep. 2023].

Random Services. (n.d.). *Expected Value.* [Online] Available at: http://www.randomservices.org/random/expect/index.html [Accessed 24 September 2023].

Ranjan, S. (2023). *Python Pandas dataframe.rolling()*. GeeksforGeeks. Available at: https://www.geeksforgeeks.org/python-Pandas-dataframe-rolling/ [Accessed 26 Sep. 2023].

Read, D. (2004). *Utility theory from Jeremy Bentham to Daniel Kahneman*. Department of Operational Research, London School of Economics. [Online] Available at: http://eprints.lse.ac.uk/22750/1/04064.pdf

Reid, S. (2015). *A Recipe for the 2008 Financial Crisis*. Turing Finance. Available at: https://www.turingfinance.com/recipe-for-the-financial-crisis/ [Accessed 25 Sept. 2023].

Richards, C. (2012). *Tomorrow's Market Probably Won't Look Anything Like Today*. Bucks Blog, The New York Times. Available at: https://archive.nytimes.com/bucks.blogs.nytimes.com/2012/02/13/tomorrows-market-probably-wont-look-anything-like-today/ [Accessed 25 Sept. 2023].

Rigamonti, A. (2020). *Mean-Variance Optimisation Is a Good Choice, But for Other Reasons than You Might Think*. Risks, 8(1), 29. Available at: https://doi.org/10.3390/risks8010029 [Accessed 26 September 2023].

Roberson, M. (2019). *The Practical Problems with Coherent Risk Measure. SSRN*. [Online] Available at: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3373063 [Accessed 25 September 2023].

Rollinger, T.N. & Hoffman, S.T., 2012. *Sortino: A 'Sharper' Ratio*. Red Rock Capital. Available at: https://www.cmegroup.com/education/files/rr-sortino-a-sharper-ratio.pdf [Accessed 25 Sept. 2023].

S&P Dow Jones Indices. (n.d.). *S&P 500®*. [Online] Available at: https://www.spglobal.com/spdji/en/indices/equity/sp-500/#data [Accessed 24 September 2023].

Salkar, T., Shinde, A., Tamhankar, N., & Bhagat, N. (2021). *Algorithmic Trading using Technical Indicators.* 2021 International Conference on Communication information and Computing Technology (ICCICT). Available at: https://ieeexplore.ieee.org/abstract/document/9510135 [Accessed 25 Sept. 2023].

Samrhitha, A. (2023) 'Netflix falls as benefits from password-sharing crackdown to take time', Reuters. Available at: https://www.reuters.com/technology/netflix-tumbles-revenue-hit-overshadows-subscriber-jump-2023-07-20/ (Accessed: [Date of Access]).

Samuelsson, S. (2023). *How Many Trading Days in a Year?*. The Robust Trader. Available at: https://therobusttrader.com/how-many-trading-days-in-a-year/ [Accessed 26 September 2023].

Saturn Cloud. (2023). *Getting started with scipy minimize with constraints*. Available at: https://saturncloud.io/blog/getting-started-with-scipy-minimize-with-constraints/ [Accessed 26 Sep. 2023].

Scandinavian Capital Markets. (2021). *Sharpe Ratio vs. Sortino Ratio*. Available at: https://scandinavianmarkets.com/2021/02/17/sharpe-ratio-vs-sortino-ratio/ [Accessed 25 Sept. 2023].

Schmidt, U. & Zank, H., 2005. *What is Loss Aversion?.* J Risk Uncertainty, 30, pp.157–167. Available at: https://doi.org/10.1007/s11166-005-6564-6 [Accessed 29 September 2023].

SciPy. (n.d.a). *Optimisation and Root Finding (scipy.optimise)*. Available at: https://docs.scipy.org/doc/scipy/tutorial/optimise.html [Accessed 26 Sep. 2023].

SciPy. (n.d.b). *scipy.optimise.minimize*. Available at: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimise.minimize.html [Accessed 26 Sep. 2023].

Sekan, F. (2021). *Use Cases & Limitations of np.where() Function in NumPy*. Medium. Available at: https://medium.com/@filip.sekan/use-cases-limitations-of-np-where-function-in-NumPy-f7ff714e60b7 [Accessed 26 Sep. 2023].

Sen, J., Dutta, A. & Mehtab, S., 2021. Stock Portfolio Optimization Using a Deep Learning LSTM Model. [pdf] arXiv.org. Available at: https://arxiv.org/abs/2111.04709 [Accessed 24 September 2023].

Sharpe, W.F. (1966). *Mutual Fund Performance*. The Journal of Business, 39(1), 119-138. Available at: http://www.jstor.org/stable/2351741 [Accessed 25 Sept. 2023].

SIFMA. (2023). *Research Quarterly: Equities*. [Online] Available at: https://www.sifma.org/resources/research/research-quarterly-equities/ [Accessed 24 September 2023].

Singh, M., & Sawers, P. (2023). *Meta to cut another 10,000 jobs, Zuckerberg says*. TechCrunch. Available at: https://techcrunch.com/2023/03/14/meta-to-cut-another-10000-jobs-zuckerberg-says/?guccounter=1 [Accessed 26 Sept. 2023].

Sortino, F.A. and Satchell, S.E. (2001). *Managing Downside Risk in Financial Markets: Theory, Practice and Implementation*. Butterworth-Heinemann.

Spillers, F. (2023). *Progressive Disclosure. Interaction Design Foundation*. Available at: https://www.interaction-design.org/literature/book/the-glossary-of-human-computer-interaction/progressive-disclosure [Accessed 26 Sep. 2023].

Stack Overflow. (2014). *Why to convert a python list to a NumPy array?* Available at: https://stackoverflow.com/questions/23023878/why-to-convert-a-python-list-to-a-NumPy-array [Accessed 26 Sep. 2023].

StatMuse (2023). *The closing price for Apple stock from 2018 to 2023*. Available at: https://www.statmuse.com/money/ask/apple+stock+price+in+2018+to+2023 [Accessed 26 Sept. 2023].

Stewart, E., 2021. *Why stocks soared while America struggled*. [online] Vox. Available at: https://www.vox.com/business-and-finance/22421417/stock-market-pandemic-economy [Accessed 24 September 2023].

SwingTradeSystems. (2023). 2023 Trading Days Calendar. Available at: https://www.swingtradesystems.com/trading-days-calendars.html [Accessed 24 Sep. 2023].

Taleb, N. N. (2007). *The Black Swan: The Impact of the Highly Improbable*. 1st Edition.

Taleb, N.N. (2015). *Doing Statistics Under Fat Tails*. [Online] Available at: https://www.fooledbyrandomness.com/FatTails.html [Accessed 24 September 2023].

Tamplin, T. (2023). Conditional Value at Risk (CVaR) | Meaning, Pros, and Cons. Finance Strategists. Available at: https://www.financestrategists.com/wealth-management/risk-profile/conditional-value-at-risk-cvar/ [Accessed 25 September 2023].

Tan, R. (2021). *Changes in the Portfolio Management and Construction under the Pandemic Era.* In Proceedings of the E3S Web of Conferences, Qingdao, China, 28–30 May.

TeamGantt. (2023). *TeamGantt: Project Management Software & Online Gantt Charts.* [Online]. Available at: https://www.teamgantt.com/ [Accessed 29 Sep. 2023].

Thune, K. (2023). *Stock Market Sectors: A List of The 11 GISC Sectors. Seeking Alpha*. [Online] Available at: https://seekingalpha.com/article/4475586-stock-market-sectors [Accessed 24 September 2023].

TM. (2018). *Monte Carlo project*. In Quant Trading. GitHub. [Accessed 29 September 2023]. Available at: https://github.com/je-suis-tm/quant-trading/tree/master/Monte%20Carlo%20project.

TM. (2022). *Quant Trading*. GitHub. [Accessed 29 September 2023]. Available at: https://github.com/je-suis-tm/quant-trading.

Tollander, J. (2020). *Measuring Tail Risk Using Conditional Value at Risk*. Available at: https://jaantollander.com/post/measuring-tail-risk-using-conditional-value-at-risk/ [Accessed 29 September 2023].

Tran, T. (2022) *Functional Testing vs. Unit Testing*. Orient Software. Available at: https://www.orientsoftware.com/blog/functional-testing-vs-unit-testing/ [Accessed: 25 September 2023].

Tu, J. & Zhou, G. (2011). *Markowitz meets Talmud: A combination of sophisticated and naive diversification strategies*. Journal of Financial Economics, 99(1), pp. 204–215.

Turney, S. (2022). What Is Kurtosis? | Definition, Examples & Formula. Scribbr. [Online] Available at: https://www.scribbr.com/statistics/kurtosis/ [Accessed 24 September 2023].

Van der Wijst, N. (n.d.). *Option Pricing in Continuous Time*. Cambridge University Press. Available at: https://www.cambridge.org/ba/files/5613/6698/2500/chpt8optioncont.pdf.

Vega, N. (2022). *What experts say to do during a bear market*. CNBC. Available at: https://www.cnbc.com/2022/06/14/what-experts-say-to-do-during-a-bear-market.html [Accessed 24 Sep. 2023].

Vishwajit (2020) *What is NumPy?*, Numpy Ninja. Available at: https://www.numpyninja.com/post/what-is-numpy [Accessed 25 September 2023].

Visual Paradigm (2023). *Visual Paradigm Online*. Available at: https://online.visual-paradigm.com/ [Accessed 26 Sep. 2023].

Wall Street Prep. (n.d.). *Alpha (α)*. [Online] Available at: https://www.wallstreetprep.com/knowledge/alpha/ [Accessed 24 September 2023].

Wang, J. (2021). *Variables in C++.* Imperial College London. Available at: https://intro2ml.pages.doc.ic.ac.uk/autumn2021/modules/lab-cpp/variables [Accessed 26 Sept. 2023].

Yeo, L.L.X., Cao, Q. & Quek, C. (2023). *Dynamic portfolio rebalancing with lag-optimised trading indicators using SeroFAM and genetic algorithms.* Expert Systems with Applications, 216, 119440. [Online]. Available at: https://www.sciencedirect.com/science/article/pii/S0957417422024599 [Accessed 26 Sep. 2023]. doi: 10.1016/j.eswa.2022.119440.

Yıldırım, S. (2021). *5 Use Cases of Pandas loc and iloc Methods*. Towards Data Science. Available at: https://towardsdatascience.com/5-use-cases-of-Pandas-loc-and-iloc-methods-a94796b1f734 [Accessed 26 Sep. 2023].

Young, J. (2023). *Market Index: Definition, How Indexing Works, Types, and Examples*. Investopedia. [Online] Available at: https://www.investopedia.com/terms/m/marketindex.asp [Accessed 24 September 2023].

Zakamulin, V. (2015). *A Comprehensive Look at the Empirical Performance of Moving Average Trading Strategies.*

Zimpelmann, C. (2022) *Skewness expectations and portfolio choice*, Experimental Economics. SpringerLink. Available at: https://link.springer.com/article/10.1007/s10683-022-09780-9 [Accessed: 24 September 2023].

Zucchi, K. (2021). *Lognormal and Normal Distribution*. Investopedia. [Online] Updated October 31, 2021. Available at: https://www.investopedia.com/articles/investing/102014/lognormal-and-normal-distribution.asp [Accessed 24 September 2023].

12 Appendicies

12.1 Appendix A: Requirements Specification

A.1 Functional Requirements

Table 2: Functional Requirements - Measures

1. Measures		Priority
1.1	The system must validate the input types for all calculations to ensure they are of the correct data structure (NumPy arrays or Pandas series).	M
1.2	The system must compute the stratified average by summing the product of individual asset values and their proportion in the portfolio.	M
1.3	The system must compute the portfolio variance using the dispersion matrix and the proportion of each investment.	M
1.4	The system must compute the volatility of a portfolio based on its variance and the relative magnitude of each asset.	M
1.5	The system must compute and return the expected yearly metrics, including return, volatility, and a performance measure based on risk and return.	M
1.6	The system must use factors derived from the number of regular trading days in a year to annualize measures like return and volatility.	M
1.7	The system must be able to calculate the Sharpe Ratio for a portfolio based on the predicted return, volatility, and an optional risk-free rate of return.	M
1.8	The system must handle scenarios where specific metrics, like portfolio volatility or downside risk, are zero, returning appropriate values (e.g., infinite Sharpe Ratio, NaN for Sortino Ratio).	M
1.9	The system must be able to calculate the Sortino Ratio for a portfolio based on the predicted return, downside risk, and an optional risk-free rate of return.	M
1.10	The system must use default benchmark return values if not provided by the user.	M
1.11	The system must calculate the Value at Risk (VaR) for a given asset based on its total value, average return, volatility, and an optional confidence level.	M
1.12	The system must compute the downside risk by identifying negative returns (returns below the risk-free rate) and calculating the mean of squared negative returns.	M
1.13	The system must utilize the inverse of the cumulative distribution function to compute the Value at Risk for an asset.	M
1.14	The system should use default values for certain parameters if not provided by the user, such as the risk-free rate of return.	S
1.15	The system should handle potential edge cases, such as when the proportions of investments do not sum up to a certain expected value.	S
1.16	The system should provide clear error messages when input validation fails, specifying which input caused the error.	S
1.17	The system should employ an internal function to ensure the consistency and correctness of input types for all calculations, including computing the downside risk based on weighted daily mean returns and the risk-free rate of return.	S
1.18	The system could allow for customization of certain parameters, like the risk-free rate of return and the number of trading days.	C
1.19	The system could incorporate logging mechanisms to record system activities, especially errors or exceptions, for troubleshooting and monitoring purposes	C
1.20	The system could be designed in a modular and extensible way to accommodate future features or extensions.	C
1.21	The system won't fetch real-time data for certain parameters, like risk-free rates or trading days.	W

1.22	The system won't provide a graphical user interface for displaying inputs and outputs.	W
1.23	The system won't integrate with external systems or modules outside of the provided/specified context.	W

Table 3: Functional Requirements - Minimisation

2. Minimisation		**Priority**
2.1	The system must validate the data structures of all inputs, ensuring they are either NumPy arrays, Pandas series, or Pandas DataFrame.	M
2.2	The system must check the consistency in dimensions across the input data for proportions, average revenue, and dispersion matrix.	M
2.3	The system must validate the risk-free rate of return, ensuring it's a numeric value between 0 and 1.	M
2.4	The system must adjust the proportions of investments such that their sum equals 1, ensuring valid portfolio calculations.	M
2.5	The system must be capable of determining the annualized volatility of a portfolio using a combination of input data and internal calculations.	M
2.6	The system must determine the inverse of a Sharpe Ratio of a portfolio, which is used as an objective function in optimisation scenarios.	M
2.7	The system must determine the annualized return of a portfolio using input data and a specific set of calculations.	M
2.8	The system should provide clear and specific feedback when the data structure or dimensions of the inputs are not as expected.	S
2.9	The system should ensure that the normalized proportions are used consistently across all calculations to maintain accuracy.	S
2.10	The system could offer flexibility in accepting additional parameters that might influence the calculations, such as different risk-free rates or other financial metrics.	C
2.11	The system could maintain a log of its activities, especially when encountering errors, to aid in troubleshooting and continuous improvement.	C
2.12	The system won't directly fetch or update data from external sources or databases.	W
2.13	The system won't have a visual interface or dashboard for users to interact with or visualize the results.	W

Table 4: Functional Requirements - Efficient Frontier

3. Efficient Frontier Analysis		**Priority**
3.1	All classes must validate the provided inputs to ensure they meet specific criteria.	M
3.2	An initialisation class must be able to initialize with parameters for average revenue, dispersion matrix, risk-free rate of return, number of regular trading days in a year, and an optimisation method.	M
3.3	An initialisation class must be able to create a DataFrame to represent asset allocation based on given allocation values.	M
3.4	An initialisation class must define a constraint function for the optimisation and set it.	M
3.5	The system must validate that the optimisation method is a supported string.	M
3.6	An initialisation class should determine the portfolio size based on the average revenue's index.	S

3.7	An initialisation class should set numerical parameters for optimisation, including limits and initial guess.	S
3.8	An optimisation class must be able to create a DataFrame to represent asset allocation using the initialization's method.	M
3.9	An optimisation class must be able to optimise the portfolio to minimize volatility.	M
3.10	An optimisation class must be able to optimise the portfolio to maximize the Sharpe ratio.	M
3.11	An optimisation class must be able to optimise the portfolio to achieve a specified target return while minimizing volatility.	M
3.12	An optimisation class must be able to optimise the portfolio to achieve a specified target volatility while maximizing the Sharpe ratio.	M
3.13	An optimisation class must be able to evaluate the Markowitz Efficient Frontier (MEF) based on provided or generated targets.	M
3.14	The system must be able to define and use constraints for the optimisation.	M
3.15	The system must be able to update the preceding optimisation type based on the optimisation performed.	M
3.16	The system must be able to calculate the efficient frontier based on provided targets.	M
3.17	The system must be able to append the annualized volatility and target to the efficient frontier list.	M
3.18	The DataFrame should be indexed by asset designations and have a single column.	S
3.19	If optimisation targets are not provided, the system should generate them based on average revenue and trading days.	S
3.20	The system should be able to create a DataFrame to represent asset allocation.	S
3.21	The system could use the optimise.minimize function from the scipy library for optimisation.	C
3.22	The system could create a DataFrame to represent asset allocation using the initialization's method.	C
3.23	A visualisation class must be initialized with instances of both the initialisation and optimisation classes.	M
3.24	A visualisation class must be able to plot the optimal Markowitz Efficient Frontier (MEF) points on a graph.	M
3.25	A visualisation class must be able to plot the optimal portfolios for minimum volatility and maximum Sharpe ratio.	M
3.26	A visualisation class must be able to calculate and display the metrics for the Markowitz Efficient Frontier.	M
3.27	If the efficient frontier has not been calculated, the system should calculate it before plotting the optimal MEF points.	S
3.28	The system could provide a formatted output of the portfolio properties, including the most favourable allocation.	C
3.29	A master class must instantiate and integrate the initialisation, optimisation, and visualisation classes.	M
3.30	A master class must provide properties to access attributes from the initialisation component, including average revenue, dispersion matrix, risk-free rate of return, regular trading days, optimisation method, stock symbols, portfolio size, asset allocation, asset allocation dataframe, and optimal MEF points.	M
3.31	A master class must delegate methods to the optimisation component, including methods for volatility minimisation, Sharpe ratio maximisation, return optimisation, volatility optimisation, and MEF evaluation.	M
3.32	A master class must delegate methods to the visualisation component, including methods to plot optimal MEF points, plot optimal portfolios for minimum volatility and maximum Sharpe ratio, and calculate MEF metrics.	M

Table 5: Functional Requirements - Monte Carlo

	4. Monte Carlo Simulation	Priority
4.1	A class must be designed to execute Monte Carlo Simulations (MCS).	M
4.2	The execution class must be initialized with an optional parameter which represents the number of iterations for the Monte Carlo simulation. If not provided, it should default to 5000.	M
4.3	The execution class must also accept a variable number of keyword arguments to be passed to the a “run” function.	M
4.4	The execution class must use a list comprehension to call the “run” function for the specified number of iterations and store the results in a list.	M
4.5	The execution class could have error handling mechanisms to handle potential issues during the simulation, such as invalid input parameters or errors in the “run” function.	C
4.6	Given that Monte Carlo Simulations can be computationally intensive, the execution class could have the capability parallelise simulations for speed-up.	C
4.7	Instead of relying solely on a fixed number of iterations, the execution class could have an optional stopping criteria based on the convergence of results or a specified confidence interval.	C
4.8	The execution class won't have mechanisms for automatic parameter tuning or optimisation based on the simulation results.	W
4.9	A method class must be able to calculate and set essential portfolio metrics such as the number of assets, average revenue, and the dispersion matrix.	M
4.10	A method class must be able to generate uniformly distributed random allocations for portfolio stocks.	M
4.11	A method class must generate a list of diversified portfolios and their respective metrics.	M
4.12	A method class must determine the optimal portfolios based on minimum volatility and maximum Sharpe ratio criteria.	M
4.13	A method class should have a method to visualise the results of the Monte Carlo simulation for users to understand the outcomes.	M
4.14	A method class should have a method to print out the properties of the Monte Carlo Simulation, including the optimal portfolios and their metrics.	M

Table 6: Functional Requirements - Technical Analysis

	5. Technical Analysis	Priority
5.1	The system must calculate moving averages for given stock prices.	M
5.2	The system must determine buy/sell signals based on moving average crossovers.	M
5.3	The system must calculate moving averages for each specified window size.	M
5.4	The system must calculate deviation based on crossover of short-term and long-term moving averages.	M
5.5	The system must determine buy/sell signals based on changes in the deviation.	M
5.6	The system must plot moving averages on a graph, while highlighting buy and sell signals.	M
5.7	The system should calculate the simple moving average, and its standard deviation, for the given input stock prices using a rolling window.	S
5.8	The system should calculate the exponential moving average, and its standard deviation, for the given input stock prices.	S
5.9	The system could be extended to support other types of moving average such as weighted or triangular.	C
5.10	The system could comprise features like zooming in on specific date ranges, highlighting specific events, or annotating the graph with additional information.	C

5.11	The system won't use more advanced trading strategies (i.e. drawdown, multiple technical indicators, positional trading).	W
5.12	The system must compute the Bollinger Bands based on the moving average and standard deviation.	M
5.13	The system must visualise the Bollinger Bands along with the original stock prices and moving average.	M
5.14	The system must validate the moving average function.	M
5.15	The system should determine the appropriate standard deviation function based on the moving average type.	S
5.16	The system should compute the upper and lower Bollinger Bands.	S
5.17	The system should visualise the Bollinger Bands by filling the area between the upper and lower Bollinger Bands.	S
5.18	The system won't incorporate any other indictors aside from Bollinger Bands (i.e. RSI, MFI, and MACD).	W
5.19	The system won't allow for compositional analysis, where a strategies are based on proportions of different indicators.	W
5.20	The system must validate that all moving average and Bollinger Band inputs and results are of the expected data type and structure.	M

Table 7: Functional Requirements - Returns

6. Performance (Returns)		**Priority**
6.1	The system must be able to calculate the cumulative return of stock prices over a given period.	M
6.2	The system must be able to compute the daily return of stock prices.	M
6.3	The system should calculate the daily return of stock prices based on given allocations.	S
6.4	The system should compute the historical average return based on daily returns.	S
6.5	The system could offer the capability to compute the logarithmic daily return of stock prices.	C
6.6	The system won't offer predictive analytics or forecasting features for stock prices or returns.	W

Table 8: Functional Requirements - Investments

7. Investment (Stock & Index)		**Priority**
7.1	An investment class must allow the initialization of an Investment object with historical price data, investment name, and an optional investment category.	M
7.2	An investment class must compute and assign key statistical properties like forecast return, volatility, skewness, and kurtosis for the investment upon initialization.	M
7.3	An investment class must be able to calculate the daily return for the investment based on its historical price data.	M
7.4	An investment class must display key attributes of the investment in a tabular format.	M
7.5	An investment class should calculate the forecasted return for the investment based on its historical price data.	S
7.6	An investment class should calculate the annualised volatility for the investment based on its historical price data.	S
7.7	An investment class could calculate the skewness of the investment's historical price data.	C

7.8	An investment class could calculate the kurtosis (tailedness) of the investment's historical price data.	C
7.9	An investment class won't have advanced visualization tools for presenting the analysed data beyond the basic tabular format.	W
7.10	A Stock class must allow the initialization of a Stock object with stock details and historical price data.	M
7.11	A Stock class must validate the type of asset price history and initialize the asset using the parent 'Investment' class.	M
7.12	A Stock class must calculate the beta coefficient of the stock based on its daily returns and the returns of a given market index.	M
7.13	A Stock class must display key attributes of the stock in a structured format.	M
7.14	A Stock class should store additional stock details provided during initialisation.	S
7.15	A Stock class could present stock attributes in a visually appealing manner, combining both stock properties and investment information.	C
7.16	A Stock class won't provide tools or functions to execute stock trades or manage a stock portfolio.	W
7.17	An Index class must allow the initialization of an Index object with historical price data, using the 'Investment' parent class.	M
7.18	An Index class must compute the daily returns for the financial index based on its historical price data.	M
7.19	An Index class won't automatically update the index data in real-time based on live market data.	W
7.20	An Index class won't provide tools or functions to execute trades based on the index or manage a portfolio based on the index.	W

Table 9: Functional Requirements - Portfolio Optimisation

8. Portfolio Optimisation		**Priority**
8.1	The system must allow the initialization of a directing Portfolio object, to be incorporated with all previously defined functions.	M
8.2	The system must allow the addition of a stock to the portfolio.	M
8.3	The system must allow the extraction of a stock from the portfolio based on its symbol.	M
8.4	The system must update the portfolio metrics based on changes in its composition.	M
8.5	The system should compute the beta coefficient for the portfolio.	S
8.6	The system must calculate the Value at Risk (VaR) for the portfolio.	M
8.7	The system must calculate the downside risk of the portfolio.	M
8.8	The system must calculate the daily return of the portfolio.	M
8.9	The system must calculate the forecasted return of the portfolio.	M
8.10	The system must calculate the dispersion matrix of the portfolio.	M
8.11	The system must calculate the annualized volatility of the portfolio.	M
8.12	The system must calculate the proportioned allocation of the portfolio.	M
8.13	The system must calculate the Sharpe Ratio of the portfolio.	M
8.14	The system must calculate the Sortino Ratio of the portfolio.	M
8.15	The system should calculate the cumulative return of the portfolio.	S
8.16	The system should calculate the logarithmic daily return of the portfolio.	S
8.17	The system should calculate the historical average return of the portfolio.	S
8.18	The system should calculate the skewness of the stocks in the portfolio.	S
8.19	The system should calculate the kurtosis of the stocks in the portfolio.	S
8.20	The system should calculate annualised volatility for each stock in the portfolio.	S
8.21	The system must initialize or retrieve the Monte Carlo Simulation instance, from the Monte Carlo Simulation module.	M
8.22	The system must optimise the portfolio using Monte Carlo Simulation.	M

8.23	The system must initialise or retrieve the Markowitz Efficient Frontier instance, from the Monte Carlo Simulation module.	M
8.24	The system must optimise the portfolio for minimum volatility using the Efficient Frontier.	M
8.25	The system must optimise the portfolio for maximum Sharpe Ratio using the Efficient Frontier.	M
8.26	The system should visualise the results and print the attributes of the Monte Carlo Simulation.	S
8.27	The system should optimise the portfolio for a specific target return using the Efficient Frontier.	S
8.28	The system should optimise the portfolio for a specific target volatility using the Efficient Frontier.	S
8.29	The system should evaluate the portfolio on the Efficient Frontier.	S
8.30	The system could plot the optimal points on the Efficient Frontier.	C
8.31	The system could plot the portfolio's volatility and Sharpe Ratio on the Efficient Frontier.	C
8.32	The system must visualise the annualized returns of stocks against their volatility.	M
8.33	The system must print the portfolio's various attributes, including return metrics, risk metrics, distribution metrics, and other metrics.	M
8.34	The system should return a string representation of the portfolio, listing the stocks it contains.	S
8.35	The system could have advanced tools for deeper portfolio analysis, such as stress testing, scenario analysis, or backtesting.	C
8.36	The system won't have tools to automatically rebalance the portfolio based on certain criteria or thresholds.	W
8.37	The system won't schedule automated reports to be generated and sent at specified intervals.	W
8.38	The system won’t offer enhanced visualization features, such as interactive charts, 3D plots, or heat maps.	W
8.39	The system won't provide tools or functions to execute trades based on the portfolio or manage a portfolio based on live trading.	W

Table 10: Functional Requirements - Portfolio Composition

9. Portfolio Composition		**Priority**
9.1	The system must invoke the yfinance API to retrieve stock data for given symbols and a specified date range.	M
9.2	The system should raise an error if the yfinance package is not installed.	S
9.3	The system should raise a general error for any issues encountered during the data fetch from Yahoo Finance.	S
9.4	The system could offer enhanced data formatting options, such as additional date formats or data transformations.	C
9.5	The system should handle DataFrames with MultiIndex columns and identify the appropriate sub-columns.	S
9.6	The system could offer enhanced data formatting options, such as additional column transformations or data aggregations.	C
9.7	The system must determine the allocation for the stocks in the portfolio.	M
9.8	The system must be able to retrieve the 'Adjusted Close' data from a provided DataFrame.	M
9.9	The system could offer more comprehensive data verification checks, such as checking for missing values or outliers in the 'Adjusted Close' data.	C
9.10	The system should generate an error message for conflicting column names.	S
9.11	The system should default to "Adj Close" if no column ID tags are provided.	S
9.12	The system should cascade any changes made to the portfolio.	S

9.13	The system should set the financial index for the portfolio if index data is provided.	S
9.14	The system must offer utility functions to compare elements within two sets. This is done for further error handling.	M
9.15	The system must validate the provided keyword arguments.	M
9.16	The system must check for input argument conflicts.	M
9.17	The system must validate the attributes of the final portfolio.	M
9.18	The system should handle portfolio assembly either via API or DataFrame.	S
9.19	The system won't integrate with other data sources besides yfinance for fetching stock data.	W
9.20	The system won't store the fetched stock data in a database or other persistent storage.	W

A.2 Non-Functional Requirements

Table 11: Non-Functional Requirements - Product

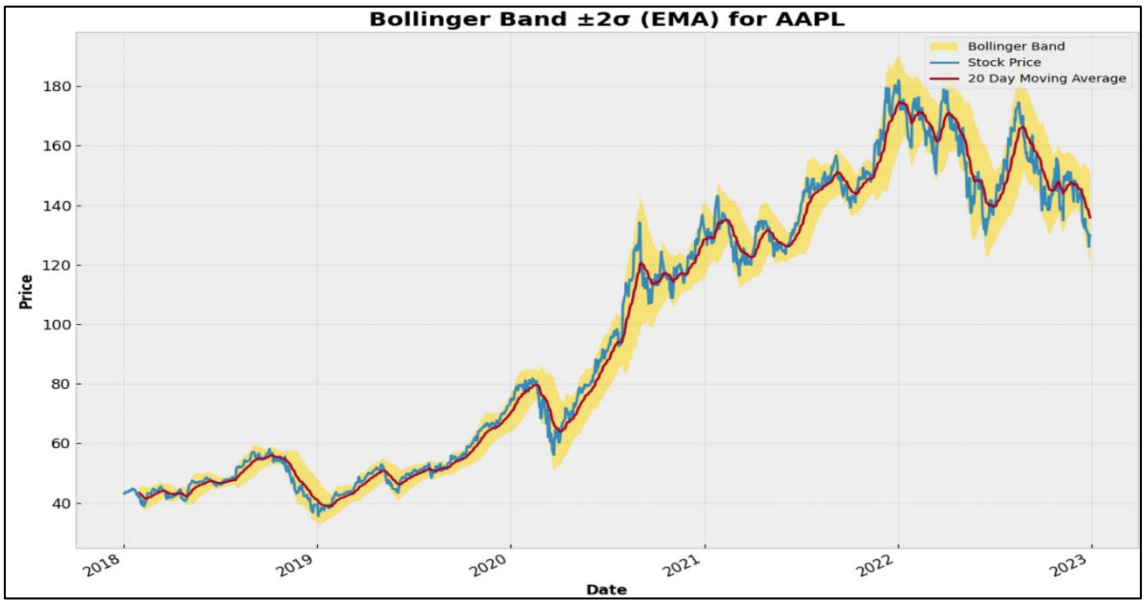
1. Product Requirements			**Priority**
1.1	**Performance**		
	1.1.1	The system must process stock data and return portfolio analysis results within a reasonable time frame, depending on the dataset's size, and the operation being performed.	M
	1.1.2	The system must handle large datasets (i.e. 5 years of yfinance data for 5 companies) without crashing or significant performance degradation.	M
	1.1.3	The system must require no more than 8GB of RAM to operate.	M
1.2	**Security**		
	1.2.1	The system must sanitise all user inputs to prevent potential vulnerabilities, especially when interacting with external APIs.	M
	1.2.2	If applicable, any stored user preferences or configurations must be encrypted.	M
1.3	**Reliability**		
	1.3.1	The system must provide accurate financial calculations consistently.	M
	1.3.2	Error handling mechanisms must be in place in each module to manage potential issues with data sources or calculations.	M
1.4	**Usability**		
	1.4.1	The system's functions and methods must be well-documented, allowing developers to understand and utilise them effectively.	M
	1.4.2	Clear error messages should be provided, especially for data inconsistencies or invalid inputs.	M
1.5	**Scalability**		
	1.5.1	The system should be designed modularly, allowing for the addition of new financial analysis methods or data sources in the future.	S
	1.5.2	The system should be compatible with various data formats commonly used in financial analysis, such as CSV.	S
1.6	**Flexiblity & Configurability**		
	1.6.1	The system could provide a user-friendly interface or visualisation tools for non-developers to understand portfolio analysis results.	C

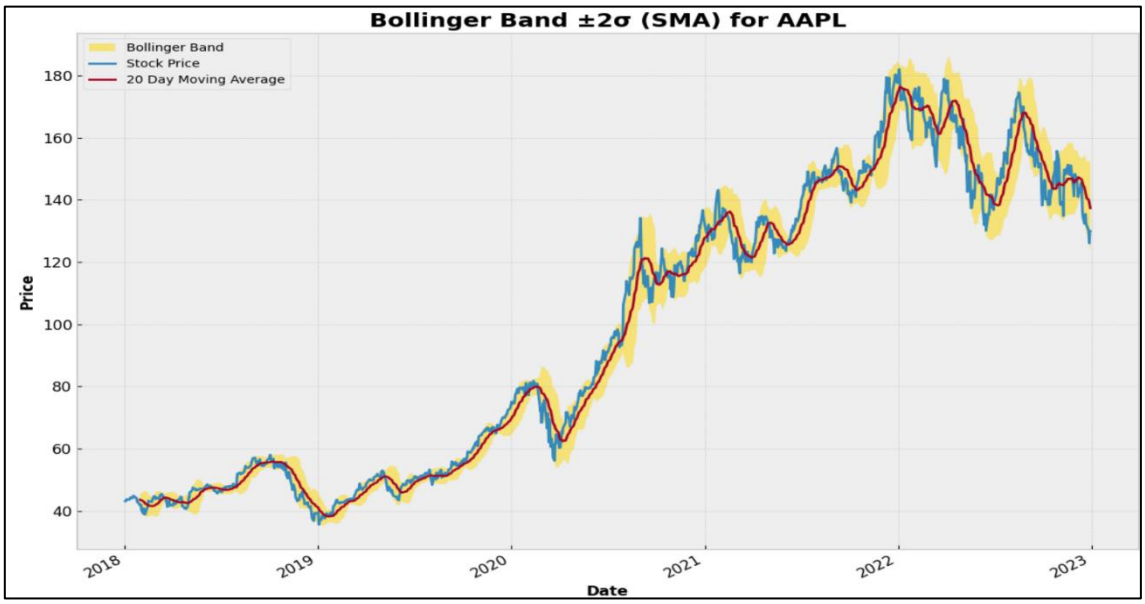
	1.6.2	The system could offer customization options for specific financial metrics or parameters used in portfolio construction (i.e. trading days, risk-free rate).	C

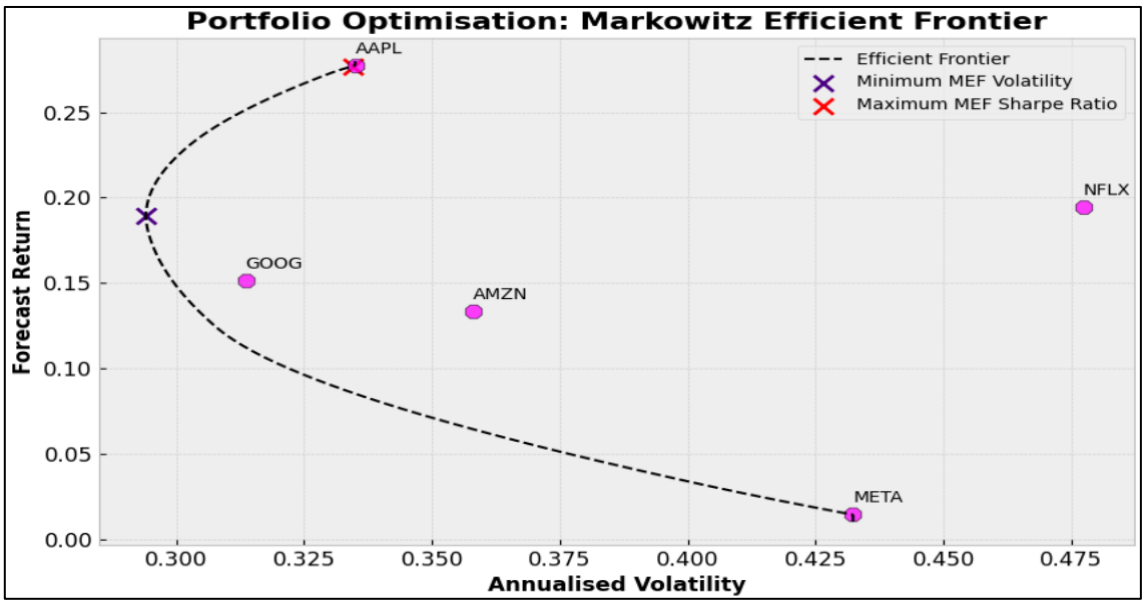
Table 12: Non-Functional Requirements - Organisational

2. Organisational Requirements			**Priority**
2.1	**Maintenance**		
	2.1.1	The system must provide detailed logs and/or error message to assist in troubleshooting, especially when interacting with external data sources.	M
2.2	**Compliance**		
	2.2.1	The system should adhere to relevant financial and data protection regulations, especially when dealing with real-time stock data.	S
2.3	**Feedback**		
	2.3.1	A mechanism won't be implemented for users or developers to suggest improvements or report issues.	W
	2.3.2	Periodic reviews won't be conducted to gather insights for system enhancements based on user feedback or evolving financial analysis methodologies.	W
2.4	**Delivery**		
	2.4.1	The product must be delivered by the 2nd of October 2023.	M
2.5	Implementation		
	2.5.1	The system must be developed in Python, using NumPy and Pandas.	M
	2.5.2	The system visualisations must be achieved using the terminal and/or Matplotlib plots.	M
2.6	**Portability**		
	2.6.1	The system must be compatible with Linux (XUbuntu).	M
	2.6.2	The system could be compatible with Windows.	C
	2.6.3	The system could be compatible with MacOS.	C

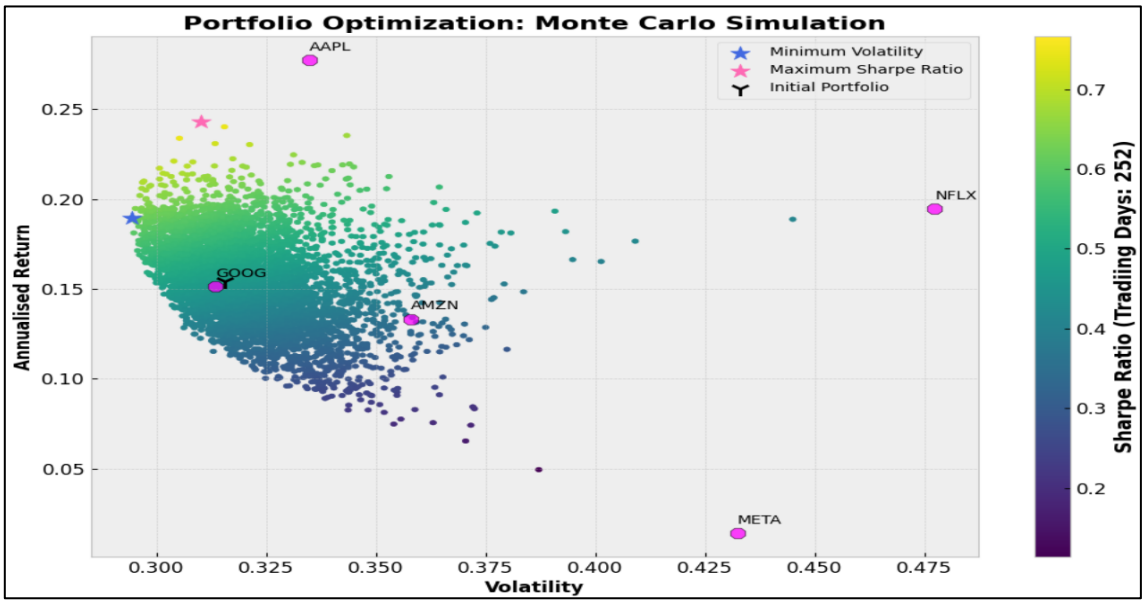
12.2 Appendix B: Complete Outputs of Example Scripts

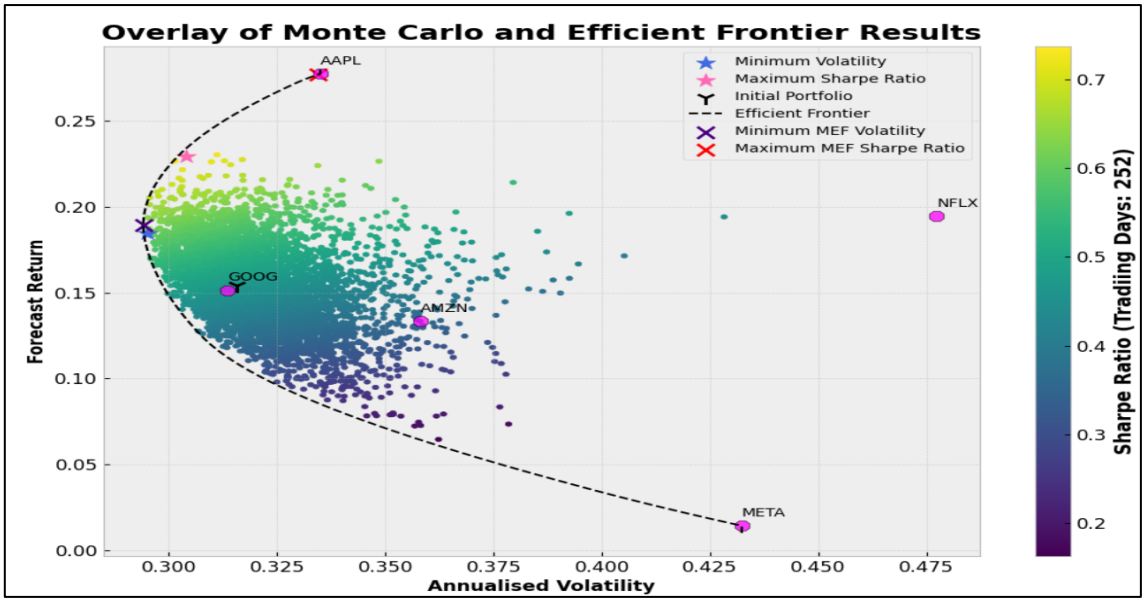


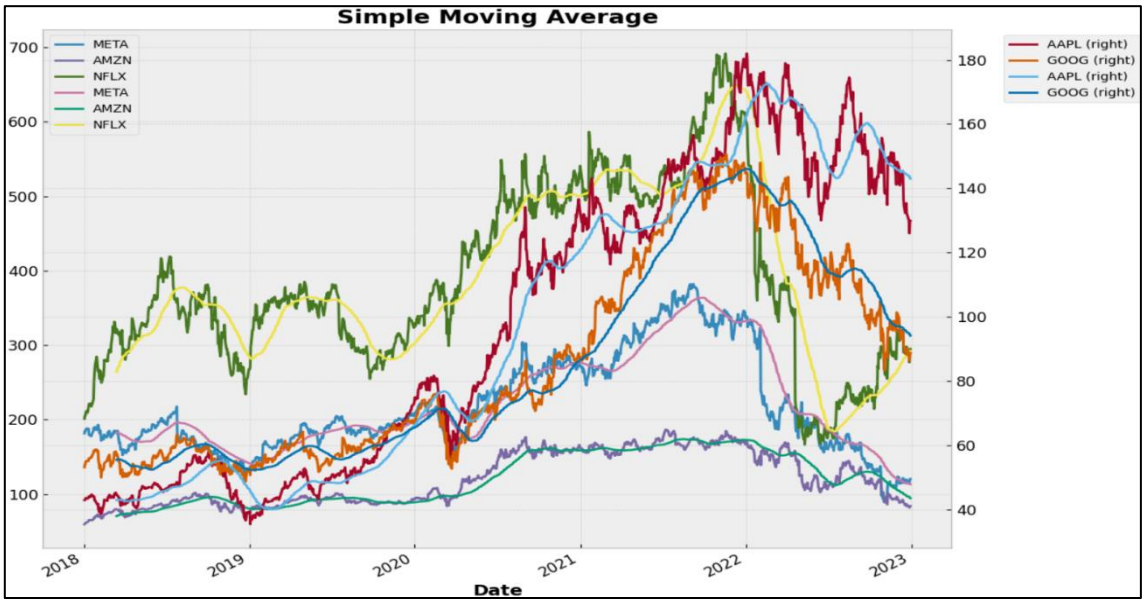


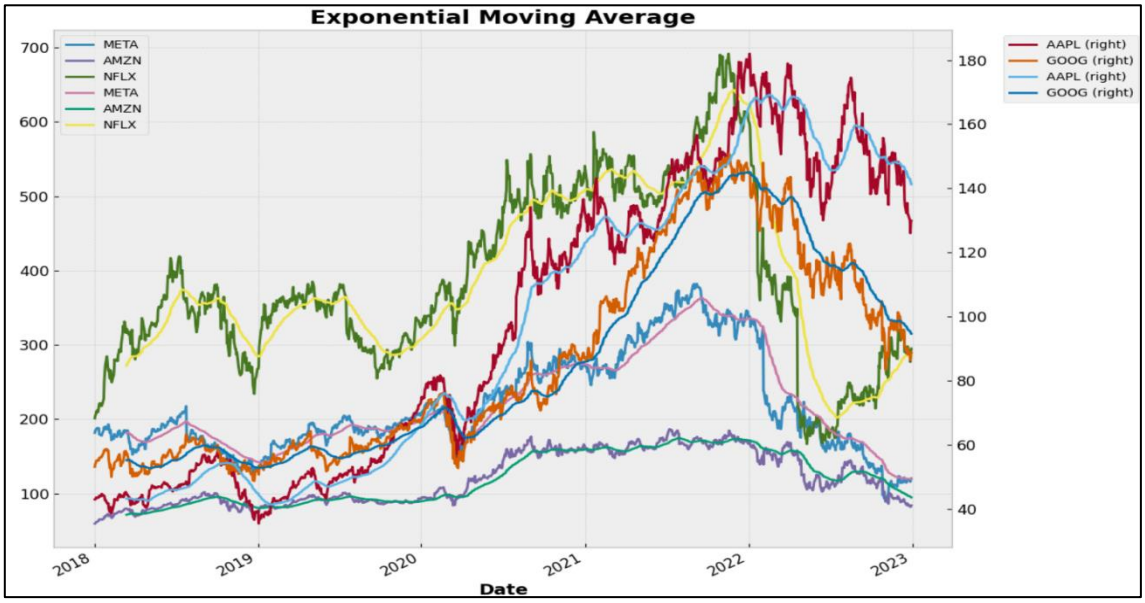


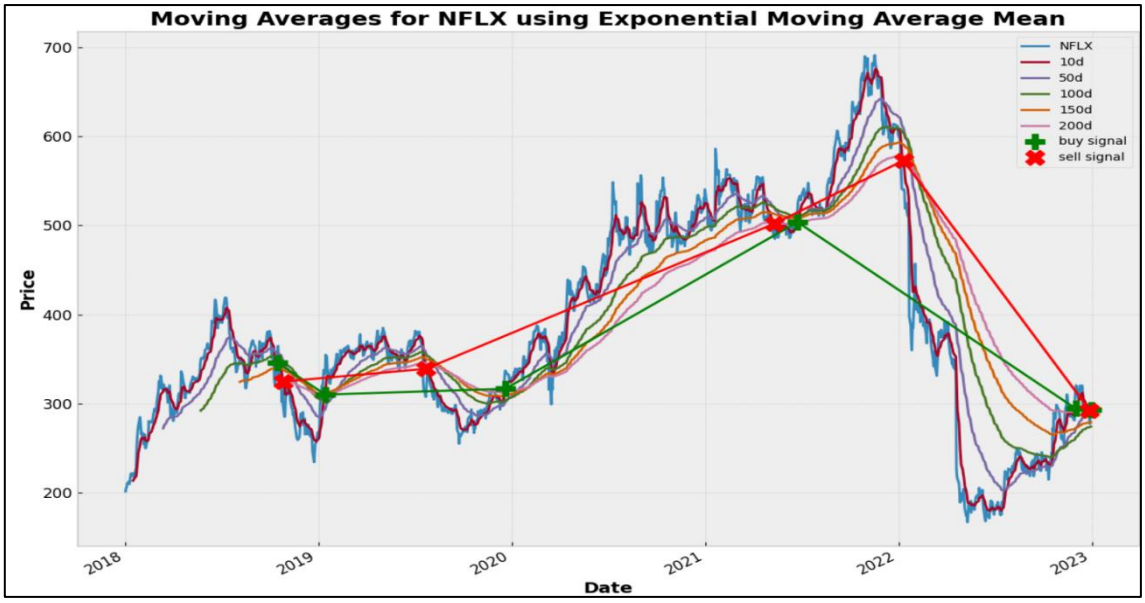
```
wasim@wasim-VirtualBox:~/Desktop/EntroPy$ python3 scripts/script_mef.py
==================================================
Optimised Portfolio for Minimum Volatility
--------------------------------------------------
Epoch/Trading Days: 252
Risk-Free Rate of Return: 0.54%
Predicted Annualised Return: 18.956%
Annualised Volatility: 29.408%
Sharpe Ratio: 0.6261
--------------------------------------------------
Most favourable Allocation:
                META      AAPL      AMZN      NFLX      GOOG
Allocation  9.745886e-18  0.315797  0.160878  0.028645  0.49468
==================================================
==================================================
Optimised Portfolio for Maximum Sharpe Ratio
--------------------------------------------------
Epoch/Trading Days: 252
Risk-Free Rate of Return: 0.54%
Predicted Annualised Return: 27.702%
Annualised Volatility: 33.448%
Sharpe Ratio: 0.8120
--------------------------------------------------
Most favourable Allocation:
                META      AAPL          AMZN      NFLX          GOOG
Allocation  4.277514e-17  0.995423  5.986342e-17  0.004577  1.022125e-16
==================================================
==================================================
Optimised Portfolio for Efficient Return
--------------------------------------------------
Epoch/Trading Days: 252
Risk-Free Rate of Return: 0.54%
Predicted Annualised Return: 20.000%
Annualised Volatility: 29.466%
Sharpe Ratio: 0.6603
--------------------------------------------------
Most favourable Allocation:
            META     AAPL      AMZN      NFLX      GOOG
Allocation   0.0  0.38949  0.120704  0.038639  0.451166
==================================================
==================================================
Optimised Portfolio for Efficient Volatility
--------------------------------------------------
Epoch/Trading Days: 252
Risk-Free Rate of Return: 0.54%
Predicted Annualised Return: 18.926%
Annualised Volatility: 29.408%
Sharpe Ratio: 0.6251
--------------------------------------------------
Most favourable Allocation:
                META      AAPL      AMZN      NFLX      GOOG
Allocation  5.839662e-21  0.313339  0.161882  0.029343  0.495435
==================================================
```

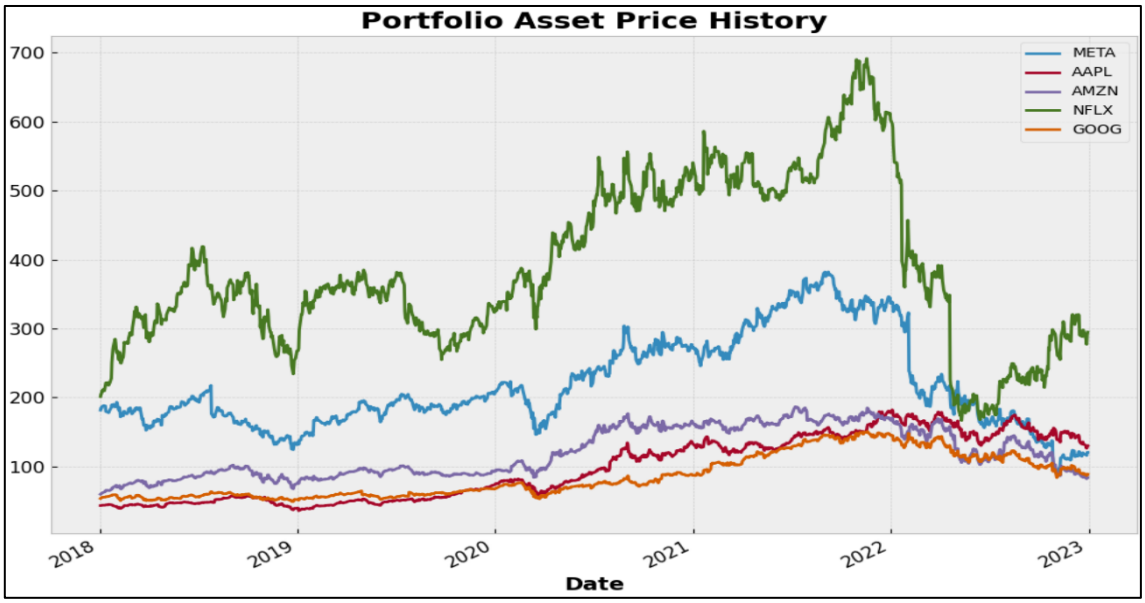


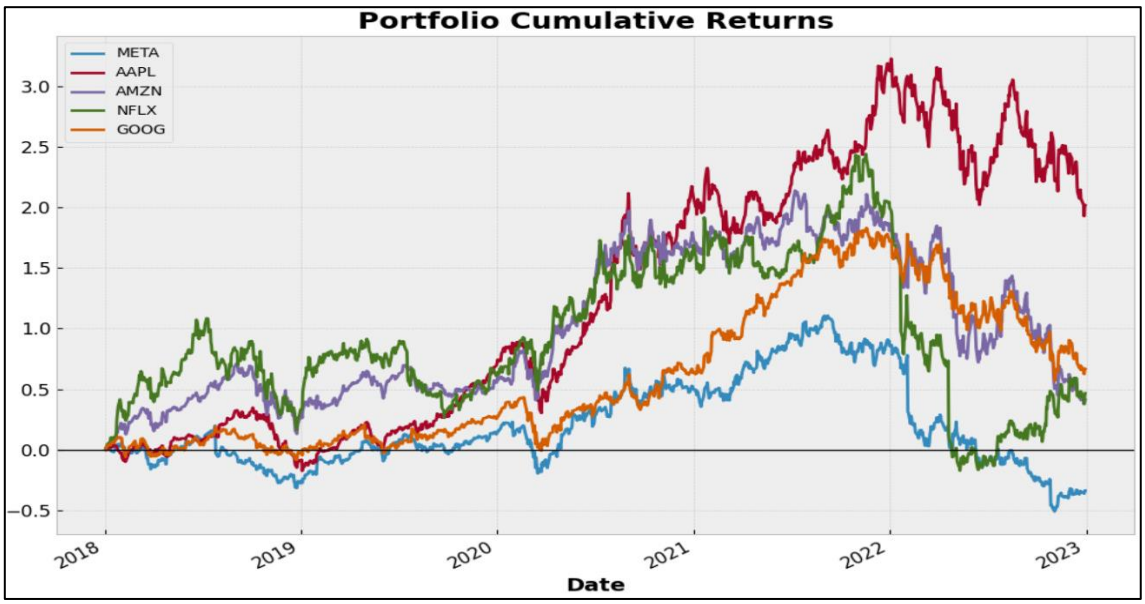

Overlay of Monte Carlo and Efficient Frontier Results
Minimum Volatility
Maximum Sharpe Ratio
Initial Portfolio
Efficient Frontier
Minimum MEF Volatility
Maximum MEF Sharpe Ratio
AAPL
NFLX
GOOG
AMZN
META
Forecast Return
Annualised Volatility
Sharpe Ratio (Tradiing Days: 252)

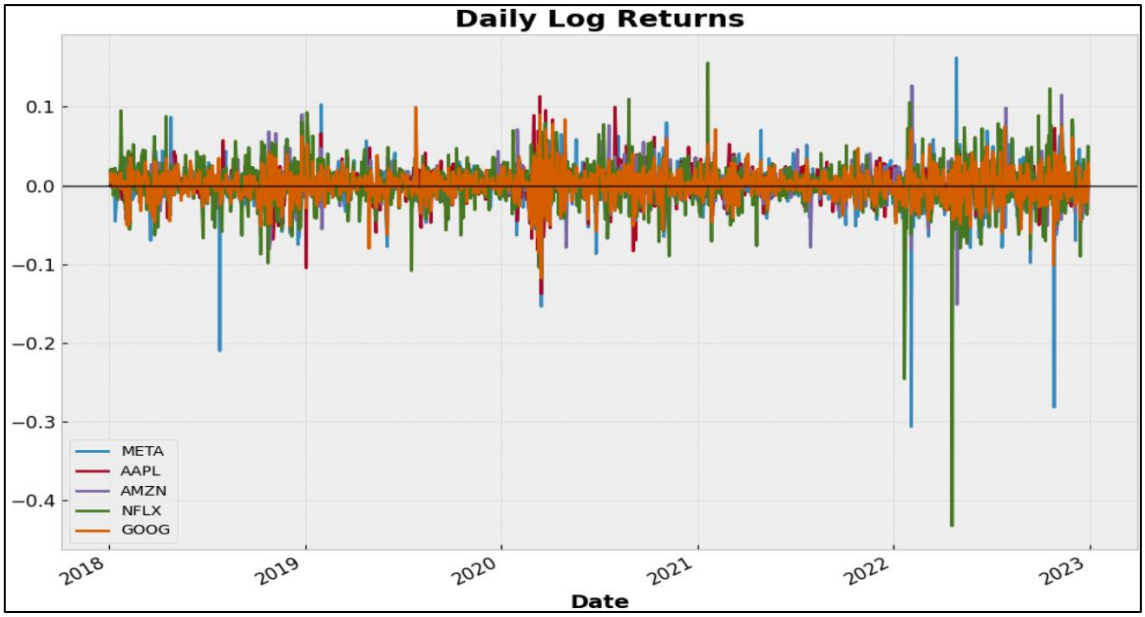

Simple Moving Average
META
AMZN
NFLX
AAPL (right)
GOOG (right)
Date

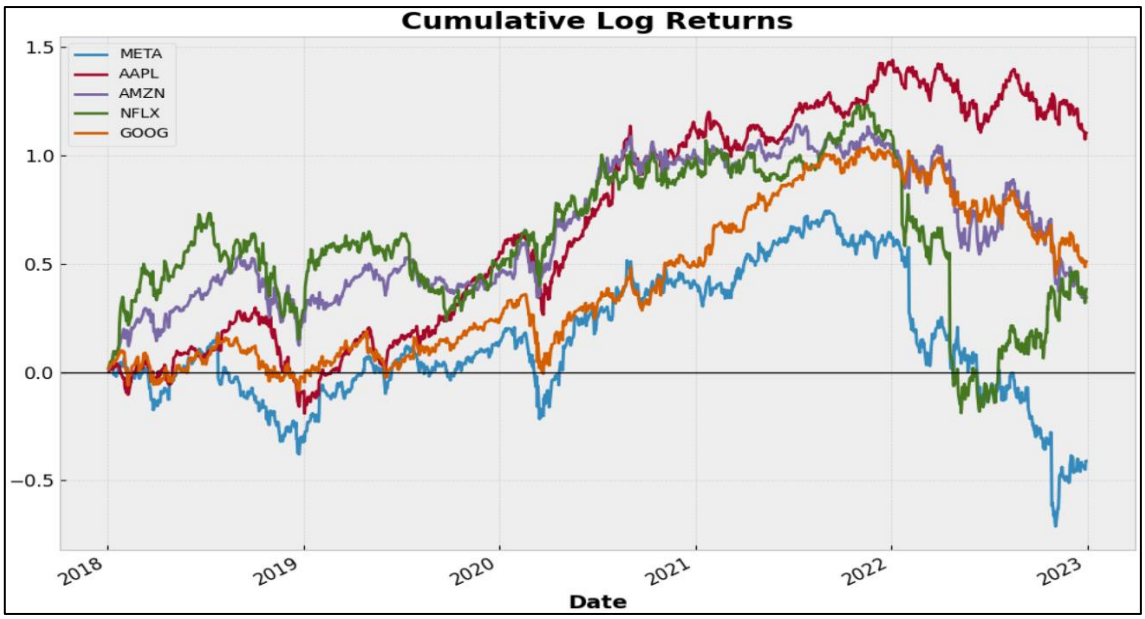

Exponential Moving Average
META
AMZN
NFLX
AAPL (right)
GOOG (right)
Date

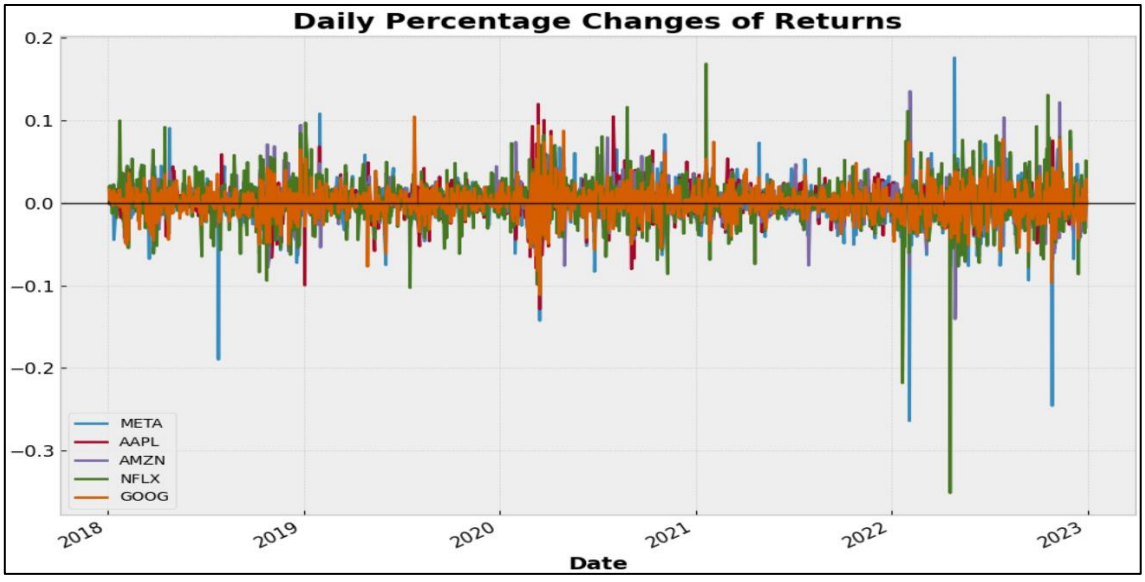

Moving Averages for NFLX using Exponential Moving Average Mean
NFLX
10d
50d
100d
150d
200d
buy signal
sell signal
Price
Date
700
600
500
400
300
200
2018
2019
2020
2021
2022
2023


Portfolio Asset Price History
META
AAPL
AMZN
NFLX
GOOG
Date
700
600
500
400
300
200
100
2018
2019
2020
2021
2022
2023


Portfolio Cumulative Returns
META
AAPL
AMZN
NFLX
GOOG
Date
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
2018
2019
2020
2021
2022
2023


Daily Log Returns
0.1
0.0
−0.1
−0.2
−0.3
−0.4
META
AAPL
AMZN
NFLX
GOOG
2018
2019
2020
2021
2022
2023
Date


Cumulative Log Returns
1.5
1.0
0.5
0.0
−0.5
META
AAPL
AMZN
NFLX
GOOG
2018
2019
2020
2021
2022
2023
Date


Daily Percentage Changes of Returns
0.2
0.1
0.0
−0.1
−0.2
−0.3
META
AAPL
AMZN
NFLX
GOOG
2018
2019
2020
2021
2022
2023
Date

```
wasim@wasim-VirtualBox:~/Desktop/EntroPy$ python3 scripts/script_pf_composition_api.py


========================================
            STOCK ALLOCATION
========================================

   Name  Allocation
0  META          20
1  AAPL          30
2  AMZN          25
3  NFLX          15
4  GOOG          10
[**********************100%**********************]  5 of 5 completed
[**********************100%**********************]  1 of 1 completed

========================================
         PORTFOLIO DISTRIBUTION
========================================

   Name Allocation
0  META         20
1  AAPL         30
2  AMZN         25
3  NFLX         15
4  GOOG         10

========================================
          ASSET PRICE HISTORY
========================================

                 META       AAPL       AMZN        NFLX       GOOG
Date
2018-01-02  181.419998  40.776527  59.450500  201.070007  53.250000
2018-01-03  184.669998  40.769413  60.209999  205.050003  54.124001
2018-01-04  184.330002  40.958794  60.479500  205.630005  54.320000
2018-01-05  186.850006  41.425125  61.457001  209.990005  55.111500
2018-01-08  188.279999  41.271263  62.343498  212.050003  55.347000

========================================
           PORTFOLIO SUMMARY
========================================

Portfolio containing information about stocks: META, AAPL, AMZN, NFLX, GOOG
 Forecast Return: 0.167
 Sharpe Ratio: 0.514
Sortino Ratio: 1.873

 Portfolio Volatility: 0.314
 Downside Risk: 0.0862
 Value at Risk: 68.3270
 Confidence Interval (Value at Risk): 95.000 %

 Skewness:
META    0.699468
AAPL    0.152113
AMZN    0.269860
NFLX    0.391759
GOOG    0.614572
dtype: float64

 Kurtosis:
META   -0.496395
AAPL   -1.573618
AMZN   -1.505556
NFLX   -0.653215
GOOG   -1.068987
dtype: float64

 Financial Index: ^GSPC with Forecast Return of 0.0949 and Annualised Volatility of
0.2187

 Beta Coefficient: 0.000
 Trading Horizon: 252
 Risk Free Rate of Return: 0.54%
```

```
wasim@wasim-VirtualBox:~/Desktop/EntroPy$ python3 scripts/script_pf_composition_csv.py

================================================================================
                          LOADED PORTFOLIO DATA
================================================================================

   Allocation  Name
0        20.0  META
1        30.0  AAPL
2        25.0  AMZN
3        15.0  NFLX
4        10.0  GOOG


================================================================================
                          FIRST ROWS OF STOCK DATA
================================================================================

                 META        AAPL        AMZN        NFLX        GOOG
Date
2018-01-02  181.419998   43.064999   59.450500  201.070007   53.250000
2018-01-03  184.669998   43.057499   60.209999  205.050003   54.124001
2018-01-04  184.330002   43.257500   60.479500  205.630005   54.320000
2018-01-05  186.850006   43.750000   61.457001  209.990005   55.111500
2018-01-08  188.279999   43.587502   62.343498  212.050003   55.347000


================================================================================
                       PORTFOLIO WITH EQUAL WEIGHTS
================================================================================

  Allocation  Name
0        0.2  META
1        0.2  AAPL
2        0.2  AMZN
3        0.2  NFLX
4        0.2  GOOG


================================================================================
                     PRICE HISTORY (EQUAL WEIGHTS)
================================================================================

                 META        AAPL        AMZN        NFLX        GOOG
Date
2018-01-02  181.419998   43.064999   59.450500  201.070007   53.250000
2018-01-03  184.669998   43.057499   60.209999  205.050003   54.124001
2018-01-04  184.330002   43.257500   60.479500  205.630005   54.320000
2018-01-05  186.850006   43.750000   61.457001  209.990005   55.111500
2018-01-08  188.279999   43.587502   62.343498  212.050003   55.347000


================================================================================
                      PORTFOLIO WITH CUSTOM WEIGHTS
================================================================================

  Allocation  Name
0       20.0  META
1       30.0  AAPL
2       25.0  AMZN
3       15.0  NFLX
4       10.0  GOOG


================================================================================
                     PRICE HISTORY (CUSTOM WEIGHTS)
================================================================================

                 META        AAPL        AMZN        NFLX        GOOG
Date
2018-01-02  181.419998   43.064999   59.450500  201.070007   53.250000
2018-01-03  184.669998   43.057499   60.209999  205.050003   54.124001
2018-01-04  184.330002   43.257500   60.479500  205.630005   54.320000
2018-01-05  186.850006   43.750000   61.457001  209.990005   55.111500
2018-01-08  188.279999   43.587502   62.343498  212.050003   55.347000
```

```
wasim@wasim-VirtualBox:~/Desktop/EntroPy$ python3 scripts/script_stock_specific_analys
is.py

Amazon Stock Attributes:

Date
2018-01-02    59.450500
2018-01-03    60.209999
2018-01-04    60.479500
2018-01-05    61.457001
2018-01-08    62.343498
Name: AMZN, dtype: float64
Date
2018-01-02         NaN
2018-01-03    0.012775
2018-01-04    0.004476
2018-01-05    0.016163
2018-01-08    0.014425
Name: AMZN, dtype: float64
0.13332817419630477
0.3580279090278777
0.2698597060446709
-1.5055563260529825
Stock: AMZN with Forecast Return of 0.1333 and Annualised Volatility of 0.3580
+---------------------------------------+-----------------------------------------
----------------+
| Property                              | Value
                |
+---------------------------------------+-----------------------------------------
----------------+
| Investment Name/Category              | Stock: AMZN
                |
| Forecast Return                       | 0.1333
                |
| Annualised Volatility                 | 0.3580
                |
| Investment Skew                       | 0.2699
                |
| Investment Kurtosis/Tailedness        | -1.5056
                |
+---------------------------------------+-----------------------------------------
----------------+

Amazon stock data on 2018-01-02 00:00:00:
59.45050048828125

Amazon stock data after 2018-01-01:
Date
2018-01-03    60.209999
2018-01-04    60.479500
2018-01-05    61.457001
2018-01-08    62.343498
2018-01-09    62.634998
                ...
2022-12-23    85.250000
2022-12-27    83.040001
2022-12-28    81.820000
2022-12-29    84.180000
2022-12-30    84.000000
Name: AMZN, Length: 1258, dtype: float64

Amazon stock data for the year 2022:
Date
2022-01-03    170.404495
2022-01-04    167.522003
2022-01-05    164.356995
2022-01-06    163.253998
2022-01-07    162.554001
                 ...
2022-12-23     85.250000
2022-12-27     83.040001
2022-12-28     81.820000
2022-12-29     84.180000
2022-12-30     84.000000
Name: AMZN, Length: 251, dtype: float64
```

12.3 Appendix C: GitLab Repository and Running the Project

The complete code for EntroPy, containing the source, test and example script code, can be found at: https://git.cs.bham.ac.uk/projects-2022-23/wan749

C.1 Repository Structure

EntroPy is structured with clarity and modularity in mind:

Name	Last commit	Last update
data	Finished and cleaned-up	2 weeks ago
scripts	Finished and cleaned-up	2 weeks ago
src	Fixed a few doc_ref strings	5 days ago
tests	Fixed a few doc_ref strings	5 days ago
.gitignore	Final commit to GitLab	5 days ago
README.md	Update README.md	2 months ago
architecture_uml.png	Add files via upload	1 week ago
requirements.txt	Finished and cleaned-up	2 weeks ago

- Data: contains the portfolio (initial allocation) and stock (price history) csv files
- Scripts: contains all example scripts to produce the plots discussed in Chapter 8
- Src: the source code of the project, aligning with Chapter 5's discussion
- Tests: all unit tests written for each module in src, supporting with Chapter 6
- Architecture_uml: this is the architecture diagram discussed in Chapter 4 to give users a quick reference to the overall structure
- Requriements.txt: contains most of the dependencies (all frameworks) outlined below

C.2 Requirements & Dependencies

The code currently runs on a Linux (Xubuntu) environment, and was built via a Debian installation of Visual Studio Code. The software requirements include:

- Python >= 3.11.4
- Numpy >= 1.15
- Pandas >= 2.0
- Scipy >= 1.11.2
- Matplotlib >= 3.0
- Yfinance >= 0.2.29
- Pytest >= 7.3.1 (should the user want to run any unit-tests)
- Pip >= 23.2.1 (should the user want a tool to manage dependencies)
- Visual Studio Code (or an IDE the user is comfortable with, and is compatible with Linux)

The hardware requirements, aligning with the development team's own Desktop PC and Laptop, include:

- 8 GB of RAM
- 30 MB of available disk space (EntroPy is 12.7 MB in size)
- 2.0 – 2.5 GHz or faster processor (EntroPy should not be overly computationally intensive)

C.3 Installation

The repository may be cloned using the command:

```
git clone https://git.cs.bham.ac.uk/projects-2022-23/wan749
```

Navigate in the command terminal to the repository using:

```
cd wan749
```

The project contains a requirements.txt file, so the user can install its dependencies using:

```
pip install -r requirements.txt
```

If any requirements are outstanding, the user can manually install them in Python using:

```
pip install [package-name]
```

```
pip install [path-to-downloaded-package]
```

The project must now be added to the Python path:

```
nano ~/.bashrc
```

Add the following line to the end of the file:

```
export PYTHONPATH=$PYTHONPATH:/path/to/your/project
```

Save and close the file. To make the changes take effect, either restart the terminal or run:

```
export PYTHONPATH=$PYTHONPATH:/path/to/your/project
```

Ensure that "/path/to/your/project" is replaced with the actual path to the user's cloned repository.

C.4 Running a Script

Using the script, script_mef, as an example, in order to run the script, the user must navigate to the correct location first, and open the project in the IDE:

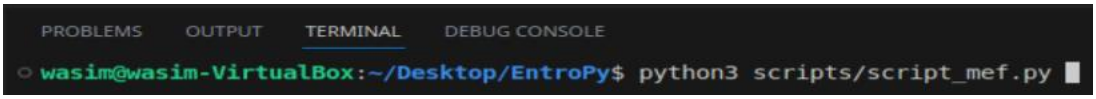
The user can subsequently open the script_mef file in the IDE. They will notice that an absolute import is used to define the data path of the price data in the line:

```
price_data_filepath = "/home/wasim/Desktop/EntroPy/data/MAANG_stock_data.csv"
```

Replace "/home/wasim/Desktop/EntroPy/data/MAANG_stock_data.csv" with the absolute path to the desired CSV file. For example:

```
price_data_filepath = "/path/to/your/datafile.csv"
```

Save and close the file, and then in the terminal run the following command:



This will run the script and should produce the desired results, as seen in Chapter 8. Absolute import has been used due to their clarity and readability, refactoring ease and to maintain "explicitness over implicitness". Tests are run using a very similar logic.