UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ADE Facultad de Administración y Dirección de Empresas /UPV

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Evaluating Score-Investing Methodologies: A Systematic Review of Tweenvest's Algorithm for Long-Term Stock Investing Using Descriptive Analytics and Predictive Modeling.

Trabajo Fin de Grado Integrado
Grado en Administración y Dirección de Empresas
Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Author: Carlos Eduardo Domínguez Martínez

Tutor: Joaquín Martínez Minaya (GADE)

Tutor: Alberto Albiol Colmener (GITST)

Extern Co-Tutor: José Tatay Sangüesa (Tweenvest)

Course: 2024-2025

Valencia, June 2025

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universitat Politècnica de València
Edificio 4D. Camino de Vera, s/n, 46022 Valencia
Tel. +34 96 387 71 90, ext. 77190
www.etsit.upv.es

VLC/ CAMPUS VALENCIA, INTERNATIONAL CAMPUS OF EXCELLENCE

EMAS Gestión ambiental verificada REG.NO.ES-CV-000030

Abstract

This study investigates the effectiveness of a factor-investing methodology developed by Tweenvest, leveraging a proprietary algorithm grounded in fundamental financial analysis. The algorithm scores companies across four key factors: Quality, Growth, Value, and Dividends.

The research aims to evaluate the profitability of investment strategies based on these scores over multiple profitability horizons from 1 month to five years. A comprehensive dataset was constructed, integrating factor scores with additional variables such as sector and geographic region, standardized for currency and timeframes.

Statistical analyses will explore relationships between these factors and returns, identifying optimal investment periods. Subsequently, predictive modeling—including econometric regressions, time series models, and neural networks—will be applied to assess the translation of these factors into market performance.

The study incorporates an interdisciplinary approach, combining financial theory and econometrics with advanced programming, data engineering, and artificial intelligence. This integration bridges Business Management and Telecommunications Engineering, offering insights into the practical application of Tweenvest's scoring algorithm and contributing to the advancement of financial technology analytics.

To my parents.

Contents

List of Figures

List of Tables

Part I

Main Report

Chapter 1

Introduction to Fundamental Analysis

1.1 Definition

Fundamental Analysis is a methodology used to evaluate the intrinsic value of a company, asset, or market by analyzing various economic, financial, qualitative and quantitative factors. Unlike technical analysis, which focuses on price movements and chart patterns, fundamental analysis seeks to determine an asset's "true value" to identify investment opportunities that may be undervalued or overvalued in the market.

This intrinsic value is defined by numerous experts and renowned investors—such as Benjamin Graham, Warren Buffett, and Pat Dorsey [6]—as a company's ability to adapt to an ever-changing environment, create value in the market, and establish barriers to entry for competitors, commonly referred to as *economic moats*.

Measuring these moats is challenging due to the difficulty of quantifying certain variables, such as brand strength and market influence. However, over the long term, these intangible factors translate into tangible financial data, reflected in a company's balance sheet, income statement, and cash flow statement; that when exposed to the public market share creates a need to buy or sell the stocks, altering the companies profitability. So from now on, this economic moats will be called *alpha*, following the common literature .

1.2 Tweenvest's Scores

This long-term advantage seen in the finance is used in many indexes such as MSCI [9], IBEX-35, etc. So following this approach, Tweenvest developed a series of factors to help users identify possible *alpha* in a company. Upon these factors, we can highlight four main scores strictly related to key focus areas that investors consider before making any decision:

- Profitability
- Financial Health
- Predictability
- Consistent Growth
- Entrance Moment
- Dividends Payed

1.2.1 Quality

This Tweenvest's score is approached in a similar way that many successful investors would when analyzing the annual reports, by distinguishing on three main categories: **profitability**, **financial health**, and **predictability**. Each of them includes inside of them multiple financial ratios that take account of different relevant data within the company's reports.

To understand the complexity of this score, we need to look at each category separately. Starting with **profitability** we can separate:

Profitability Margins

These are essential for understanding how a company is managing its costs and generating profits from its revenues.

- **Net margin**: Represents net profit as a percentage of total sales, indicates a company's efficiency in generating profits after accounting for all expenses, taxes, and costs.
- **Operating margin**: Measures operating profit (EBIT) as a percentage of total sales, providing a clear view of the profitability of a company's core operations, excluding interest and taxes. This metric is fundamental for evaluating a company's operational efficiency.
- **EBITDA margin**: Removes the effects of capital structure and accounting policies, offering a clear view of the company's pure operational profitability.
- **Gross margin**: Focuses on revenues after deducting the cost of goods sold, is a key measure of production efficiency and a company's ability to manage its direct costs.

Performance Ratios

These are used to measure the overall performance of the company.

- **ROA (Return on Assets)**: Measures how efficiently a company converts its assets into profits. This is especially important in capital-intensive sectors, where efficient asset management can make a significant difference in profitability.
- **ROE (Return on Equity)**: Focuses on the profits generated per dollar of equity invested by shareholders. This ratio is crucial for evaluating a company's overall profitability from the shareholders' perspective. It is an especially valuable metric for investors seeking to maximize their returns on equity investment.
- **ROIC (Return on Invested Capital)**: Focuses on the return generated by all the funds invested in the company, including both shareholders' equity and debt. It is a comprehensive measure of a company's ability to generate value from all its sources of financing.
- **ROCE (Return on Capital Employed)**: Measures the funds used to finance operations, regardless of the source. This ratio is useful for comparing the efficiency of companies with different capital structures, as it focuses on total capital employed rather than just equity.

Also, to not only look at actives, Tweenvest uses the cash generated to calculate: **CROIC** (Cash Return on Invested Capital), **CROCE** (Cash Return on Capital Employed), **OCF/Sales**, and **FCF/Sales**. And lastly it takes account also the Owner's income to calculate: **Owner's Income/Sales**, **Owner's CROIC** and **Owner's CROCE**.

Continuing with the **financial health**, we need to analyze the company's debt in different aspects:

Leverage Ratios

These ratios assess how much a company relies on debt to finance its assets and operations, and are essential for evaluating financial risk and long-term solvency.

- **Financial Leverage** (Total Assets / Equity): Measures the proportion of a company's assets that are financed by shareholder equity. A higher ratio suggests the company is using more debt relative to equity, indicating greater financial risk but also potential return amplification through leverage.
- **Total Debt/Assets**: Indicates what portion of the company's assets is financed through debt. A lower ratio implies a more conservative capital structure, while a higher one may indicate increased risk if the company becomes over-leveraged.
- **Total Debt/Capital**: Measures the share of total capital (debt + equity) that comes from debt. This ratio is useful for understanding how dependent the company is on borrowed funds compared to its overall capital base.
- **Total Debt/Equity**: Compares the company's total debt to its shareholder equity. It provides insight into the balance between debt and equity financing. A high ratio may signal financial risk, but also the potential for higher returns if debt is managed well.

Debt Coverage Ratios

These metrics evaluate a company's ability to cover its debt using its earnings or cash flow, reflecting the sustainability of a company's debt in relation to its operational performance.

- **Net Debt/EBIT**: Shows how many years it would take for a company to repay its net debt using EBIT (Earnings Before Interest and Taxes).
- **Net Debt/EBITDA**: Similar to the above, but adds back depreciation and amortization. This gives a more cash-focused view of a company's ability to handle its debt load, and is especially useful for comparing companies in capital-intensive industries.
- **Net Debt/FCF**: Evaluates how many years of free cash flow would be needed to pay off net debt. Since FCF includes investment needs, this ratio gives a more conservative view of debt sustainability.
- **Net Debt/Owner's Income**: Compares net debt to the income available to equity holders (after all operating and investing costs).

Interest Coverage Ratios

These ratios measure how easily a company can meet its interest payments on outstanding debt — critical for assessing short-term debt service capability.

- **EBIT/Interest**: Indicates how many times a company can cover its interest expenses with its operating income.
- **EBITDA/Interest**: Similar to the above, but adds back depreciation and amortization. This gives a clearer picture of available cash earnings before fixed financial obligations, ideal for heavily asset-based businesses.
- **FCF/Interest**: Since FCF considers investment needs, this is a stringent test of how much real, discretionary cash is available for debt servicing.
- **Owner Earnings/Interest**: Evaluates a company's ability to meet interest payments based on the earnings effectively attributable to shareholders. It accounts for operational cash flow minus necessary capital expenditures.

Liquidity Ratios

These ratios measure a company's ability to meet short-term obligations with its short-term assets. They are essential for evaluating near-term financial health and risk of insolvency.

- **Current Ratio** (Current Assets / Current Liabilities): Shows whether a company has enough assets to cover its short-term liabilities. A value above 1 is generally considered healthy, though excessively high values may imply inefficiency.
- **Quick Ratio** ((Current Assets - Inventory) / Current Liabilities): A more stringent version of the current ratio that excludes inventory, which may not be easily liquidated. It's useful in assessing true short-term liquidity.
- **Cash Ratio** (Cash and Equivalents / Current Liabilities): The most conservative liquidity metric, focusing only on cash and equivalents. It shows the immediate solvency of a company in a worst-case scenario.
- **OCF Ratio** (Operating Cash Flow / Current Liabilities): Assesses how well the company's operational cash flows can cover its current obligations. This offers a realistic view of liquidity since it's based on actual cash generation rather than accounting figures.

Predictability

And finally, for the quality score we need to look at the company's predictability. This is achieved by trying to fit values related to the company's success —such as Sales— to a exponential curve, using the ordinary least square method, which is supported by large financial literature [9].

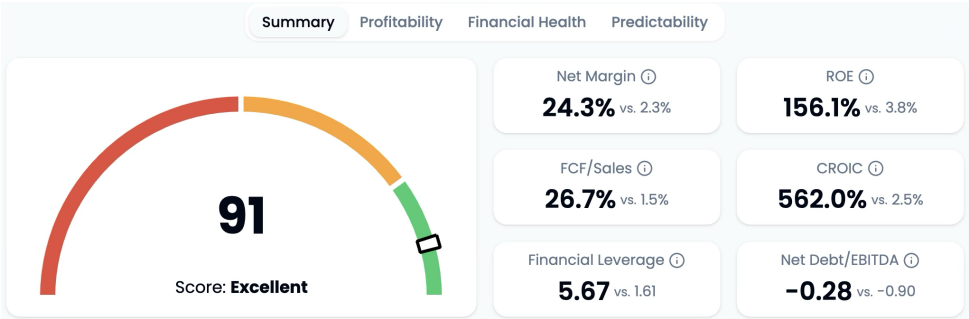

Figure 1.1: Tweenvest's Quality Score

After calculating all of this ratios, Tweenvest compares them to the sectors' median and interpolates each of them to create a single score for the ratio, then it aggregates them all using personalized weights to create the final Quality Score, which is then showed to the clients as shown in Figure 1.1.

1.2.2 Growth

The Growth Score evaluates a company's historical growth across multiple key metrics, and compares them to industry standards. This comprehensive approach ensures a balanced assessment of growth across different aspects of the business:

Revenue & Profitability Growth

- **Sales**: Measures growth in core revenue streams

- **EBITDA**: Captures growth in operational cash-generating ability before non-cash and financing impacts
- **Operating Income**: Reflects growth in profit from core business operations
- **Net Income**: Includes all income sources, showing overall profitability growth

Cash Flow Growth

- **Operating Cash Flow**: Measures growth in cash generation from operations
- **Simple FCF**: A straightforward proxy for available cash after essential investments
- **Levered/Unlevered FCF**: Provide detailed views of free cash flow with and without debt impact
- **Owner Earnings**: Useful for volatile capex cases, emphasizing cash available to shareholders

Capital Base Expansion

- **Total Assets**: Indicates expansion in overall asset base
- **Equity**: Reflects growth in shareholders' claim on the business
- **Tangible Book Value**: Highlights growth in physical net assets, excluding intangibles
- **Invested Capital**: Captures total capital being put to productive use
- **Capital Employed**: A broader measure of capital supporting business operations

Per-Share Value Growth

- **Diluted EPS**: Tracks per-share earnings growth, accounting for dilution effects
- **Diluted Shares**: Included to track share count changes, ensuring EPS growth isn't artificially inflated by buybacks or dilution
- **Ordinary DPS**: Tracks the growth of shareholder payouts, a proxy for confidence in future earnings

To compute the Growth Score, Tweenvest calculates 10-year, 5-year, and 3-year averages and then interpolates the growth rate to industry standards. This approach reinforces the long-term investment philosophy, giving lasting growing companies a better score.

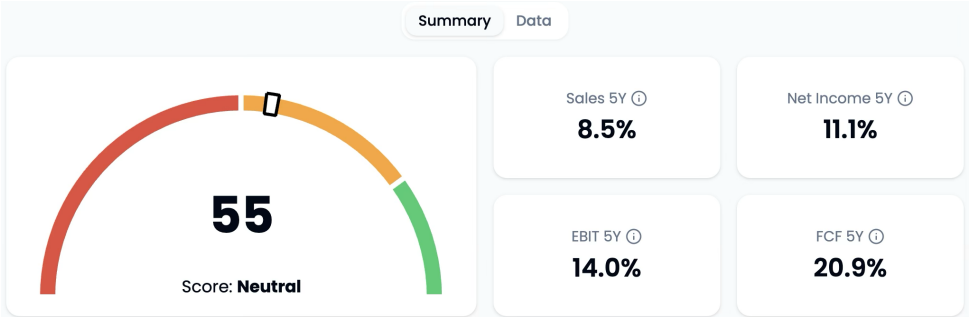

Figure 1.2: Tweenvest's Growth Score

1.2.3 Valuation

The Valuation Score measures how attractively a company is priced relative to fundamental multiples such as earnings, cash flow, sales, and dividends. This is critical for investors following value investing principles, where the goal is to buy quality companies for less than their intrinsic worth. The algorithm evaluates a series of valuation multiples—both price-based and enterprise value-based—and dividend yield.

Price-Based Multiples

- **P/E** (Market Cap / Adjusted TTM Earnings): Measures how much investors are willing to pay per dollar of earnings
- **P/S** (Market Cap / TTM Revenue): Useful when earnings are volatile; shows valuation relative to sales
- **P/CF** (Market Cap / TTM Operating Cash Flow): Reflects valuation relative to cash-generating ability
- **P/B** (Market Cap / Tangible Equity): Especially relevant for asset-heavy sectors like banks or industrials

Enterprise Value-Based Multiples

- **EV/Sales** (Enterprise Value / TTM Revenue)
- **EV/EBITDA** (Enterprise Value / TTM EBITDA)
- **EV/EBIT** (Enterprise Value / TTM EBIT)
- **EV/FCF** (Enterprise Value / TTM Free Cash Flow)

Yield-Based Valuation

- **Dividend Yield (%)** (Dividend per Share / Price per Share)

Each of these ratios is compared to multiple historical statistics and sectoral benchmarks to create individual scores and then average them.

Note:

- Market Cap = Share Price × Total Outstanding Shares
- Enterprise Value = Market Capitalization + Total Debt - Cash + Marketable Securities

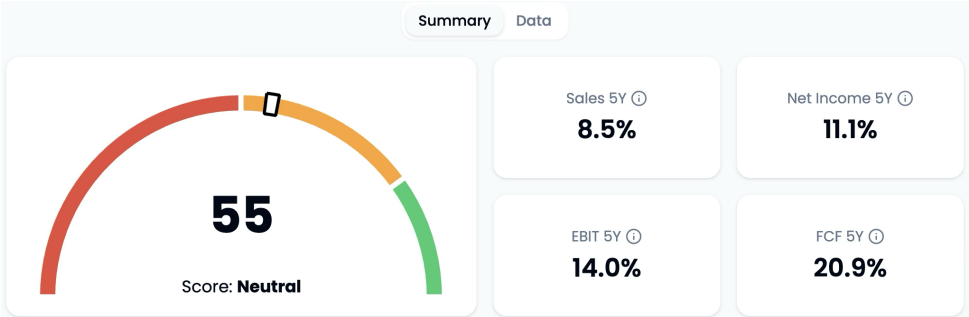

Figure 1.3: Tweenvest's Value Score

1.2.4 Dividend

When analyzing Tweenvest most common investors profile, we see a high tendency to income-focused and long-term investors using mainly dividends as their principal concern. Thats why the platform dedicated a full section to this topic.

The Dividend Score measures the attractiveness, reliability, and growth potential of a company's dividend payments. It helps investors assess whether the dividend is both rewarding today and sustainable for tomorrow. To do so, the score was built from three primary components:

Safety

This part of the score is used to assess the sustainability of the dividend.

- **Payout Ratio EPS** (DPS / Diluted EPS): Shows if dividends are covered by accounting earnings
- **Payout Ratio FCF** (DPS / Free Cash Flow per Share): Shows if dividends are funded by real cash generation
- **Payout Ratio Owner Earnings** (DPS / Owner Earnings per Share): A conservative test of sustainability (excludes CAPEX)

Growth

Measures how consistently and strongly the dividend has grown over time.

- **Ordinary DPS CAGR**: 3-Year, 5-Year, 10-Year growth rates

Yield

This evaluates the attractiveness of the dividend today relative to the company's historical averages and sector benchmarks.

- **Dividend Yield** (DPS / Price per Share): Represents how much income an investor receives annually from dividends

These components are individually scored weighted and interpolated against industry benchmarks to form a composite score.

Figure 1.4: Tweenvest's Dividend Score

Finally, as seen in Figure 1.4, the algorithm adjusts this score based on how many years the dividend has been maintained or increased, **rewarding consistency**.

1.3 Problems with the current approach

In the context of this thesis, we have to explain the need to make a systematic review of the scores to check wether the simplifications and hypotheses assumed by the financial consensus truly show a company's ability to create *alpha* over time or not.

1.3.1 Unquantified Variables

Since the current model only includes variables found in a company's financial statements, there is a significant amount of relevant information being left out, for example:

- The **perceived differentiation** of a product is one such element that may not be properly captured by traditional financial analysis. A trusted brand or strong reputation can allow a company to charge premium prices and build customer loyalty, generating extraordinary long-term profits. For instance, brands like Tiffany or Rolex have high perceived value that justifies elevated prices—something that cannot be easily quantified using standard financial metrics such as profit margins or return on capital.
- Additionally, strategies such as **cost leadership** and offering products at lower prices can provide a significant competitive advantage that is not always directly reflected in financial data.
- Companies may also establish **barriers to entry** and **high switching costs** for customers, making it more difficult for them to move to competitors. These strategies may involve investments in technology, patents, or simply building long-term customer and employees relationships.

To properly measure these variables, a more exhaustive analysis of each company would be required, pulling from multiple secondary information sources such as news articles, conference transcripts, customer blogs, competitive product reviews, and more. This task is far more difficult to automate via code, as it would require multimodal AI techniques—thus, it will fall outside the scope of this work.

1.3.2 Emergent Effects in Complex Systems

In complex systems like financial markets, network effects emerge, adding noise to actual reliable data behind it. For example:

- **Herd behavior** is a network effect where investors tend to follow the actions of others rather than rely on their own analysis. This can amplify market movements, both upward and downward. Herd behavior can lead to speculative bubbles and abrupt corrections when collective expectations shift.
- **Feedback loops** are another effect where market participants' actions reinforce existing market behavior. For example, rising asset prices may attract more investors, which in turn drives prices even higher. This type of positive feedback can cause price escalation that becomes detached from underlying fundamentals. Conversely, negative feedback can occur during a sell-off, where falling prices trigger more selling, further accelerating the decline.
- **Macro-level influences** are another crucial factor where broader structural changes—such as political decisions, economic policies, technological shifts, or industry-wide trends—can significantly impact market behavior. These larger forces often operate beyond the scope of individual company analysis and can create ripple effects throughout the entire market ecosystem.

1.3.3 Linearity and Stationarity

Non-linearity

The relationships between financial variables are often non-linear, meaning that changes in one variable may not result in proportional changes in another.

- **Economies of scale** can create non-linear relationships between production volume and costs. As a company grows, it may experience decreasing marginal costs due to better resource utilization, bulk purchasing discounts, or spreading fixed costs over larger output.
- **Market saturation** can lead to diminishing returns on marketing spend or R&D investments. Initial investments might yield significant returns, but as the market becomes saturated, additional spending may produce smaller incremental benefits.
- **Competitive dynamics** can create threshold effects where small changes in market share or pricing can trigger significant shifts in competitive position or profitability.

Trends

Traditionally, standard growth ratios have been used, based on the assumption that in competitive markets, when a new business model or opportunity emerges with above-average margins, entrepreneurs quickly move in to capitalize on it—eventually saturating the opportunity and driving margins back down to average levels over time. But what happens when there is a significant shift in trend?

Markets are "fluctuating entities", so static metrics can become problematic when underlying trends change. A clear example is the rise of artificial intelligence and the surge in stock prices of companies involved in the production and development of the necessary technologies.

Chapter 2

Objectives

Knowing the possible problems shown with the current approach, Tweenvest has the need to check if the scores actually represent an objective and accurate view of the company's ability to generate *alpha* for different time frames, and propose changes to the current algorithm if needed for being able to present their results with more accuracy to their users. To do this, we need to follow multiple steps:

1. **System Architecture Enhancement**: Modify Tweenvest's database architecture to enable historical score storage and retrieval and create the datasets for the analysis.
2. **Data Curation**: Clean and preprocess the collected data to ensure consistency and reliability, handling missing values and outliers appropriately.
3. **Exploratory Analysis**: Process and analyze the data to check the distributions and correlations between variables, looking for early on patterns to later compare and use in the modeling.
4. **Predictive Modeling**: Develop multiple predictive models to check what is the best way to use the scores for multiple time frames investment strategies.
5. **Validation & Benchmarking**: Back-test the algorithms to check their performance and consistency to see if the scores generate positive *alpha* over time.

Chapter 3

Methodology & Theoretical Framework

3.1 Code practices

Since the code that needs to be changed is used in a production environment by Tweenvest, it is important to follow some *good practices* to ensure the code is easy to understand, maintain, and to avoid introducing new bugs.

- **Understanding and Documenting**: Before starting to work on the code, it is important to understand the codebase and the purpose of the code. For this, it is recommended to use some tools like flux diagrams, code comments, and documentation. As it will be shown later on.
- **Testing**: To ensure proper functionality, every function and class should have unit tests that cover all possible scenarios. These tests are run automatically by the CI/CD pipeline.
- **Pair code review**: After completing the previous steps, the code undergoes a pair code review process, where at least one other team member reviews and approves the changes in *Github* before merging into the main branch.
- **Logging**: After each feature is implemented, it is important to analyze the actual performance in the production environment by looking at its logs to check if the feature is working as expected, and to look for any possible optimizations to be made.

3.2 Preprocessing

Once we have created our raw dataset, we need to inspect it and make sure it is consistent and ready to be used for the modeling. This part is really important to avoid any bias in the modeling process and to only use the data that is actually relevant for the analysis.

3.2.1 Data Consistency

Since all of the data is coming from a data provider that uses OCR as one of their tools whenever they don't have the data in a structured format, we need to be very restrictive with the data used. Also, as we mentioned before, some of the algorithms for calculating the scores use long-term data such as 10 year growths rates.

This is one of the biggest **limitations to the study**: we will be missing the data of companies that stopped existing or that were acquired by other companies, if they didn't exist for 10 or more years.

For inspecting the data, we will use the *pandas* [11], *matplotlib* [7], and *seaborn* [12], which are powerful tools for data manipulation and analysis. The focus will be on the following aspects:

- **Missing Values**: We need to check if there are any missing values in the data.
- **Data Distributions**: We need to make sure that the data is distributed in a way that is suitable for the modeling.
- **Variable Correlations**: We also have to see if there are any correlations between the variables. For this, we will use the correlation matrix defined as:

$$\rho_{X,Y} = \frac{\text{Cov}(X,Y)}{\sigma_X \sigma_Y} \tag{3.1}$$

  where $\text{Cov}(X,Y)$ is the covariance between the variables X and Y, σ_X is the standard deviation of X, and σ_Y is the standard deviation of Y. This matrix will be created with a custom heatmap function (see Appendix 7.4).
- **Data Patterns**: We need to check if there are any visible data patterns between the variables. To address this, we will use a custom *pairplot* function from *seaborn*, which is a powerful tool for data visualization and exploration depending on the type of variable (see Appendix 7.4).

Additionally, we decided to assign a value of 0 to all null Dividend scores, as a null value indicates that the company does not pay dividends. This choice was made to ensure that the Dividend score remains part of the analysis when evaluating its relevance to long-term profitability, rather than being excluded due to missing values.

3.2.2 Data Transformation

To ensure a robust and well-performing model, it is often necessary to transform the data into a format more suitable for learning algorithms, with the obligatory condition of being able to revert it back to its original scale when needed.

This step is particularly important because many components of machine learning models—such as the RBF kernel in Support Vector Machines or the regularization terms (L1 and L2) in linear models—implicitly assume that features are centered around zero and have comparable variances. Without this adjustment, features with significantly larger variances could dominate the optimization process, leading the model to underweight or ignore more informative but lower-variance features.

Standard Scaler

When training the neural networks, it is really important to standardize the data. We can easily use the *StandardScaler* from the *scikit-learn* library that transforms a feature x_i with the following formula:

$$z_i = \frac{x_i - \mu}{\sigma} \tag{3.2}$$

where μ is the mean of the feature and σ is its standard deviation. This transformation results in a distribution with a mean of 0 and a standard deviation of 1.

Gaussian Transformation

In numerous modeling applications, having normally distributed features is advantageous. **Power transformations** represent a set of parametric, monotonic functions that are an extension of the Box-Cox transformation. They are designed to convert data from various distributions into approximately Gaussian distributions, thereby reducing variance fluctuations and decreasing distribution asymmetry.

We decided to use the **Yeo-Johnson transformation** because it allows for negative values and it is reversible.

$$x_i^{(\lambda)} = \begin{cases} \frac{[(x_i+1)^\lambda - 1]}{\lambda}, & \text{if } x_i \geq 0, \lambda \neq 0 \\ \ln(x_i + 1), & \text{if } x_i \geq 0, \lambda = 0 \\ -\frac{[(-x_i+1)^{2-\lambda} - 1]}{2-\lambda}, & \text{if } x_i < 0, \lambda \neq 2 \\ -\ln(-x_i + 1), & \text{if } x_i < 0, \lambda = 2 \end{cases} \tag{3.3}$$

Where λ is a power parameter that helps minimize the skewness of the data. And since we have already deleted extreme outliers, the final distribution shouldn't be too distorted.

Here are some exammples of the transformations for different distributions:

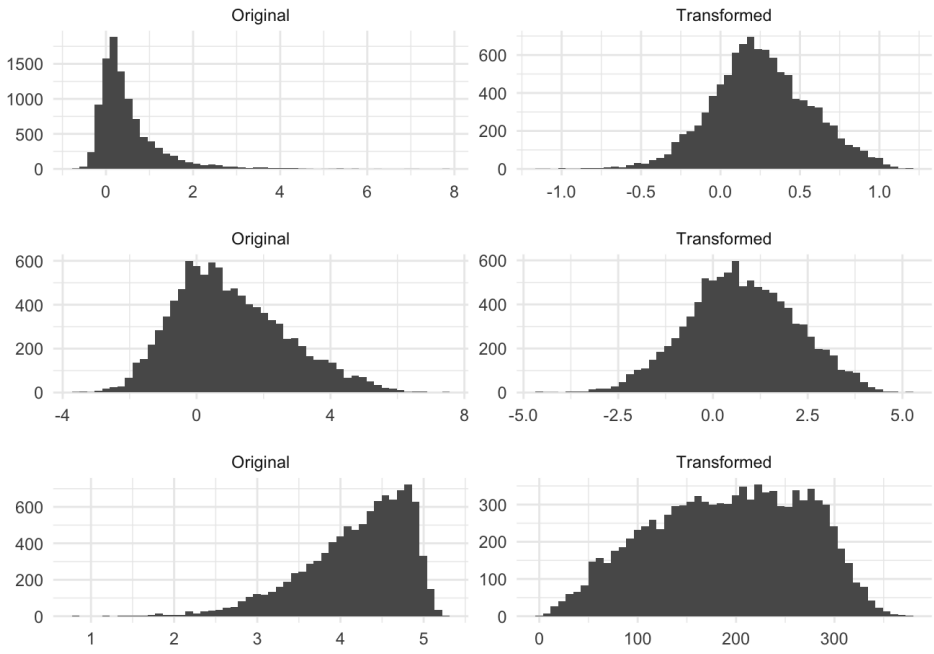

Figure 3.1: Yeo-Johnson Transformations examples

3.3 Outlier Detection

Anomalies are data patterns that have different data characteristics from normal instances. The ability to detect anomalies has significant relevance since often provide critical and actionable information in many different contexts.

For example, anomalies in credit card transactions could signify fraudulent use of credit cards, or an unusual computer network traffic pattern could stand for an unauthorised access.

3.3.1 Inter Quartile Range

Given a univariate dataset, the Interquartile Rule identifies outliers based on the interquartile range (IQR). The steps are as follows:

1. Compute the first quartile (Q_1), which is the 25th percentile of the data.
2. Compute the third quartile (Q_3), which is the 75th percentile of the data.
3. Calculate the interquartile range:

$$\text{IQR} = Q_3 - Q_1$$

4. Define the lower and upper bounds for non-outlier values following the 1.5 rule:

$$\text{Lower bound} = Q_1 - 1.5 \times \text{IQR}$$

$$\text{Upper bound} = Q_3 + 1.5 \times \text{IQR}$$

5. Any data point x is considered an outlier if:

$$x < \text{Lower bound} \quad \text{or} \quad x > \text{Upper bound}$$

As we can see this is a very simple and intuitive method, but it has some limitations:

Pros	**Cons**
Simple to compute and interpret.	Not suitable for multivariate data with correlated features.
Non-parametric.	May misclassify values in skewed distributions as outliers.
Robust to extreme values, since it relies on percentiles rather than the mean.	Fixed multiplier is arbitrary and may need tuning for each dataset.

Table 3.1: Pros and Cons of the Interquartile Rule

Even though this method is simple to compute and interpret, it is not suitable for multivariate data with correlated features, and that's where the rest of the methods come into play.

3.3.2 Single Vector Machine

Support Vector Machines (SVMs) represent a robust class of unsupervised learning algorithms that can be effectively adapted for anomaly detection tasks. This algorithm establishes itself on the premise that the majority of real-world data is inherently normal. Where the goal is to define a boundary encapsulating the normal instances in the feature space, thereby creating a region of familiarity.

To capture the bases of the *scikit-learn* implementation [3] used in this work, we can say that the main idea is to find a minimum volume sphere that contain all the training samples. This sphere, described by its center c and its radius r, is obtained by solving the constrained optimization problem: [13]

$$\min_{r,c} r^2 \quad \text{subject to} \quad \|\Phi(x_i) - c\|^2 \leq r^2 \quad \text{for} \quad i = 1, 2, \ldots, n \tag{3.4}$$

This boundary is strategically positioned to maximize the margin around the normal data points, allowing for a clear delineation between what is considered ordinary and what may

be deemed unusual. This emphasis on margin maximization is akin to creating a safety buffer around the normal instances, fortifying the model against the influence of potential outliers or anomalies.

In summary, this method has the following characteristics:

Pros	**Cons**
Effective for high-dimensional data and complex boundaries.	Sensitive to the choice of kernel and hyperparameters (e.g., ν, γ).
Works well when the training set contains mostly or only normal instances.	Computationally intensive, especially on large datasets.
Can capture nonlinear patterns using kernels (e.g., RBF kernel).	Difficult to interpret or explain the decision function.
No need for labeled data from the outlier class.	–
Solid theoretical foundation and widely supported in machine learning libraries.	–

Table 3.2: Pros and Cons of One-Class SVM for Outlier Detection

3.3.3 Isolation Forest

As we are seeing in this small fraction of outlier detection methodologies, most of the existing anomaly detection approaches are based on the premise of normal distributions, then identify anomalies as those that do not conform to the normal profile.

But to solve this issue we can use the Isolation Forest algorithm, which is explained in the original paper Liu, Ting, and Zhou [8]. The algorithm constructs multiple isolation binarytrees (iTrees) from the dataset, where anomalies are identified as data points that exhibit shorter average path lengths across these trees.

Lets consider a dataset $X = \{x_1, \ldots, x_n\}$ containing n points in d-dimensional space, and a subset $X' \subset X$. An Isolation Tree (iTree) is constructed as follows:

- Each node T in the tree is either:
  - An external node (leaf) with no children, or
  - An internal node with exactly two children (T_l and T_r) and a test condition
- The test condition at each internal node consists of:
  - A randomly selected attribute q
  - A split value p
  - A test $q < p$ that determines whether a point goes to T_l or T_r

The tree construction process recursively partitions X' by randomly selecting attributes and split values until either:

- The node contains only one instance, or
- All instances at the node have identical values

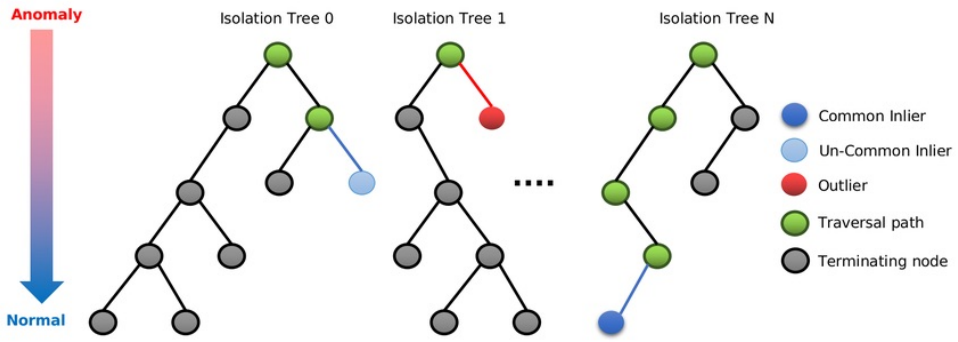

Figure 3.2: Visualization of how Isolation Forest works.

As we can see in Figure 3.2, once the iTree is fully constructed, each point $x_i \in X$ is isolated at a leaf node. The path length $h(x_i)$ of a point x_i is defined as the number of edges traversed from the root to its leaf node. Points with shorter path lengths are considered more likely to be anomalies, as they require fewer splits to be isolated from the rest of the data.

This method works very good in most cases, but it has also its own limitations:

Pros	**Cons**
Efficient and scalable to large datasets due to its tree-based structure.	Less effective for detecting local outliers in densely clustered regions.
Non-parametric: does not assume any data distribution.	Performance can vary with the choice of parameters like number of estimators and subsample size.
Naturally handles high-dimensional data.	Not suitable for detecting subtle anomalies that closely resemble normal data.
Requires little preprocessing (no need for feature scaling).	–
Robust to irrelevant features due to random sub-sampling.	–

Table 3.3: Pros and Cons of the Isolation Forest Method for Outlier Detection

3.3.4 Local Outlier Factor

Since we are looking at outliers from the perspective of correlations between the different response variables to see which companies don't behave naturally. We also checked on the Local Outlier Factor (LOF) for being based on hyper-plane densities of data.

The algorithm measures the local deviation of the density of a given sample with respect to its neighbors. It is local in that the anomaly score depends on how isolated the object is with respect to the surrounding neighborhood. More precisely, locality is given by k-nearest neighbors, whose distance is used to estimate the local density. By comparing the local density of a sample to the local densities of its neighbors, one can identify samples that have a substantially lower density than their neighbors. These are considered outliers. Breunig et al. [2]

Pros	Cons
Can detect outliers relative to their local neighborhood, allowing it to identify context-specific anomalies.	LOF scores are relative and lack a universal interpretation threshold for outliers.
Effective in datasets with varying densities — unlike global methods, it can recognize outliers near dense clusters.	Sensitivity to parameter selection (e.g., number of neighbors) can affect performance and consistency.
Applicable in any domain where a dissimilarity measure is defined, not limited to vector spaces.	Not inherently scalable to very large datasets without approximation or optimization.
Works well across domains, such as network intrusion detection or classification tasks.	–
Easily generalizable and adaptable for use in spatial data, temporal data, and network structures.	–

Table 3.4: Pros and Cons of the Local Outlier Factor (LOF) Method

3.3.5 Multi-Criteria Outlier Detection

There is a big controversy in the field about the best method to detect the outliers, since the results can vary a lot depending on the algorithm used.

So to attempt to fix this issue, after reading some aggregative methods such as Abro, Taşci, and Uğur [1], we created a multi-modal method that combines the results of the different methods to get a more robust result.

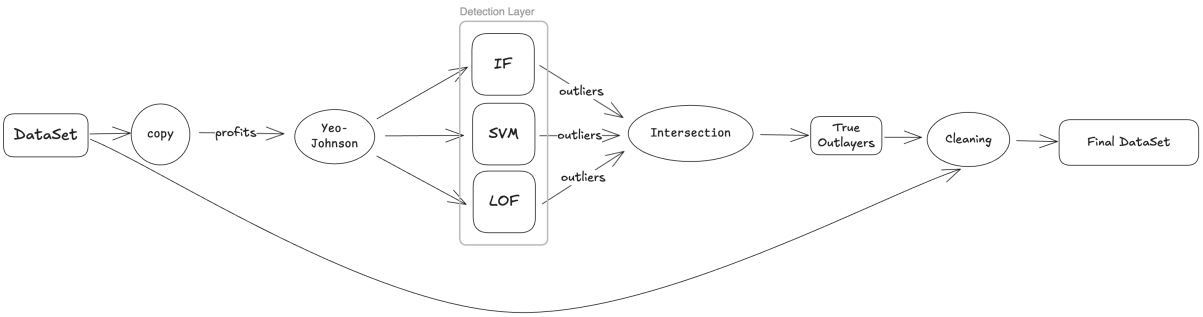

Figure 3.3: Multi-Criteria Outlier Detection Method

As shown in the Figure 3.3, this starts by homogenizing the variables distribution with a common data transformation so they are more Normal, but to make sure that the final data isn't being transformed we copy the original dataset earlier. After this, we use multiple methods to detect the different outliers and then we make the intersection between them to get the common outliers. By the end of this process we have tagged the original dataset rows, assuring that the data that will be excluded is only formed by *"true outliers"*.

We noticed that since this method is very restrictive, we have to set a threshold multiplier to the original dataset to get the final outliers. So for example, a 20% threshold would only exclude around 4% of the original dataset because of the intersection of all the methods makes it very restrictive. Here we **propose for future work** to fine tune the different thresholds for each

method to get a more robust result.

3.4 Predictive Models

The main objective of this thesis is to check if the scores have any relation with the future profits of the companies. For doing this in the most generalized way, instead of creating a portfolio following some subjective criteria on the scores, markets and other variables, we used predictive modeling to check for the actual relationships between the response variables and the explanatory ones.

As a first approach, we created a set of models that separate the scores and the different returns.

Response Variables

To help users understand how to use the scores depending on their investment horizon, we used different profits that from know one will be representad as P_i for each time horizon: $i \in \{1 \text{ month}; 3 \text{ months}; 6 \text{ months}; 1 \text{ year}; 2 \text{ years}; 5 \text{ years}\}$

Explanatory Variables

For being able to train the models, we have different variable categories:

- **Main variables**, these are numerical variables from **Tweenvest's Scores**, that are in the range of 0 to 100: S_j where $j \in \{$ Quality; Value; Dividend; Growth$\}$
- **Dummy variables**, these are boolean variables that allow us to tag the companies by:
  - **Industry**: I_k where $k \in \{$Financials; Healthcare; etc.$\}$
  - **Region**: R_l where $l \in \{$Europe; North America; etc.$\}$
- **Market Variables**, these are numeric variables that represent the companies *"size"* using the: **Market Cap** and **Volume**.

3.4.1 Regression Models

To create the econometrical models, we want to be able to understand the actual relationship between the response variables and the explanatory ones. For this we started with the creation of multiple regression models.

Linear Regression

As for the first models, we created a series of linear regression models for each factor score that includes the interactions between the dummy variables and the score used in each model.

$$\hat{P}_i = \beta_0 + \beta_1 \cdot S_j + \sum_{n=2}^{N} \beta_n \cdot D_n + S_j \cdot \sum_{m=N+1}^{M} \beta_m \cdot D_m \tag{3.5}$$

where:

- P_i is the estimated profit for each time horizon i
- S_j is the score for each factor j
- $D_{n,m}$ are the dummy variables for the industry and region, where $D_{n,m} := I_k \cup R_l$
- β are the coefficients that weight each variable.

For fitting the models, we also applied a backward selection to remove the variables that are not statistically significant, using the **p-value** criterion.

Generalized Additive Models

Because the relationships between the variables are not linear, we also used the Generalized Additive Models (GAM) to check if they were able to capture the non-linear relationships between the variables.

According to Servén and Brummitt [10], GAMs are smooth semi-parametric models that can capture non-linear relationships between variables. They take the form:

$$g(\mathbb{E}[y|X]) = \beta_0 + f_1(X_1) + f_2(X_2, X_3) + \cdots + f_M(X_N) \tag{3.6}$$

where:

- $X^T = [X_1, X_2, \ldots, X_N]$ are the independent variables
- y is the dependent variable.
- $g()$ is the link function that relates the predictor variables to the expected value of the dependent variable.
- $f_m()$ are feature functions built using penalized B-splines, which automatically model non-linear relationships without requiring manual transformation of variables.

In our case, we used pyGAMs LinearGAM since the model gives a Normal error distribution, and an identity link.

3.4.2 Time Series

In the stock market, the prices of the stocks are not independent of each other, they are correlated in the time series and they also have memory on past behavior of the company. This is what is known as **Momentum**, so companies that have a good past performance are more likely to have a good future performance, and vice versa.

For this reason, we contemplated two possibilities:

ARIMA Models

To take account the profit tendency, we implemented ARIMA models to improve the regression models performance by using the residues of the already fitted model:

$$P_i = \hat{P}_i + \varepsilon_i \quad \longleftrightarrow \quad \varepsilon_i \sim ARIMA_i(p, d, q) \tag{3.7}$$

The ARIMA model is defined as:

$$\phi_i(B_i)(1 - B_i)^{d_i}\varepsilon_i^t = \theta_i(B_i)\eta_i^t \tag{3.8}$$

where for each i profit:

- ε_i^t is the residual at time t.
- η_i^t is a white noise error term (innovation).
- B_i is the backshift operator, such that $B\varepsilon_i^t = \varepsilon_i^{t-1}$.

- $\phi_i(B) = 1 - \phi_{i1}B - \phi_{i2}B^2 - \cdots - \phi_{ip}B^p$ is the autoregressive (AR) polynomial.
- $\theta_i(B) = 1 + \theta_{i1}B + \theta_{i2}B^2 + \cdots + \theta_{iq}B^q$ is the moving average (MA) polynomial.
- d_i is the order of differencing.

Windowed Models

And for capturing possible memory of the companies behavior in different aspects, we also implemented windowed models that used the last N scores statistical properties to predict future profits.

So for each score j, we will calculate the average and the standard deviation of the scores over the last N periods. Let S_j^t be the score at time t, then for a window of size N:

$$\bar{S}_j^N = \frac{1}{N} \sum_{i=t-N+1}^{t} S_j^i \tag{3.9}$$

where:

- $\bar{S}_j^N$ is the average score over window N for score j
- N can be 3 months, 6 months, 1 year, or 2 years.
- t is the current time point

For adding also the deviation from the average, we also calculated the standard deviation of the scores over the same window:

$$\sigma_j^N = \sqrt{\frac{1}{N} \sum_{i=t-N+1}^{t} (S_j^i - \bar{S}_j^N)^2} \tag{3.10}$$

These statistical measures will be used as additional features in our predictive models to capture the temporal behavior of each score.

3.4.3 Neural Networks

Finally, for trying to capture the non-linear relationships between all the available variables, including the windowed statistical properties, we will use the Neural Networks.

Chapter 4

Development and Results

4.1 Creation of the Dataset

At the current moment of beginning this work, Tweenvest kept a large DB of most of the needed information for creating the dataset. This included all of the price histories, historical currency multipliers to dollar, dividends payed per stock... but when it came to the scores we had a traceability problem.

For optimizing the costs and structure of the DB, Tweenvest chose to only save the latest score for each company and overwriting it each day when calculating the newest one, which led to the first main task.

4.1.1 Updating Tweenvest Code

After locating where the score calculators are called, we have to understand the code's architecture and the relationships between the different database models to see exactly what needs to be changed. Tweenvest uses Django as the main backend's technology to handle a relational database with PostgreSQL, this is due to the nature of the financial data where one the information is related to another.

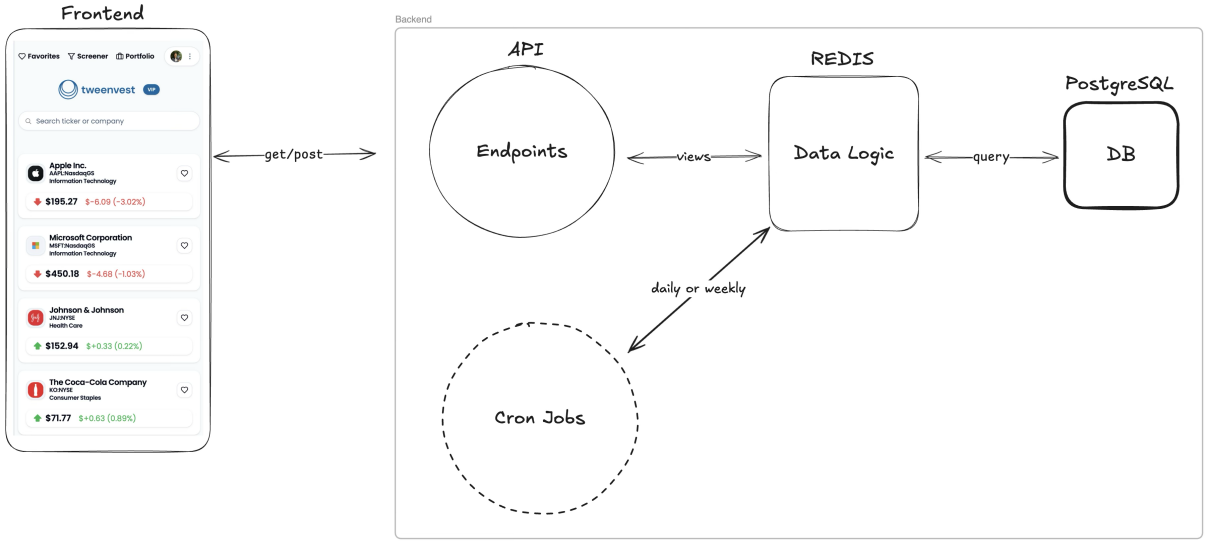

Figure 4.1: General Tweenvest's Architecture

As we can see, there are two different actions occurring at the same time. For one side we have all of the requests coming from the users interactions, and for the other hand the programmed

jobs that need to be executed every day to update all of the financial information of the stocks, or necessary actions such as sending emails to get authentication pins.

Since we are changing the data logic for calculating and storing historical data of the factor scores, we need to assure that the systems capabilities don't exceed the platforms needs, and that the changes don't affect the normal calculation of the factor scores, so we had to look at the code corresponding to the factor scores.

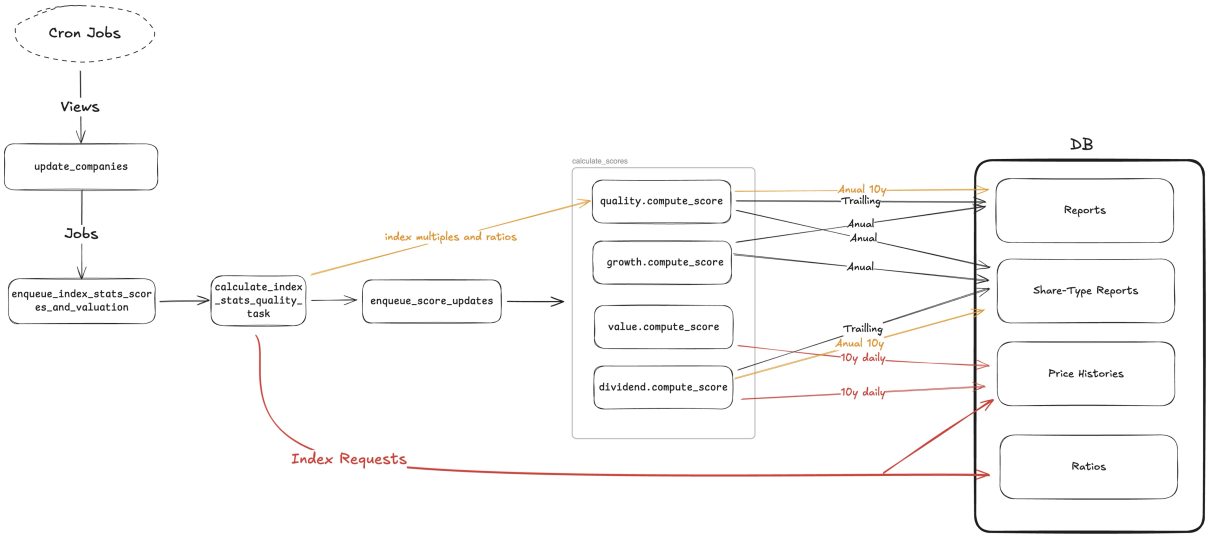

Figure 4.2: Scores Calculations Code Schema

With this schema we have simplified the logic behind several functions related to the calculations of each factor score. Quickly we realized that we had to modify the queries to include an extra filter, *calculation_date* to get models related to the score at any time, this way we can set it as "today" whenever we are just calculating the current ones.

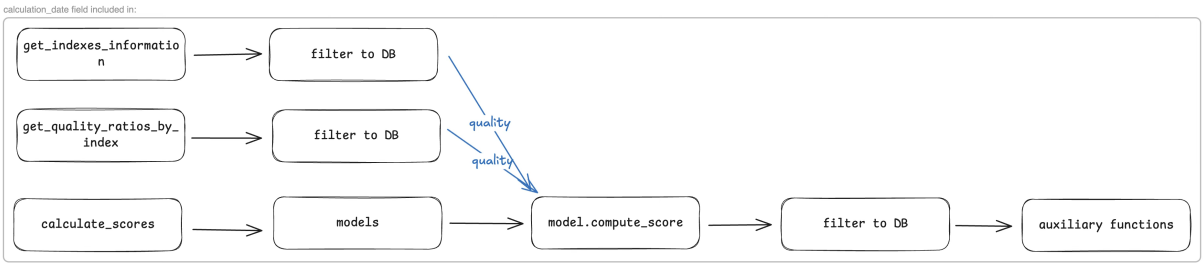

Figure 4.3: *calculation_date* Field Propagation Schema

Following the propagation schema in Figure 4.3, the root change comes from the functions *calculate_scores*, *get_indexes_information* and *get_quality_ratios_by_index*. These are the ones used by the production cron jobs to calculate the factor scores daily, so for a normal function call we set the *calculation_date* to None and whenever we want to calculate the scores for a specific date we set it to the desired date.

This helped us create very readable logic to adapt the filters to both scenarios, for example in the value score shown in Listing 4.1:

```
1 if calculation_date is None:
```

```
    calculation_date = date.today()
    filters = {
        "stock": stock,
        "date__gte": calculation_date - relativedelta(years=10),
    }
else:
    filters = {
        "stock": stock,
        "date__gte": calculation_date - relativedelta(years=10),
        "date__lte": calculation_date,
    }
phs = (
    PriceHistory.objects.filter(**filters)
)
```

Listing 4.1: Value Score Filtering

So as we can observe, the main change for this part of the code was to add a date filter to get only data "lower than or equal" to the *calculation_date*. This may seem a bad approximation, but it has to be done this way due to the nature of the data: most of the financial data come from annual, semestral, and quarterly reports, except to the price history that is updated daily.

Along the way of analyzing the code's structure we made some adjustments to simplify the legibility:

- Renaming functions whenever they are internal methods or generic auxiliary methods called by different functions.
- Externalizing mocking functions that help create fake models and data for unit testing.
- Homogenize the code's structure, field types, to help with readability and keeping a standard order along the backend and frontend.

Another very important part of these updating procedure was to adapt the current unit tests to make sure that the original code still behaved as expected, and creating new ones to check that the changes calculated the factor scores properly.

4.1.2 Designing new Jobs

Once all tests have passed and the team had approved the changes, the next step was to create scripts —from now on, *jobs*—to populate the current DB with the historic factor scores while maintaining the servers performance in normal levels. For this, we had to update a basic method, *bulk_create_or_update*, used many times through the whole codebase and add a new conditioning so it can handle conflicts depending if the models are empty or not. After reading the official Django Documentation [4], we saw that in newest versions they had included new fields to its original method *bulk_create*, so we had to update the Django version in our Docker machine and make sure this didn't introduce any unwanted issues.

After this was implemented, we decided to create three different jobs for enqueueing the tasks and being able to separate the calculations. This way we could restart them in case something broke during the process (see Appendix 7.1). Also, we had to manage the server resources, so for assuring the platform's performance, we used the following hierarchy (see Listing **??**):

1. Seven workers for primary daily jobs from different API data integration

2. One dedicated worker for guaranteeing email sendings, and 6 prioritized shared workers

3. Four shared workers with lower priority, except for 1 which is set higher than email.

A crucial part of this code was replicability, equal distribution through different sectors, and mostly to assure that the companies existed during enough time for the factor scores to calculate the 10 years averages.

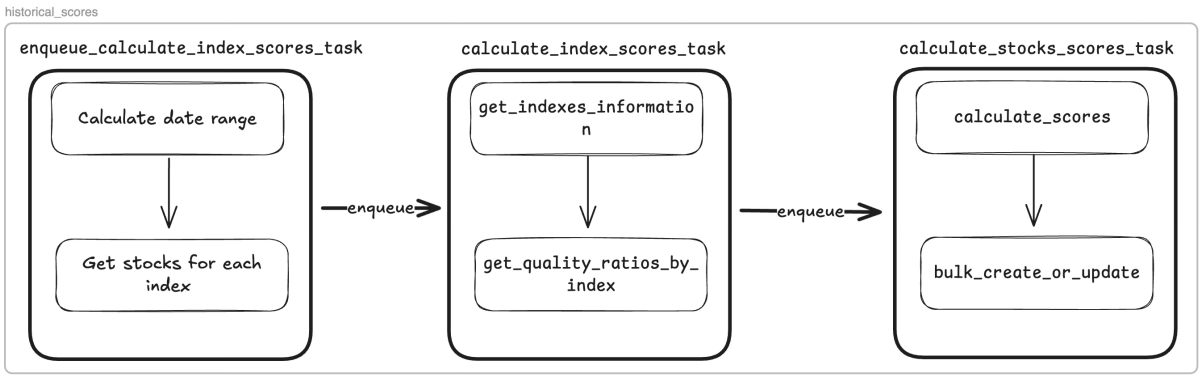

Figure 4.4: Historical Jobs First Iteration Schema

We created the first jobs with the schema shown in Figure 4.4, and during the first small test it worked in local, but when tested with the production DB we started to see important issues with how the functions where designed due to the large amount of data being processed:

- Passing incorrect arguments between jobs—such as entire lists of stock objects—led to the REDIS database reaching its maximum storage capacity.

- Jobs ran out of time.

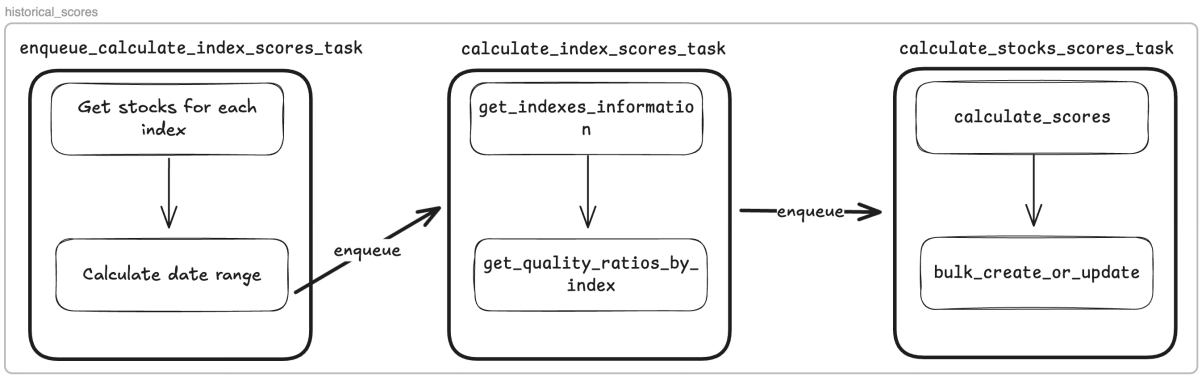

Figure 4.5: Historical Jobs Final Version Schema

We then changed the day range logic to enqueue a *calculate_index_scores_task* for each day, so the job didn't have to calculate all of the index data for the dates at all at once, as shown in Figure 4.5. Also, we modified the function args to only uses models identifiers "id" to reduce the REDIS memory used. The complete implementation of this job can be found in Appendix 7.1.

4.1.3 Telemetry Tracing

Finally we had the functional jobs for calculating the scores, and ran the first calculation for a dataset of 1,500 stocks. After the first jobs finished we quickly realized that they were taking too much time to process, and saw a disproportionate amount of queries to the DB so we started a process to check on where was the bottleneck using telemetry tracing tools.

To do so, we had to inspect each method and function that was being used through the calculations of the factor scores. Luckily we had just recently implemented *DataDog* which is a software that provides monitoring and analytics for applications and infrastructure, offering real-time metrics, event monitoring, and end-to-end tracing.

To apply the tracer, we simply had to add a specific decorator before each method and set up the environment for being able to trace not only production launched jobs, but also development tests.

Here's an example of how the tracer was implemented in the quality score calculation:

```python
from opentelemetry import trace

tracer = trace.get_tracer(__name__)

@tracer.start_as_current_span("compute_quality_score")
def compute_quality_score(
    stock: "Stock",
    index_quality_ratios: Dict[str, IndexRatio],
    calculation_date: date | None,
) -> tuple[dict, dict]:
    ...
    return final_score
```

Listing 4.2: Telemetry Tracing Implementation

We then looked at the *DataDog* dashboard and clearly saw an excess of queries to the DB when calculating each score, and tried to reduce them by looking for a simple fault in the logic of the queries. Similar issues had been resolved before by adding a *select_related* in the queries. To clarify what this does, here is the definition with a simple use-case:

"Returns a QuerySet that will follow foreign-key relationships, selecting additional related-object data when it executes its query. This is a performance booster which results in a single more complex query but means later use of foreign-key relationships won't require database queries." – Documentation [5].

In Django, *select_related()* and *prefetch_related()* are designed to stop the deluge of database queries that are caused by accessing related objects. *select_related()* "follows" foreign-key relationships, selecting additional related-object data when it executes its query. *prefetch_related()* does a separate lookup for each relationship and does the "joining" in Python. So one uses *select_related* when the object that you're going to be selecting is a single object, so *OneToOneField* or a *ForeignKey*.

But after many attempts, and due to the lack of time left to just calculate the factor scores, we had to leave this "minor issue" unsolved and change the strategy to fix the calculation time problem. Which led to the realization that we were sharing most of the servers power with the normal usage of the platform, and for the normal calculations times (3-4h) didn't matter

because Tweenvest do them daily after market hours, but that didn't work for calculating 5 or more years in a reasonable time, even when calculating only one score per month.

4.1.4 Dedicated Server Deployment

We made the decision to establish a dedicated server for this thesis, equipped with enhanced computational power to facilitate the calculations and data fitting. After thorough research, we opted for Hetzner servers due to their exceptional quality-price ratio, we ended up with a dedicated server with the following characteristics:

VCPUS	RAM	SSD Storage	Extra Volume	Bandwidth	Location
16	32 GB	320 GB	480 GB	20TB	Germany

Table 4.1: Hetzner Server Specifications

Deploying the project in a scalable and reproducible environment involved several structured steps, encompassing server provisioning, secure remote access configuration using SSH, and initializing the containerized application deployment via Docker, with the application codebase managed through GitHub.

After all of the setup was done, we saw an amazing **x100 performance improvement**, going from 15-minute calculation times for index statistics to 8 seconds. Finally, we could launch the factor scores historical calculations, so we created two subsets of 1,500 random companies each with the factor scores calculated for the 1st day of the month for the following periods: 2015-2020 and 2020-2025, which took a total of 3 hours.

Now that we have filled the DB with the necessary factor scores for the study, we ran a big calculation to get all of the possible factor scores for the +100,000 companies, which took several days.

4.1.5 Aggregating the data

For being able to continue, we needed to create the final dataset with all the necessary data for later on doing the analysis and developing the predictive models, so we created a simple algorithm with this logic:

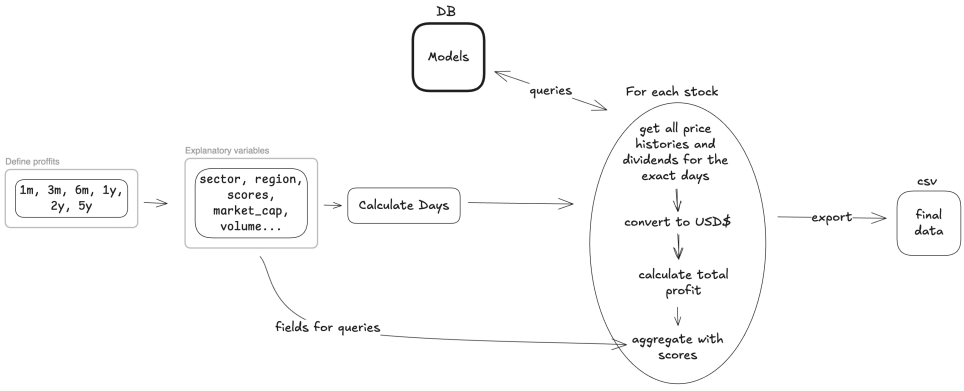

Figure 4.6: First Data Aggregation Schema

Once we had the final csv file we noticed that we had too many empty values for profits, the problem was because some of the days calculated were empty for a specific company, so there was no price history for them. Since the companies belong to multiple countries, the different

holidays could be causing another major loss in the information. To fix this in a general way we implemented an internal method for estimating price histories if it didn't exist for the wanted date. The complete implementation of this data export and price estimation process can be found in Appendix 7.3.

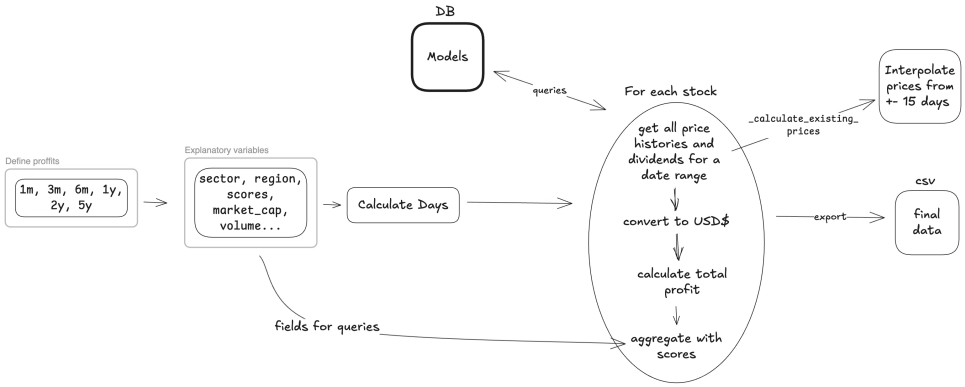

Figure 4.7: Final Data Aggregation Schema

This can be done due to the fact that we are only looking for long term profitabilities, so we can **approximate the price** on a day x by interpolating the price on $x - y$ and $x + z$, weighted by how close they are to the original day x, with a maximum distance from x of 15 days. This significantly reduced the amount of missing values.

4.1.6 Datasets Creation

For this study, we have attacked the main objective from multiple perspectives. First we made a descriptive analysis to understand the data, and then we proposed different models to see if there were any relationships between the scores factors and long term profitabilities. Thats why when creating the datasets, we made two different approaches to have more information for the analysis:

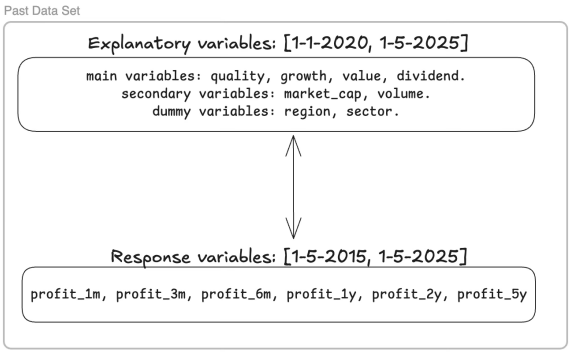

Figure 4.8: Past Dataset

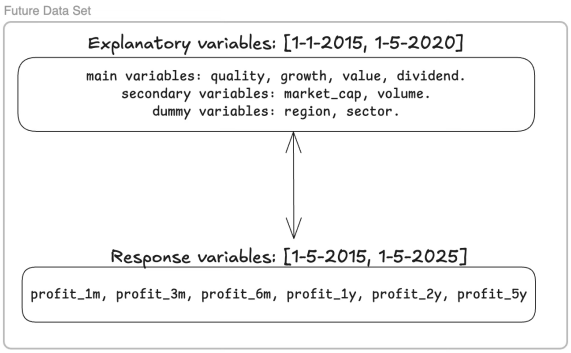

Figure 4.9: Future Dataset

For the past dataset, we calculated the scores for the range 2020-2025 and then related them to the profits obtained until the specified date. For example, the date 1-1-2021 would be linked to the 2 years' profitability obtained until that date:

$$Profit_{\text{1-1-2021}}(\%) = \frac{P_{\text{1-1-2021}} - P_{\text{1-1-2019}} + D_{\text{acc}}}{P_{\text{1-1-2019}}} \tag{4.1}$$

Which interprets to how earlier profits have influenced on the explanatory variables. This will give us valuable information later on, but for the main analysis we actually want the opposite.

So we created the future dataset, we calculated the scores for the range 2015-2020 and then related them to the profits obtained until the specified date. For example, the date 1-1-2021 would be linked to the 2 years' profitability that will be obtained in the future:

$$Profit_{\text{1-1-2021}}(\%) = \frac{P_{\text{1-1-2023}} - P_{\text{1-1-2021}} + D_{\text{acc}}}{P_{\text{1-1-2021}}} \tag{4.2}$$

As this approach is what we actually want to use for the predictive models, we will be using the future dataset structure for most of the analysis.

4.2 Descriptive Analysis

As a summary to know how many datasets we will be using through the study, we can use the following table:

Dataset	Raw	Cleaned	No Outliers
Small Data Past	83,243	95,000	5,000
Big Data Past	16,119,852	95,000	5,000
Small Data Future	65,166	95,000	5,000
Big Data Future	100,000	95,000	5,000
Small Data Future USA	60,503	380,000	20,000
Small Data Future EU	62,725	380,000	20,000
Small Data Future AS	65,772	380,000	20,000
Small Data Future LAT	48,412	380,000	20,000
Small Data Future AF	58,341	380,000	20,000

Table 4.2: Data Points per Dataset

4.2.1 Preprocessing the Data

Even with all of these protocols to create the most reliable and filled datasets, we still had to follow the methodologies explained in the previous chapter to analyze the data.

To begin, we looked at the data structure and saw that **about 50% of growth scores were empty**. This turned on many alerts from problems with the algorithm, because compared to the rest of the variables there was a significant difference, since the **normal absence was only around 8%**. But after digging into the data we realized that it was due to the fact that many companies stopped sending reports or selling but keep existing, so we decided to deleted these companies from our dataset because they weren't behaving as a "normal company", and this could create a bias in the models. Additionally, as noted earlier, we replaced all NaN values in the dividend scores with 0, since Tweenvest's algorithm omits a score when a company does not pay dividends.

Once we did those small adjustments on the dataset, we ended up with the following distributions:

The complete implementation of this preprocessing process can be found in Appendix 7.9.

Chapter 5

Discussion

Chapter 6

Conclusion

Bibliography

[1] A. A. Abro, E. Taşci, and A. Uğur. "A Stacking-based Ensemble Learning Method for Outlier Detection". In: *Balkan Journal of Electrical & Computer Engineering* 8.2 (Apr. 2020). URL: https://dergipark.org.tr/en/download/article-file/1106223.

[2] Markus M. Breunig et al. "LOF: identifying density-based local outliers". In: *ACM sigmod record*. 2000. URL: https://dl.acm.org/doi/pdf/10.1145/335191.335388.

[3] scikit-learn developers. *OneClassSVM*. scikit-learn Documentation. URL: https://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html. 2025.

[4] Official Documentation. *bulk_create Method*. Django Documentation. URL: https://docs.djangoproject.com/en/5.2/ref/models/querysets/#bulk-create. 2025.

[5] Official Documentation. *select_related Method*. Django Documentation. URL: https://docs.djangoproject.com/en/5.2/ref/models/querysets/#django.db.models.query.QuerySet.select_related. 2025.

[6] P. Dorsey and J. Mansueto. *The Five Rules for Successful Stock Investing*. 2011.

[7] John D. Hunter et al. *Matplotlib: Python plotting package*. Online documentation. URL: https://matplotlib.org/stable/. 2025.

[8] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. "Isolation-based anomaly detection". In: *ACM Transactions on Knowledge Discovery from Data (TKDD)* 6.1 (2012), p. 3. URL: https://www.lamda.nju.edu.cn/publication/tkdd11.pdf.

[9] MSCI. "Fundamental Data Methodology". In: (June 2025). URL: https://www-cdn.msci.com/documents/10199/0b41b3a3-9da8-46f6-fce7-1dc1177e4a23.

[10] Daniel Servén and Charlie Brummitt. *Generalized Additive Models*. pyGAM Documentation. URL: https://pygam.readthedocs.io/en/latest/notebooks/tour_of_pygam.html. 2018.

[11] The pandas development team. *pandas Documentation*. pandas Documentation. URL: https://pandas.pydata.org/. 2024.

[12] Michael L. Waskom. *seaborn: statistical data visualization*. Seaborn Documentation. URL: https://seaborn.pydata.org/index.html. 2024.

[13] Noumir Zineb, Paul Honeine, and Cedue Richard. "On simple one-class classification methods". In: *IEEE International Symposium on Information Theory Proceedings* (2012). URL: https://www.researchgate.net/publication/261248728_On_simple_one-class_classification_methods.

Part II

Appendices

Appendix 7

Code Listings

7.1 Historical Scores Job Implementation

The following code snippets shows the complete implementation of the historical scores job that was used to populate the database with historical factor scores data:

7.1.1 Enqueueing Historical Scores Job

```python
@job("historical_scores", timeout="3h")
def enqueue_calculate_index_scores_task(
    start_date: datetime.date,
    end_date: datetime.date,
    index_names: list[str] | None = None,
    region_names: list[str] | None = None,
    total_stocks: int | None = None,
    seed: float | None = None,
    daily: bool = False,
):
    logger.info(
        "[H1] Starting enqueue_year_tasks from %s to %s",
        start_date,
        end_date,
    )

    days = get_range_from_dates(
        start_date,
        end_date,
        delta=relativedelta(days=1) if daily else relativedelta(
            months=1),
    )
    days.reverse()

    if not index_names:
        index_names = list(
            Index.objects.exclude(name__in=[Index.INDEX_ALL, "Other"
                ])
            .only("name")
            .values_list("name", flat=True)
        )
    logger.info("[H1] Enqueuing tasks for %s indexes", len(index_
        names))

    stocks_per_index = (
```

```
33          int(total_stocks / len(index_names)) if total_stocks else
                None
34      )
35
36      stock_count = 0
37      total_stocks_dict = {}
38      for index_name in index_names:
39          stock_ids = get_stock_ids_for_index(
40              index_name,
41              stocks_per_index,
42              start_date,
43              end_date,
44              seed,
45              region_names,
46          )
47          if not stock_ids:
48              raise ValueError(
49                  f"[H1] No stocks found for index '{index_name}' after
                        {start_date}"
50              )
51          total_stocks_dict[index_name] = stock_ids
52          for day in days:
53              calculate_index_scores_task.delay(
54                  index_name=index_name,
55                  day=day,
56                  stock_ids=stock_ids,
57              )
58
59          stock_count += len(stock_ids)
60
61      logger.info(
62          "[H1] Enqueued days %s for stocks:%s .", len(days), stock_
                count
63      )
64
65      return {
66          "stocks_processed": stock_count,
67          "days": len(days),
68          "total_stocks": total_stocks_dict,
69      }
```

Listing 7.1: Enqueueing Historical Scores Job

This job function is responsible for:

- Accepting date ranges and configuration parameters for historical score calculation
- Generating a list of dates to process (either daily or monthly intervals)
- Retrieving stock IDs for each index based on the specified criteria
- Enqueuing individual calculation tasks for each day and index combination
- Providing detailed logging and return statistics about the processing

7.1.2 Calculate Index Scores Data

```python

@job("historical_scores", timeout="45m")
def calculate_index_scores_task(
    index_name: str,
    day: datetime.date,
    stock_ids: list[int],
):
    start_time = time.perf_counter()
    logger.info(
        "[H2] Calculating index",
        extra={
            "index": index_name,
            "day": day,
            "stocks": len(stock_ids),
        },
    )

    # ---------------- Index calculation ----------------- #
    index_stats = get_indexes_information(
        index_names=[index_name], calculation_date=day
    )
    index_quality = get_quality_ratios_by_index(
        index_names=[index_name], calculation_date=day
    )
    stats = index_stats.get(index_name)
    quality = index_quality.get(index_name)

    # ------------------------------------------------------ #
    if (
        not stats
        or not quality
        or all(value is None for value in quality.values())
        or all(
            value is np.nan
            for stat in stats.values()
            for value in stat.values()
        )
    ):
        raise RuntimeError(
            f"[H2] Missing index data for {index_name} on {day},
               stats: {stats}, quality: {quality}"
        )

    calculate_stocks_scores_task.delay(
        day=day,
        stats=stats,
        quality=quality,
        stock_ids=stock_ids,
    )
    return {
        "index_calculated": index_name,
        "total_time": time.perf_counter() - start_time,
    }
```

Listing 7.2: Calculate Index Scores Data

This job function is responsible for:

- Calculating all the necessary index stats data
- Enqueuing the calculation of the stocks scores for the given index and date
- Providing detailed logging and return statistics about the processing

7.1.3 Calculate Stocks Scores Task

```python
@tracer.start_as_current_span("calculate_stocks_scores_task")
@job("historical_scores", timeout="4h")
def calculate_stocks_scores_task(
    day: datetime.date,
    stats: dict,
    quality: dict,
    stock_ids: list[int],
):
    """
    Job 3: For each stock, calculate scores and bulk save.
    """
    start_time = time.perf_counter()
    logger.info(
        "[H3] Calculating scores",
        extra={"day": day, "stocks": len(stock_ids)},
    )

    if not stock_ids:
        return {
            "stocks_processed": 0,
        }
    # Select related for trying to avoid multiple queries to the DB
    stocks = Stock.objects.filter(id__in=stock_ids).select_related(
        "share_type",
        "share_type__company",
        "share_type__company__industry",
        "share_type__company__industry__industry_group",
        "share_type__company__industry__industry_group__sector",
    )
    factor_scores = []
    for stock in stocks:
        score = calculate_scores(
            stock,
            stats,
            quality,
            calculation_date=day,
            bulk_create_or_update=True,
        )
        if not score:
            logger.warning(
                "[H3] No score for calculated",
                extra={"stock_id": stock.pk, "day": day, "score":
                    score},
            )
            continue  # Not as important as the others, need to check
                errors in DataDog
        factor_scores.append(score)

    bulk_create_or_update(
        model=FactorScore,
```

```
49          object_list=factor_scores,
50          unique_fields=["stock", "date"],
51          update_fields=["quality", "growth", "value", "dividend"],
52      )
53
54      return {
55          "stocks_processed": len(stock_ids),
56          "total_time": time.perf_counter() - start_time,
57          "time_per_stock": (time.perf_counter() - start_time) / len(
                stock_ids),
58      }
```

Listing 7.3: Calculate Stocks Scores Task

This job function is responsible for:

- Calculating the factor scores for each stock in the given index and date
- Bulk saving the factor scores for the given index and date
- Providing detailed logging and return statistics about the processing

7.2 Workers Management

In this section we have 2 different configurations for the workers management, since we had to switch servers.

7.2.1 Tweenvest's Production Server

```
1 [program:worker1]
2 command=python manage.py rqworker internal_tasks ciq_priority fmp_
      priority ciq indices_partial indices_full historical_scores fmp
      emails --with-scheduler
3 directory=/app
4 autostart=true
5 autorestart=true
6 redirect_stderr=true
7 stdout_logfile=/dev/fd/1
8 stdout_logfile_maxbytes=0
9 environment = SERVICE_NAME="tweenvest-worker"
10
11 [program:worker2]
12 command=python manage.py rqworker internal_tasks ciq_priority fmp_
      priority ciq indices_partial indices_full historical_scores fmp
      emails --with-scheduler
13 autostart=true
14 directory=/app
15 autorestart=true
16 redirect_stderr=true
17 stdout_logfile=/dev/fd/1
18 stdout_logfile_maxbytes=0
19 environment = SERVICE_NAME="tweenvest-worker"
20
21 [program:worker3]
22 command=python manage.py rqworker internal_tasks ciq_priority fmp_
      priority ciq indices_partial fmp emails historical_scores --with-
      scheduler
```

```
23 autostart=true
24 directory=/app
25 autorestart=true
26 redirect_stderr=true
27 stdout_logfile=/dev/fd/1
28 stdout_logfile_maxbytes=0
29 environment = SERVICE_NAME="tweenvest-worker"
30
31 [program:worker4]
32 command=python manage.py rqworker ciq_priority fmp_priority ciq
     indices_partial fmp emails historical_scores --with-scheduler
33 autostart=true
34 directory=/app
35 autorestart=true
36 redirect_stderr=true
37 stdout_logfile=/dev/fd/1
38 stdout_logfile_maxbytes=0
39 environment = SERVICE_NAME="tweenvest-worker"
40
41 [program:worker5]
42 command=python manage.py rqworker fmp_priority ciq_priority ciq fmp
     emails --with-scheduler
43 autostart=true
44 directory=/app
45 autorestart=true
46 redirect_stderr=true
47 stdout_logfile=/dev/fd/1
48 stdout_logfile_maxbytes=0
49 environment = SERVICE_NAME="tweenvest-worker"
50
51 [program:worker6]
52 command=python manage.py rqworker emails --with-scheduler
53 autostart=true
54 directory=/app
55 autorestart=true
56 redirect_stderr=true
57 stdout_logfile=/dev/fd/1
58 stdout_logfile_maxbytes=0
59 environment = SERVICE_NAME="tweenvest-worker"
60
61 [program:worker7]
62 command=python manage.py rqworker ciq_priority ciq emails summaries
     --with-scheduler
63 directory=/app
64 autostart=true
65 autorestart=true
66 redirect_stderr=true
67 stdout_logfile=/dev/fd/1
68 stdout_logfile_maxbytes=0
69 environment = SERVICE_NAME="tweenvest-worker"
70
71 [program:worker8]
72 command=python manage.py rqworker ciq_priority ciq emails summaries
     --with-scheduler
73 directory=/app
74 autostart=true
75 autorestart=true
76 redirect_stderr=true
```

```
77 stdout_logfile=/dev/fd/1
78 stdout_logfile_maxbytes=0
79 environment = SERVICE_NAME="tweenvest-worker"
80
81 [program:worker9]
82 command=python manage.py rqworker historical_scores ciq_priority
       summaries ciq emails --with-scheduler
83 directory=/app
84 autostart=true
85 autorestart=true
86 redirect_stderr=true
87 stdout_logfile=/dev/fd/1
88 stdout_logfile_maxbytes=0
89 environment = SERVICE_NAME="tweenvest-worker"
90
91 [program:worker10]
92 command=python manage.py rqworker summaries ciq_priority fmp_priority
       ciq fmp emails --with-scheduler
93 directory=/app
94 autostart=true
95 autorestart=true
96 redirect_stderr=true
97 stdout_logfile=/dev/fd/1
98 stdout_logfile_maxbytes=0
99 environment = SERVICE_NAME="tweenvest-worker"
```

Listing 7.4: Production Workers Management

7.2.2 Personal Development Server

```
1 [program:worker1]
2 command=python manage.py rqworker historical_scores
3 process_name=%(program_name)s_%(process_num)02d
4 numprocs=16
5 directory=/app
6 autostart=true
7 autorestart=true
8 redirect_stderr=true
9 stdout_logfile=/dev/fd/1
10 stdout_logfile_maxbytes=0
11 environment = SERVICE_NAME="tweenvest-worker"
```

Listing 7.5: Hetzner Workers Management

7.3 Data Export Job Implementation

The following code shows the complete implementation of the data export job that was used to create the final dataset, including the price estimation for missing values due to weekends and holidays:

```
1 def export_factors_and_pricehistory_task(
2     stocks_id: list[int],
3     start_date: datetime.date,
4     end_date: datetime.date,
5     export_name: str,
```

```
    daily: bool = False,
):
    """
    Export FactorScore, PriceHistory y rentabilidades a CSV sin cargar todo en memoria.
    """
    start_time = time.perf_counter()
    print("Starting CSV export...")

    periods = {
        "profit_1m": relativedelta(months=1),
        "profit_3m": relativedelta(months=3),
        "profit_6m": relativedelta(months=6),
        "profit_1y": relativedelta(years=1),
        "profit_2y": relativedelta(years=2),
        "profit_5y": relativedelta(years=5),
    }

    factor_fields = [
        "stock__ticker",
        "stock__share_type__company__name",
        "stock__share_type__company__industry__industry_group__sector__name",
        "stock__share_type__company__country__region",
        "date",
        "quality",
        "growth",
        "value",
        "dividend",
    ]
    price_fields = ["market_cap_usd", "volume"]

    days = get_range_from_dates(
        start_date,
        end_date,
        delta=relativedelta(days=1) if daily else relativedelta(months=1),
    )
    days.reverse()

    profitability_fields = list(periods.keys())
    export_fields = factor_fields + price_fields + profitability_fields
    missing_prices = 0
    missing_factors = 0
    fs_path = f"api/factors/data_exports/{export_name}.csv"
    with open(fs_path, mode="w", newline="", encoding="utf-8") as f:
        writer = csv.DictWriter(f, fieldnames=export_fields)
        writer.writeheader()

        for stock_id in tqdm(stocks_id, desc="Stocks"):

            price_qs = PriceHistory.objects.filter(
                stock__id=stock_id,
                date__range=(
                    start_date,
                    end_date + relativedelta(years=5),
                ),
```

```
60              ).values("date", "close_price", "market_cap", "volume", "
                    fx_mult")
61              price_lookup_stock = {
62                  (stock_id, ph["date"]): ph
63                  for ph in price_qs
64                  if ph["fx_mult"] is not None
65              }
66
67              dividend_qs_stock = DividendHistory.objects.filter(
68                  stock__id=stock_id,
69                  ex_dividend_date__range=(start_date, end_date),
70              ).values("ex_dividend_date", "adjusted_dividend", "fx_
                    mult")
71              dividend_lookup = {
72                  (stock_id, dh["ex_dividend_date"]): (
73                      float(dh["adjusted_dividend"]) * float(dh["fx_
                            mult"])
74                  )
75                  for dh in dividend_qs_stock
76                  if dh["fx_mult"] is not None
77              }
78              fs_qs_stock = (
79                  FactorScore.objects.filter(stock_id=stock_id, date__
                        in=days)
80                  .select_related(
81                      "stock__share_type__company__industry__industry_
                            group__sector",
82                      "stock__share_type__company__country",
83                  )
84                  .values(*factor_fields)
85              )
86
87              fs_lookup = {(stock_id, fs["date"]): fs for fs in fs_qs_
                    stock}
88
89              for day in days:
90                  factor = fs_lookup.get((stock_id, day))
91                  if not factor:
92                      missing_factors += 1
93                      continue
94                  day = factor["date"]
95                  price_day = day
96                  while price_day.weekday() >= 5:
97                      price_day -= datetime.timedelta(days=1)
98
99                  price = price_lookup_stock.get((stock_id, price_day))
100                 if not price or price["close_price"] is None:
101                     price = _calculate_existing_prices(
102                         price_day, price_lookup_stock, stock_id
103                     )
104                     if not price:
105                         missing_prices += 1
106                         continue
107
108                 try:
109                     market_cap_usd = (
110                         float(price.get("market_cap", 0) or 0)
111                         * price["fx_mult"]
```

```
                    )
                except Exception:
                    market_cap_usd = None
                try:
                    volume = (
                        float(price.get("volume", 0) or 0) * price["
                            fx_mult"]
                    )
                except Exception:
                    volume = None

                if price["close_price"] is None or price["fx_mult"]
                    is None:
                    continue
                close_now_usd = float(price["close_price"]) * price["
                    fx_mult"]

                profitabilities = {}
                for field, delta in periods.items():
                    future_date = day + delta
                    while future_date.weekday() >= 5:
                        future_date -= datetime.timedelta(days=1)

                    future_price = price_lookup_stock.get(
                        (stock_id, future_date)
                    )
                    if not future_price or future_price["close_price"
                        ] is None:
                        future_price = _calculate_existing_prices(
                            future_date, price_lookup_stock, stock_id
                        )
                        if not future_price:
                            profitabilities[field] = None
                            missing_prices += 1
                            continue

                    close_future_usd = float(
                        future_price["close_price"]
                    ) * float(future_price["fx_mult"])
                    if close_future_usd == 0:
                        profitabilities[field] = None
                        missing_prices += 1
                        continue
                    # Dividend sum
                    dividend_sum = sum(
                        v
                        for (s_id, ex_div_date), v in dividend_lookup
                            .items()
                        if s_id == stock_id
                        and day <= ex_div_date < future_date
                    )
                    profitabilities[field] = (
                        close_future_usd + dividend_sum - close_now_
                            usd
                    ) / close_now_usd

                row = {**factor}
                row["market_cap_usd"] = market_cap_usd
```

```
                row["volume"] = volume
                row.update(profitabilities)
                writer.writerow(row)

    elapsed = time.perf_counter() - start_time
    print(f"Exportado a {fs_path} en {elapsed:.2f}s")
    print("MISSING PRICES:", missing_prices)
    print("MISSING FACTOR SCORES:", missing_factors)
```

Listing 7.6: Data Export Job Implementation

This job function is responsible for:

- Exporting factor scores, price history and profitability data to CSV without loading everything into memory
- Handling missing price data due to weekends and holidays by estimating prices from nearby dates
- Converting all monetary values to USD using appropriate exchange rates
- Calculating profitability for different time periods (1m, 3m, 6m, 1y, 2y, 5y)
- Including dividend payments in profitability calculations
- Tracking and reporting missing data points for both prices and factor scores

7.4 Custom Python Functions for Data Analysis

```
def correlation_matrix(
    df: pd.DataFrame, columns: list[str], file_path: str | None =
        None
) -> None:
    """
    Plot a correlation matrix for the DataFrame showing only the
        lower triangle.
    """
    df_columns = df[columns]
    corr_matrix = df_columns.corr()

    # Create a mask for the lower triangle
    mask = np.triu(np.ones_like(corr_matrix, dtype=bool))

    # Plot using seaborn for better visualization
    plt.figure(figsize=(12, 8))
    sns.heatmap(
        corr_matrix,
        mask=mask,
        annot=True,
        cmap="coolwarm",
        center=0,
        fmt=".2f",
        square=True,
    )
    plt.title("Correlation Matrix of Numeric Variables")
    plt.tight_layout()

```

```
    if file_path:
        plt.savefig(file_path)
        plt.show()
    else:
        plt.show()
```

Listing 7.7: Correlation Matrix Plot Function

```
def pairplot(
    df: pd.DataFrame,
    file_path: str | None = None,
    kde: bool = False,
    hue: str = "region",
    hue_order: list[str] | None = None,
    palette: dict[str, str] | None = None,
) -> None:

    if len(df) > 10000:
        df = df.sample(10000, random_state=42)

    g = sns.pairplot(
        df,
        hue=hue,
        diag_kind="kde",
        hue_order=hue_order,
        palette=palette,
        corner=True,
    )
    if kde:
        g.map_lower(sns.kdeplot, levels=4, color=".2")

    if file_path:
        plt.savefig(file_path)
        plt.close()
    else:
        plt.show()
```

Listing 7.8: Custom Pairplot Function

7.5 Data Preprocessing

7.5.1 Data Cleaning

```
    import sys
    from pathlib import Path

    # Add the parent directory (code/) to sys.path
    sys.path.append(str(Path().resolve().parent))
    from utils import load_data, plot_numeric_distributions
    import pandas as pd

    # %% [markdown]
    # # Loading the data

    # %%
```

```
13  file_name = "Small Data future AF"
14  data_path = f"../data/raw/{file_name}.csv"
15
16  # Read the CSV file
17  df = load_data(data_path)
18
19  # %% [markdown]
20  # ## Cleaning and renaming
21
22  # %%
23  df = df.rename(
24      columns={
25          "stock__share_type__company__industry__industry_group__
                sector__name": (
26              "sector"
27          ),
28          "stock__share_type__company__country__region": "region",
29          "market_cap_usd": "market_cap",
30          "stock__share_type__company__name": "company",
31          "stock__ticker": "ticker",
32      }
33  )
34
35  df["dividend"] = df["dividend"].fillna(0.0)
36  df["date"] = pd.to_datetime(df["date"])
37  df_no_nan = df.dropna()
38
39
40  # %% [markdown]
41  # # Analyzing
42
43  # %%
44  export_path = f"../data/cleaned/basic/plots/{file_name}.png"
45  plot_numeric_distributions(df, file_path=export_path)
46
47  # %% [markdown]
48  # # Export cleaned data
49
50  # %%
51  # Export the cleaned dataset to CSV
52  cleaned_data_path = f"../data/cleaned/basic/{file_name}.csv"
53  df.to_csv(cleaned_data_path, index=False)
54
55  print(f"Cleaned data exported to: {cleaned_data_path}")
```

Listing 7.9: Basic Data Cleaning

7.5.2 Outlier Detection

Listing 7.10: Outlier Detection