Erasmus School of Economics

MSc. Quantitative Finance: Master Thesis

Simulation of Multivariate Financial Time Series Data for Portfolio Optimization

Student:

Laurent Balesse (580755)

Supervisor:

dr. Rasmus Lonn

Company supervisor:

Emiel Lemahieu

Abstract

This thesis researches different models for the generation of multivariate asset return data including a DCC(1,1)-GARCH(1,1) model, a moving block bootstrap model and a Variational Autoencoder. Their performance is evaluated based on univariate path distribution similarity, univariate stylized facts appearance frequency, and drawdown distribution similarity related to InvestSuite Value at Risk portfolio optimization. The models show significant difference in performance when looking at the test results. Furthermore, the Variational Autoencoder is successfully adapted to improve drawdown distribution similarity between sample and generated data. This result shows the strength and flexibility of the use of artificial networks for financial data generation.

August 8, 2021

Contents

1 Introduction

Quantitative research for investment strategies is a rigorous procedure consisting of many intermediate steps. However, once a strategy is developed, one expects a flawless test that shows its performance. Traditionally, such a test, also called a backtest, checks the effectiveness of the strategy by charting the performance in real-world conditions using historical data. Such a test however, gives you only one version of the story. Relying on only one single version of a story out of an infinite amount of possibilities might lead to adverse outcomes such as overfitting of a strategy to that test. Overfitting can make a strategy less suitable for future conditions as these future conditions can diverge from the tested scenario, on which the strategy depends. To avoid these dangers, there is a need for a richer toolkit that can provide quality time series data for backtesting. This thesis provides a basis for this toolkit, and is written in collaboration with InvestSuite, providing practical knowledge and data resources.

The research discusses and compares multiple methods that can be used to simulate realistic multivariate financial return data for portfolio optimization testing purposes. Furthermore, it assesses the quality of these generated financial time series on different levels. Three different models are discussed, covering the spectrum from parametric models to non-parametric models. These include Dynamic Conditional Correlation - General Autoregressive Conditional Heteroskedasticity (DCC-GARCH), Variational Autoencoder (VAE) and block bootstrap. The DCC-GARCH model will advocate for the parametric models, the block bootstrap for the non-parametric models, and the VAE is somewhere in between being an artificial neural network.

To assess the output of our generating models, a qualitative analysis of the time series is done by checking if the stylized facts of historic financial time series appear in the generated data as well. Furthermore, a quantitative analysis comparing the sample data and the simulations is done using normalized mathematical signatures of a time series. Signatures is a data mapping used to capture the characteristics of a time series with a finite dimensional vector. Normalising the moments of the signatures of the time series allows to find the underlying data generating process or law. These laws can be compared. As multivariate data is considered, an analysis is done on the correlation of the considered assets in the simulated data as well. Subsequently, this paper discusses the application of the generated data as backtesting data for InvestSuite Value at Risk (iVaR). IVaR is based on the frequency, magnitude and duration of drawdowns, and we compare the distribution of drawdowns in sample data and generated data to analyse the performance of the generating model in the iVaR environment. Furthermore, the Variational Autoencoder method is further adapted to improve drawdown distribution matching between sample and simulated data.

The sample data that is used is provided through the financial data servers of InvestSuite, where data is used from either Morningstar, Datastream Webservices API (DSWS), or TRKD. The sample data contains 7 years of financial return data (2014-01-01 until 2021-01-01) of the current companies under the EURO STOXX 50 universe, containing 50 blue-chip companies from 8 European countries.

The text starts with a discussion of the literature on financial time series characteristics and financial return data generating models in the literature section. Next, three models are selected and explained in detail together with the different tests in the methodology section. After methodology, the data set that is used, is discussed before moving on to the results section where the results of the different tests are discussed for each model. The paper finishes with a conclusion on the results, and further research possibilities.

The results show the generative performance of all three models by discussing all scenarios of one simulated month and by discussing one time series of all simulated one month scenarios combined. The DCC-GARCH model produces scenarios where the correlation between different assets is overestimated and there is no significant similarity in drawdown distribution, but where the similarity of underlying process for each asset is partly present. When combining the scenarios, four out of six checked stylized facts tend to return regularly for different assets. The block bootstrap returns a correlation more realistic compared to sample correlations, improved similarity in univariate underlying distribution, and significant similarity in drawdown distribution. The combined scenarios show regular appearance of four out of six stylized facts.

While the variational autoencoder returns correlations comparable to sample correlations, and finds similarity in underlying process for almost all assets, the main result is the improvement in similarity of univariate drawdown distribution between sample and simulation by adaptation of the objective function of the Variational Autoencoder. The objective function allows to make the autoencoder focus on a desired characteristic of the data and these results indicate that it is possible to improve set characteristic. The contribution of this research to current literature is threefold. First, an adapted version of the variational autoencoder is provided in order to improve similarity in loss distribution. Second, as far as my knowledge goes, there is currently no comparison made between the considered models when focusing on generating financial return data. Third, a testing suite is provided for assessment of generated financial data. This suite contains a scoring system for presence of stylized facts in a universe of financial data, a test comparing the underlying distribution of return paths, and a test for analyzing the similarity in drawdown distribution of sample paths and generated paths.

2 Literature review

High quality return data is not easy to define nor to reproduce. However, understanding financial time series and its characteristics is very critical, which is why financial time series and their characteristics have been researched for decades by academia and practitioners. This section describes what insights the current literature provides for describing, but also reproducing, asset return time series.

Over more than half a century, academics have been researching the statistical properties of prices of financial assets. Over time, a set of properties common among the return data of financial instruments from different markets has been observed by independent studies. These common properties for prices and returns of stocks, commodities and market indexes are further classified as stylized facts. Cont (2001) presents a pedagogical overview of the stylized facts. The 11 stylized facts in this paper are set for financial log returns defined as $r(t, \Delta t) = ln(S(t + \Delta t)) - ln(S(t))$ with $S(t)$ the price of a financial asset at time t, and Δt the time scale indicating the time between each price measurement. When Δt is small, we speak of a fine scale, whereas if Δt is large we speak of a coarse-grained scale. The following stylized facts are outlined in their research.

1. **Absence of autocorrelation in financial log returns**: The linear autocorrelations of asset returns are very small or not existing. Only for small intraday time scales, there might be a significant autocorrelation, but this is out of the scope of this research as we only consider daily asset returns.

2. **Distribution of log returns of a financial asset is heavy tailed**: The tail index of the distribution expresses higher kurtosis than the tail index of a normal distribution.

3. **Distribution of log returns of a financial asset is skewed**: There is an asymmetry in the downwards, and the upward movements of prices. In general, there are more positive returns, but larger negative returns. This asymmetry causes a deformation of the distribution of log returns when compared to a normal distribution with the tip of the distribution more to the right but a longer tail on the left. By this characteristic, asset are said to be negatively skewed.

4. **Distribution of log returns shows Aggregational Gaussianity**: If the time scale becomes more coarse-grained (Δt larger e.g., weeks or months), the distribution of log returns tends to look more like a normal distribution. This to show that the distribution of the log returns changes under a changing time scale Δt.

5. **Intermittency in log returns**: Irregular alternations of asset returns at any time scale. Irregular price changes are common in assets. This due to irregular news events or other irregular events that can influence the price of an asset.

6. **Volatility clustering in log returns**: We find a positive autocorrelation in different volatility measures over several days. A day of high volatility is highly likely to be followed by a day of high volatility and likewise for low volatility.

7. **Conditional heavy tails in log returns**: When corrected for heteroskedasticity, the residual returns distribution still show heavy tails. However, the tails are less fat than for the unconditional log returns

8. **Slow decay of autocorrelations in nonlinear function of log returns**: Absolute log returns or squared log returns are examples of a nonlinear function of log returns. The resulting time series of these functions show slow decay in autocorrelation in function of time lag over which the autocorrelation is determined. In what follows this stylized fact will be defined as nonlinear autocorrelation.

9. **Leverage effect in log returns**: Volatility in asset price is often negatively correlated with the returns of that asset. In times of uncertainty (high volatility), investors are tempted to sell, lowering the price of assets.

10. **Volume/volatility asset correlation**: The volume at which an asset is traded is positively correlated with the volatility in asset prices.

11. **Asymmetry in volatility prediction over time scales**: Coarse-grained (large Δt) measures of volatility predict fine-scale volatility better than the other way around.

Depending on the research topic, researches focus on different stylized facts, but not often on all of them. The ones that are most often focused on are the distribution related and dynamic (autocorrelation and correlation related) stylized facts (Francq and Zakoian, 2019).

While stylized facts are a good qualitative way of describing characteristics of a time series of financial returns, Lyons et al. (2007) describes a different, more mathematical, method of characterizing a time series with mathematical signatures. The approach interprets a time series or path as a discretisation of an underlying continuous path. The signature approach transforms this information in a vector of real-valued features that are known to characterise set path. Main applications of this method are feature mappings of time series in machine learning models, but they can also be used in order to detect certain characteristics. For example, (Wilson-Nunn et al., 2018) describes the use of a signatures approach to develop a recognition methodology for Arabic handwriting, while (Vauhkonen, 2017) describes the applications of the signatures method in financial time series analysis in his master thesis. The method can be used to analyse and compare the underlying distribution of different time series.

Regarding replicating financial returns, available literature provides an abundance of ideas for replication

strategies. When discussing these models or strategies one needs to consider the entire spectrum from parametric models to non-parametric models. We identify three major classes: parametric models, non-parametric models, and artificial neural networks.

Parametric methods assume the underlying stochastic process of the time series has a structural form that can be described by utilizing a small number of parameters. Under these assumptions the handful of parameters should be able to capture the key features of the distribution of a financial time series. One can simulate a time series using such model in a Monte Carlo engine which results in plausible return paths following the underlying structure that was predetermined by the parametric model. Parametric models have multiple advantages. Firstly, by describing the characteristics of time series by only a handful of parameters, the models have a highly simplistic and transparent character, providing an idea of the impact of every parameter. Secondly, the simplicity of the models requires less computing power when fitting the parameters on sample data. Thirdly, there is an abundance of available research on parametric models describing financial time series. The disadvantage of parametric models is however, that they are often a poor approximation of reality. The small number of parameters is unable to capture the complexities of financial time series and give more of a broad approximation. That is why simple models have been adapted towards more and more complex models using many more parameters. This however decreases the transparency of the models and makes them vulnerable to overfitting and being inflexible to new data. The classic models used in a Monte Carlo approach regarding financial time series simulation are stochastic market models and autoregressive models. Classic stochastic market models like the Heston model (Heston, 1993) and stochastic alpha, beta, rho (SABR) model (Hagan et al., 2002) assume that the volatility of the price of an asset follows a random path. An autoregressive model on the other hand, is a serially dependent forecasting model which predicts the future time series values based on a relation between past values of the observed time series and past values of the volatility. A famous example of an autoregressive model is the autoregressive conditional heteroskedasticity (ARCH) model (Engle, 1982) and generalized ARCH (GARCH) model (Bollerslev, 1986), which contrary to the stochastic market model assumes that the volatility of asset returns depends on the previous volatilities and previous returns in a structural way. Trying to further improve the model, many different alternatives and advanced versions were published. Bollerslev et al. (1992) gives an extensive overview of different versions of ARCH and GARCH that were developed shortly after publication of the original. Francq and Zakoian (2019) is a very recent work on GARCH models, providing theoretical background, advanced versions, and financial applications.

Opposite to parametric methods, non-parametric methods do not assume any underlying structure in the time series process. Simulations are made by analysing and reusing historical data. Non-parametric mod-

elling can go from extremely simple historical simulation (Sharma, 2012), to more advanced methods like bootstrapping and its alternatives (Efron, 1979). The advantages of non-parametric modeling are that there are no parameters that need fitting, and that the most relevant features of time series stay intact as we reuse the original data. However, with financial data, things are not as simple as the stylized facts mention for example nonlinear autocorrelation in log returns. This stylized fact indicates that there are dependencies in time series resulting of applying a nonlinear functions on the log returns. By rearranging the historical data one might destroy the present dependencies. For this reason, solutions like block bootstrap (James et al., 2017) and other, more advanced, alternatives were introduced (Berkowitz and Kilian, 2000).

The third category of models are machine learning (ML) methods and more specific artificial neural network (ANN) methods. In recent years, with advanced computing power becoming more readily available, ML techniques have gotten much attention for market generation purposes. Data driven generative models are interesting alternatives to the parametric models as they allow for a much more realistic and complex approximation of the behaviour of asset returns in a more flexible way compared to the advanced parametric models. Furthermore, the explicit knowledge of the underlying data generating process is no longer required. Instead, generative models approximate the underlying distribution of sample data implicitly by generating samples from the data set and comparing the generated samples' similarity to the original data set with respect to certain similarity metrics. There are three techniques that are common for generating financial time series: Generative Adversarial Networks (GAN), Variational Autoencoders (VAE) and Restriced Boltzman Machines (RBM). Lezmi et al. (2020) gives an extensive explanation of both GAN and RBM models and is an example of how these methods can be used to improve the robustness of trading strategy backtesting. Wiese et al. (2020) provides another example of GANs for deep generation of financial time series. Bühler et al. (2020) provides an example of a VAE for data generation, where they focus on data generation for small data environments. Kondratyev and Schwarz (2020) describes the application of RBM in a financial time series simulation. Both GANs and VAEs are relatively new techniques (ca. 2014) making the application of these methods in financial time series generation very interesting and up to date with current research. The VAE specifically, is very interesting as it allows for operation in scarce data environments and it is theoretically straightforward.

3 Methodology

This section consists of two major parts. The first part discusses the financial returns simulation methods, and the second part discusses the time series quality assessment methods. Three generative methods are considered: DCC-GARCH, block bootstrap, and Variational Autoencoders. Furthermore three types of analysis are done on the simulated data: a qualitative analysis by checking the stylized facts of the generated

data, a quantitative analysis by comparing the underlying law of the data process of sample and simulation paths through normalized signatures, and a second quantitative test comparing the drawdown distribution of sample and simulation paths.

3.1 Simulation methods

3.1.1 Parametric model: DCC-GARCH

In section 2, two major types of parametric models made their appearance. On the one hand we mentioned stochastic market models, and on the other hand autoregressive models. Where the stochastic market models consider the volatility of the market as a random stochastic process, the autoregressors assume that a relation between previous returns and previous volatility predicts the present volatility of a financial time series. This is, the volatility at time t is a function of the previous returns and the previous volatilities, with this function depending on the type of autoregressor. The idea of modeling the volatility process with a handful of parameters to determine the financial returns is most in line with our definition of a parametric model, which is why this type of parametric model was chosen.

As mentioned before, many different versions of autoregressive models exists. The focus in this work is on GARCH models. GARCH models are very interesting as they consider heteroskedasticity in the data, so there is no assumption that the volatility is constant throughout the process. This model characteristic matches the stylized fact of volatility clustering. Furthermore, by only modeling the volatility, the returns are calculated using random vectors at each time step, which matches the stylized fact that there would be no linear autoregression in log returns. This is, these random vectors are considered i.i.d, making them independent of each other. A nonlinear function of returns is used however in the modeling of the volatility, which causes nonlinear autocorrelation in log returns, which is again a stylized fact. The indication of the presence of these stylized facts, makes the GARCH model an interesting model for this research. While dynamic facts like absence of linear autocorrelation, presence of nonlinear autocorrelation, and volatility clustering might be stylized facts that can be found in simulated data, GARCH models do make assumptions about the process determining the distribution of the data.

With portfolio optimization backtesting in mind as application of the simulated data, it is important to not only correctly simulate the return path for each asset separately, but also to consider the correlations between the different assets. Multivariate GARCH (MGARCH) models are models that consider these correlations. Bauwens et al. (2006) gives an overview of appearing types of MGARCH models. They consider three major types: generalizations of the univariate GARCH model, linear combinations of univariate GARCH models, and nonlinear combinations of univariate GARCH models. The latter option has several

attractive advantages: It allows for separate specification of the volatility of each asset, and the dependence between the individual assets. This makes that the model needs less parameters and is easier to estimate, while, in the mean time, adding more dynamics to the correlations than the linear combinations of univariate GARCH models. The most straightforward method of the nonlinear combinations class is the constant conditional correlation (CCC)-GARCH model. In this model, the correlation between time series is determined once. This correlation is based on the residuals of the univariate analysis of each time series. From that point, the correlation matrix is assumed constant. This however, does not match with reality, where correlation between different assets changes over time (Bauwens et al., 2006). To cope with this problem, Engle (2002) published the DCC-GARCH model, which is a generalization of the CCC-GARCH model and allows for dynamic correlations over time. Considering that the DCC-GARCH model is relatively easy to estimate and provides dynamic correlations, while being less complex compared to other MGARCH models, the DCC-GARCH is chosen to advocate for the parametric models.

In what follows, we explain the different components of the DCC-GARCH model, the estimation of the parameters, and the simulation of asset return paths in a Monte Carlo engine. For the explanation of the different components, we will start from the expression for the entire model and work our way down to cover every element. We use the mathematical setup from Orskaug (2009) in our explanation.

We define the DCC-GARCH model with Equations (1:3).

$$\boldsymbol{r_t} = \boldsymbol{\mu_t} + \boldsymbol{a_t} \tag{1}$$

$$\boldsymbol{a_t} = \boldsymbol{H_t^{1/2}} \boldsymbol{z_t} \tag{2}$$

$$\boldsymbol{H_t} = \boldsymbol{D_t} \boldsymbol{R_t} \boldsymbol{D_t} \tag{3}$$

With $\boldsymbol{r_t}$ the $n\,x\,1$ vector of log returns of n assets at time t, $\boldsymbol{\mu_t}$ the $n\,x\,1$ vector of expected values of the conditional $\boldsymbol{r_t}$, and $\boldsymbol{a_t}$ the $n\,x\,1$ vector of mean corrected returns of n assets at time t. $\boldsymbol{H_t}$ is the $n\,x\,n$ covariance matrix of $\boldsymbol{a_t}$ at time t, calculated with $\boldsymbol{D_t}$, the $n\,x\,n$ diagonal matrix of conditional standard deviations of $\boldsymbol{a_t}$ at time t, and $\boldsymbol{R_t}$, the $n\,x\,n$ conditional correlation matrix of $\boldsymbol{a_t}$ at time t. Finally, $\boldsymbol{z_t}$ is a $n\,x\,1$ vector of i.i.d. errors such that $\mathrm{E}[\boldsymbol{z_t}] = 0$ and $\mathrm{E}[\boldsymbol{z_t}\ \boldsymbol{z_t^T}] = I$.

Estimation of the conditional covariance matrix $\boldsymbol{H_t}$ is the eventual goal of the DCC-GARCH model. Equation (3) shows that the covariance matrix can be split in two parts. One being the diagonal matrix of the standard deviations at time t, the other being the conditional correlation matrix of the returns of the considered assets at time t. The second part of the name of the model: GARCH, refers to the estimation of the standard deviation diagonal matrix $\boldsymbol{D_t}$, while the first part of the name of the model: DCC, refers to

the estimation of the conditional correlation matrix $\boldsymbol{R_t}$.

For the calculation of every element of the standard deviation diagonal matrix $\boldsymbol{D_t}$, a univariate GARCH model is used. A univariate GARCH model is defined by Equations (4:6).

$$r_t = \mu_t + a_t \tag{4}$$

$$a_t = h_t^{1/2} z_t \tag{5}$$

$$h_t = \alpha_0 + \sum_{q=1}^{Q} \alpha_q a_{t-q}^2 + \sum_{p=1}^{P} \beta_p h_{t-p} \tag{6}$$

The equations are very similar to the equations of the DCC-GARCH, but here we only focus on one single asset. α_q for $q = 0, ..., Q$ and β_p for $p = 0, ..., P$ are the parameters of the model. If these parameters are known and if the return data is available, the variance can be determined with Equation (6). P and Q indicate the order of the GARCH model, and are free of choice but do determine the number of parameters used. If we consider Δt to be one day, then the conditional variance at time t is defined as the sum of a linear combination of the squared daily returns and a linear combination of the previous variances. From this definition, a first sense of some stylized facts is found. Large volatility on the previous day will return large volatility the next day as well, and the same goes for small volatilities. This indicates volatility clustering. Furthermore, in an indirect, nonlinear way the mean-adjusted returns depend on the previous returns as well through the volatility, indicating possible nonlinear autocorrelation in log returns. Lastly, when calculating the fourth moment of the log return distribution of the asset returns one can see that this construction does not introduce fat tails under the assumption that the standardized errors z_t are normally distributed (e.g. for $P = Q = 1$ in Orskaug (2009) p6.) for the conditional volatilities. Equations (7:9) are important constraints that have to be followed by the parameters in order for the variances to be positive.

$$\sum_{q=1}^{Q} \alpha_q + \sum_{p=1}^{P} \beta_p < 1 \tag{7}$$

$$0 \leq \alpha_q < 1 \tag{8}$$

$$0 \leq \beta_p < 1 \tag{9}$$

We use this model in order to calculate the volatility at time t for each asset in the data set. The volatilities at time t are the squared diagonal elements of matrix $\boldsymbol{D_t}$ resulting in Equation (10). The calculation of the independent volatilities is however not limited to the standard univariate GARCH(P,Q) and can be done using any univariate GARCH process that ensures the unconditional variances to exist. In this work

however, only the standard univariate GARCH is considered.

$$\boldsymbol{D}_t = \begin{bmatrix} \sqrt{h_{1t}} & 0 & \cdots & 0 \\ 0 & \sqrt{h_{2t}} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \sqrt{h_{nt}} \end{bmatrix} \tag{10}$$

The next step is the calculation of conditional correlation matrix $\boldsymbol{R_t}$. $\boldsymbol{R_t}$ is defined as the conditional correlation matrix of the standardized residuals $\boldsymbol{\epsilon_t}$ according to Equation (11).

$$\boldsymbol{\epsilon_t} = \boldsymbol{D_t}^{-1}\boldsymbol{a_t} \sim N(0, \boldsymbol{R_t}) \tag{11}$$

with the shape of $\boldsymbol{R_t}$ given by Equation (12).

$$\boldsymbol{R}_t = \begin{bmatrix} 1 & \rho_{12,t} & \rho_{13,t} & \cdots & \rho_{1n,t} \\ \rho_{12,t} & 1 & \rho_{23,t} & \cdots & \rho_{2n,t} \\ \rho_{13,t} & \rho_{23,t} & 1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \rho_{n-1,n,t} \\ \rho_{1n,t} & \rho_{2n,t} & \cdots & \rho_{n-1,n,t} & 1 \end{bmatrix} \tag{12}$$

with element $\rho_{i,j,t}$ the correlation between asset i and asset j at time t. Just like for univariate GARCH, there are requirements for $\boldsymbol{R_t}$. Two requirements have to be considered. The first requirement is the fact that $\boldsymbol{H_t}$ has to be positive definite as it is a covariance matrix. In order to make sure that $\boldsymbol{H_t}$ is positive definite, $\boldsymbol{R_t}$ has to be positive definite, as $\boldsymbol{D_t}$ is already positive definite with the diagonal elements the variance of each asset at time t. The second requirement is that, because $\boldsymbol{R_t}$ is a correlation matrix, by definition the absolute value of all the elements should be equal or less than one. By decomposing $\boldsymbol{R_t}$ using Equations (13:14), these requirements can be imposed on $\boldsymbol{R_t}$.

$$\boldsymbol{R_t} = \boldsymbol{Q_t^{*-1}} \boldsymbol{Q_t} \boldsymbol{Q_t^{*-1}} \tag{13}$$

$$\boldsymbol{Q_t} = \left(1 - \sum_{m=1}^{M} d_m - \sum_{\eta=1}^{N} b_\eta\right) \bar{\boldsymbol{Q}} + \sum_{m=1}^{M} d_m \left(\boldsymbol{\epsilon_{t-m}} \boldsymbol{\epsilon'_{t-m}}\right) + \sum_{\eta=1}^{N} b_\eta \boldsymbol{Q_{t-\eta}} \tag{14}$$

This decomposition is the DCC part of the model, as we calculate the dynamic conditional correlations at time t. Just like P and Q indicate the order of the univariate GARCH model, M and N in Equation (14) indicate the order of the DCC equation. These orders are a measure for how far back in history we look to calculate current values of $h_{i,t}$ and Q_t. Parameters $\mathbf{d}$ with $d_m \in R_+$ with $m = 1, ..., M$ and $\mathbf{b}$ with $b_\eta \in R_+$ with $\eta = 1, ..., N$, are the parameters of the DCC equation. The choice of order determines the number of parameters used in the model. In order for $\boldsymbol{H_t}$ to be positive definite $\boldsymbol{d}$ and $\boldsymbol{b}$ must satisfy Equation (15). This fulfills the first requirement for $\boldsymbol{R_t}$.

$$\sum_{m=1}^{M} d_m + \sum_{\eta=1}^{N} b_\eta < 1 \tag{15}$$

Further in Equation (14), $\bar{\boldsymbol{Q}} = Cov[\boldsymbol{\epsilon_t}\boldsymbol{\epsilon_t'}] = E[\boldsymbol{\epsilon_t}\boldsymbol{\epsilon_t'}]$ is the unconditional covariance matrix of standardized residuals $\boldsymbol{\epsilon_t}$ and can be estimated with Equation (16). $\boldsymbol{Q_t^*}$ is a diagonal matrix with the square root of the diagonal elements of $\boldsymbol{Q_t}$. $\boldsymbol{Q_t^*}$ is needed in Equation (14) to rescale the elements in $\boldsymbol{Q_t}$ such that the correlations are less or equal than one, which fulfills the second requirement for $\boldsymbol{R_t}$.

$$\overline{\boldsymbol{Q}} = \frac{1}{T}\sum_{t=1}^{T} \epsilon_t \epsilon_t^T \tag{16}$$

With every element of the DCC-GARCH model defined, the parameters $\boldsymbol{d}$, $\boldsymbol{b}$, $\boldsymbol{\alpha_0}$, $\boldsymbol{\alpha_i}$, $\boldsymbol{\beta_i}$ with $i = 1, ..., n$ with n the number of assets considered, have to be estimated. One of the main advantages of this type of MGARCH models is that the estimation of the volatility of each time series, and the estimation of the time series dependencies can be executed separately. This divides the estimation of the DCC-GARCH model in two steps. The first step is the estimation of $\boldsymbol{\alpha_0}, \boldsymbol{\alpha_i}, \boldsymbol{\beta_i}$ for each asset i separately with univariate GARCH. The second step is the estimation of $\boldsymbol{d}, \boldsymbol{b}$ with the DCC theory. The estimation steps strongly depend on the distribution for the standardized errors $\boldsymbol{z_t}$ in Equation (2). Multiple distributions are possible for these standardized errors: multivariate Gaussian, multivariate Student's t-distribution, and multivariate skew Student's t-distribution are some examples. In this paper, we work with multivariate Gaussian distributed errors. With the joint distribution of $z_1, ..., , z_t, ..., z_T$ given by Equation (17), with $t = 1, ..., T$ the time period over which the parameters are estimated.

$$f\left(\boldsymbol{z}_t\right) = \prod_{t=1}^{T} \frac{1}{(2\pi)^{n/2}} \exp\left\{-\frac{1}{2}\boldsymbol{z}_t^T \boldsymbol{z}_t\right\} \tag{17}$$

From the joint distribution of the standardized errors, the likelihood function for $\boldsymbol{a_t} = \boldsymbol{H_t^{1/2}}\boldsymbol{z_t}$ can be determined using the rule for linear transformation of variables and is given by Equation (18). The likelihood function parameter $\boldsymbol{\theta}$ contains all the parameters required in the model. Taking the log of the likelihood function and substituting $\boldsymbol{H_t} = \boldsymbol{D_t}\boldsymbol{R_t}\boldsymbol{D_t}$ results in the log-likelihood function that allows for the estimation of the model parameters, given by Equation (19).

$$L(\boldsymbol{\theta}) = \prod_{t=1}^{T} \frac{1}{(2\pi)^{n/2}\left|\boldsymbol{H}_t\right|^{1/2}} \exp\left\{-\frac{1}{2}\boldsymbol{a}_t^T \boldsymbol{H}_t^{-1}\boldsymbol{a}_t\right\} \tag{18}$$

$$
\begin{aligned}
\ln(L(\boldsymbol{\theta})) &= -\frac{1}{2}\sum_{t=1}^{T}\left(n\ln(2\pi) + \ln\left(|\boldsymbol{H}_t|\right) + \boldsymbol{a}_t^T\boldsymbol{H}_t^{-1}\boldsymbol{a}_t\right) \\
&= -\frac{1}{2}\sum_{t=1}^{T}\left(n\ln(2\pi) + \ln\left(|\boldsymbol{D}_t\boldsymbol{R}_t\boldsymbol{D}_t|\right) + \boldsymbol{a}_t^T\boldsymbol{D}_t^{-1}\boldsymbol{R}_t^{-1}\boldsymbol{D}_t^{-1}\boldsymbol{a}_t\right) \\
&\quad -\frac{1}{2}\sum_{t=1}^{T}\left(n\ln(2\pi) + 2\ln\left(|\boldsymbol{D}_t|\right) + \ln\left(|\boldsymbol{R}_t|\right) + \boldsymbol{a}_t^T\boldsymbol{D}_t^{-1}\boldsymbol{R}_t^{-1}\boldsymbol{D}_t^{-1}\boldsymbol{a}_t\right)
\end{aligned} \tag{19}
$$

For the estimation of the independent conditional variance of each asset, we impose independence on Equation (19) by setting the diagonal elements of $\boldsymbol{R_t}$ to 1, and all non-diagonal elements to zero. By doing this, only the univariate GARCH parameters for each asset are left in the equation and can be estimated. The entire set of parameters is denoted by $\boldsymbol{\phi}$ and we rewrite (19) resulting in the quasi-log-likelihood function L_1 given by Equation (20). The equation shows that the total log likelihood of all assets can be written as the sum of the log likelihood of each separate asset. This allows us to estimate the univariate GARCH parameters $\boldsymbol{\alpha_i}$ and $\boldsymbol{\beta_i}$ for each asset i denoted as $\boldsymbol{\phi}$ with a maximum log likelihood estimation scheme. The code for this maximum likelihood can be found in class `DCC_GARCH` Appendix A.1, Function `garch_fit()`.

$$
\begin{aligned}
\ln\left(L_1(\boldsymbol{\phi})\right) &= -\frac{1}{2}\sum_{t=1}^{T}\left(n\ln(2\pi) + 2\ln\left(|\boldsymbol{D}_t|\right) + \ln\left(|\boldsymbol{I}_n|\right) + \boldsymbol{a}_t^T\boldsymbol{D}_t^{-1}\boldsymbol{I}_n\boldsymbol{D}_t^{-1}\boldsymbol{a}_t\right) \\
&= -\frac{1}{2}\sum_{t=1}^{T}\left(n\ln(2\pi) + 2\ln\left(|\boldsymbol{D}_t|\right) + \boldsymbol{a}_t^T\boldsymbol{D}_t^{-1}\boldsymbol{I}_n\boldsymbol{D}_t^{-1}\boldsymbol{a}_t\right) \\
&= -\frac{1}{2}\sum_{t=1}^{T}\left(n\ln(2\pi) + \sum_{i=1}^{n}\left[\ln\left(h_{it}\right) + \frac{a_{it}^2}{h_{it}}\right]\right) \\
&= \sum_{i=1}^{n}\left(-\frac{1}{2}\sum_{t=1}^{T}\left[\ln\left(h_{it}\right) + \frac{a_{it}^2}{h_{it}}\right] + \text{ constant }\right)
\end{aligned} \tag{20}
$$

With all parameters in $\boldsymbol{\phi}$ estimated, the conditional variance $h_{i,t}$ can be calculated for each asset $i = 1, ..., n$ for each time point $t = 1, ..., T$. This allows us to estimate $\boldsymbol{\epsilon_t} = \boldsymbol{D_t^{1/2}}\boldsymbol{a_t}$ and $\bar{\boldsymbol{Q}} = E[\boldsymbol{\epsilon_t}\boldsymbol{\epsilon_t'}]$. With $\boldsymbol{\epsilon_t}$, $\boldsymbol{D_t}$, and $\bar{\boldsymbol{Q}}$ we have all the elements for the second estimation step, estimating the set of parameters $\mathbf{d}$ and $\mathbf{b}$, denoted by $\boldsymbol{\psi}$. We rewrite Equation (19) taking the parameters from estimation step 1 as given, which results in a second log-likelihood function L_2 given by Equation (21). This equation can be further simplified by assuming that $\boldsymbol{D_t}$ is constant when conditioning on the parameters from step one, which allows us to take out the constant terms in the log-likelihood function, which results in Equation (22).

$$
\begin{aligned}
\ln\left(L_2(\boldsymbol{\psi})\right) &= -\frac{1}{2}\sum_{t=1}^{T}\left(n\ln(2\pi) + 2\ln\left(|\boldsymbol{D}_t|\right) + \ln\left(|\boldsymbol{R}_t|\right) + \boldsymbol{a}_t^T\boldsymbol{D}_t^{-1}\boldsymbol{R}_t^{-1}\boldsymbol{D}_t^{-1}\boldsymbol{a}_t\right) \\
&= -\frac{1}{2}\sum_{t=1}^{T}\left(n\ln(2\pi) + 2\ln\left(|\boldsymbol{D}_t|\right) + \ln\left(|\boldsymbol{R}_t|\right) + \boldsymbol{\epsilon}_t^T\boldsymbol{R}_t^{-1}\boldsymbol{\epsilon}_t\right)
\end{aligned} \tag{21}
$$

$$\ln\left(L_2(\psi)\right) = -\frac{1}{2}\sum_{t=1}^{T}\left(\ln\left(|\boldsymbol{R}_t|\right) + \boldsymbol{\epsilon}_t^T \boldsymbol{R}_t^{-1}\boldsymbol{\epsilon}_t\right) \tag{22}$$

Equation (22) can also be estimated using a maximum likelihood scheme. The code for the estimation of the parameters of the DCC equation is given in Appendix A.1, Function `dcc_fit()`.

With all the parameters of the DCC-GARCH model estimated, the model can be used in order to simulate different scenarios for the considered assets. The pseudo algorithm is given by Algorithm 1. It is important to note that instead of predicting the covariance matrix up to k steps in the future and generating paths of random standardized vectors to multiply with the covariance matrix, we use a recursive strategy. Each calculated return has an effect on the covariance matrix of the next day.

With the mean-corrected returns $\boldsymbol{a_t}$, there is one final step left for finding log returns $\boldsymbol{r_t}$, which is the calculation of mean log return $\boldsymbol{\mu_t}$. Among prominent options for calculation of the mean are calculating the mean log return as the mean of the log returns in the sample data or working with an approximated moving average that is updated every time step (Orskaug, 2009). We use the latter. At every time step t, we update the mean value using Equation (23).

$$\boldsymbol{\mu_{t+k}} = \frac{\boldsymbol{\mu_{t+k-1}} * (T+k-1) + (\boldsymbol{a_{t+k}} + \boldsymbol{\mu_{t+k-1}})}{T+k} \tag{23}$$

Algorithm 1 finalises the section on DCC-GARCH path generation. With the proposed structure, it is expected that for the generated paths, we will find volatility clustering, slowly decreasing autocorrelation in a nonlinear function of log returns, and an absence of autocorrelation in log returns. Furthermore, we predict absence of the leverage effect, skewness, and perhaps fat tails in log returns as we did not account for these appearances in the proposed structure of the model. In the calculations we will consider a DCC-GARCH with $M = N = P = Q = 1$. In multiple works this was indicated as sufficient (Orskaug, 2009). Furthermore, finding starting values for the estimation of the parameters for higher order DCC-GARCH, and the estimation of these parameters itself is already a complete research topic by itself, which is why it is not further considered. The provided code however, is build to support higher order DCC-GARCH.

Algorithm 1: DCC-GARCH scenario generation

Result: DCC-GARCH mean-corrected returns

for *s in range(0, number_of_scenarios)* **do**

  for *k in range(1, number_of_days)* **do**

    (a) Calculate independent conditional variances h_{it+k} for each asset i with

$$h_{i,t+k} = \alpha_0 + \sum_{q=1}^{Q} \alpha_q a_{i,t+k-q}^2 + \sum_{p=1}^{P} \beta_p h_{i,t+k-p} \tag{24}$$

    (b) Calculate the conditional correlation matrix $\boldsymbol{R_{t+k}}$ with

$$\boldsymbol{R_{t+k}} = \boldsymbol{Q^*_{t+k}}^{-1} \boldsymbol{Q_{t+k}} \boldsymbol{Q^*_{t+k}}^{-1} \tag{25}$$

    (c) Calculate the conditional covariance matrix $\boldsymbol{H_{t+k}}$ with

$$\boldsymbol{H_{t+k}} = \boldsymbol{D_{t+k}} \boldsymbol{R_{t+k}} \boldsymbol{D_{t+k}} \tag{26}$$

    (d) Generate a random vector $\boldsymbol{z_{t+k}}$. Vectors $\boldsymbol{z_i}$ are standardized and follow a multivariate Gaussian distribution

$$\boldsymbol{z_{t+k}} = random.default\_rng().normal(0, 1, size = (n, 1)) \tag{27}$$

    (e) Calculate mean-corrected returns at time k for scenario s $\boldsymbol{a_{t+k}}$ with:

$$\boldsymbol{a_{t+k}} = \boldsymbol{H_{t+k}}^{1/2} \boldsymbol{z_{t+k}} \tag{28}$$

    (f) Update the data set of mean corrected returns, update the data set of variance of mean corrected returns, update data set of standardized residuals, update data set of matrices $\boldsymbol{Q_i}$

  end

end

3.1.2 Non-parametric model: Block Bootstrap

The bootstrap will advocate for the non-parametric models. The bootstrap is a widely applicable and extremely powerful statistical tool that allows obtaining new sample sets from the original data without assuming any underlying process for the data or having to train the algorithm (James et al., 2017). The main idea of the bootstrap is to resample the sample data by replacement of the data points. By replacing data points of the sample with different data points, a new time series is generated. A lot of different types of bootstrapping exist, but not all are suitable for financial time series data. For example, standard boot-

strapping assumes that every data point is i.i.d. and creates a new time series by randomly mixing the data points. If standard bootstrapping would be applied directly to dependent observations, the resampled data would not preserve the properties of the original data set. Such a bootstrapping method is not usable for financial time series, as there are dependencies in the time series. Advanced alternatives of bootstrapping exist ((Hongyi Li and Maddala, 1996), (Berkowitz and Kilian, 2000)), but as the methods become more advanced, more hyperparameters are added. These extra parameters, often chosen arbitrarily or through a grid search, determine how a certain resampling strategy is applied, which can steer the algorithm in a certain direction. Although these references show that bootstrap algorithms that make use of some parametric assumption about the sample distribution are preferred for many applications in time series econometrics, the choice of including a non-parametric model was made with the goal to have a model that does not make any assumptions about distributions at all. For this reason, we opted for a simple moving block bootstrap, that only depends on two parameters and does not make any assumptions about the sample distribution or other characteristics of the time series sample.

Moving block bootstrap divides the data set in overlapping blocks containing a certain amount of data points and randomly selects a number of these blocks to form a new time series with a desired length. The algorithm depends on two parameters. The first parameter is the amount of data points in each block. When considering daily returns, this determines how many consecutive days of data we will keep untouched. The second parameter is the size of the overlap of the blocks. How many data points of block one, will also be in block two. Figure 1 provides a illustration of this approach with block size three and overlap two. To then build a new time series of a certain length, blocks are randomly selected and put behind each other without overlap until the desired length of the new time series is reached.

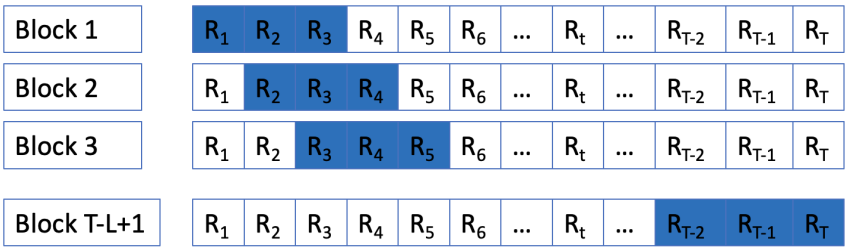

Figure 1: A graphical illustration of the creation of blocks for block bootstrap with block size 3 and overlap 2. The daily return at time t is depicted as R_t with $t = 1...T$

Algorithm 2 shows the pseudo algorithm of the approach that is used to generate new scenarios in this work using moving block bootstrap. Important to note is that for the multivariate setting, the R_t return of Figure 1 is a vector $\boldsymbol{R_t}$ that contains the returns of all considered assets at time t. This way we try to conserve

the relation between the different assets. Furthermore, we use each block only once. This should avoid the distribution of the new scenario becoming very different from the original sample distribution. This however also depends on the length of the new time series that has to be generated. The code that is written for the block bootstrap method is the class `MovingBlockBootstrap` in Appendix A.2.

Algorithm 2: Block bootstrap scenario generation

Result: Block bootstrap mean-corrected returns

max_index = len(data)/(length_block - overlap)

number_of_blocks = number_of_days / length_block

for *s in range(0, number_of_scenarios)* **do**

(a) Randomly select number_of_blocks blocks from all possible blocks with index 0 to max_index, without using a block twice

(b) Connect the blocks and create a time series of length number_of_days by cutting of the surplus of days if needed.

end

We consider a block size of five days with an overlap of one day. The thought behind five days is that this matches an entire trading week. This means that if we put four blocks behind each other for a path of 20 days, this results in four weeks of stock returns that are dependent intra weekly and considered independent on a larger time scale. This could induce the stylized fact of preserving nonlinear autocorrelations in log returns for at least five days, which also goes for the volatility clustering. The block bootstrap also favours the absence of linear autocorrelation in returns as the different blocks are expected to be independent and the intra block time series structure should already follow that stylized fact as it is sample data. Another advantage of taking blocks that are large enough is that the leverage effect might be preserved. Furthermore, if few blocks are used, the distribution of the simulated path could turn out very differently from the sample distribution, which may or may not remove heavy tails and skewness. The choice for one day of overlap comes from the reasoning that a large overlap for short scenarios enhances the probability of a resulting distribution that does not match the original distribution at all.

3.1.3 Machine Learning model: Variational Autoencoder

As mentioned earlier, three popular machine learning methods are popular in regards to generating financial return paths: Restricted Boltzman Machines, Variational Autoencoders and Generative Adversarial Networks. All three of these methods are stochastic artificial neural networks that learn the probability distribution of a real data sample. All three of these methods do not initially assume a certain structure for the underlying data generating process, and applications of all three of them can be found in current research on market generation. With not assuming any structure for the underlying data generating process initially, the

artificial neural networks completely depend on learning the underlying process through the sample data. By learning on the sample data, the artificial neural network tries to determine this underlying process, from which it can then generate new data samples. The important difference between these artificial neural networks is the objective function. RBMs train on log-likelihood maximization problem, GANs consider a minimax two-player game, and a VAE optimizes on an evidence lower bound (ELBO). According to Bühler et al. (2020) GANs are the most popular generative networks and has multiple applications in generation of financial time series, while the VAE approach is more new for this purpose. Although GANs are more popular, they require lots of data, and often have difficulties with convergence and stability. The VAE on the other hand is more elegant and simple, and handles data scarce environments better. Bühler et al. (2020) describes that considering return data of several years is already a data set of magnitudes smaller than the standard required amount of data for training of a neural network. Furthermore, the VAE is straightforward to explain, simple to implement and flexible. The unpopularity of VAEs compared to GANs comes from drawbacks of use of VAE in image generation settings, where the VAE tends to return somewhat blurry images. This is caused by attribution of high probability to nearby points other than the exact points in the training set. This returns images that are very much like the original, but not exact, hence blurry. In a time-series setting this would however not form a disadvantage for the VAE (Bühler et al., 2020). Being theoretically straightforward, elegant, easy to implement and strong performing in data scarce environments is why the VAE advocates for the artificial neural networks in this research. The following section explains the principles of VAE and the design of VAE applied in our research.

Artificial neural networks are an artificial representation of the human brain: a network of neurons that transfer information in order to perform a task at hand. In a very simple representation, a neural network is given an input and this input is transferred through the network and returns an output.The network of neurons that is considered in our research is not random, but build of different layers and the neurons of one layer can only be connected to the neurons of the previous layer and the next layer. Figure 2 gives a representation of a layered neural network. We see an input layer, two hidden layers to do some transformation of the data and an output layer containing the possible outputs. In each node, one calculates a weighted sum of the values of all the nodes in the previous layer. The group of weights associated with each node are the parameters that are trained during the training process. On the weighted sum of inputs of a node an activation function is applied and the output of that activation function is passed to the neurons in the next layers. This activation function rescales the weighted sum to a number between 0 and 1. Examples of an activation function are the sigmoid function or a Rectified Linear Unit (ReLU) function. The values that are eventually returned by the last hidden layer, should be usable to determine the desired output.

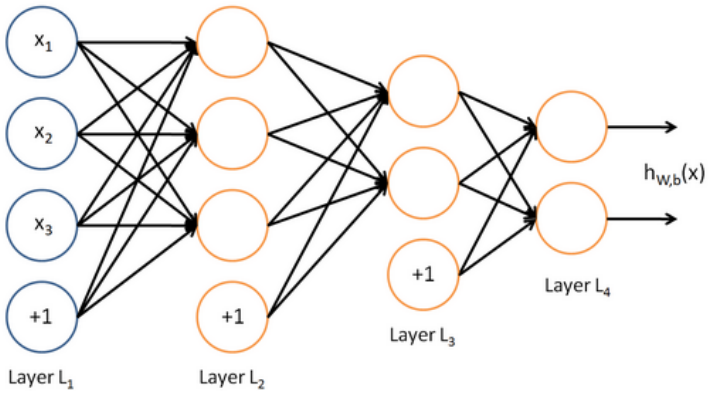

Figure 2: A graphical illustration of an artificial neural network consisting of 4 layers, with the first layer where the input is entered, 2 hidden layers that do a transformation, and the final layer containing the output (Arnx, 2019)

The network weighting parameters need to be trained in order for the data transformation to make sense. We hand both the input and output to the neural network. This way the neural network knows what output is expected. When it calculates an output and it matches the expected output, the neural network saves the parameters that are used, as this is desired. When the output does not match the expected output, the weights are altered. The magnitude of change of weights determines the speed of the training and is called the learning rate. A small change slows down the learning process but improves accuracy, while a large change speeds up the learning process but might reach a less optimal solution. Furthermore, there exists a process which is called backpropagation. Backpropagation goes against the stream of the network and investigates for every connection how the output would look if the weight of that connection was changed. Backpropagation is used in order to find the optimal weights for the neural network.

The principles of an artificial neural network are at the core of an autoencoder. An autoencoder uses two neural networks. The first neural network, the encoder, takes the data in its original shape as input and compresses the data to a lower dimension. The second neural network, the decoder, takes the compressed data from the encoder as input and returns the data in its original shape as output. The training goal is to reduce the error between the input of the encoder and the output of the decoder. The encoder changes the representation of the input data x with a function $f(x) \rightarrow z$ with z the output of your encoder. The decoder takes the encoded information z as input and decodes it with a function $g(z) \rightarrow \tilde{x}$ back to the original data shape. The objective in training the autoencoder is to make x and $\tilde{x}$ as similar as possible according to a set metric. This metric represents reconstruction loss of the model. This approach is traditionally used for dimensionality reduction of data or feature learning (Carlsson and Lindgren, 2020). Examples of applications

are image segmentation or large data transfers.

A variational autoencoder is an autoencoder that, instead of mapping the input to a lower-dimensional latent vector $\mathbf{z}$, maps the input into a distribution. The output of the encoder, vector $\mathbf{z}$, is now replaced by a vector $\boldsymbol{\mu}$, containing the parameters needed to describe the mean of the distribution, and a vector $\boldsymbol{\sigma}$, containing the parameters needed to describe the standard deviation of the distribution. The dimensions of $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ do not depend on the data but are determined by the number of nodes in the latent layer of the encoder. Complex data requires more parameters to correctly capture the underlying distribution, thus a higher amount of nodes in the latent layer of the encoder is needed. The decoder input will then be a sample from this created distribution, which is again a latent vector $\mathbf{z}$. The decoder output is again $\tilde{\boldsymbol{x}}$. The loss-function to train the variational autoencoder consists of two terms. The first term is again the reconstruction loss or decoded loss as defined for the autoencoder. The second term of the loss is the called the Kullback-Leiberd (KL)-divergence, or latent loss. The latent loss prohibits the distribution created by the encoder of diverging to much from a certain set distribution. The total loss function of the variational autoencoder is given by Equation (29).

$$\mathcal{L}(\theta, \phi; \mathbf{x}, \mathbf{z}) = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})} \left[\log p_\theta(\mathbf{x} \mid \mathbf{z})\right] - D_{KL} \left(q_\phi(\mathbf{z} \mid \mathbf{x}) \| p(\mathbf{z})\right) \tag{29}$$

In this equation $\mathbf{x}$ is the input data, $\mathbf{z}$ the latent vector, $q_\phi(\mathbf{z}|\mathbf{x})$ the encoding distribution with ϕ the encoding parameters, and $p_\theta(\mathbf{x}|\mathbf{z})$ the decoding distribution with θ the generative model parameters. The first term corresponds to the probability of observing data $\mathbf{x}$ on average over all possible samples from latent vectors $\mathbf{z}$ and should be as close to one as possible. The second term corresponds to KL-divergence, which encourages the shape of the distribution over the latent factors $\mathbf{z}$ to be closer to a certain chosen distribution, for example a standard Gaussian distribution. Maximising for the first term and minimizing for the second term results in objective Function (29). In order to be able to train the variational autoencoder, one additional step is needed. Between the encoder and decoder, a sampling operation takes place sampling $\mathbf{z}$ out of a distribution defined by $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$. This sampling operation prohibits us to properly use backpropagation to train the network as sampling of $\mathbf{z}$ is random. This can be facilitated however by defining the sample operation for $\mathbf{z}$ as $\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} * \boldsymbol{\epsilon}$ with $\boldsymbol{\epsilon}$ a random variable with standard normal distribution. The random sampling now happens in creation of $\boldsymbol{\epsilon}$, which does not need training as it is already defined. This means $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ can now be properly trained and we can make use of backpropagation. This practise is called the reparametrization trick. The dimension of vector $\boldsymbol{\epsilon}$ depends on the dimensions of $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ and thus on the number of nodes in the latent layer. Figure 3 gives a schematic representation of the described variational autoencoder. Once the VAE is trained, new data can be generated by feeding random $\boldsymbol{\epsilon}$ to $\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} * \boldsymbol{\epsilon}$ and this sampled $\mathbf{z}$ to the decoder. The decoder is trained and takes the sampled $\mathbf{z}$ and returns a generated data sample in the shape of the original input which was used to train.

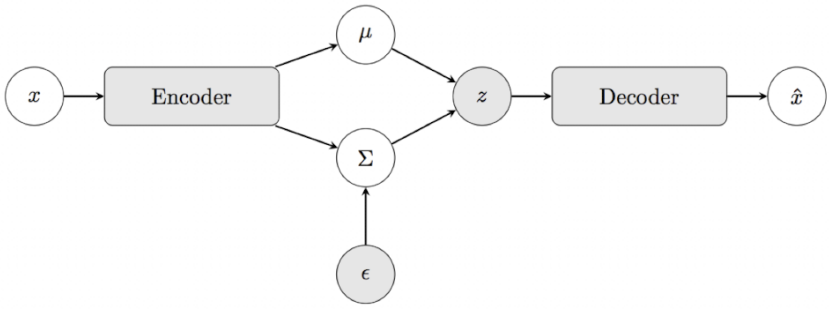

Figure 3: A flow chart representation of a variational autoencoder. The encoder maps input $\mathbf{x}$ to latent space $\mathbf{z}$ through mapping of mean and variance of sample $\mathbf{x}$ with a stochastic layer described by ϵ, μ, and σ. The decoder takes latent space $\mathbf{z}$ as input and creates a reconstruction of input $\mathbf{x}$

The Variational Autoencoder used for simulation of multivariate return data in our research is based on the VAE provided by the research in Bühler et al. (2020). The code of their VAE market generator is available in the Github repository provided for this paper (Perez, 2020). In this work they use a conditional VAE (CVAE). The conditional VAE adapts the generation process to certain market conditions by calculating and storing relevant market conditions such as the current level of volatility or current price level of an asset. This translates in the fact that the input $\mathbf{x}$ for the encoder is accompanied by additional, market relevant, data. This additional data influences the learning process of the neural network. In generation of new data, the decoder is now fed the sampled $\mathbf{z}$ and a condition. The condition we consider is a month of return paths. That is, one generated scenario is generated conditional on a month of sample data.

The details of the neural networks are found in the code for the Variational Autoencoder, class `CVAE`, Appendix A.3. Training of the VAE is done with an Adaptive Moment Estimation (Adam) optimizer. An advantage of this technique is fast convergence, a disadvantage is that the method is computationally costly (Doshi, 2019). The loss function is defined as a linear combination of the autoencoder loss (or decoded loss) and the latent loss shown in Equation (30). The decoded loss is defined as the squared difference between input and output (Equation 31) and the latent loss is defined so that the distribution of the latent vector $\mathbf{z}$ does not diverge too much from a standard Gaussian distribution (Equation 32). δ represents the trade-off between the decoded loss and the latent loss. The value for δ is set at 0.2. The market conditions that are used are the log returns of different sample paths. The function `tensor.reduced_sum()` reduces the the

dimension of a vector by summation of the elements.

$$loss = (1 - \delta) * decoded\_loss + \delta * latent\_loss \tag{30}$$

$$decoded\_loss = \|\boldsymbol{x} - \tilde{\boldsymbol{x}}\|^2 \tag{31}$$

$$latent\_loss = -0.5 * (1 + tensor.reduced\_sum(\boldsymbol{\sigma} - \boldsymbol{\mu} - exp(\boldsymbol{\sigma}))) \tag{32}$$

The inputs for the VAE should have the same shape as the desired output so input and output can be compared for training purposes. The raw sample data has dimensions (# days, # assets) with (# days) the total number of trading days considered and (# assets) the number of assets in the considered universe. The shape of our desired output is (# trading days in one month, # assets) as we want to generate monthly scenarios. As we need multiple inputs, the sample data is split up in trading months, which results in a total input dimension of (# months in sample data, # trading days in one month, # assets). This results in (# months in data) input elements of shape (# trading days in one month, # assets) resulting in output elements of shape (# trading days in one month, # assets). In other words, the inputs that are used are the monthly paths for all the different assets in the universe. In addition to each input element, a condition is given. The condition has the same shape and is the monthly data previous of the considered input data. The shape of our input required a change in the VAE provided by (Bühler et al., 2020) from a univariate VAE to a multivariate VAE to facilitate for multivariate inputs and outputs and multivariate conditions. The multivariate character asked for change of tensor operations and also for an increase of hidden nodes and latent nodes in every layer of the neural network due to the increase in complexity. The resulting code for the multivariate VAE generator can be found in Appendix A.3 in class `CVAE` and class `MarketGenerator`. The pseudo algorithm for the scenario generation is given by Algorithm 3.

Algorithm 3: Variational Autoencoder scenario generation

Result: VAE log returns

(a) Data split-up in cumulative monthly paths of all assets.

(b) Train the neural network based on loss Function 30 with tensorflow.AdamOptimizer and learning rate = 0.005 for 10.000 iterations.

(c) Generate paths by feeding sample vectors $\mathbf{z}$ and an input element as condition to the decoder.

The main advantage of a machine learning method is that instead of determining a certain underlying structure to reach an objective, the method uses a predefined objective and determines the ideal underlying structure required to optimise for the objective. This means that one can alter the objective when desired. Due to the accessibility of the VAE algorithm, adapting its objective is made possible without doing major harm to the algorithm. In the spirit of the iVaR portfolio optimization, which is explained further, the backtesting data should have realistic drawdown distributions for each asset. In order to improve the

similarity of drawdown distributions in the generated paths of each asset, we adapt the previously defined objective Function (31)) to Equation (33), with the co_drawdown loss defined by Equation (34) and γ a hyperparameter to regulate the influence of the co_drawdown loss on the total loss. The co_drawdown of an asset i, defined as the difference in drawdown between the input and output time series of asset i, is given by Equation (35) with M_t indicating the running max of a time series at moment t and V_t the value at moment t in that time series. This change in loss function is aimed at adaption of the training of the neural network in order for the neural network to specifically take into account drawdown distribution and to be able to generate data samples with a more realistic drawdown.

$$decoded\_loss = \|\boldsymbol{x} - \tilde{\boldsymbol{x}}\|^2 + \gamma * co\_drawdown\_loss \tag{33}$$

$$co\_drawdown\_loss = \sum_{i=1}^{n} co\_drawdown_i \tag{34}$$

$$co\_drawdown_i = ((0.5 * M_{t,i_{org}} - V_{t,i_{org}}) + (0.5 * M_{t,i_{gen}} - V_{t,i_{gen}}))/(M_{t,i_{diff}} - V_{t,i_{diff}}) \tag{35}$$

$$V_{t,i_{diff}} = V_{t,i_{org}} - V_{t,i_{gen}} \tag{36}$$

3.2 Quality assessment methods

In the following subsection we describe the methods that are used to assess the quality of the generated financial time series. The first test consists of checking the stylized facts for each financial time series that is generated. The second test is a process discriminator based on mathematical signatures, and is used in order to assess the equality between sample time series and simulated time series in underlying distribution. The third and final test compares the distribution of drawdowns in the sample paths against the distribution of drawdowns in the generated paths of each asset based on the Kolmogorov-Smirnov (KS) test.

3.2.1 Stylized facts in financial time series

The stylized facts that will be checked are the absence of linear autocorrelations in log returns, heavy tailed distribution of log returns, negatively skewed distribution of log returns, volatility clustering in log returns, slow decay of nonlinear autocorrelation in log returns and the leverage effect in log returns. Considering that these are stylized facts for univariate financial time series, there is need for a system that can provide information on the average univariate quality of the simulated time series, without having to manually check the quality of the time series of each asset with standard visual methods. Therefore a scoring system was developed that can provide the frequency of appearance of a stylized fact in the time series of the assets in the universe answering the question: "For how many assets does the time series exhibit a certain stylized fact?". For each stylized fact, a rule is designed. If, for a single asset, the univariate time series passes this rule, a point is awarded for that asset. After all assets are checked on that stylized fact, an overall score for that stylized fact is returned as the number of points awarded divided by the total number of assets in the universe.

This results in the relative frequency of the appearance of a stylized fact in the generated time series. With a rule for every considered stylized fact, a scoring chart for the generated universe is created. This scoring chart is also provided for the sample data, which then gives us an idea of which frequency of each stylized fact to aim for. For each stylized fact the thinking process and eventual rules are provided in the following section.

We divide these six stylized facts in two groups. The first group are the dynamic stylized facts: linear autocorrelation, nonlinear autocorrelation, volatility clustering and leverage effect. The second group are the distribution related stylized facts: heavy tails and skewness. For the first group, the analysis of the different stylized facts is very similar. For each stylized fact we define a specific correlation function, the values to expect from this correlation function, and the scoring rule that is eventually used. A visual interpretation of the rules are provided for better understanding. For the second group the rules are provided from theoretical basis.

The first correlation based stylized fact is the absence of linear autocorrelation in log returns. Linear autocorrelation is defined by Equation (37) with $corr(x, y)$ representing the correlation between variable x and variable y. In this case variable x are the log returns and variable y are the delayed log returns. It can be assumed that once the time difference τ for which we calculate the linear autocorrelation is larger than 15 minutes, the correlation becomes zero (Cont, 2001). As we are considering daily asset returns, the autocorrelation should be zero for any considered lag τ. Figure 4 (a) shows the autocorrelation for different lags τ, with the blue band indication the 99% confidence interval around zero. The autocorrelation for different lags can be calculated using the `acf()` function of the `statsmodels` python package, which additionally returns the confidence interval around zero. Using this information from the python package, the rule becomes that if a sufficient percentage of the autocorrelations defined by Equation (37) for different values of τ are in the calculated confidence interval, a point is awarded to the asset. Visually this means that sufficient blue dots have to be in the light blue band. The considered confidence interval is the 99% confidence interval, the lags considered are between 1 and 10, and the required percentage of autocorrelations in the 99% interval is 80%. The code for this test is Function `absence_of_linear_autocorrelations()` in class `StylizedFacts` in Appendix A.4.

$$C(\tau) = corr(r(t, \Delta t), r(t + \tau, \Delta t)) \tag{37}$$

Although there is no autocorrelation in log returns according to the first test, a stylized fact was discovered saying that autocorrelation is found in nonlinear functions of the log returns. This would mean that the increments of the time series are not independent, implying that there is a structured process behind the log returns and they are not just random walks. This is derived from the definition of a random walk that includes the independence of the increments. Such a nonlinear dependency is also called persistence and

in the case of financial log returns, this persistence is positive. In general, the autocorrelation function of a nonlinear function of log returns is given by Equation (38). For the stylized facts test, the considered nonlinear function is $f(x) \rightarrow ln(1 + x^2)$. This function shows the most significant values in the tests of (Cont, 2001). Figure 4(b) shows the nonlinear autocorrelation based on Equation (38) and function $f(x)$ for different values of τ and the 95% confidence interval around zero. The autocorrelation and confidence interval are again calculated with the `acf()` function. The rule goes: For an asset to be rewarded a point for presence of nonlinear autocorrelation, a percentage of the the autocorrelations has to be higher than the confidence interval around zero. The considered confidence interval is 95%, the considered lags are 1 to 3 days, and the required percentage to be above the confidence interval is 66%. The function for this test named `nonlinear_autocorrelations_in_returns` can also be found under class `StylizedFacts`, Appendix A.4.

$$C_f(\tau) = corr(f(r(t + \tau, \Delta t)), f(r(t, \Delta t))) \tag{38}$$

A special case of nonlinear autocorrelation in log returns is with $f(x) \rightarrow |x|^2$. This results in autocorrelation Equation (39) and is commonly used as a measure for autoregressive conditional heteroskedasticity, better known as volatility clustering. This is, days with high volatility are followed by days with high volatility and days with low volatility are followed by low volatility. This is very intuitive as high volatility induces uncertainty for investors, which will encourage investors to take action. Low volatility has the opposite effect and encourages the investor to hold. These autocorrelations between squared returns can be seen as an advocate for autocorrelations between volatilities, hence the name of the stylized fact.Figure 4(c) gives an illustration of volatility clustering for different values of τ in Equation (39) with a confidence interval of 95%. The rule is exactly the same as the rule for presence of nonlinear autocorrelation with the same rule parameter values. The function called `volatility_clustering()` can be found in the same class `StylizedFacts`, Appendix A.4.

$$C_2(\tau) = corr(|r(t + \tau, \Delta t)|^2, |r(t, \Delta t)|^2) \tag{39}$$

A final case a dynamic stylized fact is the leverage effect. For the leverage effect we do not assume serial dependence but dependence between two different time series. The first time series is the absolute squared return, the second time series is the log return. The correlation equation is given by Equation (40). Only a positive τ is considered as in general the leverage effect is negligible for a negative τ (Bouchaud et al., 2001). Figure 4(d) gives a visual representation of the leverage effect for different values of τ and a 95% confidence interval around zero. As far as our knowledge goes, there is no predefined python package for this correlation function and its confidence interval. That is why, based on the source code of the `acf()` function, the functions `leverage_effect()` and `leverage_effect_plot()`. The first function gives the correlations for different values of τ, using the build in pandas function `corr()` and the confidence interval, calculated according to the formula applied in the `statsmodels` package, which is the Bartlett formula. The second

function provides a plot that is inspired by the `plot_acf()` function. The rule for presence of the leverage effect is then, for an asset to be rewarded a point for the presence of the leverage effect, 50% of the correlation values for τ going from 1 to 10, has to be below the 95% confidence interval around zero. The test is coded as function `leverage_effect_fact()` under class `StylizedFacts`.

$$L(\tau) = corr((|r + \tau, \Delta t)|^2, r(t, \Delta t)) \tag{40}$$

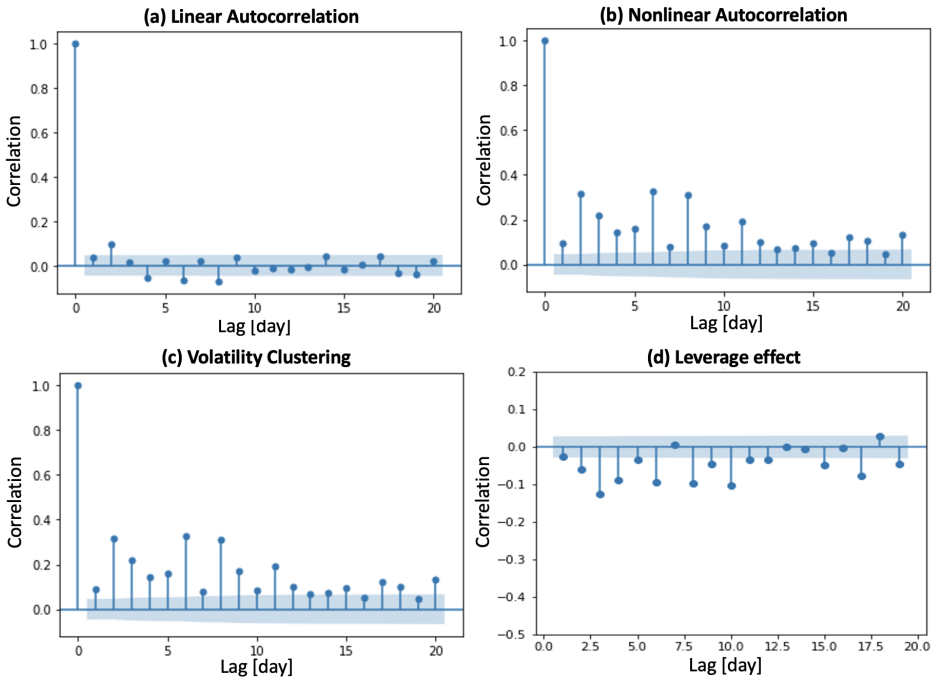

Figure 4: (a) linear autocorrelation for lag 0 to 20 and with confidence interval 95% of a random asset of the asset universe. (b) nonlinear autocorrelation based on Equation 38 for lag 0 to 20 and with confidence interval 95% of same asset as in Figure (a). (c) volatility clustering based on Equation (39) for lag 0 to 20 and with confidence interval 95% of same asset as in Figure (a). (d) leverage effect based on Equation (40) for lag 0 to 20 and with confidence interval 95% of same asset as in Figure (a).

The two remaining stylized facts are related to the distribution of the log returns. In many methods in quantitative finance, the assumption is made that the log returns of an asset are normally distributed. However, the stylized facts tell us that the distribution of the log returns of an asset are heavy tailed and negatively skewed. Figure 5, gives a visual comparison between a normal distribution and the approximated distribution of the log returns of a financial asset from the considered universe. Firstly, from the figure it can be derived that the distribution of the log returns of the considered asset is asymmetric to the right compared to the symmetric distribution of the normal distribution. This results in a negative third moment

of the underlying process, $E[X^3]$, while the third moment of a normal distribution is equal to zero $E[D^3] = 0$. Secondly, from the figure can also be derived that the tails of the log return distribution are heavier than the tails of the normal distribution. This translates in a higher fourth moment of the underlying process, $E[X^4]$, compared to the fourth moment of the normal distribution $E[D^4] = 3$. This results in two rules. Firstly, the asset is awarded a point for skewness if the third moment of the generated time series of the asset is smaller than zero. Secondly, the asset is awarded a point for heavy tails if the fourth moment of the time series of the asset is larger than 3. The frequency of appearance is determined by summing the points for each stylized facts and divided the sum by the number of assets. The functions used in the test are named `gain_loss_asymmetry()` and `heavy_tails()` respectively and are part of class `StylizedFacts`.

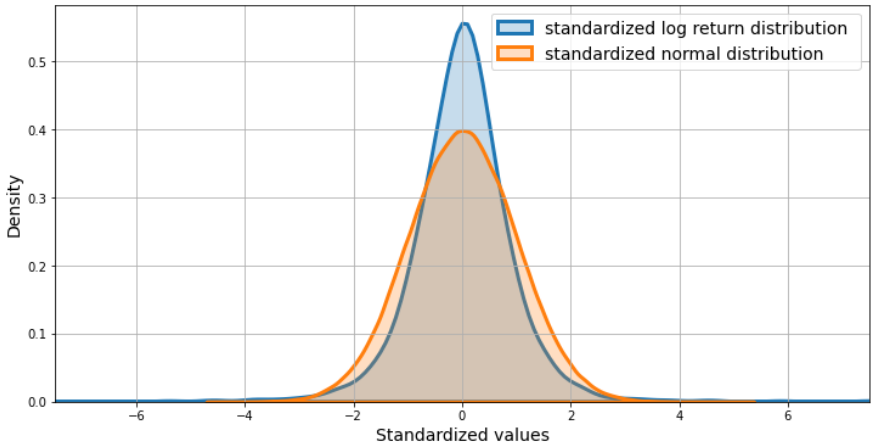

Figure 5: Comparison between the standardized log return distribution of a random asset and a standardized normal distribution. The peak and heavy tails indicate higher kurtosis for the log return distribution. The peak being on the right of the 0.0 vertical grid line shows that the distribution is negatively skewed

3.2.2 Process discriminator for financial time series

The stylized facts test allows us to check if certain characteristics that are expected return in the simulated data, however the test does not give us any quantitative information regarding similarity between sample data and generated data. A complete theoretical description of the test would take another thesis, so we focus on the intuition behind the test and what the outcome means. For the technical details I refer to Chevyrev and Oberhauser (2018).

Chevyrev and Oberhauser (2018) developed a process discriminator that provides quantitative information regarding similarity of underlying distribution of data sets. Applied to our case, this means that we split the sample data set into multiple monthly scenarios as described for the variational autoencoder. For each asset, we take the path of that asset out of each scenario. These paths of one asset is the first set used

for the test. The second set of the test are the generated paths for one asset. The process discriminator first determines an underlying law or process for the first set of paths, the sample paths. The idea is that because they represent the same asset, the paths should have the same underlying distribution. Second, the process discriminator determines the underlying process of the generated path set, the second set. Third, the process discriminator determines if the underlying process of the sample paths of an asset is the same as the underlying process of the generated paths of that asset. Applied to every asset, this gives us a quantitative measure for how many assets we have replicated the underlying process correctly.

The test is based on a theoretical concept called the signature of a path. A formal definition of a path is given in Appendix B.1. In essence, the signature of a path in a finite set of numbers characterising set path. This means that the signature is a mapping of the path to a finite set of numbers. Such a mapping of data from the original path space to a smaller vector space, allows characterization of a path in a lower dimensional space. The main application of such a mapping is machine learning, as a mapping to a lower dimensional space allows for improved learning. Appendix B.1 describes how the signature of a path is calculated mathematically.

Chevyrev and Oberhauser (2018) uses the signature mapping to determine the underlying process of several paths of one asset. The paper starts from the idea that a normalized sequence of moments characterizes the law of any finite-dimensional random variable. The problem is that this reasoning cannot be applied to paths as the path-space is infinite dimensional and this rule only works for a finite-dimensional random variable. This is where signatures come in. Signatures are a mapping that map from the path space to a finite dimensional space. After the mapping, a normalized sequence of signature moments can be used to describe the underlying law of the paths. This can be done for both the sample paths of an asset and the generated paths of that asset, resulting in the underlying law for both sets of paths, which can then be compared. If these laws are equal, it can be said that both sample paths and generated paths have the same underlying distribution.

As we are considering multiple assets, this test has to be carried out for all the assets separately. We gather the sample paths and generated paths of each asset and compare the underlying processes of these sets of paths estimated with the method of normalised signature moments. Analogous to the stylized facts check, we build a scoring system. For each asset for which we find that the underlying distribution is the same for the generated paths as for the sample paths, a point is awarded. The overall score of the model, is the total amount of points, divided by the number of assets. This score represents the frequency of appearance of equivalence in underlying process between sample paths and generated paths of the different assets.

3.2.3 iVaR and drawdown distribution in financial time series

InvestSuite provides banks, brokers, wealth managers and other financial institutions with investment solutions with at its core a portfolio construction methodology based on an in-house measure of risk: InvestSuite Value-at-Risk (iVaR). This risk measure captures the frequency, the magnitude and the duration of losses, also called drawdowns, and can be understood as the average expected drawdown percentage in a portfolio. Minimising risk against this measure has the goal to provide the smoothest possible evolution for the considered portfolio. Figure 6 gives a visual representation of how iVaR can be understood.

The mathematics behind the measure are very intuitive. M_t is defined as the rolling maximum of an

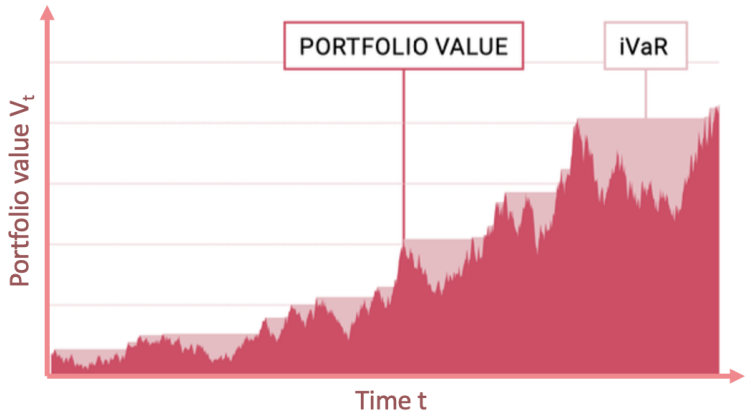

Figure 6: Visual representation of iVaR. The dark pink area indicates the evolution of the portfolio value throughout time. The light pink areas indicate the drawdown contributions, and the iVaR value is calculated as the average of all these drawdown areas (InvestSuite, 2019).

asset or portfolio with value V_t at time t and given by Equation (41). The corresponding drawdown, the difference between current rolling maximum and current value is then given by Equation (42). The drawdowns are summed over the entire considered time interval defining the accumulated drawdown with Equation (43). The iVaR value is then calculated using Equation (44).

$$M_t = \max_{s \leq t} V_s \tag{41}$$

$$D_t = M_t - V_t \tag{42}$$

$$AD_T = \sum_{t=1}^{T} (D_t) \tag{43}$$

$$iVaR = \frac{\sum_{t=1}^{T} (D_t)}{T} \tag{44}$$

Next to the standard financial time series characteristics that we considered in previous tests, applying the generated data for backtesting of an iVaR based portfolio optimization algorithm requires realistic drawdowns in the simulated data. The performance of the different generating methods in terms of conserving the drawdown distribution in the data is determined by comparing the drawdown distribution between the sample paths and the generated paths of each asset. That is, for each asset, we collect the sample paths and the generated paths and then the drawdown distribution of one set of paths is given by the mean of D_t for every path at time t, with D_t calculated with Equation (42) for each path. Figure 7 gives a visual representation of the original distribution of the drawdowns of the paths of one asset from the universe. The sample drawdown distribution and the drawdown distribution of the generated paths is compared by

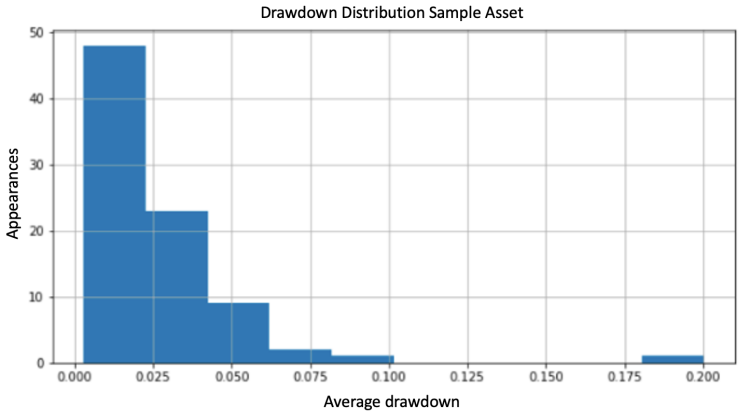

Figure 7: Visual representation of the drawdown distribution of the sample paths of one asset of the considered universe. Mainly small drawdowns with only a few extremes in the distribution.

means of the two-sample Kolmogorov-Smirnov test for goodness of fit. The test compares the underlying continuous distributions $F(x)$ and $G(y)$ of two independent variables x and y, and returns the KS statistic and the p-value of the statistic. If the value of the KS statistic is small and/or the p-value is high, the null-hypothesis $F(x) = G(y)$ for all x and all y, cannot be rejected meaning that stochastic variables x and y have the same distribution (Scipy, 2021). For each model, the minimum KS-value, the maximum KS-value, and the median KS-value accompanied by their p-value are given, to give an idea of the performance of the different models on conserving the drawdowns of the original data.

4 Data

This section provides a summary description of the considered sample data. First, we discuss the content and the choice of the data universe. Second, we provide an overview of standard characteristics of the data, and apply the stylized facts test on the sample data.

4.1 Sample data: EURO STOXX 50

In portfolio optimization, there are often constraints regarding asset classes, asset regions, or other asset specific characteristics. We consider the case that equity is the only allowed asset class, the asset region is Europe and we want to only invest in the largest public companies as we believe these give stable returns. This translates into the asset universe covered under the EURO STOXX 50 index. The EURO STOXX 50 Index is a European based index that covers 50 supersector leaders in 8 European countries: Belgium, Finland, France, Germany, Ireland, Italy, the Netherlands, and Spain (STOXX, 2021). This index is chosen for our research as it is known for being relatively stable as only the largest European blue-chip companies are considered. The index covers 50 assets, a sizeable universe which challenges both the univariate and multivariate side of the applied methods. A larger universe would cause trouble with computational effort and run times of the code.

Figure 8 provides an overview of the historic performance of a portfolio containing all assets of the universe, with a $1/n$ weight allocation rebalanced at every time point. This very simple portfolio describes the market conditions of the sample data. The 50 assets in our portfolio are the 50 assets covered by the index on the last day of our data set. This means that we do not take into account the rebalacing of the index and constantly work with the same assets. The period that is considered is a 7 year period from 2014-01-01 until 2021-01-01. The number of years is chosen in function of the needed amount of data for proper training of parameters and to avoid overfitting on a certain period. This period does include the substantial COVID-19 dip, followed by the steep regeneration of the market over the year 2020. This event might challenge the different models. Access to the data is provided by InvestSuite, which uses the Datastream data base of Refinitiv (REFINITIV, 2021).

4.2 Sample data characteristics

Table 13 in Appendix C.1 provides a list with the name of all constituents covered in the index accompanied by their annualized return and annualized standard deviation of returns. In general we see a positive mean for the different assets and a comparable standard deviation. The best performing stock is the large semiconductors related company, ASML Holding, with an annualized return of 30.8%. The worst performing company based on annualized returns is BAYER, with an annualized return of 5,9%. The asset with the highest annualized volatility is the airplane constructor AIRBUS, with an annualized volatility of 36.7%. The asset with the lowest volatility is PROSUS, with an annualized volatility of 16%.

The correlation of the sample data, and of the generated data further, is analysed by looking at the maximum correlation between different assets, the mean correlation between assets, and the minimum correlation

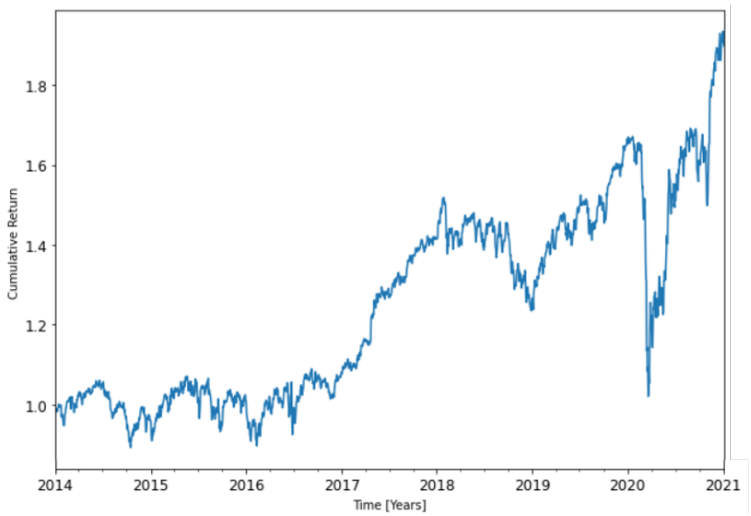

Figure 8: Plot of universe portfolio with $1/n$ weight allocation with daily rebalancing. The plot is made for the 50 assets currently covered under the EURO STOXX 50 index over a period of 7 years: 2014-01-01 - 2021-01-01. There is a steep drop in cumulative returns of the portfolio due to COVID-19, after which there follows an even stronger regeneration.

between assets. These values give us a range of to expect correlations from the different simulation models and are given in Table 1.

Table 1: Test results for the maximum, mean, and minimum correlation of the sample data. There is a small to large positive correlation between the different assets of the universe. The final years with COVID and the regeneration after will have increased the correlation values.

Correlation	Maximum	Mean	Minimum
Sample data	0.8644	0.4928	0.1043

As we are generating financial returns for the considered universe, we need to have an idea of the frequency of appearance of the different stylized facts in the sample data. This allows us to compare the quality of the generated data with the sample data regarding presence of stylized facts in the generated time series. These results form a benchmark for the results of the different models. The results of the stylized facts test for the sample data over the entire time period are given in Table 2. As expected, the results of the stylized facts test for the sample data indicate presence of these stylized facts in most of the time series of the different assets.

Table 2: Stylized fact scores of the sample data using the stylized facts tests presented in methodology. As we know that this is real financial data, the scores should be reasonably high. The test is adapted to find a reasonable score for every stylized fact.

Stylized fact	No lin. acorr.	Nonlin. acorr.	Fat tails	Skewed	Vol. clus.	Lev. eff.
Sample data	0.80	0.88	1.00	0.90	0.98	0.86

5 Results

In section 3, three different approaches were considered for the generation of multivariate financial time series. This section covers the results of the discussed approaches with the EURO STOXX 50 universe as sample data, discussed in section 4. Figure 9 gives a schematic overview of the implementation of the discussed models and tests. We start from raw sample data, which needs pre-treatment specific to each model. The model then produces the different scenarios according to the specified algorithms. Each model reproduces 84 scenarios. 84 because we generate new scenarios for each condition with the CVAE, and there are 84 different conditions when considering the current data set. From these scenarios, we take one scenario for correlation comparison and simulation analysis. For the other tests a rearrangement of the data is needed. For both process discriminator test and drawdown distribution test, we look at all scenario paths of one single asset, which corresponds to 84 scenarios of one asset. For the stylized facts test a longer data set is needed for each asset, which is why all the scenarios are combined to one long scenario resulting in a 84 month time series for each asset. For this combination of scenarios, the correlation is also compared with the sample correlation.

5.1 DCC-GARCH

Before Algorithm 1 can be applied to generate financial returns with the DCC-GARCH approach, the parameters of the DCC-GARCH model have to be estimated based on the available data. The estimation of parameters for the DCC($M = 1$,$N = 1$)-GARCH($P = 1$,$Q = 1$) model returns for each asset α_0, α, and β. These parameters are summarized in Table 14 in Apppendix C.2. Furthermore, the estimation results for d, and b, the parameters of the dynamical conditional correlation estimation, are given in Table 3. Looking at the values of the different parameters, they are comparable with estimations for the DCC parameters found in literature (Lewinson, 2020). We see a large dependence on $\boldsymbol{Q_{t-1}}$ for the calculation of $\boldsymbol{Q_t}$, and a small dependence on $\boldsymbol{\epsilon_t \epsilon_t^T}$ with $\boldsymbol{\epsilon_t}$ the residuals of the univariate GARCH estimation at time t based on Algorithm 1.

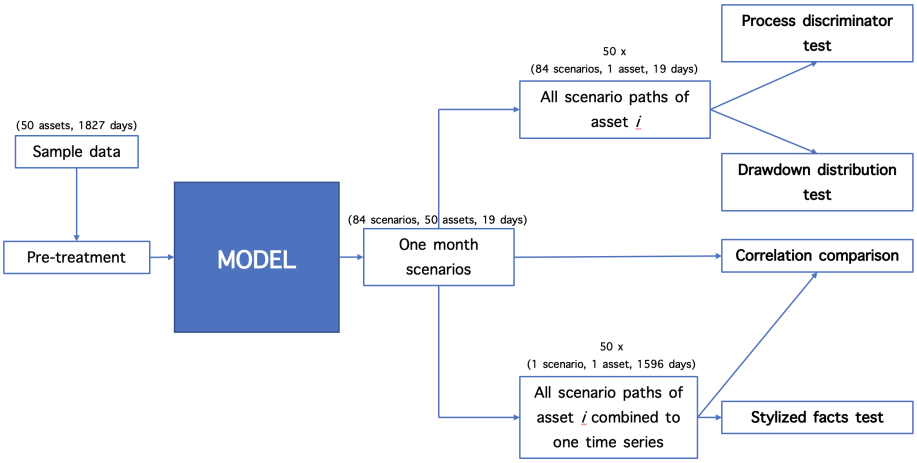

Figure 9: Graphical representation of the presented methodology, accompanied by the dimensions of the datasets that are used for each test. We start from raw sample data that is processed. The model uses the processed data and generates 84 one-month scenarios. This data is then rearranged to accommodate for the different tests.

Table 3: DCC parameter estimation values. A large weight is assigned to the correlation of the previous time step, a small weight is assigned to the residuals matrix, and a small weight is assigned to the constant conditional correlation: 1-(a+b).

DCC parameter	d	b	1-(d+b)
Value	0.004538	0.8719	0.1236

First, we investigate one generated scenario from the model. Figure 10 gives a visual representation of one scenario of multivariate paths generated for the universe of 50 assets over a period of 19 days. The figure suggests that the estimated correlations between assets are very high. Table 4 proves this statement. The minimum correlation of the scenario of Figure 10 is higher than the mean correlation between assets in the sample data. The high correlation could be caused by a number of reasons. With the high value of b, a high estimated $\boldsymbol{Q_{t+k-1}}$ could cause a high $\boldsymbol{Q_{t+k}}$ in Algorithm 1, which on its term causes a high $\boldsymbol{Q_{t+k+1}}$ and so on. This means that if the correlation calculated for the final day of the sample data is high, this might result in high correlations for the rest of the predicted path. Figure 8 indicates a strong upwards movement of the market over the final year. Such large movements in the market goes hand in hand with higher correlations. These higher correlations at the end of the sample period might cause high correlations in the DCC-GARCH scenarios. Another reason for the high correlation might be estimation errors in the volatility, causing high residuals in the GARCH estimation. These high residuals then result in a high correlation. Furthermore, the same high correlations are found when considering the combination of all scenarios. This tells us that

over all the different scenarios, the correlation should be very similar.

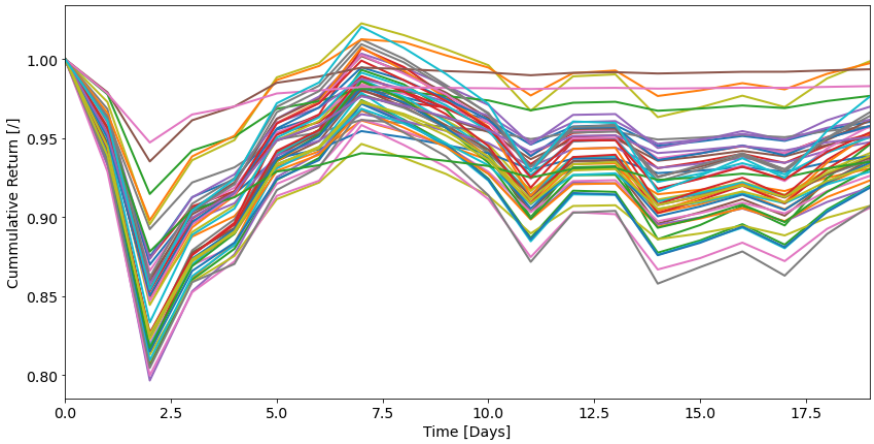

Figure 10: Plot of one generated scenario for the considered universe using the DCC-GARCH approach. The plot indicates very high correlation between assets but stays in reasonable return bounds.

Table 4: Test results for the correlation of 1 DCC scenario shown in Figure 10 and all scenarios combined in one time series. Both short period and long period correlations are high compared to the correlation of the sample data. The cause of the high correlations can be the high parameter b value or high estimation errors in the GARCH model.

Type	Sample data	DCC-GARCH 1 scenario	DCC-GARCH all scenarios
Maximum	0.8644	0.9998	0.9996
Mean	0.4928	0.9759	0.9694
Minimum	0.1043	0.7760	0.7461

A second test for the generated paths over one predicted period is the process discriminator test provided by (Chevyrev and Oberhauser, 2018). 84 paths are gathered for each asset separately and it is tested if these paths have the same underlying generative process as the sample paths of that asset. Under the circumstances described in section 3, we find a score of 0.86. This means that according to this test 86% of the assets, the generated paths have the same underlying distribution as the sample paths. This outcome is interesting as by construction, the DCC-GARCH defines a underlying distribution for the generated paths. This result shows that the assumed structure in the DCC-GARCH approximates the underlying structure of the sample paths for 86% of the assets.

A third test for the generated paths of each asset is to test if the drawdown distribution is similar to the drawdown distribution of the sample paths of that asset by means of the Kolmogorov-Smirnov test.

Table 8 gives the maximum, median and minimum KS values accompanied by their respective p-value. The hypothesis of equal distributions is rejected for maximum, median and minimum KS-value with a confidence level of 95%. This result shows that the DCC-GARCH model is not capable of simulating the drawdown distributions of the sample paths.

Table 5: Test results for the distribution comparison of the DCC-GARCH generated paths and the sample paths. A KS-value accompanied by a p-value is found for every asset. This table shows the Maximum KS-value, the median KS-value, and the minimum KS-value accompanied by their respective p-values. For maximum, median and minimum KS-value the test of similar drawdown distribution is rejected with a confidence level of 95%.

Type	KS-value	p-value
Maximum	0.8095	< 0.01
Median	0.5238	< 0.01
Minimum	0.2500	0.0102

The last test is testing the stylized facts on the time series created by combination of all generated scenarios. Table 6 shows the results of the stylized facts test for the DCC-GARCH model. High appearance frequency is found for absence of linear autocorrelation. nonlinear correlation, heavy tails, and volatility clustering. This partly matches with the hypothesis that the DCC-GARCH would generate absence of linear autocorrelation, nonlinear correlation, and volatility clustering. The heavy tails score indicates that while conditionally there are no heavy tails according to the structure of the DCC-GARCH, unconditionally this result changes. The appearance frequency for skewness and the leverage effect is lower compared to the sample data. This is not surprising as the structure of the model, does not account for skewness or leverage effect.

Table 6: Stylized facts scores of the DCC-GARCH generated data compared to the stylized facts scores of the sample data. For 4 out of 6 stylized facts, the DCC-GARCH output performs likewise to the sample data or better. For 2 out of 6 stylized facts, the appearance frequency of these stylised facts is lower than the appearance frequency in the sample data.

Stylized Fact	No lin. acorr.	Nonlin. acorr.	Fat tails	Skewed	Vol. clus.	Lev. eff.
Sample data	0.80	0.88	1.00	0.90	0.98	0.86
DCC-GARCH	1.00	0.98	1.00	0.16	0.98	0.00

5.2 Block Bootstrap

For the block bootstrap there is no estimation of parameters. This means that Algorithm 2 can be immediately applied to the sample data. We consider one generated scenario in Figure 11, which plots one generated scenario of the block bootstrap method. This figure immediately indicates that the correlation between assets is much lower compared to the correlation simulated by the DCC-GARCH model. Table 7 confirms this statement. The mean and minimum correlation of the considered scenario are lower than the sample mean and minimum correlation, which were already lower than the minimum correlation from the generated DCC-GARCH scenario. Furthermore, the combination of scenarios gives a likewise result. These correlations are close to the sample correlations. This result confirms the hypothesis that applying the block bootstrap resampling method keeps the relation between the different assets.

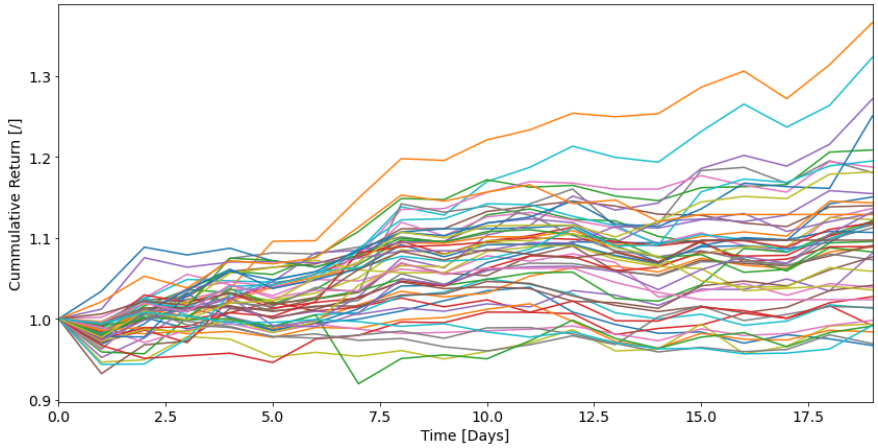

Figure 11: Plot of one generated scenario for the considered universe using the block bootstrap approach.

Table 7: Test results for the maximum, mean, and minimum correlation of one generated scenario with block bootstrap and all generated scenarios with block bootstrap combined. The results indicate that the correlation between assets is kept during resampling with moving block bootstrap.

Type	Sample data	Bootstrap 1 scenario	Bootstrap all scenarios
Maximum	0.8644	0.9873	0.8690
Mean	0.4928	0.3539	0.4895
Minimum	0.1043	-0.6580	0.0917

The process discriminator test returns a results of 1.0. This means that for every asset, the generated paths have the same underlying distribution as the sample paths. From this result it can be concluded that when

using a moving block bootstrap resampling with blocksize five and overlap one, the newly formed one month paths, consisting of four blocks give the same underlying distribution for the paths as for the sample paths.

The results of the drawdown distribution test are summarized in Table 8, which gives the max, median and minimum KS values accompanied by their respective p-value. The hypothesis of equal distributions is rejected only for the maximum KS-value and not for the median and minimum value considering a confidence interval of 95%. We see an improvement of the KS-values compared to the DCC-GARCH model, as the KS-values are lower and the p-values are higher for the median and minimum KS-value. This means that the block bootstrap model generates paths such that for more assets the drawdown distribution from its paths matches the drawdown distribution from its sample paths.

Table 8: Test results for the drawdown distribution comparison of the block bootstrap generated paths and the sample paths. This table shows the Maximum KS-value, the median KS-value, and the minimum KS-value accompanied by their respective p-values. The hypothesis of equal distributions is rejected only for the maximum KS-value and not for the median and minimum value considering a confidence interval of 95%.

Type	KS-value	p-value
Maximum	0.3929	< 0.01
Median	0.1190	0.5940
Minimum	0.0714	0.9839

Table 9 returns the results of the stylized facts test on the long period time series constructed by adding the different paths. The stylized fact test indicates that combining all the different scenarios does not hurt the frequency of appearance of absence of linear autocorrelation, nonlinear autocorrelation, heavy tails distribution or volatility clustering. The appearance of skewness and the leverage effect is however lower compared to the sample data. The under performance of the leverage effect compared to the other dynamic stylized facts has to do with the fact that it considers significantly more lags in the scoring rule, meaning that the effect of the resampling has a larger influence on this stylized fact than on the other dynamic stylized facts. The results however are very sensitive to the blocks that are selected in the generation of scenarios, as each scenario is generated from the same pool of blocks, there is change that many block return in the combined scenario time series.

Table 9: Stylized facts scores of the block bootstrap generated data compared to the stylized facts scores of the sample data. Comparable results are obtained for absence of linear autocorrelation, nonlinear autocorrelation, heavy tails distribution, and volatility clustering. Skewness and leverage effect appear less frequent in the time series of the different assets.

Stylized Fact	No lin. acorr.	Nonlin. acorr.	Fat tails	Skewed	Vol. clus.	Lev. eff.
Sample data	0.80	0.88	1.00	0.90	0.98	0.86
Bootstrap	0.96	0.84	1.00	0.56	0.84	0.18

5.3 Variational Autoencoder

The Variational Autoencoder generates 84 scenarios, which are all based on a different condition. The different conditions are the different monthly sample paths, and these conditions influence the outcome of the generator in the sense that the generator assumes that is generates a scenario that follows after the scenario that is given as a condition. The reason the VAE was selected as our neural network model was because of its simplicity and the opportunity to change the objective function. That is why the first and main result discussed for the VAE is the performance of the model regarding the simulation of realistic drawdown distributions. Table 10 gives an overview of the maximum, median and minimum KS values and their matching p-values for different values of γ. The table shows that when $\gamma = 0$, the hypothesis of equal drawdown distributions between sample paths and generated paths is rejected at a confidence level of 95% for the maximum KS value and the median KS value. When $\gamma > 0$ there is a general improvement of the median and minimum KS values as they become smaller. For $\gamma = 1.5$ and $\gamma = 2$, the hypothesis cannot be rejected at a confidence interval of 95% for the median KS values indicating further improvement. This result indicates that the additional term in the decoded loss that accounts for similarity in drawdowns between input and output does have a positive effect on the similarity in drawdowns of the generated paths, which suggests that the VAE can be improved towards market generation for specific applications like iVaR.

The remaining results of the VAE for analysis of one scenario and all scenarios combined are given for a $\gamma = 0$. Figure 12 plots one generated scenario of the VAE and Table 11 gives the correlations of the plotted scenario and of all the scenarios combined. The correlation values of 1 scenario and all scenarios combined are comparable to the sample correlation values.

For the single scenarios of the VAE, we check if the different paths of separate assets have the the same underlying distribution as their sample paths using the signature based process discriminator. The process

Table 10: Maximum, median, and minimum KS values with their respective p-values for different values of γ. The table shows that there is an improvement in three out of four cases when γ is not zero. For a γ of two the best improvement is found.

γ	Max KS	p-value	Median KS	p-value	Min KS	p-value
$\gamma = 0$	0.8095	< 0.01	0.2976	< 0.01	0.1310	0.4697
$\gamma = 1.2$	0.8214	< 0.01	0.2500	0.0102	0.0833	0.9347
$\gamma = 1.5$	0.8095	< 0.01	0.2261	0.0269	0.0952	0.8438
$\gamma = 1.8$	0.8100	< 0.01	0.3333	< 0.01	0.1190	0.5940
$\gamma = 2.0$	0.8095	< 0.01	0.1905	0.0949	0.0833	0.9347

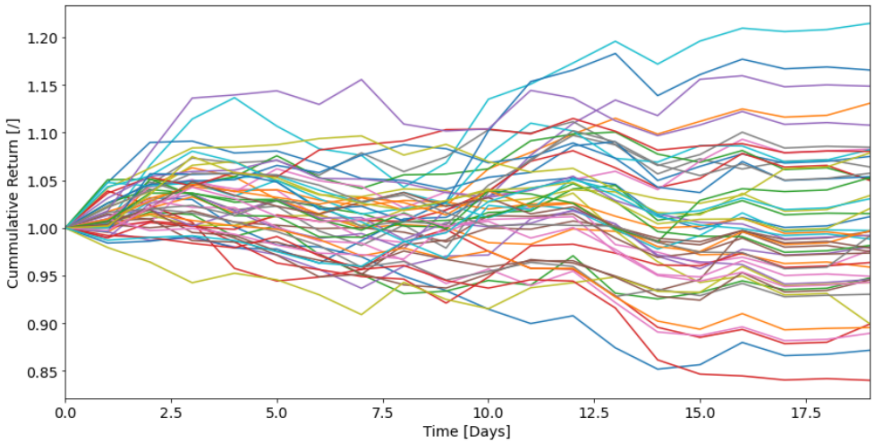

Figure 12: Plot of one generated scenario for the considered universe using the variational autoencoder method. The scenario indicates realistic monthly returns and seems to stabilise near the end of the generated month.

discriminator returns a value of 0.98, meaning that for 98% of the assets, the generated paths and the sample paths of an asset have the same underlying distribution according to the process discriminator test. This result suggests that the VAE is capable of learning the underlying process of the sample paths.

Combining the different scenarios results in one long term simulation of the universe on which the stylized facts test can be applied. The results of the test are given in Table 12. The table indicates that only the heavy tails stylized fact appears frequently in the time series of different assets. The other five stylized facts appear much less frequent compared to the sample data. An explanation for this could be the influence of the conditions on which the scenarios are generated. Due to the conditions there might be a lack of uniformity between the different scenarios.

Table 11: Test results for the maximum, mean, and minimum correlation of one generated scenario with the variational autoencoder and all generated scenarios combined. The results indicate that the correlation between assets is learned by the variational autoencoder.

Type	Sample data	VAE 1 scenario	VAE all scenarios
Maximum	0.8644	0.9168	0.8215
Mean	0.4928	0.5855	0.5349
Minimum	0.1043	-0.0182	0.2041

Table 12: Stylized facts scores of the variational autoencoder generated data compared to the stylized facts scores of the Sample data. Heavy tails seems to appear in the time series of most assets, but the other five stylized facts lack appearance compared to the sample data.

Stylized Fact	No lin. acorr.	Nonlin. acorr.	Fat tails	Skewed	Vol. clus.	Lev. eff.
Sample data	0.80	0.88	1.00	0.90	0.98	0.86
VAE	0.14	0.04	0.86	0.28	0.04	0.40

6 Conclusion

The goal of this paper was to generate multivariate financial time series data that can be applied in a backtest of a portfolio optimization algorithm. In order to be useful for a general backtest of a portfolio optimization, the generated data had to contain the basic financial time series characteristics or stylized facts, show realistic multivariate behaviour, and match in underlying distribution. When backtesting for InvestSuite Value at Risk however, an additional criterion for the generated data was added. The generated data needed to show similarity in drawdown distribution for each asset. iVaR uses the drawdown distribution to reduce overall losses in the portfolio. For the standard financial data generation three models where compared in regards to the different requirements. A DCC-GARCH based generator, a moving block bootstrap generator, and a variational autoencoder. Furthermore, the models were analysed on their capability of reproducing the drawdown distributions in the assets. All models had their advantages and disadvantages regarding the different tests, but the VAE had one main advantage. Where the other models where stiff in regards to focusing on certain characteristics, the VAE could be adapted to specifically improve drawdown characteristic. This would mean that a VAE can be used to generate financial data for any application that requires similarity in a certain specific characteristic of a time series. This opens up numerous possibilities for finance research where data is scarce. The VAE can operate in a scarce data environment and generate new data that can specifically focus on certain characteristics of the data at hand that one want to research. An example of this could be generation of extreme value data which could then be used for improved risk analysis.

For further research, two interesting directions can be indicated regarding improvement of the VAE. The first direction is looking at the nature of the input data. In this work, the inputs where the paths of the assets. A possible improvement however, could be found in changing the nature of the input by making use of a mapping. A multivariate signature mapping of the paths would reduce the dimension of the input which helps the learning of the VAE and might preserve the multivariate relations better. Next to signature mapping, other possible mappings exist that might improve the performance of the VAE. The second direction of research, is additional research regarding the loss function of the VAE. The loss function that was used now was very simple. This could however be more sophisticated which would again improve the training of the VAE.

In conclusion, the research itself gives an insightful look on the generative performance of different types of models both on a univariate and multivariate level and shows where different models could be improved. Furthermore, a basis is formed of building generative models that can be optimized in regards to the specific application of the generated data. Lastly, several research ideas are given for further research on this topic.

Bibliography

Arnx, A. (2019). First neural network for beginners explained (with code) .

Bauwens, L., Laurent, S., and Rombouts, J. V. K. (2006). Multivariate GARCH models: a survey. *Journal of Applied Econometrics*, 21(1).

Berkowitz, J. and Kilian, L. (2000). Recent developments in bootstrapping time series. *Econometric Reviews*, 19(1).

Bollerslev, T. (1986). Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics*, 31(3).

Bollerslev, T., Chou, R. Y., and Kroner, K. F. (1992). ARCH modeling in finance* A review of the theory and empirical evidence. Technical report.

Bouchaud, J.-P., Matacz, A., and Potters, M. (2001). Leverage Effect in Financial Markets: The Retarded Volatility Model. *Physical Review Letters*, 87(22).

Bühler, H., Horvath, B., Lyons, T., Arribas, I. P., and Wood, B. (2020). A Data-driven Market Simulator for Small Data Environments.

Carlsson, F. and Lindgren, P. (2020). *Deep Scenario Generation of Financial Markets*. PhD thesis, KTH Royal Institute of Technology, Stockholm.

Chevyrev, I. and Oberhauser, H. (2018). Signature moments to characterize laws of stochastic processes.

Cont, R. (2001). Empirical properties of asset returns: stylized facts and statistical issues. *Quantitative Finance*, 1:223–236.

Doshi, S. (2019). Various Optimisation Algorithms For Training Neural Network .

Efron, B. (1979). Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics*, 7(1).

Engle, R. (2002). Dynamic Conditional Correlation. *Journal of Business & Economic Statistics*, 20(3).

Engle, R. F. (1982). Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation. *Econometrica*, 50(4).

Francq, C. and Zakoian, J.-M. (2019). *GARCH models : structure, statistical inference and financial applications*. Wiley, Lille, 2 edition.

Hagan, P. S., Kumar, D., Lesniewski, A. S., and Woodward, D. E. (2002). Managing Smile Risk. *Wilmott Magazine*, pages 84–108.

Heston, S. L. (1993). A Closed-Form Solution for options with Stochastic Volatility with Applications to Bond and Currency Options. *The Review of Financial Studies*, 6:327–343.

Hongyi Li, G. S. and Maddala (1996). Bootstrapping time series models. *Econometric Reviews*, 15(2).

InvestSuite, L. (2019). The human-centred, next-generation quantitative approach to constructing portfolios. Technical report, Leuven.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2017). *An Introduction to statistical learning* . Springer.

Kondratyev, A. and Schwarz, C. (2020). The Market Generator. Technical report.

Lewinson, E. (2020). *Python for Finance Cookbook: Over 50 recipes for applying modern Python libraries to financial data analysis*, volume 1. Packt Publishing Ltd., Birmingham, 1 edition.

Lezmi, E., Roche, J., Roncalli, T., and Xu, J. (2020). Improving the Robustness of Trading Strategy Backtesting with Boltzmann Machines and Generative Adversarial Networks.

Lyons, T. J., Caruana, M., and Lévy, T. (2007). *Differential Equations Driven by Rough Paths*, volume 1908. Springer Berlin Heidelberg, Berlin, Heidelberg.

Orskaug, E. (2009). *Multivariate DCC-GARCH Model- With Various Error Distributions*. PhD thesis, The Norwegian University of Science and Technology, Trondheim.

Perez, I. (2020). Market simulator.

REFINITIV, L. (2021). Datastream.

Scipy, S. (2021). scipy.stats.ks_2samp.

Sharma, M. (2012). The Historical Simulation Method for Value-at-Risk: A Research Based Evaluation of the Industry Favorite. *SSRN Electronic Journal*.

STOXX, D. (2021). EURO STOXX 50 .

Vauhkonen, A. K. (2017). *Rough Paths Theory and its Application to Time Series Analysis of Financial Data Streams*. PhD thesis, University of Oxford, Trinity.

Wiese, M., Knobloch, R., Korn, R., and Kretschmer, P. (2020). Quant GANs: deep generation of financial time series. *Quantitative Finance*, 20(9):1419–1440.

Wilson-Nunn, D., Lyons, T., Papavasiliou, A., and Ni, H. (2018). A Path Signature Approach to Online Arabic Handwriting Recognition. In *2018 IEEE 2nd International Workshop on Arabic and Derived Script Analysis and Recognition (ASAR)*. IEEE.

A Appendices

A Code

Written code for models and testing methods.

A.1 DCC-GARCH

```
class DCC_GARCH:
    """Class that generates scenarios with dcc_garch model."""

    def __init__(self, M: int, N: int, P: int, Q: int, data: pd.DataFrame) -> None:
        self.M = M
        self.N = N
        self.P = P
        self.Q = Q
        self.T = len(data)
        self.n = len(data.columns)
        self.data = data

    def garch_var(self, params_garch: Any, data: np.array) -> np.array:
        """Calculate variance for one asset over the whole data set."""
        alpha0 = params_garch[0]
        alpha = params_garch[1 : self.P + 1]
        beta = params_garch[self.P + 1 :]
        var_t = np.zeros(self.T)
        lag = max(self.Q, self.P)
        for t in range(0, self.T):
            if t < lag:
                var_t[t] = data[t] ** 2
            else:
                if self.P == 1:
                    var_alph = alpha * (data[t - 1] ** 2)
                if self.Q == 1:
                    var_beta = beta * var_t[t - 1]
                else:
                    var_alph = np.dot(alpha, data[t - self.Q : t] ** 2)
                    var_beta = np.dot(beta, var_t[t - self.P : t])
                var_t[t] = alpha0 + var_alph + var_beta
        assert np.all(var_t > 0)
        assert not np.isnan(var_t).any()
        return var_t
```

```
35
36      def garch_loglike(self, params_garch: Any, data: np.array) -> Any:
37          """Calculate loglikelihood for each asset separatly."""
38          var_t = self.garch_var(params_garch, data)
39          Loglike = np.sum(-np.log(var_t) - (data ** 2) / var_t)
40          return -Loglike
41
42      def garch_fit(self, data: np.array) -> Any:
43          """Minimize the negative loglikelihood to estimate the parameters."""
44          total_parameters = 1 + self.P + self.Q
45          start_params = np.zeros(total_parameters)
46          start_params[0] = 0.01
47          start_params[1 : self.P + 1] = 0.01
48          start_params[self.P + 1 :] = 0.97
49          bonds = []
50          for _i in range(0, total_parameters):
51              bonds.append((1e-6, 0.9999))
52          # If you would want a working algorithm for P,Q>1 this could be used but chosing ...
                the start params is notoriously hard
53          # if max(self.P,self.Q)>1:
54          # constraint = {'type': 'ineq', 'fun': lambda x: 1 - sum(x[1:self.P+1]) - ...
                sum(x[self.P+1:])}
55          # res = minimize(self.garch_loglike, (start_params), args=(data), bounds= bonds, ...
                constraints= constraint, options={'disp':True})
56          res = minimize(self.garch_loglike, (start_params), args=(data), bounds=bonds)
57          return res.x
58
59      def dcc_covar(self, data: pd.DataFrame, params_dcc: Any, D_t: np.array) -> Any:
60          """Calculate the dynamic conitional correlation matrix and residuals."""
61          # parameters a and b
62          a = params_dcc[: self.M]
63          b = params_dcc[self.M :]
64          # calculation of residuals and Q_bar (constant conditional correlation matrix)
65          et = np.zeros((self.n, self.T))
66          Q_bar = np.zeros((self.n, self.n))
67          for t in range(0, self.T):
68              et[:, t] = np.matmul(np.linalg.inv(np.diag(D_t[t, :])), ...
                    np.transpose(data.iloc[t, :]))
69              et_i = et[:, t].reshape((self.n, 1))
70              Q_bar = Q_bar + np.matmul(et_i, et_i.T)
71          Q_bar = (1 / self.T) * Q_bar
72          # calculation of Q_t, the building stone of Rt, the dynamic conditional ...
                correlation matrix
```

```
73          lag = max(self.M, self.N)
74          Q_tn = np.zeros((self.T, self.n, self.n))
75          R = np.zeros((self.T, self.n, self.n))
76          Q_tn[0] = np.matmul(np.transpose(data.iloc[0, :] / 2), data.iloc[0, :] / 2)
77          for t in range(1, self.T):
78              # start values, niet van toepassing voor M=N=1, source is the dcc code on ...
                    which this structure is based
79              if t < lag:
80                  Q_tn[t] = np.matmul(np.transpose(data.iloc[t, :] / 2), data.iloc[t, :] / 2)
81                  assert not np.isnan(Q_tn[t]).any()
82              if lag == 1:
83                  et_i = et[:, t - 1].reshape((self.n, 1))
84                  Q_tn[t] = (1 - a - b) * Q_bar + a * np.matmul(et_i, et_i.T) + b * Q_tn[t - 1]
85                  assert not np.isnan(Q_tn[t]).any()
86              else:
87                  a_sum = np.zeros((self.n, self.n))
88                  b_sum = np.zeros((self.n, self.n))
89                  if self.M == 1:
90                      a_sum = a * np.matmul(
91                          et[:, t - 1].reshape((self.n, 1)),
92                          np.transpose(et[:, t - 1].reshape((self.n, 1))),
93                      )
94                  if self.N == 1:
95                      b_sum = b * Q_tn[t - 1]
96                  else:
97                      for m in range(1, self.M):
98                          a_sum = a_sum + a[m - 1] * np.matmul(
99                              et[:, t - m].reshape((self.n, 1)),
100                             np.transpose(et[:, t - m].reshape((self.n, 1))),
101                         )
102                     for n in range(1, self.N):
103                         b_sum = b_sum + b[n - 1] * Q_tn[t - n]
104                 Q_tn[t] = (1 - np.sum(a) - np.sum(b)) * Q_bar + a_sum + b_sum
105             Q_star = np.diag(np.sqrt(np.diagonal(Q_tn[t])))
106             R[t] = np.matmul(np.matmul(np.linalg.inv(Q_star), Q_tn[t]), np.linalg.inv(Q_star))
107         self.Q_bar = Q_bar
108         self.Q_tn = Q_tn
109         self.et = et
110         return R, et
111
112     def dcc_loglike(self, params_dcc: Any, data: pd.DataFrame, D_t: np.array) -> Any:
113         """Calculate loglikelihood for dcc estimation."""
114         Loglike = 0
```

```
115         R, et = self.dcc_covar(data, params_dcc, D_t)
116         for t in range(1, self.T):
117             et_i = et[:, t].reshape((self.n, 1))
118             residual_part = np.matmul(et_i.T, np.matmul(np.linalg.inv(R[t]), et_i))
119             determinant_part = np.log(np.linalg.det(R[t]))
120             assert determinant_part != 0
121             Loglike = Loglike + determinant_part + residual_part[0][0]
122         return Loglike
123
124     def dcc_fit(self, data: pd.DataFrame) -> Any:
125         """Fit the parameters for the dynamic conditional correlation."""
126         # Estimation of garch params and calculation of the variances
127         D_t = np.zeros((self.T, self.n))
128         par_garch_n = np.zeros((self.n, 1 + self.P + self.Q))
129         for i in range(0, self.n):
130             par_garch_i = self.garch_fit(data.iloc[:, i].to_numpy())
131             par_garch_n[i, :] = par_garch_i
132             D_t[:, i] = np.sqrt(self.garch_var(par_garch_i, data.iloc[:, i].to_numpy()))
133         # Estimation of dcc params, both low starting values to give the algorithm more ...
                freedom
134         total_params = self.M + self.N
135         start_params = np.zeros(total_params)
136         start_params[: self.M] = 0.05
137         start_params[self.M :] = 0.05
138         bounds = []
139         for _i in range(0, total_params):
140             bounds.append((0.001, 0.999))
141         constraint = {"type": "ineq", "fun": lambda x: 0.999 - x[0] - x[1]}
142         res = minimize(
143             self.dcc_loglike,
144             (start_params),
145             args=(data, D_t),
146             constraints=constraint,
147             bounds=bounds,
148             options={"disp": True},
149         )
150         # possible other option to find a global maximum or minimum
151         # res = optimize.shgo(self.dcc_loglike, bounds, args = (data, D_t), ...
                options={'disp':True})
152         par_dcc = res.x
153         return par_garch_n, par_dcc, D_t
154
155     def dcc_garch_scenarios(self, data: pd.DataFrame, ndays: int, npaths: int) -> Any:
```

```
156         """Generate scenarios for universe."""
157         data = np.log(np.array(data) + 1)  # set to log returns
158         mean_n = data.mean(axis=0)
159         self.mean = mean_n
160         demean_data = data - mean_n
161         demean_data = pd.DataFrame(demean_data)
162 
163         par_garch, par_dcc, D_t = self.dcc_fit(demean_data)
164 
165         self.par_garch = par_garch
166         self.par_dcc = par_dcc
167         print(par_garch, par_dcc)
168 
169         all_log_returns = np.zeros((npaths, ndays, self.n))
170         for s in range(npaths):
171             all_log_returns[s] = self.dcc_garch_predict(par_garch, par_dcc, D_t, ...
                    demean_data, ndays)
172 
173         all_paths, all_log_returnsT = self.cumulative_returns(all_log_returns, ndays, npaths)
174         all_returns = np.exp(all_log_returnsT) - 1
175 
176         return all_log_returnsT, all_returns, all_paths
177 
178     def dcc_garch_predict(
179         self,
180         par_garch: Any,
181         par_dcc: Any,
182         D_t: Any,
183         demean_data: pd.DataFrame,
184         ndays: int,
185     ) -> Any:
186         """Predict the future return scenarios."""
187         a = par_dcc[: self.M]
188         b = par_dcc[self.M :]
189 
190         lag = max(self.M, self.N)
191 
192         data_update = np.array(demean_data)
193         Dt1 = D_t
194         Q_bar_update = self.Q_bar
195         Qt_update = self.Q_tn
196         et_update = self.et
197         mean_n1 = self.mean
```

```
198
199         returns = np.zeros((ndays, self.n))
200
201         for k in range(ndays):
202             # step 1: garch prediction => D_t+1
203             ht1 = np.zeros(self.n)
204
205             for i in range(self.n):
206
207                 alpha0 = par_garch[i][0]
208                 alpha = par_garch[i][1 : self.P + 1]
209                 beta = par_garch[i][self.P + 1 :]
210
211                 if self.P == 1:
212                     var_alph = alpha * data_update[-1, i] ** 2
213                 if self.Q == 1:
214                     var_bet = beta * Dt1[-1][i]
215                 else:
216                     var_alph = np.dot(alpha, data_update[-1 - self.P : -1, i] ** 2)
217                     var_bet = np.dot(beta, Dt1[-1 - self.Q : -1, i])
218
219                 ht1[i] = alpha0 + var_alph + var_bet
220             Dt1 = np.append(Dt1, [ht1], axis=0)
221
222             # step 2: dcc prediction => R_t+1
223             if lag == 1:
224                 et_i = et_update[:, -1].flatten().reshape((self.n, 1))
225                 Qt1 = (1.0 - a - b) * Q_bar_update + a * np.matmul(et_i, et_i.T) + b * ...
                        Qt_update[-1]
226
227             else:
228                 a_sum = np.zeros((self.n, self.n))
229                 b_sum = np.zeros((self.n, self.n))
230
231                 if self.M == 1:
232                     a_sum = a * np.matmul(
233                         et_update[:, -1].reshape((self.n, 1)),
234                         np.transpose(et_update[:, -1].reshape((self.n, 1))),
235                     )
236                 if self.N == 1:
237                     b_sum = b * Qt_update[-1]
238                 else:
239                     for m in range(1, self.M):
```

```
                        a_sum = a_sum + a[m - 1] * np.matmul(
                            et_update[:, -1 - m].reshape((self.n, 1)),
                            np.transpose(et_update[:, -1 - m].reshape((self.n, 1))),
                        )
                    for order in range(1, self.N):
                        b_sum = b_sum + b[order - 1] * Qt_update[-order]

                Qt1 = (1 - np.sum(a) - np.sum(b)) * self.Q_bar + a_sum + b_sum

            Q_star = np.diag(np.sqrt(np.diagonal(Qt1)))
            Rt1 = np.matmul(np.matmul(np.linalg.inv(Q_star), Qt1), np.linalg.inv(Q_star))

            # step 3: return calculation => at1 = H_t+1 * z_t+1

            Ht1 = np.matmul(np.diag(Dt1[-1]), np.matmul(Rt1, np.diag(Dt1[-1])))
            zt1 = np.random.default_rng().normal(0, 1, size=(self.n, 1))

            at1 = np.matmul(np.sqrt(Ht1), zt1)
            at1 = at1.flatten()

            # calculate mean_t+k
            mean_n1 = (mean_n1 * (self.T + k) + at1 + mean_n1) / (self.T + k + 1)
            return_k = mean_n1 + at1
            returns[k] = return_k

            # step 4: update of relevant data
            data_update = np.append(data_update, [at1], axis=0)
            et1 = np.matmul(np.linalg.inv(np.diag(Dt1[-1])), np.transpose(data_update[-1, :]))
            et1 = et1.reshape((self.n, 1))
            et_update = np.append(et_update, et1, axis=1)
            # Q_bar_update = (Q_bar_update*(len(data_update)-1) + ...
                np.matmul(et1,et1.T))/(self.T+1)
            Qt_update = np.append(Qt_update, [Qt1], axis=0)

        return returns

    def cumulative_returns(self, all_returns: np.array, ndays: int, scenarios: int) -> Any:
        """Create paths instead of daily returns."""
        real_returns = np.exp(all_returns)
        paths = np.ones((scenarios, self.n, ndays + 1))
        log_returns = np.ones((scenarios, self.n, ndays))
        for s in range(scenarios):
            for k in range(1, ndays + 1):
```

```
                for i in range(self.n):
                    paths[s][i][k] = real_returns[s][k - 1][i]
                    log_returns[s][i][k - 1] = all_returns[s][k - 1][i]
            paths[s] = np.cumprod(paths[s], axis=1)
        return paths, log_returns

    def visualize(
        self,
        paths_per_asset: np.array,
        number_of_assets: int,
        number_of_scenarios: int,
        number_of_days: int,
    ) -> None:
        """Visualize the simulated returns."""
        days = list(range(number_of_days))
        fig, ax = plt.subplots(figsize=(14, 7))
        for i in range(number_of_assets):
            for s in range(number_of_scenarios):
                ax.plot(days, paths_per_asset[i][s], linewidth=2)
        ax.set_xlabel("Time [Days]", fontsize=14)
        ax.set_ylabel("Cummulative Return [/]", fontsize=14)
        ax.set_xlim(0, 19)
        ax.tick_params(axis="both", which="major", labelsize=14)
```

A.2 Block Bootstrap

```
class MovingBlockBootstrap:
    """Class that generates scenarios with Moving Block Bootstrap Method."""

    def __init__(
        self, block_size: int, overlap: int, data: pd.DataFrame, scenarios: int, ndays: int
    ) -> None:
        self.block_size = block_size
        self.overlap = overlap
        self.scenarios = scenarios
        self.ndays = ndays
        self.data = data

    def block_bootstrap(self) -> Any:
        """Create new scenarios by block bootstrapping the original sample."""
        # Otherwise the algorithm cannot work properly
```

```
16          assert self.overlap < self.block_size
17          max_index = math.floor(len(self.data) / (self.block_size - self.overlap))
18          num_blocks = math.ceil(self.ndays / self.block_size)
19          all_returns = np.zeros((self.scenarios, self.ndays, len(self.data.columns)))
20          for s in range(self.scenarios):
21              indices: List[int] = []
22
23              while len(indices) < num_blocks:
24                  index = random.randrange(max_index - 1)
25                  if index not in indices:
26                      indices.append(index)
27
28              for i in range(len(indices)):
29                  if i == 0:
30                      returns = self.data.iloc[
31                          (self.block_size - self.overlap)
32                          * indices[i] : (self.block_size - self.overlap)
33                          * indices[i]
34                          + self.block_size,
35                          :,
36                      ]
37                  else:
38                      data = self.data.iloc[
39                          (self.block_size - self.overlap)
40                          * indices[i] : (self.block_size - self.overlap)
41                          * indices[i]
42                          + self.block_size,
43                          :,
44                      ]
45                      returns = returns.append(data)
46              all_returns[s] = returns.iloc[: self.ndays, :]
47          paths, log_returns = self.paths(all_returns)
48          return log_returns, all_returns, paths
49
50      def paths(self, all_returns: np.array) -> Any:
51          """Create paths out of returns."""
52          N_assets = len(self.data.columns)
53          paths = np.ones((self.scenarios, N_assets, self.ndays + 1))
54          log_returns = np.ones((self.scenarios, N_assets, self.ndays))
55          for n in range((len(self.data.columns))):
56              for s in range(self.scenarios):
57                  for k in range(self.ndays):
58                      paths[s][n][k + 1] = all_returns[s][k][n] + 1
```

```
                    log_returns[s][n][k] = np.log(all_returns[s][k][n] + 1)
                paths[s][n] = np.cumprod(paths[s][n])
        return paths, log_returns

    def visualize(
        self,
        returns_per_asset: np.array,
        number_of_assets: int,
        number_of_scenarios: int,
        number_of_days: int,
    ) -> None:
        """Visualize the simulated returns."""
        days = list(range(number_of_days))
        fig, ax = plt.subplots(figsize=(14, 7))
        for i in range(number_of_assets):
            for s in range(number_of_scenarios):
                ax.plot(days, returns_per_asset[i][s])
        ax.set_xlabel("Time [Days]", fontsize=14)
        ax.set_ylabel("Cummulative Return [/]", fontsize=14)
        ax.set_xlim(0, 19)
        ax.tick_params(axis="both", which="major", labelsize=14)
```

A.3 Variational Autoencoder

```
class CVAE(object):
    """Conditional Variational Auto Encoder (CVAE)."""

    def __init__(self, n_latent, n_hidden=1000, alpha=0.2):
        self.n_latent = n_latent
        self.n_hidden = n_hidden
        self.alpha = alpha

    def lrelu(self, x, alpha=0.3):
        return tf.maximum(x, tf.multiply(x, alpha))

    def encoder(self, X_in, cond, input_dim):
        with tf.variable_scope("encoder", reuse=None):
            x = tf.concat([X_in, cond], axis=2)
            x = tf.layers.flatten(x)
            x = tf.layers.dense(x, units=self.n_hidden, activation=self.lrelu)
            mn = tf.layers.dense(x, units=self.n_latent, activation=self.lrelu)
```

```
18              sd = tf.layers.dense(x, units=self.n_latent, activation=self.lrelu)
19              epsilon = tf.random_normal(tf.shape(mn), 0, 1, dtype=tf.float32, seed = 580)
20              z = mn + sd*epsilon
21              return z, mn, sd
22
23      def decoder(self, sampled_z, cond, input_dim):
24          with tf.variable_scope("decoder", reuse=None):
25              cond = tf.layers.flatten(cond)
26              x = tf.concat([sampled_z, cond], axis=1)
27              x = tf.layers.dense(x, units=self.n_hidden, activation=self.lrelu)
28              x = tf.layers.dense(x, units=input_dim[0]*input_dim[1], activation=tf.nn.sigmoid)
29              # we have to make sure that a reshape is possible.
30              x = tf.reshape(x, shape=[-1, input_dim[0], input_dim[1]])
31
32              return x
33
34      def train(self, data, data_cond, n_epochs=10000, learning_rate=0.0005,
35                show_progress=True, beta=0):
36
37          data = utils.as_float_array(data)
38          data_cond = utils.as_float_array(data_cond)
39
40          if len(data_cond.shape) == 1:
41              data_cond = data_cond.reshape(-1, 1)
42
43          assert data.max() ≤ 1. and data.min() ≥0., \
44              "All features of the dataset must be between 0 and 1."
45
46          tf.reset_default_graph()
47
48          input_dim = data[1].shape
49          self.input_dim = input_dim
50          dim_cond = data_cond[1].shape
51
52          X_in = tf.placeholder(dtype=tf.float32, shape=[None, input_dim[0], input_dim[1]], ...
                name="X")
53          self.cond = tf.placeholder(dtype=tf.float32, shape=[None, dim_cond[0], ...
                dim_cond[1]], name="c")
54          Y = tf.placeholder(dtype=tf.float32, shape=[None, input_dim[0], input_dim[1]], ...
                name="Y")
55
56          Y_flat = Y
57
```

```
58
59          self.sampled, mn, sd = self.encoder(X_in, self.cond, input_dim=input_dim)
60          self.dec = self.decoder(self.sampled, self.cond, input_dim=input_dim)
61          unreshaped = tf.reshape(self.dec, [-1, input_dim[0], input_dim[1]])
62
63          D_reshape = tf.reshape(Y_flat, [-1,input_dim[1]])
64          G_reshape = tf.reshape(unreshaped, [-1, input_dim[1]])
65          co_drawdown = self.co_dd(G_reshape, D_reshape)
66          co_drawdown = tf.reduce_sum(tf.reshape(co_drawdown, [-1,input_dim[0]]),1)
67
68          decoded_loss1 = tf.reduce_sum(tf.squared_difference(unreshaped, Y_flat),1)
69          decoded_loss = tf.reduce_sum(decoded_loss1,1) + beta * co_drawdown
70          latent_loss = -0.5 * tf.reduce_sum(1. + sd - tf.square(mn) - tf.exp(sd),1)
71          self.loss = tf.reduce_mean((1 - self.alpha) * decoded_loss + self.alpha * latent_loss)
72
73          optimizer = tf.train.AdamOptimizer(learning_rate).minimize(self.loss)
74          self.sess = tf.Session()
75          self.sess.run(tf.global_variables_initializer())
76
77
78          for i in tqdm(range(n_epochs), desc="Training"):
79              self.sess.run(optimizer, feed_dict={X_in: data, self.cond: data_cond, Y: data})
80              if not i % 1000 and show_progress:
81                  ls, d = self.sess.run([self.loss, self.dec], feed_dict={X_in: data, ...
                        self.cond: data_cond, Y: data})
82                  print(i, ls)
83
84      def generate(self, cond, seed, n_samples=None):
85          cond = utils.as_float_array(cond)
86
87          if n_samples is not None:
88              np.random.seed(seed)
89              randoms = np.random.normal(0, 1, size=(n_samples, self.n_latent))
90              cond = [list(cond)] * n_samples
91          else:
92              np.random.seed(seed)
93              randoms = np.random.normal(0, 1, size=(1, self.n_latent))
94              cond = [list(cond)]
95
96          samples = self.sess.run(self.dec, feed_dict={self.sampled: randoms, self.cond: cond})
97
98          if n_samples is None:
99              return samples[0]
```

```
100
101         return samples
102
103
104     def co_dd(self, ts1, ts2):
105         ts_diff = 0.5 * ts1 + 0.5 * ts2
106         co_dd = tf.reduce_mean((0.5 * (tf_run_max(ts1) - ts1)) + (0.5 * (tf_run_max(ts2) - ...
                ts2)), 1) / tf.reduce_mean(tf_run_max(ts_diff) - ts_diff)
107         return co_dd
108
109
110 def tf_maintain_loop(x, loop_counter):
111     return tf.not_equal(loop_counter, 0)
112
113 def tf_run_max_loop(x, loop_counter):
114     loop_counter -= 1
115     y = tf.concat(([x[0]], x[:-1]), axis=0)
116     z = tf.maximum(x, y)
117     return z, loop_counter
118
119 def tf_run_max(ts):
120     cumulative_max, _ = tf.while_loop(cond=tf_maintain_loop, body=tf_run_max_loop, ...
            loop_vars=(ts, ts.shape[1]))
121     return cumulative_max
122
123 class MarketGenerator:
124     def __init__(self, ticker, start=pd.Timestamp('2017-01-01'),
125                  end= pd.Timestamp('2021-01-01'), freq="BM", n_latent=20, n_hidden=500):
126
127         self.ticker = ticker
128         self.start = start
129         self.end = end
130         self.freq = freq
131
132         self._load_data()
133         self._build_dataset()
134         self.generator = CVAE(n_latent, n_hidden, alpha=0.03)
135
136     def _load_data(self):
137         dat = Data(self.ticker, self.start, self.end)
138         data = dat.get_data()
139         self.data = data
140
```

```
141         self.windows = []
142         for _, window in self.data.resample(self.freq):
143             values = window.values
144             path = []
145             for i in range(len(self.data.columns)):
146                 values_i = []
147                 for k in range(len(values)):
148                     values_i.append(values[k,i])
149                 path_i = leadlag(values_i)
150                 path.append(path_i)
151             self.windows.append(path)
152 
153     def _build_dataset(self):
154         orig_logsig = []
155         for portfolio in self.windows:
156             orig_logsig_i = np.array([np.diff(np.log(path_i[::2, 1])) for path_i in ...
                    portfolio])
157             orig_logsig.append(orig_logsig_i)
158         self.orig_logsig = [p for p in orig_logsig if len(p[0]) ≥ 4]
159         steps = np.inf
160         for port in range(len(self.orig_logsig)):
161             minn = min(map(len, self.orig_logsig[port]))
162             if minn < steps:
163                 steps = minn
164         self.scaler = MinMaxScaler(feature_range=(0.00001, 0.99999))
165         logsig = np.zeros((len(orig_logsig), len(self.data.columns), steps))
166         for port in range(len(self.orig_logsig)):
167             self.orig_logsig[port] = np.array([val[:steps] for val in self.orig_logsig[port]])
168             logsig_i = self.scaler.fit_transform(self.orig_logsig[port])
169             logsig[port] = logsig_i
170 
171         self.logsigs = logsig[1:]
172         self.conditions = logsig[:-1]
173 
174     def train(self, n_epochs=10000, beta=0):
175         self.generator.train(self.logsigs, self.conditions, n_epochs=n_epochs, beta=0)
176 
177     def generate(self, logsig, seed, n_samples=None, normalised=False):
178         generated = self.generator.generate(logsig, seed, n_samples=n_samples)
179         if normalised:
180             return generated
181 
182         if n_samples is None:
```

```
            return self.scaler.inverse_transform(generated)

        return self.scaler.inverse_transform(generated)
```

A.4 Stylized Facts

```
class StylizedFacts:
    """Class that tests 6 stylized facts for all assets in the asset universe."""

    def __init__(self, data: pd.DataFrame):
        self.data = data

    def leverage_effect(self, data: pd.DataFrame, lag: int, alpha: float) -> Any:
        """Calculate correlation delayed squared returns and returns."""
        datax = data
        datay = abs(datax) ** 2
        corrs = []
        for i in range(lag):
            corrs.append(datax.corr(datay.shift(-i)))
        np.array(corrs).shape
        nobs = len(corrs)
        varacf = np.ones_like(corrs) / nobs
        varacf[0] = 0
        varacf[1] = 1.0 / nobs
        varacf[2:] *= 1 + 2 * np.cumsum(np.power(corrs[1:-1], 2))
        interval = stats.norm.ppf(1 - (1 - alpha) / 2.0) * np.sqrt(varacf)
        confint = np.array(lzip(corrs - interval, corrs + interval))

        return corrs, confint

    def leverage_effect_plot(self, data: pd.DataFrame, lag: int, alpha: float) -> None:
        """Return graph with leverage correlaltions for different lags."""
        corrs, confint = self.leverage_effect(data, lag, alpha)
        lags = np.array(list(range(lag)))
        fig, ax = plt.subplots(figsize=(14, 7))
        ax.margins(0.05)
        ax.vlines(lags[1:], [0], corrs[1:])
        ax.axhline()
        ax.plot(lags[1:], corrs[1:], "o")
        ax.set_title("leverage effect", fontsize=14)

```

```
36          lags = lags[1:]
37          confint = confint[1:]
38          corrs = corrs[1:]
39          lags = lags.astype(float)
40          lags[0] -= 0.5
41          lags[-1] += 0.5
42          ax.fill_between(lags, confint[:, 0] - corrs, confint[:, 1] - corrs, alpha=0.25)
43
44      def absence_of_linear_autocorrelations(self) -> float:
45          """Verify no autocorrelation in daily returns."""
46          # Test no autocorrelation in returns
47          data = self.data
48          lags = 5
49          alpha = 0.01
50          # assert len(data)/lags > lags # otherwise the autocorrelation calculations might ...
                be off
51          succes = np.zeros(len(data.columns))
52          for i in range(len(data.columns)):
53              # determine autocorrelation and confidence interval
54              ACF, confint = acf(data.iloc[:, i], nlags=lags, alpha=alpha, fft=False)
55              score = 0
56              for auto_i in range(1, len(ACF)):
57                  if (
58                      ACF[auto_i] > confint[auto_i][0] - ACF[auto_i]
59                      and ACF[auto_i] < confint[auto_i][1] - ACF[auto_i]
60                  ):
61                      score += 1
62              if score / (lags) ≥ 0.80:
63                  succes[i] = 1
64          score = np.sum(succes) / len(succes)
65          return score
66
67      def nonlinear_autocorrelation_in_returns(self) -> float:
68          """Verify slow deacy of autocorrelation in absolute daily returns."""
69          # Test nonlinear autocorrelation in returns: ln(1+x**2) # more expressed in ...
                absolute values but can be any function basically.
70          data = self.data
71          lags = 3
72          alpha = 0.05
73          nonlin_data = np.log(1 + np.array(data) ** 2)
74          nonlin_data = pd.DataFrame(nonlin_data)
75          succes = np.zeros(len(data.columns))
76          for i in range(len(data.columns)):
```

```
77              # determine autocorrelation and confidence interval
78              ACF, confint = acf(nonlin_data.iloc[:, i], nlags=lags, alpha=alpha, fft=False)
79              score = 0
80              for auto_i in range(1, len(ACF)):
81                  if (
82                      ACF[auto_i] > confint[auto_i][1] - ACF[auto_i]
83                  ):  # put in slowly dying measure as well?
84                      score += 1
85              if score / lags ≥ 0.65:
86                  succes[i] = 1
87          score = np.sum(succes) / len(succes)
88          return score
89
90      def heavy_tails(self) -> Any:
91          """Verify heavy tails in financial data."""
92          # Test kurtosis:
93          data = self.data
94          succes = np.zeros(len(data.columns))
95          for i in range(len(data.columns)):
96              # Determine kurtosis value. Pandas uses Fisher's value so kurtosis of normal ...
                    dist == 0, we add three for the standard defintion
97              kurtos = data.iloc[:, i].kurtosis()
98              if kurtos > 0:
99                  succes[i] = 1
100         score = np.sum(succes) / len(succes)
101         return score
102
103     def gain_loss_asymmetry(self) -> Any:
104         """Verify a gain loss asymmetry in financial data."""
105         # Test skewnes:
106         data = self.data
107         succes = np.zeros(len(data.columns))
108         for i in range(len(data.columns)):
109             skewn = data.iloc[:, i].skew()
110             if skewn < 0:
111                 succes[i] = 1
112         score = np.sum(succes) / len(succes)
113         return score
114
115     def volatility_clustering(self) -> float:
116         """Verify volatility clustering in financial data."""
117         # Test volatility clustering in returns
118         data = self.data
```

```
        lags = 3
        alpha = 0.05
        succes = np.zeros(len(data.columns))
        for i in range(len(data.columns)):
            # determine autocorrelation and confidence interval
            ACF, confint = acf(abs(data.iloc[:, i] ** 2), nlags=lags, alpha=alpha, fft=False)
            score = 0
            for auto_i in range(1, len(ACF)):
                if ACF[auto_i] > confint[auto_i][1] - ACF[auto_i]:
                    score += 1
            if score / lags ≥ 0.65:
                succes[i] = 1
        score = np.sum(succes) / len(succes)
        return score

    def leverage_effect_fact(self) -> float:
        """Verify leverage effect in each separate time series."""
        data = self.data
        lag = 10
        alpha = 0.05
        succes = np.zeros(len(data.columns))
        for i in range(len(data.columns)):
            lev, confint = self.leverage_effect(data.iloc[:, i], lag, alpha)
            score = 0
            for k in range(len(lev[1:])):
                if lev[k] < (confint[k, 0] - lev[k]):
                    score += 1
            if score / lag ≥ 0.50:
                succes[i] = 1
        score = np.sum(succes) / len(succes)
        return score

    def get_style_score(self) -> pd.DataFrame:
        """Return score chart of stylized facts test."""
        style_score = {}
        style_score[
            "No linear autocorrelation in returns"
        ] = self.absence_of_linear_autocorrelations()
        style_score[
            "Nonlinear autocorrelation in returns"
        ] = self.nonlinear_autocorrelation_in_returns()
        style_score["Heavy tails distribution"] = self.heavy_tails()
        style_score["Gain loss assymmetry distribution"] = self.gain_loss_asymmetry()
```

```
        style_score["volatility_clustering in returns"] = self.volatility_clustering()
        style_score["leverage effect in returns"] = self.leverage_effect_fact()

        style_score_table = pd.DataFrame.from_dict(style_score, orient="index")

        return style_score_table
```

B Theory

Section that gives additional explanations on some theoretic concepts not explained in the core text.

B.1 Signatures

A signature is defined as a transformation of the path space. A transformation that anticipates typical properties of the data and irregularities of sampling. A transformation that is capable of describing the interaction of complex oscillatory systems. A signature of path $X : [a, b] - >^d$ denoted as $S(X)_{a,b}$ is a collection of all iterated integrals of X. An iterated integral for path $X : [a, b] - >^d$ is defined as follows. Denote the coordinate paths by $(X_t^1, ..., X_t^d)$ where each $X^i : [a, b] - >^d$ is a real-valued path. For any single index $i \in \{1, ..., d\}$ we let equation 45 define quantity $S(X)_{a,t}^i$, which is the increment of the i-th coordinate of the path at time $t \in [a, b]$. Next, for every pair of coordinates $i, j \in \{1, ..., d\}$, the double integral $S(X)_{a,t}^{i,j}$ is defined by equation 46. Likewise, for any triplet of coordinates $i, j, k \in \{1, ..., d\}$, the triple-integrated is defined by equation 47. This can be continued recursively so that for any n indexes the n-fold integral of X is defined by equation 48.

$$S(X)_{a,t}^i = \int_{a<s<t} dX_s^i = X_t^i - X_0^i \tag{45}$$

$$S(X)a, t^{i,j} = \int_{a<s<t} S(X)a, s^i dX_s^j = \int_{a<r<s<t} dX_r^i dX_s^j \tag{46}$$

$$S(X)a, t^{i,j,k} = \int_{a<s<t} S(X)a, s^{i,j} dX_s^k = \int_{a<q<r<s<t} dX_q^i dX_r^j dX_s^k \tag{47}$$

$$S(X)a, t^{i_1,\ldots,i_n} = \int_{a<s<t} S(X)_{a,s}^{i1,\ldots,i_{n-1}} dX_s^{i_n} \tag{48}$$

With the iterated integrals of X a formal definition for the signature of path X can be given as the collection of all the iterated integrals of X. That is, $S(X)_{a,t}$ is defined as a sequence of real numbers by equation 49.

$$S(X)_{a,b} = \left(1, S(X)_{a,b}^1, \ldots, S(X)_{a,b}^d, S(X)_{a,b}^{1,1}, S(X)_{a,b}^{1,2}, \ldots\right) \tag{49}$$

With this definition we can show that the signature of a path is a finite sequence of numbers that is able to describe set path. The signature technique allows to translate a path from the infinite dimensional and non-

locally compact pathspace $C^0([0,1],^d$ to a finite dimensional space where a sequence of numbers describes the feature of a path. (Chevyrev and Oberhauser, 2018) takes it even a step further by normalizing the signature moments in order to be able to describe the full distribution where the paths originate from.

C Figures and Tables

Additional figures and tables of data and results section.

C.1 Asset Universe

Table 13: Annualized return and annualized standard deviation of the constituents of the asset universe.

Constituents	Annualized return	Annualized std. dv.
DEUTSCHE BOERSE (XET)	0.193942	0.238111
INDITEX	0.110377	0.260562
ASML HOLDING	0.307927	0.286868
FLUTTER (DUB) ENTERTAINMENT	0.210059	0.327861
ENEL	0.220826	0.261580
DEUTSCHE POST (XET)	0.151716	0.245130
INFINEON TECHS. (XET)	0.256679	0.341433
AIRBUS	0.145112	0.367288
ANHEUSER-BUSCH INBEV	0.083602	0.281083
BNP PARIBAS	0.094237	0.324451
ENGIE	0.098907	0.254882
ING GROEP	0.091791	0.341695
DAIMLER (XET)	0.111802	0.313819
AMADEUS IT GROUP	0.162182	0.294035
BANCO SANTANDER	0.059009	0.346326
VINCI	0.161297	0.277433
KONE 'B'	0.184714	0.222930
ALLIANZ (XET)	0.163300	0.244253
BASF (XET)	0.102241	0.258750
BAYER (XET)	0.061599	0.285563
BMW (XET)	0.103811	0.276420
SIEMENS (XET)	0.137952	0.248039
ENI	0.074323	0.289188
MUENCHENER RUCK. (XET)	0.160634	0.240483
ADIDAS (XET)	0.223801	0.286854

DEUTSCHE TELEKOM (XET)	0.132699	0.217079
VOLKSWAGEN PREF. (XET)	0.084532	0.348711
SAP (XET)	0.151765	0.245918
VONOVIA (XET)	0.252296	0.223265
CRH	0.166114	0.301647
TOTALENERGIES	0.106257	0.284408
DANONE	0.111955	0.200381
KONINKLIJKE AHOLD DELHAIZE	0.168177	0.214563
LVMH	0.269063	0.264517
VIVENDI	0.152451	0.234147
L AIR LQE.SC.ANYME. POUR L ETUDE ET L EPXTN.	0.161970	0.209715
L'OREAL	0.192269	0.210567
PERNOD-RICARD	0.163860	0.203475
KERING	0.266813	0.297525
SAFRAN	0.183617	0.342829
INTESA SANPAOLO	0.127502	0.349019
ADYEN	0.269371	0.248170
PHILIPS ELTN.KONINKLIJKE	0.155973	0.234700
ESSILORLUXOTTICA	0.145379	0.241283
AXA	0.118504	0.283129
LINDE (XET)	0.159004	0.192932
PROSUS	0.127394	0.160012
SANOFI	0.117123	0.218967
SCHNEIDER ELECTRIC	0.171858	0.265480
IBERDROLA	0.222586	0.204038

C.2 GARCH estimated values

Table 14: Overview of the estimated values for α_0, α, and β for each asset in the considered universe.

Asset	α_0	α	β
OERSE (XET)	2.00861680e-05	1.21683289e-01	7.81371826e-01
INDITEX	4.95888727e-05	4.70508384e-03	7.96721333e-01
ASML HOLDING	8.02840616e-06	2.02741776e-02	9.52137534e-01
FLUTTER (DUB) ENTERTAINMENT	3.38858776e-05	1.11404290e-01	8.10439842e-01
ENEL	1.29214250e-05	1.20561065e-01	8.35631081e-01
DEUTSCHE POST (XET)	7.25034156e-06	6.03007219e-02	9.06444578e-01

INFINEON TECHS. (XET)	4.25833711e-05	6.61363875e-02	8.42262518e-01
AIRBUS	1.77262346e-05	1.21495215e-01	8.35849695e-01
ANHEUSER-BUSCH INBEV	1.05904167e-05	9.02344515e-02	8.73157364e-01
BNP PARIBAS	9.04161291e-06	1.25598912e-01	8.57501180e-01
ENGIE	7.14573646e-05	6.97478621e-03	6.82018708e-01
ING GROEP	9.93909209e-06	1.18513030e-01	8.64461178e-01
DAIMLER (XET)	3.36926920e-06	8.57794036e-02	9.07396331e-01
AMADEUS IT GROUP	5.74580281e-06	9.27288082e-02	8.90831244e-01
BANCO SANTANDER	1.62392093e-05	1.48983505e-01	8.26213223e-01
VINCI	8.29884253e-06	1.23718130e-01	8.50111314e-01
KONE 'B'	1.54431587e-05	1.27328129e-01	8.18273041e-01
ALLIANZ (XET)	3.85186915e-05	2.49866885e-01	5.52966624e-01
BASF (XET)	4.88877474e-05	3.00789145e-02	8.26661646e-01
BAYER (XET)	1.00500484e-05	4.33185415e-02	9.20071800e-01
BMW (XET)	4.67576444e-06	6.40324007e-02	9.19512150e-01
SIEMENS (XET)	1.35340445e-05	7.05262433e-02	8.69841689e-01
ENI	2.92526241e-06	1.06280677e-01	8.92995660e-01
MUENCHENER RUCK. (XET)	8.51294805e-06	1.29566257e-01	8.27605406e-01
ADIDAS (XET)	5.31556912e-05	7.64382489e-02	7.56287669e-01
DEUTSCHE TELEKOM (XET)	4.03134591e-06	7.98130353e-02	8.98090560e-01
VOLKSWAGEN PREF. (XET)	8.03337814e-06	9.97741250e-02	8.89824034e-01
SAP (XET)	3.19171494e-06	7.16701023e-02	9.27980918e-01
VONOVIA (XET)	5.33475311e-06	7.94677599e-02	8.94790638e-01
CRH	2.08893652e-05	1.05604482e-01	8.29705988e-01
TOTALENERGIES	3.88348956e-06	1.18515222e-01	8.76254937e-01
DANONE	7.31064835e-06	2.15543938e-03	9.51874186e-01
KONINKLIJKE AHOLD DELHAIZE	4.21941969e-05	1.54919513e-01	6.11448528e-01
LVMH	7.13938734e-06	6.19268344e-02	9.12454235e-01
VIVENDI	2.01717255e-05	1.17931383e-01	7.88899860e-01
L AIR LQE.SC.ANYME.	1.31828145e-05	1.44142411e-01	7.83343220e-01
L'OREAL	6.11013396e-06	7.42579380e-02	8.98736246e-01
PERNOD-RICARD	2.90936950e-06	6.34077531e-02	9.19564615e-01
KERING	2.13029226e-05	8.43066951e-02	8.55798107e-01
SAFRAN	1.03466956e-05	1.27841972e-01	8.43465382e-01
INTESA SANPAOLO	1.00711741e-05	1.69341782e-01	8.28452790e-01
ADYEN	3.41219908e-05	1.06364564e-01	8.02599570e-01
PHILIPS ELTN.KONINKLIJKE	8.78917015e-05	8.63093302e-03	5.84368652e-01

ESSILORLUXOTTICA	9.85576452e-06	6.04091077e-02	8.94453073e-01
AXA	1.75407651e-05	2.34554514e-01	7.27764119e-01
LINDE (XET)	1.00000000e-06	9.99900000e-01	4.49271320e-01
PROSUS	1.00000000e-06	9.99900000e-01	3.37975987e-01
SANOFI	5.16197578e-06	6.20560410e-02	9.12410222e-01
SCHNEIDER ELECTRIC	1.31332430e-05	1.07102568e-01	8.46905239e-01
IBERDROLA	4.24047470e-06	3.26262493e-03	9.67890130e-01