Assignment 3, due June 09, 2023

Topics in Advanced Time Series Econometrics

Prof. Dr. Marc Paolella

presented by

Migle Kasetaite: 21-733-779

Kilian Sennrich: 18-051-060

Abstract

This is our submission for Assignment 3 of the lecture *Topics in Advanced Time Series Econometrics*. In the first part, Value at Risk prediction using Students-t GARCH(1,1) is investigated. In the second part, The SMESTI distribution and the CCC-GARCH model are used to perform portfolio optimization.

1 Mixed-Normal GARCH

1.1 Theoretical Considerations

A GARCH(r, s) (Generalized Autoregressive Conditional Heteroskedasticity) process $\{R_t\}, t \in \mathbb{Z}$ is given by $R_t = \epsilon_t = Z_t \cdot \sigma_t$, where $\{Z_t\}$ refers to a sequence of i.i.d. random variables from location-zero, scale-one density $f_Z(\cdot)$, and scale parameter σ_t coming from the variance recursion (Paolella, 2018):

$$\sigma_t^2 = c_0 + \sum_{i=1}^{r} c_i \epsilon_{t-i}^2 + \sum_{j=1}^{s} d_j \sigma_{t-j}^2.$$

Mixed normal GARCH models are GARCH type models, that deviate in an important way from the previous structures. This class of models is beneficial because it gives rise to more complicated dynamics that result in better in-sample fits and, crucially, better risk and density forecasts (Paloella, 2018).

Consider a probability space $(\Omega; \mathcal{F}; \mathbb{P})$ equipped with the Filtration $\{\mathcal{F}\}_{t\in\mathbb{Z}}$ of its σ-algebra $\mathcal{F}$. Strictly following the notation from Paolella (2018), the p.d.f. of the k-component mixed normal distribution, denoted MixN$(\boldsymbol{\omega}, \boldsymbol{\mu}, \boldsymbol{\sigma})$ is given by

$$f_{\mathrm{MN}}(y; \boldsymbol{\omega}, \boldsymbol{\mu}, \boldsymbol{\sigma}) = \sum_{j=1}^{k} \omega_j \phi\left(y; \mu_j, \sigma_{jt}^2\right),$$

where ϕ is the Gaussian p.d.f., $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_k)' \in \mathbb{R}^k, \boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_k)' \in \mathbb{R}_{>0}^k$, and $\boldsymbol{\omega} = (\omega_1, \ldots, \omega_k)' \in (0,1)^k$ such that $\sum_{j=1}^{k} \omega_j = 1$. Note that $\mu_k = -\sum_{j=1}^{k-1} (\omega_j/\omega_k)\, \mu_j$ is imposed to ensure that the associated random variable has zero mean. A time series $\varepsilon = (\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_T)$ is generated by a k-component mixed normal GARCH(r,s) process, denoted MixN(k)-GARCH(r,s), if its conditional distribution of ε_t is a k-component mixed normal with zero mean,

$$\varepsilon_t \mid \mathcal{F}_{t-1} \sim \mathrm{MixN}\left(\boldsymbol{\omega}, \boldsymbol{\mu}, \boldsymbol{\sigma}_t\right)$$

where $\boldsymbol{\omega}$ and $\boldsymbol{\mu}$ are as above, $\boldsymbol{\sigma}_t = (\sigma_{1,t}, \ldots, \sigma_{k,t})'$. The component variances $\sigma_{i,t}^2, i = 1, \ldots, k$, follow the GARCH-like structure

$$\boldsymbol{\sigma}_t^{(2)} = \gamma_0 + \sum_{i=1}^{r} \gamma_i \varepsilon_{t-i}^2 + \sum_{j=1}^{s} \boldsymbol{\Psi}_j \sigma_{t-j}^{(2)},$$

where $\boldsymbol{\gamma}_i = (\gamma_{i,1}, \gamma_{i,2}, \ldots, \gamma_{i,k})', i = 0, \ldots, r$, are $k \times 1$ vectors, $\boldsymbol{\Psi}_j, j = 1, \ldots, s$, are $k \times k$ matrices, and $\boldsymbol{\sigma}_t^{(\delta)} = \left(\sigma_{1,t}^\delta, \sigma_{2,t}^\delta, \ldots, \sigma_{k,t}^\delta\right)'$, for $\delta \in \mathbb{R}_{>0}$. The parameters of the model need to be such that $\boldsymbol{\sigma}_t^{(\delta)} > 0$, where, in case of non-scalars, $>$ indicates element-wise inequality. As above, a (possibly time-varying) mean term can be incorporated, so that the model becomes (using c_t instead of μ_t as above for the mean) $R_t = c_t + \epsilon_t$ (p. 478, Paolella, 2018). Code-Section 1 & 2 in the Appendix hold Python code to simulate the p.d.f. as well as RVs of *any* Mixture distribution. That is, k can be chosen arbitrarily and with slight modifikation, different distributions can be added to the mixture. Code-Section 3 can be used to generate the component variances using the mixture distribution for the errors.

1.2 Simulation

A simulation was conducted to investigate the suitability of the MixN(k)-GARCH(r, s) for Value at Risk (VaR) predictions as described in Hull and White (1998) and Barone-Adesi et al. (1999). The mixture was simulated with two normals $X \sim \mathcal{N}(-0.1, 0.3)$ and $\omega_X = 0.6$ and $Y \sim \mathcal{N}(0.2, 0.1)$ with $\omega_Y = 0.4$. $r = 1$ and $s = 1$ along with $c_0 = 0.04$, $c_1 = 0.05$ and $d_1 = 0.08$ was used to simulate the volatility with $T = 1000$ observations based on the innovations from the mixture. The volatility plotted over time is displayed in Figure 5. Volatility was then multiplied with $z \sim \mathcal{N}(0, 1)$ to arrive at a stationary time series mimicking a financial returns time series. The simulated returns are displayed in Figure 2.

The one-step-ahead prediction $VaR_{t+1|t}$ of the Student-t-GARCH model, corresponds to a univariate Student t distribution, where the scale parameter is determined by the GARCH recursion. In the following the experiment is outlined step-by-step:

1. Commencing from time point $t = 250$ and progressing up to $t = 999$, the Students-t-GARCH(1,1) model was estimated using data points of the returns.
2. The parameters from the fitted GARCH model to the time series of returns were used to derive the deterministic GARCH forecast of the scale term $\hat{\sigma}_{t+1}$, and also the filtered innovation sequence $\{\hat{Z}\}_t$.
3. Using the estimate of the degrees of freedom of the Student-t distribution, the 1% quantile $q_{0.01}$ of the estimated model density was derived.
4. The VaR was then derived using the following equality: $VaR_{t+1|t} = \hat{\mu}_{t+1|t} + \hat{\sigma}_{t+1|t} \cdot q_{0.01}$

The result is displayed in Figure 3. In particular, both the returns series (blue) and the series of $VaR_{t+1|t}$ (green) each for $t > 249$ and $t < 1000$ are shown. For each t where the return was smaller than $VaR_{t+1|t}$, a red point is plotted. This happended a total of *5* times. Notice that with infinite repetitions, this value converges to 7.5. Our simulation experiment provides further evidence for the practical usefulness of Hull and Whites (1998) VaR calculation using filtered historical simulation.

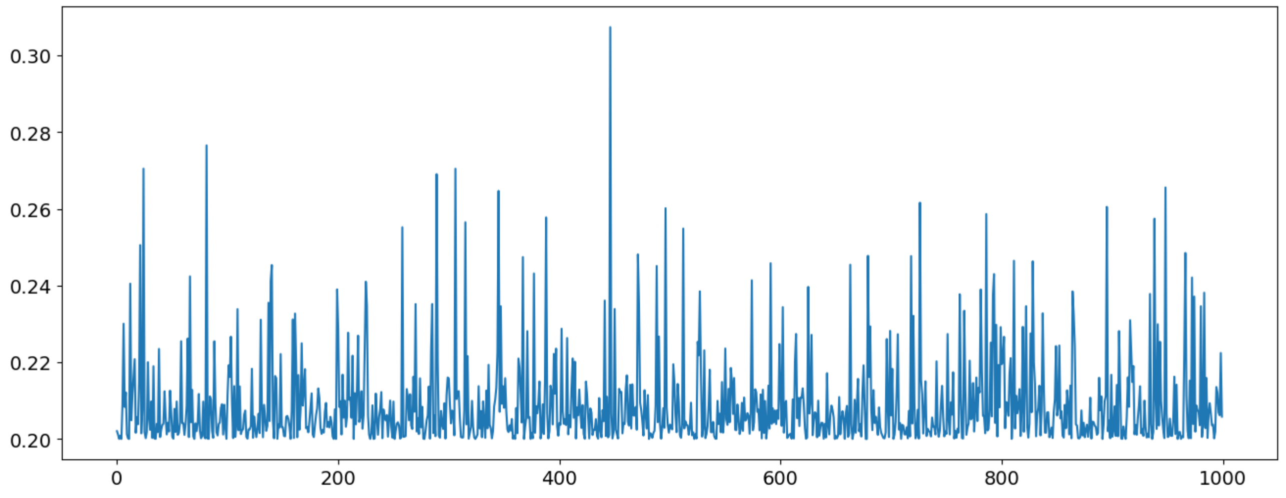

Figure 1: Volatility process using the MixN(k) distribution for the innovation process and a GARCH(1,1) with k = 2 and, $\omega_1 = 0.6, \mu_1 = -0.1, \sigma_1^2 = 0.3$ and $\omega_2 = 0.4, \mu_2 = 0.2, \sigma_2^2 = 0.1$. In accordance with Paolella, 2018, c_0 was chosen as 0.04, c_1 was chosen as 0.05 and d_1 was chosen as 0.8. A total of $T = 1'000$ timesteps were simulated.

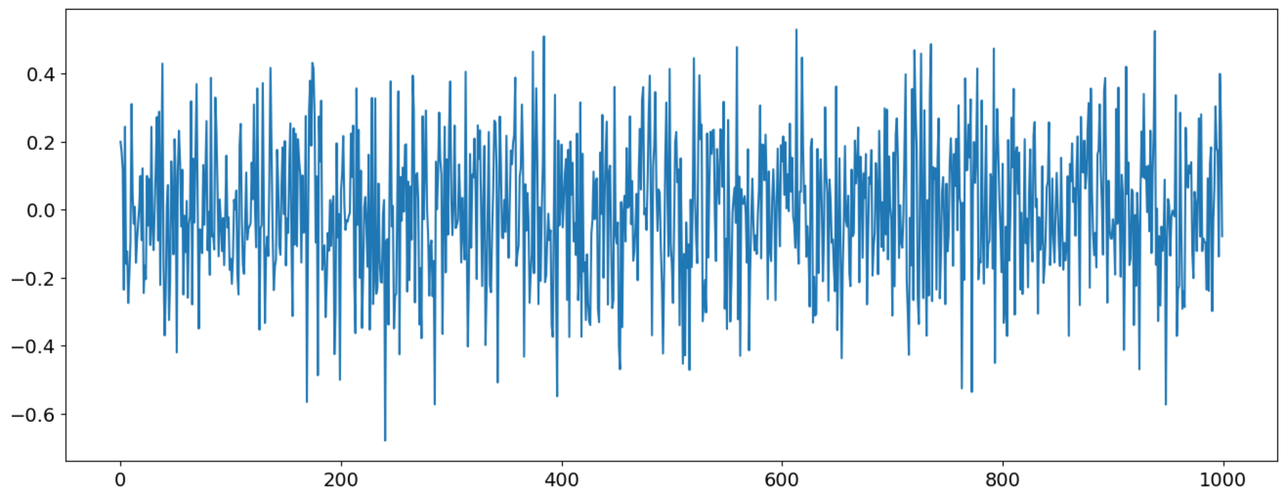
Figure 2: Simulated returns resulting from $R_t = Z_t + \sigma_t$ with $Z \sim \mathcal{N}(0, 1)$

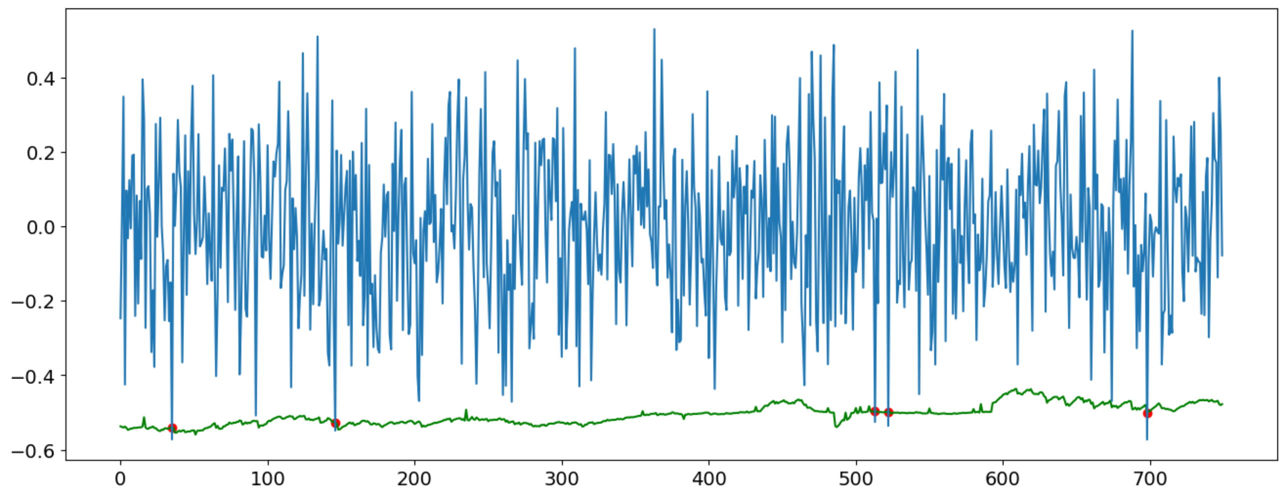
Figure 3: 1% $VaR_{t+1|t}$ (green) of the simulated returns. The return series (blue) includes the last 750 observations of the series displayed in Figure 3. The five times the return at time t was smaller than the 1% $VaR_{t+1|t}$ are indicated with the red dots.

2 Constant Conditional Correlation (CCC-GARCH)

In this part we have to run CCC-GARCH model on a multivariate (5-variate) time series data simulated by SMESTI model and do the mean-variance portfolio optimization.

2.1 SMESTI Distribution

First task was to simulate 5-variate time series data from SMESTI model.

SMESTI Distribution (symmetric marginally endowed Student's t: independent case) is a special case of MEST (marginally endowed Student's t). It is frequently employed in statistical modeling when working with data that could display heavy-tailed behavior or when the multivariate normality assumption is broken. Additionally, applications like portfolio optimization frequently use the symmetric marginally endowed Student's t-distribution, when returns on various assets are thought to be independently distributed with symmetric marginal distributions.

Consider taking $G_i \sim \text{IGam}\left(\frac{k_i}{2}, \frac{k_i}{2}\right)$, $i = 1, \dots, d$, with $k = (k_1, k_2, \dots, k_d)' \in R_+^d$ and $m(G) = \mu$. Then $\boldsymbol{X} = \boldsymbol{\mu} + \boldsymbol{D}\boldsymbol{R}^{1/2}\boldsymbol{Z}$, implying $(X \mid G = g) \sim \mathcal{N}(\mu, DRD)$. Here:

$Z \sim \mathcal{N}(0, I)$

$D = \text{diag}\left(\left[{G_1}^{1/2}, {G_2}^{1/2}, \dots, {G_d}^{1/2}\right]\right)$

$G = (G_1, G_2, \dots, G_d)'$ is a vector of possibly dependent non-negative continuous scalar-valued random variables (but still independent of Z)

R is a positive definite correlation matrix

$m\colon \mathbb{R}^d \to \mathbb{R}^d$ is a measurable mean function

Using the above formulas we have simulated T = 1000 data point for 5 stock return series that jointly follow the SMESTI distribution. We picked degrees of freedom to be k = [4, 4, 4, 4, 4] and generated random correlation matrix R with pre-defined eigenvalues, which will have to be estimated by CCC-GARCH in the next part of the task. Python implementation can be found in Code-Section 6.

	Stock 1	Stock 2	Stock 3	Stock 4	Stock 5
Stock 1	1	0.174	-0.279	0.083	0.007
Stock 2	0.174	1	-0.002	-0.017	0.182
Stock 3	-0.279	-0.002	1	-0.031	-0.311
Stock 4	0.083	-0.017	-0.031	1	-0.21
Stock 5	0.007	0.182	-0.311	-0.21	1

Table 1: Random correlation matrix R, generated for SMESTI distribution.

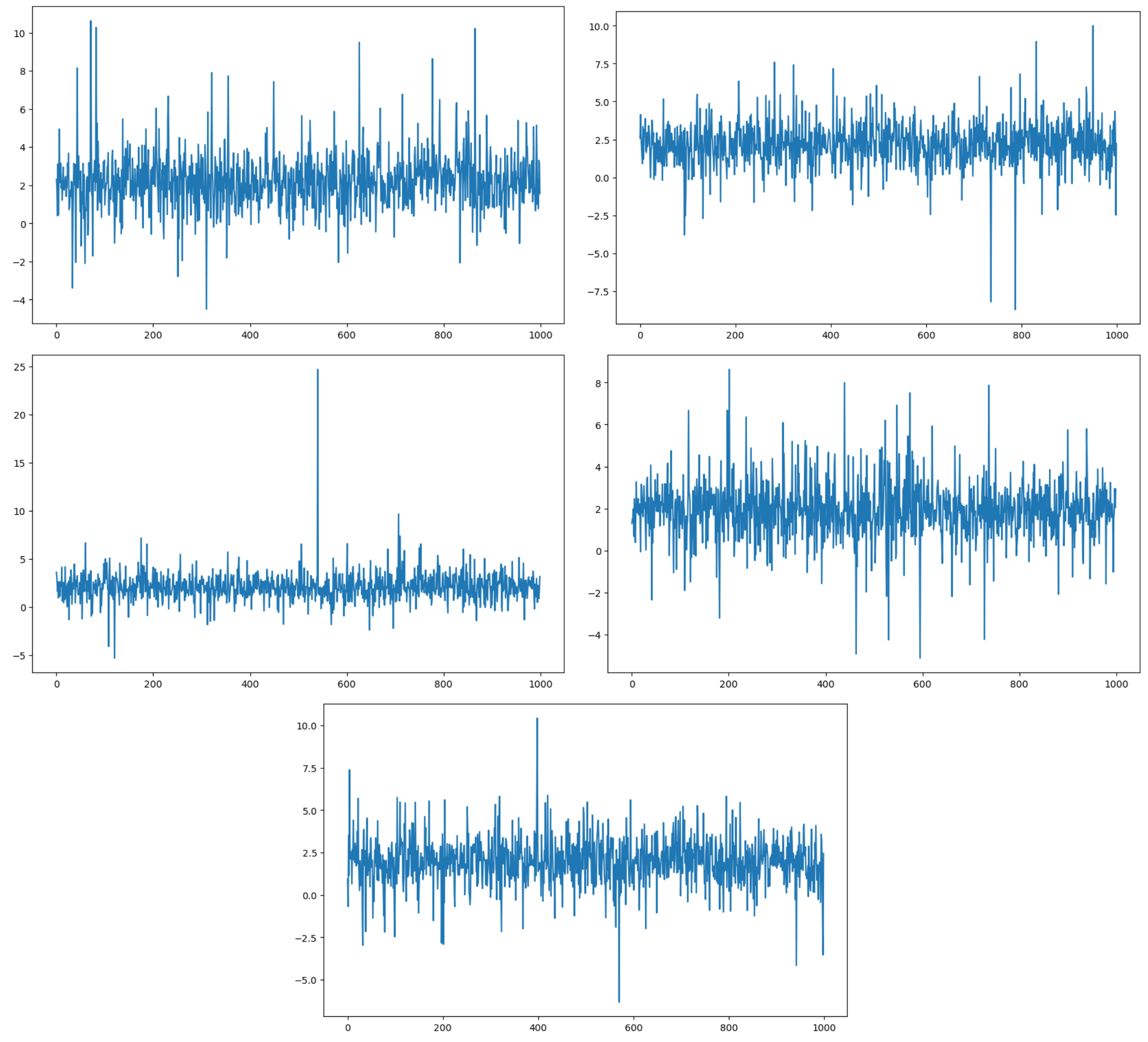

Figure 4: Five stock return series generated by SMESTI distribution. Here T = 1000 and k = [4,4,4,4].

2.2 CCC-GARCH

Second task is to fit data simulated by SMESTI distribution to CCC-GARCH model in order to compute correlation parameters that are later used in portfolio optimization task.

A multivariate volatility modeling approach called CCC-GARCH (Constant Conditional Correlation GARCH) focuses on estimating and forecasting the conditional correlations between various assets. While permitting the conditional variances to change, it makes the assumption that the conditional correlations won't change over time. For the purpose of capturing the interdependencies and volatility dynamics among various assets, CCC-GARCH models are frequently used in risk management, portfolio creation, and asset allocation.
CCC model of Bollerslev (1990) is arguably the most often used technique for producing an MGARCH model via univariate GARCH. A univariate GARCH model is fitted for each component series, and the filtered innovations are arranged as columns in a matrix. To estimate the correlations, this matrix's sample correlation is used. So, in two ways, our MGARCH model

satisfies the required aspect of ease of estimation.

First, when it comes to univariate GARCH models, these only need joint estimate of two (for the Gaussian IGARCH case) to five (for the Gaussian APARCH case) parameters apiece, and the optimization may be parallelized across assets, which would further reduce processing time.
Second, using the sample correlation estimator on the matrix of filtered innovations, set of correlation parameters is computed quickly and easily. This, however, requires that the correlations remain stable throughout time.

Let $\varepsilon_t = Z_t$, where $\varepsilon_t = (\varepsilon_{t,1}, \ldots, \varepsilon_{t,d})'$. We abbreviate $Y_{t|\Omega_{t-1}}$, where Ω_t is, the information set at time t, as just $Y_{t|t-1}$.
The CCC model can then be expressed as:

$$Y_{t|t-1} \sim N_d(\mu, H_t), \quad H_t = D_t R D_t \tag{1}$$

where $\mu = (\mu, \ldots, \mu)'$, $D^2 = \text{diag}([\sigma^2, \ldots, \sigma^2])$, and R is the set of $d \times d$ matrices of time-varying conditional correlations with dynamics specified by:

$$R_t := E[\varepsilon\varepsilon' \mid \Omega_{t-1}] = \text{diag}(Q_t)^{1/2} Q_t \text{diag}(Q_t)^{1/2} \tag{2}$$

$t = 1, \ldots, T$. Observe that

$$\varepsilon_t = D_t^{-1}(Y_t - \mu) \tag{3}$$

The $\{Q_t\}$form a sequence of conditional matrices parameterized by:

$$Q_t = S \tag{4}$$

with S being the $d \times d$ unconditional correlation matrix (Engle, 2002, p. 341) of the ε_t. Matrix S can be estimated with the usual plug-in sample correlation based on the filtered ε_t.

First, from our Python implementation of univariate GARCH(1,1), the below parameters were estimated:

	μ	ω	α_1	β_1
Stock 1	1.841	0.102	0.000	0.943
Stock 2	2.112	0.210	0.000	0.899
Stock 3	1.971	0.123	0.006	0.927
Stock 4	2.129	1.572	0.000	0.295
Stock 5	2.184	0.321	0.019	0.812

Table 2: Model parameters estimated by univariate GARCH(1,1)

Additionally, we retrieved conditional volatilises and standardized residuals for each stock return series. Code implementation can be found in Code-Section 7.
Second, we estimated stock correlation matrix R (equation (2)) that will be used for optimizing portfolio. Python implementation can be found in Code-Section 8.

	Stock 1	Stock 2	Stock 3	Stock 4	Stock 5
Stock 1	1	0.188	-0.167	0.068	0.039
Stock 2	0.188	1	0.005	-0.003	0.089
Stock 3	-0.167	0.005	1	-0.038	-0.301
Stock 4	0.068	-0.003	-0.038	1	-0.161
Stock 5	0.039	0.089	-0.301	-0.161	1

Table 3: Correlation Matrix R estimated by CCC-GARCH(1,1)

Comparing the original correlation matrix R, generated for SMESTI Distribution (Table 1) and the one estimated by CCC-GARCH (Table 3), we can conclude that CCC-GARCH did quite a good job and almost correctly captured negative and positive correlations between different stocks, as well as correlation coefficients.

The CCC-GARCH model assumes that the conditional correlations between assets remain constant over time. In practise, correlations can alter as a result of numerous market circumstances, occasions, or economic considerations. As a result, time-varying correlations might not be accurately captured by the model.
One way to improve CCC-GARCH model accurracy is incorporating moving windows. This enables computational efficiency, reduced data dependency, higher forecasting accuracy, and adaptive estimation. It gives more recent and precise estimates of conditional correlations and volatilities and aids in capturing time-varying market relationships.
Alternatively, one can use a DCC-GARCH model instead of assuming constant conditional correlations. The DCC model better captures the dynamics of the correlation structure by allowing the conditional correlations to change over time. Using prior observations of the standardized residuals, it calculates the matrix of conditional correlation that changes over time.

2.3 Portfolio Optimization

In this part, mean-variance portfolio optimization is performed using the correlation matrix estimated by CCC-GARCH.

A methodology created by Harry Markowitz is known as mean-variance portfolio optimization (MVO), which is frequently linked to Modern Portfolio Theory (MPT). It seeks to build the best portfolio possible by balancing the projected rewards and risks of various assets. The objective is to identify the asset allocation that either maximizes expected portfolio return for a given level of risk or reduces risk for a given level of expected return. In order to efficiently generate a frontier of ideal portfolios, mean-variance portfolio optimization considers the predicted returns, variances, and covariances of the assets.
Using the mean-variance optimization framework and the estimated conditional correlations and volatilities from a CCC-GARCH model, one can create portfolios that not only take expected returns and risks into account, but also capture the changing market relationships and volatility patterns. A stronger and more efficient portfolio allocation that can adjust to shifting market conditions and offer better risk-return trade-offs can result from this integration.

Consider a group of d financial assets, such as highly liquid equities on major exchanges, for which returns are observed over a particular length of time and frequency (e.g., daily), and suppose that short-selling will not be used (as is customary in many real-world scenarios). The set of valid portfolio weights is thus

$$\mathcal{W} = \{w \in [0,1]^d : 1^T w = 1\}.$$

Returns are considered to be an i.i.d. multivariate normal process with mean mu and variance $Sigma$ in the conventional portfolio optimization framework. One wishes to determine the portfolio weight vector, say $w*$, that yields the lowest variance of the predictive portfolio percentage return at time $t+1$, given information up to time t, say $P_{t+1|t}$, conditional on its expected value being greater than some positive threshold τ_{daily}. That is,

$$w^* = \arg\min_{w \in \mathcal{W}} \mathbb{V}(P_{t+1|t}, w), \text{ such that } \mathbb{E}[P_{t+1|t}, w] \geq \tau_{\text{daily}},$$

$$\tau_{\text{daily}} = 100\left(\left(1 + \frac{\tau}{100}\right)^{\frac{1}{250}} - 1\right), \quad \tau = 100\left(\left(1 + \frac{\tau_{\text{daily}}}{100}\right)^{250} - 1\right)$$

for $\tau = \tau_{\text{annual}}$ the desired annual percentage return, here calculated assuming 250 business days per year. In this i.i.d. Gaussian Markowitz setting, note that

$$\hat{\mathbb{E}}[P_{t+1|t}, w] = w^T \hat{\mu}, \qquad \hat{\mathbb{V}}(P_{t+1|t}, w) = w^T \hat{\Sigma} w,$$

where $\hat{\mu}$ and $\hat{\Sigma}$ refer to the usual plug-in unbiased estimators of the Gaussian distribution parameters.

For optimizing mean-variance d = 5 portfolio weights we used correlation matrix generated by CCC-GARCH insted of covariance of stock returns. We employed 2 methods for Markowitz portfolio optimization: Quadratic Programming (Code-Section 9) and Monte Carlo Simulation (Code-Section 10).
Quadratic Programming: This method determines the optimal portfolio using a mathematical model. Typically, the goal is to maximize return for a given level of risk (or, conversely, decrease risk for a given level of return - our implementation). This is a constrained optimization problem, with the constraints requiring that the portfolio weights be non-negative and add up to one. QP is efficient and delivers an exact solution if the problem is well-posed and the inputs are not ambiguous. The fundamental restriction is that QP is based on the assumption that returns are normally distributed and that the relationships between assets can be adequately characterized by pairwise covariances (correlation in our case).
Monte Carlo Simulation: This is a simulation-based technique that considers hundreds or even millions of alternative situations. Random portfolio weights are generated for each scenario, and the portfolio's return and risk are calculated. The best portfolio is then picked depending on certain criteria (in our case the portfolio with the highest Sharpe ratio). The fundamental advantage of the MC approach is that it does not rely on assumptions of normalcy or specific asset arrangements. It is capable of dealing with complex, nonlinear interactions and non-normal distributions. The MC method's accuracy, on the other hand, is proportional to the number of simulations: the more simulations, the better the estimate, but the longer the computation time.
Analyzing the data presented in Table 4, it's apparent that both Quadratic Programming and Monte Carlo methods provide strikingly similar results in terms of portfolio optimization. However, it's worth noting that the Monte Carlo method tends to be somewhat more computationally demanding, indicating a trade-off between computational efficiency and the flexibility that this method offers.

	Stock 1	Stock 2	Stock 3	Stock 4	Stock 5
Quadratic Programming	19.45 %	6.7 %	28.56 %	19.28 %	26.01 %
Monte Carlo	20.67 %	7.67 %	26.59 %	19.93 %	25.14 %

Table 4: Optimal portfolio weights retrieved by Quadratic Programming and Monte Carlo method.

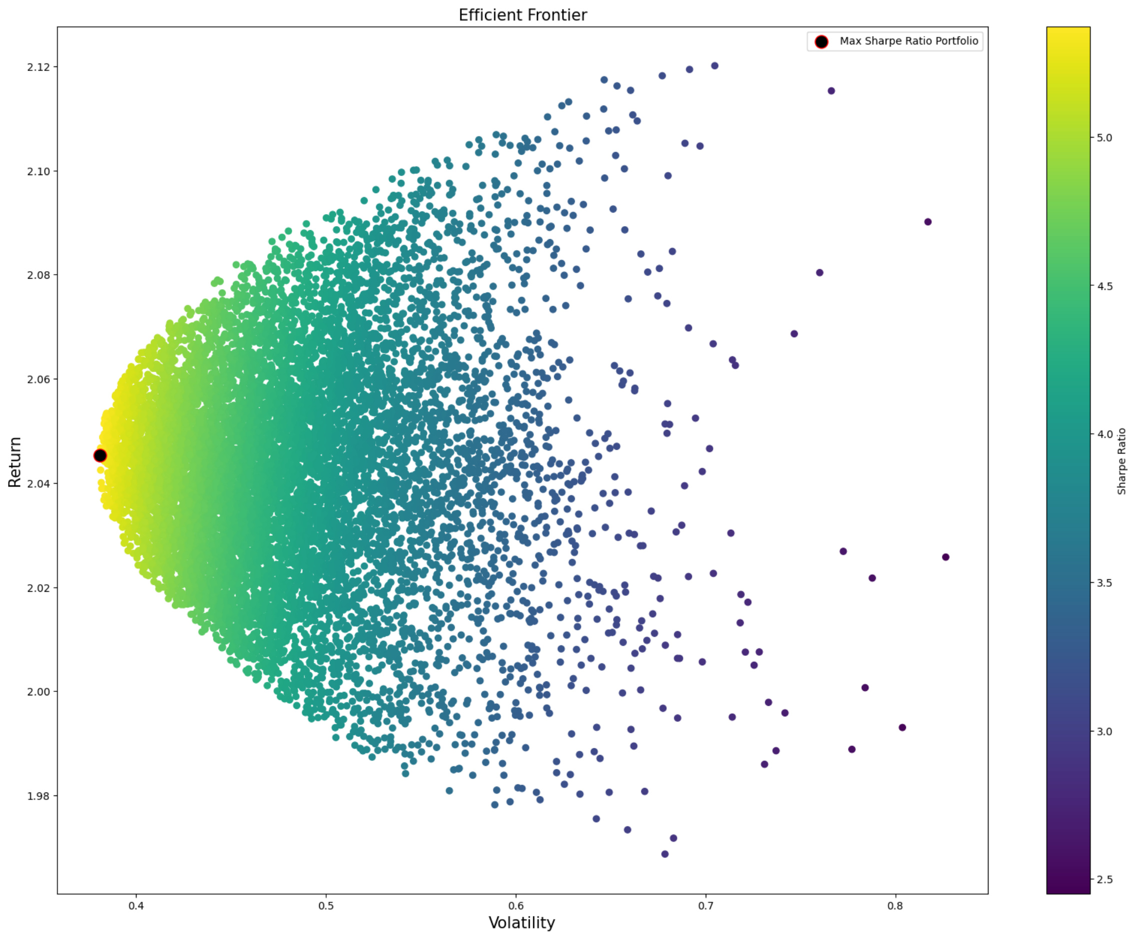

Figure 5: Portfolio optimization using Monte Carlo simulation method. Each dot corresponds to portfolio volatility and expected return computed by randomly selected weights. Portfolio marked in black is an optimal portfolio (Max Sharpe Ratio).

3 Bibliography

Paolella, M. S. (2018). *Linear models and time-series analysis: regression, ANOVA, ARMA and GARCH*. John Wiley & Sons.

Robert Engle (2002). *Dynamic Conditional Correlation*. Journal of Business & Economic Statistics, 20:3, 339-350. DOI: 10.1198/073500102288618487

Hull, J.,& White, A. (1998). Incorporating volatility updating into the historical simulation method for value-at-risk.*Journal of risk, 1*(1), 5-19.

Barone-Adesi, G., Giannopoulos, K., & Vosper, L. (1999). VaR without correlations for portfolios of derivative securities. *Journal of Futures Markets, 19*(5), 583-602.

4 Appendix

```python
class MixtureModel(rv_continuous):
    def __init__(self, submodels, omegas, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.submodels = submodels
        self.omegas = omegas

        if len(self.omegas) != len(self.submodels):
            raise ValueError('omega must have the same length as submodels')

        if sum(self.omegas) != 1:
            raise ValueError('omega must sum to 1')

    def _pdf(self, x):

        # Compute pdf of the initial distribution
        pdf = self.omegas[0] * self.submodels[0].pdf(x)

        # Add each additional distribution weighted by its omega to the pdf
        for omega, submodel in enumerate(self.submodels[1:]):
            pdf += self.omegas[1+omega] * submodel.pdf(x)
        return pdf

    def rvs(self, size):
        submodel_choices = np.random.choice(len(self.submodels), size=size,
                                            p=self.omegas)
        submodel_samples = [submodel.rvs(size=size) for submodel in self.
                                            submodels]
        rvs = np.choose(submodel_choices, submodel_samples)
        return rvs
```

Code-Section 1: Scipy-Stats has a base class for mixtures of RVs. Our code inherits from this class and constructs the pdf and its RV as described in Paolella (2018).

```
def MixN(omega: list, mu: list,  sigma: list, n: int, method: str = "rvs"):

    # throw errors if the mu, omega and sigma lists do not have the same
    # length, that is, if some parameters are missing.
    if len(omega) != len(sigma) != len(mu):
        raise ValueError('omega, mu and sigma must have the same length')

    if method not in ["rvs", "pdf"]:
        raise ValueError('method must be either "rvs" or "pdf"')

    # init the gaussians for the mixture from the input lists.
    submodels = [stats.norm(mu[i], sigma[i]) for i in range(len(mu))]
    mixture_gaussian_model = MixtureModel(submodels=submodels, omegas=omega)

    if method == "pdf":
        return mixture_gaussian_model.pdf(n)

    if method == "rvs":
        return mixture_gaussian_model.rvs(n)
# e.g.
mixture_gauss_process = MixN([0.1, 0.9], [1, 0], [1, 1], 1000, method="rvs")
# or
mixture_gauss_process2 = MixN([0.1, 0.6, 0.3], [1, 0, 2], [1, 1, 0.5], 1000,
                              method="rvs")
```

Code-Section 2: Code incorporating the class from Code-Section 1 for a pdf/RVs from a mixture distrubtion. The function is fully scalable: adding more parameters to the lists "omega", "mu" and "sigma". Will initiate a mixture with more gaussian baseline distributions.

```
def GARCH_volatility(r: int, s: int, omega: float,
                     alpha: list, beta: list, T: int):

    sigma = np.zeros(T)
    e = MixN([0.6, 0.4], [-0.1, 0.2], [0.3, 0.1], T, method="rvs")

    for t in range(T):
        sigma[t] = omega +
                   sum(( alpha[i] * (e[t-i]**2) ) for i in range(0, r)) +
                   sum(( beta[j] * (sigma[t-j]**2) ) for j in range(0, s))
        sigma = np.sqrt(sigma)
    return sigma, e

# e.g.
volatility, e = GARCH_volatility(3, 2, 0, [0.1, 0.6, 0.9], [0.9, 0.01], 1000
                                 )
```

Code-Section 3: Generates the component variances using the mixture distribution for the errors.

```
# generates a mixN(k)-Garch(r, s) process
Z = np.random.normal(loc=0, scale=1, size=1000)

garch_process = Z*volatility
```

Code-Section 4: Generates a MixN(k)-GARCH(r, s) process is defined as $R_t = \epsilon_t = Z_t \cdot \sigma_t$

```
garch_process2 = pd.DataFrame(garch_process)

pred = list()
for i in range(249,999):
    # init the model and fit it to the simulated returns
    am = arch_model(garch_process2, vol="Garch", p=1, o=0, q=1, dist="
                                             StudentsT")
    res = am.fit(disp="off", show_warning=False, last_obs=i)

    # one-step ahead forecast
    forecasts = res.forecast(start=i+1, reindex=False)
    cond_mean = forecasts.mean
    cond_var = forecasts.variance

    # 1\% quantile of the student t
    q = am.distribution.ppf([0.01], res.params[-1])

    # compute the VaR
    value_at_risk = cond_mean.values + np.sqrt(cond_var).values * q[None:]

    pred.append(value_at_risk[0])

# unpack numpy arrays
pred = [i[0] for i in pred]
```

Code-Section 5: Experiment to derive number of times, the return at time t was below the $VaR_{t+1|t}$.

```
import numpy as np
from scipy.stats import random_correlation
from scipy.linalg import sqrtm
from arch import arch_model
import pandas as pd
import cvxpy as cp



T = 1000
d = 5
k = np.array([4, 4, 4, 4, 4])

Y = np.random.gamma(k/2, scale=1, size=(T,d)) / (k/2)
G = 1/Y

sqrtG = np.sqrt(G)

Z = np.random.normal(loc=0.0, scale=1.0, size=(T,d))

#Random positive definite correlation matrix R
rng = np.random.default_rng()
R = random_correlation.rvs((.5, .8, 1.2, 1.5, 1), random_state=rng)
sqrtR = sqrtm(R)


# computing sigma; sigma=m(G)
sigma = G.mean(0)

#initializing X matrix
X = np.zeros((T,d))

for i in range(T):
    D = np.diag(sqrtG[i])
    z = Z[i]
    # computing samples
    X[i] = sigma + np.dot(D, np.dot(sqrtR,z))
    i += 1
```

Code-Section 6: Generates 5 stock return series that jointly follow the SMESTI distribution with DoF = 4 for all stocks.

```
# define lists for storing objects
coeffs = []
cond_vol = []
std_resids = []
models = []

# estimate the univariate garch models

for i in range(d):
    model = arch_model(X[:,i], mean = 'Constant', vol = 'GARCH', p = 1, o =
                                             0, q = 1).fit(update_freq = 0,
                                                 disp = 'off')

    #getting GARCH coefficients for all 5 time series
    coeffs.append(model.params)
    # Getting conditional standard deviations
    cond_vol.append(model.conditional_volatility)
    # Standardizing residuals
    std_resids.append(model.resid / model.conditional_volatility)
    models.append(model)

# store the results in df

coeffs_df = pd.DataFrame(coeffs, index=["X_1", "X_2", "X_3", "X_4", "X_5"])
cond_vol_df = pd.DataFrame(cond_vol, index=["X_1", "X_2", "X_3", "X_4", "X_5
                                       "]).transpose()
std_resids_df = pd.DataFrame(std_resids, index=["X_1", "X_2", "X_3", "X_4",
                                       "X_5"]).transpose()

print(coeffs_df)
```

Code-Section 7: Fitting 5 stock return data to univariate GARCH(1,1) model and getting GARCH parameters and standartized residuals.

```
#unconditional correlation matrix of the standartized residuals
Q = np.cov(std_resids_df, rowvar=False)
diag_Q = np.diag(Q)
inv_diag_Q = np.diag(1 / np.sqrt(diag_Q))

# matrix of time varying conditional correlations
R = np.dot(inv_diag_Q, np.dot(Q, inv_diag_Q))
print(R)
```

Code-Section 8: Estimates correlation matrix of the standardized residuals obtained from Code-Section 7. The correlation matrix is assumed to be constant over time.

```
mu = X.mean(0)
Sigma = R

# Number of stocks
n = 5

# The variables vector
w = cp.Variable(n)

# The minimum return
req_return = 0.05

# The return
ret = mu.T @ w

# The risk in wT.R.w format
risk = cp.quad_form(w, Sigma)

# The core problem definition with the Problem class from CVXPY
prob = cp.Problem(cp.Minimize(risk), [sum(w)==1, ret >= req_return, w >= 0])

stocks = ['Stock 1','Stock 2', 'Stock 3', 'Stock 4', 'Stock 5']

try:
    prob.solve()
    print ("Optimal portfolio")
    print ("----------------------")
    for s in range(len(stocks)):
       print (" Investment in {} : {}% of the portfolio".format(stocks[s],
                                                    round(100*x.value[s],2)))
    print ("----------------------")
    print ("Exp ret = {}%".format(round(100*ret.value,2)))
    print ("Expected risk    = {}%".format(round(100*risk.value**0.5,2)))
except:
    print ("Error")
```

Code-Section 9: Markowitz No Short (selling) portfolio optimization using CVXPT python package for minimizing portfolio risk where minimum required daily return is 5%. Here *Sigma* is correlation matrix generated by CCC-GARCH and *mu* is mean of each five stock returns.

```
# number of simulations
n = 10000

# initialiing arrays
port_weights = np.zeros(shape=(n,len(X[1,])))
port_volatility = np.zeros(n)
port_sr = np.zeros(n)
port_return = np.zeros(n)

num_securities = len(X[1,])

# run the simulation
for i in range(n):
    # weight each stock
    weights = np.random.random(5)

    # normalize weights
    weights /= np.sum(weights)
    port_weights[i,:] = weights

    # expected returns
    exp_ret = X.mean(0).dot(weights)
    port_return[i] = exp_ret

    # expexted volatility (risk)
    exp_vol = np.sqrt(weights.T.dot(R.dot(weights)))
    port_volatility[i] = exp_vol

    # sharpe ratio (expected returns / expected risk)
    sr = exp_ret / exp_vol
    port_sr[i] = sr


# Index of max Sharpe Ratio
max_sr = port_sr.max()
ind = port_sr.argmax()

# Return and Volatility values of max Sharpe Ratio
max_sr_ret = port_return[ind]
max_sr_vol = port_volatility[ind]

# plotting simulated portfolio volatility and return values and marking
                                       optimal portfolio
plt.figure(figsize=(20,15))
plt.scatter(port_volatility,port_return,c=port_sr, cmap='viridis')
plt.colorbar(label='Sharpe Ratio')
plt.xlabel('Volatility', fontsize=15)
plt.ylabel('Return', fontsize=15)
plt.title('Efficient Frontier', fontsize=15)
plt.scatter(max_sr_vol, max_sr_ret, c='black', s=150, edgecolors='red',
                                       marker='o', label='Max \
Sharpe Ratio Portfolio')
plt.legend()
```

Code-Section 10: Markowitz mean variance portfolio optimization using Monte Carlo Method with n = 10000 simulations.