program

December 16, 2024

[352]:
```
#%pip install yfinance numpy statsmodels pandas matplotlib arch openpyxl
%pip install nbconvert
```

```
Collecting nbconvert
  Downloading nbconvert-7.16.4-py3-none-any.whl.metadata (8.5 kB)
Requirement already satisfied: beautifulsoup4 in c:\programming\risk-management-
project\.venv\lib\site-packages (from nbconvert) (4.12.3)
Collecting bleach!=5.0.0 (from nbconvert)
  Downloading bleach-6.2.0-py3-none-any.whl.metadata (30 kB)
Collecting defusedxml (from nbconvert)
  Downloading defusedxml-0.7.1-py2.py3-none-any.whl.metadata (32 kB)
Collecting jinja2>=3.0 (from nbconvert)
  Downloading jinja2-3.1.4-py3-none-any.whl.metadata (2.6 kB)
Requirement already satisfied: jupyter-core>=4.7 in c:\programming\risk-
management-project\.venv\lib\site-packages (from nbconvert) (5.7.2)
Collecting jupyterlab-pygments (from nbconvert)
  Downloading jupyterlab_pygments-0.3.0-py3-none-any.whl.metadata (4.4 kB)
Collecting markupsafe>=2.0 (from nbconvert)
  Downloading MarkupSafe-3.0.2-cp313-cp313-win_amd64.whl.metadata (4.1 kB)
Collecting mistune<4,>=2.0.3 (from nbconvert)
  Downloading mistune-3.0.2-py3-none-any.whl.metadata (1.7 kB)
Collecting nbclient>=0.5.0 (from nbconvert)
  Downloading nbclient-0.10.1-py3-none-any.whl.metadata (8.2 kB)
Collecting nbformat>=5.7 (from nbconvert)
  Downloading nbformat-5.10.4-py3-none-any.whl.metadata (3.6 kB)
Requirement already satisfied: packaging in c:\programming\risk-management-
project\.venv\lib\site-packages (from nbconvert) (24.2)
Collecting pandocfilters>=1.4.1 (from nbconvert)
  Downloading pandocfilters-1.5.1-py2.py3-none-any.whl.metadata (9.0 kB)
Requirement already satisfied: pygments>=2.4.1 in c:\programming\risk-
management-project\.venv\lib\site-packages (from nbconvert) (2.18.0)
Collecting tinycss2 (from nbconvert)
  Downloading tinycss2-1.4.0-py3-none-any.whl.metadata (3.0 kB)
Requirement already satisfied: traitlets>=5.1 in c:\programming\risk-management-
project\.venv\lib\site-packages (from nbconvert) (5.14.3)
Requirement already satisfied: webencodings in c:\programming\risk-management-
project\.venv\lib\site-packages (from bleach!=5.0.0->nbconvert) (0.5.1)
Requirement already satisfied: platformdirs>=2.5 in c:\programming\risk-
```

```
management-project\.venv\lib\site-packages (from jupyter-core>=4.7->nbconvert)
(4.3.6)
Requirement already satisfied: pywin32>=300 in c:\programming\risk-management-
project\.venv\lib\site-packages (from jupyter-core>=4.7->nbconvert) (308)
Requirement already satisfied: jupyter-client>=6.1.12 in c:\programming\risk-
management-project\.venv\lib\site-packages (from nbclient>=0.5.0->nbconvert)
(8.6.3)
Collecting fastjsonschema>=2.15 (from nbformat>=5.7->nbconvert)
  Downloading fastjsonschema-2.21.1-py3-none-any.whl.metadata (2.2 kB)
Collecting jsonschema>=2.6 (from nbformat>=5.7->nbconvert)
  Downloading jsonschema-4.23.0-py3-none-any.whl.metadata (7.9 kB)
Requirement already satisfied: soupsieve>1.2 in c:\programming\risk-management-
project\.venv\lib\site-packages (from beautifulsoup4->nbconvert) (2.6)
Collecting attrs>=22.2.0 (from jsonschema>=2.6->nbformat>=5.7->nbconvert)
  Downloading attrs-24.3.0-py3-none-any.whl.metadata (11 kB)
Collecting jsonschema-specifications>=2023.03.6 (from
jsonschema>=2.6->nbformat>=5.7->nbconvert)
  Downloading jsonschema_specifications-2024.10.1-py3-none-any.whl.metadata (3.0
kB)
Collecting referencing>=0.28.4 (from jsonschema>=2.6->nbformat>=5.7->nbconvert)
  Downloading referencing-0.35.1-py3-none-any.whl.metadata (2.8 kB)
Collecting rpds-py>=0.7.1 (from jsonschema>=2.6->nbformat>=5.7->nbconvert)
  Downloading rpds_py-0.22.3-cp313-cp313-win_amd64.whl.metadata (4.2 kB)
Requirement already satisfied: python-dateutil>=2.8.2 in c:\programming\risk-
management-project\.venv\lib\site-packages (from jupyter-
client>=6.1.12->nbclient>=0.5.0->nbconvert) (2.9.0.post0)
Requirement already satisfied: pyzmq>=23.0 in c:\programming\risk-management-
project\.venv\lib\site-packages (from jupyter-
client>=6.1.12->nbclient>=0.5.0->nbconvert) (26.2.0)
Requirement already satisfied: tornado>=6.2 in c:\programming\risk-management-
project\.venv\lib\site-packages (from jupyter-
client>=6.1.12->nbclient>=0.5.0->nbconvert) (6.4.2)
Requirement already satisfied: six>=1.5 in c:\programming\risk-management-
project\.venv\lib\site-packages (from python-dateutil>=2.8.2->jupyter-
client>=6.1.12->nbclient>=0.5.0->nbconvert) (1.16.0)
Downloading nbconvert-7.16.4-py3-none-any.whl (257 kB)
Downloading bleach-6.2.0-py3-none-any.whl (163 kB)
Downloading jinja2-3.1.4-py3-none-any.whl (133 kB)
Downloading MarkupSafe-3.0.2-cp313-cp313-win_amd64.whl (15 kB)
Downloading mistune-3.0.2-py3-none-any.whl (47 kB)
Downloading nbclient-0.10.1-py3-none-any.whl (25 kB)
Downloading nbformat-5.10.4-py3-none-any.whl (78 kB)
Downloading pandocfilters-1.5.1-py2.py3-none-any.whl (8.7 kB)
Downloading defusedxml-0.7.1-py2.py3-none-any.whl (25 kB)
Downloading jupyterlab_pygments-0.3.0-py3-none-any.whl (15 kB)
Downloading tinycss2-1.4.0-py3-none-any.whl (26 kB)
Downloading fastjsonschema-2.21.1-py3-none-any.whl (23 kB)
Downloading jsonschema-4.23.0-py3-none-any.whl (88 kB)
Downloading attrs-24.3.0-py3-none-any.whl (63 kB)
Downloading jsonschema_specifications-2024.10.1-py3-none-any.whl (18 kB)
Downloading referencing-0.35.1-py3-none-any.whl (26 kB)
Downloading rpds_py-0.22.3-cp313-cp313-win_amd64.whl (235 kB)
Installing collected packages: fastjsonschema, tinycss2, rpds-py, pandocfilters,
mistune, markupsafe, jupyterlab-pygments, defusedxml, bleach, attrs,
referencing, jinja2, jsonschema-specifications, jsonschema, nbformat, nbclient,
nbconvert
Successfully installed attrs-24.3.0 bleach-6.2.0 defusedxml-0.7.1
fastjsonschema-2.21.1 jinja2-3.1.4 jsonschema-4.23.0 jsonschema-
specifications-2024.10.1 jupyterlab-pygments-0.3.0 markupsafe-3.0.2
mistune-3.0.2 nbclient-0.10.1 nbconvert-7.16.4 nbformat-5.10.4
pandocfilters-1.5.1 referencing-0.35.1 rpds-py-0.22.3 tinycss2-1.4.0
Note: you may need to restart the kernel to use updated packages.
```

1 Introduction

In this paper we will be pricing index basket options, using a combination of time-series forecasts and copula based simulation methods. Specifically, we will be pricing basket options on the DAX, AEX, FTSE 100 Nikkei 225, S&P 500, S&P/TSX indices, with a payoff function $\max(\text{average return of indices} - k, 0)$, for various strikes k, with a time to maturity of one month.

To do this, we will first fit several GARCH based volatility forecasts to each index, assuming normal, Student's-t, and skewed Student's-t residuals. Then, we will fit a multivariate-t distribution to our data. Goodness of fit tests will be conducted for each model. For each assumed residual distribution, we will use Monte Carlo methods to forecast the payoff of our basket option. We will conclude by simulating the performance of the covered call strategy, and backtesting our results.

1.0.1 Background Concepts

We assume that the reader has a basic understanding of time-series models, model fitting, and options[1]. First, let us define our time-series models.

ARCH Model Let us do a brief overview of the Autoregressive Conditional Heteroskedasticity (ARCH) model.

Let $\{y_t\}$ be a time-series. Let $\{\epsilon_t\}$ denote the error terms (known as residuals, with respect to a mean process). We will assume for each time-series that the mean process is constant. We assume that $\epsilon_t = \sigma_t z_t$, where σ_t repsents the time-dependent volatility/standard deviation, and z_t is a white noise process (a stochastic process with mean 0, constant volatility, and no autocorrelation). We will assume that our white noise processes have a constantly volatility of 1.

We forecast conditional volatility as:

$$\sigma_t^2 = \alpha_0 + \sum_{i=1}^{q} \alpha_i \epsilon_{t-i}^2$$

for some $\alpha_0 > 0, \alpha_i \geq 0$ where q represents the order of the ARCH process (i.e. the number of previous residuals to consider).

The main feature of this model is that it is able to capture volatility clustering in the data, by considering past "shocks" or errors.

GARCH Model The above can be generalized, by also including an moving average of the previous volatilities. This results in a Generalized Autoregressive Conditional Heteroskedasticity (GARCH) model.

We forecast conditional volatility as:

$$\sigma_t^2 = \omega + \sum_{i=1}^{q} \alpha_i \epsilon_{t-i}^2 + \sum_{i=1}^{p} \beta_i \sigma_{t-i}^2$$

for some $\omega > 0, \alpha_i \geq 0, \beta_i \geq 0$ where q represents the order of the ARCH process, and p repsents the order of the GARCH process (i.e. the number of previous volatilities to consider).

This model has the added capability of capturing the persistance of previous volatility.

For an intuitive understanding of the parameters, consider the yield of US Government bonds. Note that in the bond market, there are few market moving events, apart from the occasional interest rate change. The response to each interest rate change is generally small, as the effect has been priced in, yielding low values of α_i. However, once volatility increases, it tends to be persistant, yielding high values of β_i. For empirical justification of this situation, see the fitted parameters here.

Noise Processes We will assume that our noise process follows one of the following: 1. A Standard Normal distribution $N(0, 1)$ 2. A Standard Student's-t distribution with a fitted degrees of freedom $t_\nu(0, 1)$ 3. A Skewed Student's-t distribution with fitted degrees of freedom and skew parameter

We assume that the reader is familiar with the first two distributions, and thus we will only highlight the latter distribution. ##### Skewed Student's-t distribution

We use the density function given by Hansen in [1], and refer the reader there for a more indepth analysis:

$$g(z|\eta, \lambda) = \begin{cases} bc\left(1 + \frac{1}{\eta-2}\right)\left(\frac{bz+a}{1+\lambda}\right)^{-(\eta+1)/2} & \text{if } z < -a/b, \\ bc\left(1 + \frac{1}{\eta-2}\right)\left(\frac{bz+a}{1+\lambda}\right)^{-(\eta+1)/2} & \text{if } z \geq -a/b, \end{cases}$$

where $2 < \eta < \infty$ and $-1 < \lambda < 1$, and \$ a $= 4$ c $\left(\frac{\eta-2}{\eta-1}\right)$ \$, $b^2 = 1 + 3A\lambda^2 - a^2$, $c = \frac{\Gamma\left(\frac{\eta+1}{2}\right)}{\sqrt{\pi(\eta-2)}}\left(\frac{\eta}{2}\right)$

The idea behind this distribution is to generalize the Student's-t distribution, to allow for skewness. Fortunately, the arch library, from [5], allows us to easily sample from this skewed Student's-t distribution.

1.1 Preliminary Work

Import Libraries We will use: 1. The yfinance library from [2] to download our data, 2. numpy from [3] and pandas from [4] for data structures, 3. the arch library from [5] for fitting GARCH models, and conducting Goodness of Fit tests 4. the scipy library from [6] for fitting multivariate models, conducting simulations, and conducting Goodness of Fit tests 5. the statsmodels library from [7] for graphing autocorrelation functions and fitting t-copulas 6. the matplotlib library from [8] for data visualization

[308]:
```
import yfinance as yf
import pandas as pd
import numpy as np
from arch import arch_model
from arch.univariate import SkewStudent
from scipy.stats import norm, t, multivariate_t, chi2, f
from scipy.optimize import minimize
from statsmodels.distributions.copula.api import StudentTCopula
from statsmodels.graphics.tsaplots import plot_acf
import matplotlib.pyplot as plt
```

Import Data This cell imports index data from Yahoo Finance, using the yFinance library, or alternatively, loads previously cached data.

[309]:
```
# List of ticker symbols for each index
all_tickers = ["^GDAXI", "^AEX", "^FTSE", "^N225", "^GSPC", "^GSPTSE", "^TYX"] ␣
 ↪# DAX, AEX, FTSE 100, Nikkei 225, S&P 500, S&P/TSX Composite
index_datasets = {}

#Set to True to download data from Yahoo Finance using the yFinance library
#Set to False to use previously cached data
DOWNLOAD_DATA = False


if DOWNLOAD_DATA:
    # Loop over each ticker symbol to fetch the data
    for ticker in all_tickers:
        # Retrieve the historical data starting from November 20, 1992
        index = yf.Ticker(ticker)
        #Save to dictionary
        index_datasets[ticker] = index.history(start="1992-11-20")  ␣
 ↪#(start="1992-11-20")

        # Convert the Date column to Datetime
        (index_datasets[ticker]).index = pd.
 ↪to_datetime((index_datasets[ticker]).index.normalize().tz_localize(None))

        index_datasets[ticker] = round(index_datasets[ticker],12)

        #Write data to disk
        index_datasets[ticker].to_excel(".\\Saved Data\\"+ticker+'.xlsx',␣
 ↪index=True)
else:
    import os
    # Loop over each ticker symbol
    for ticker in all_tickers:
        # File path to saved data
        file_path = os.path.join(".\\Saved Data\\"+ticker+'.xlsx')

        if os.path.exists(file_path):
            # Load the saved data from Excel
            index_data = pd.read_excel(file_path, index_col=0, parse_dates=True)

            # Convert the Date column to Datetime
            index_data.index = pd.to_datetime(index_data.index.normalize().
↪tz_localize(None))

            # Save it to the dictionary
            index_datasets[ticker] = round(index_data,12)

            # Optional: Print the first few rows of the data for verification
            print(f"Loaded data for {ticker}")
```

```
Loaded data for ^GDAXI
Loaded data for ^AEX
Loaded data for ^FTSE
Loaded data for ^N225
Loaded data for ^GSPC
Loaded data for ^GSPTSE
Loaded data for ^TYX
```

Compute log-returns We merge all of the data stored in the index_datasets dictionary into one Pandas dataframe. We sample the data on a monthly basis, using a forward fill to account for any missing dates. The forward fill is required to account for different exchanges having different holidays. We then compute the monthly log-returns for each index. We then drop all data apart from the date and monthly log-returns from our dataframe.

[310]:
```
# Initialize a DataFrame to store merged data
merged_data = pd.DataFrame()
k=0

# Loop over each ticker symbol to fetch the data
for ticker in all_tickers:

    index_data = index_datasets[ticker]
    # Extract only the Date and Close columns, reset the index
    close_data = (index_data[['Close']]).copy()

    close_data.rename(columns={'Close': ticker},inplace=True)

    # Merge with the previously merged data
    if merged_data.empty:
        merged_data = close_data
    else:
        merged_data = pd.merge(merged_data, close_data, on='Date')

for ticker in all_tickers:
    #Forward fill missing data
    merged_data[ticker] = merged_data[ticker].ffill()

#Resample at end of each month
merged_data = merged_data.resample("ME").last()

#Compute Log Returns
merged_data[:] = np.log(1+merged_data.pct_change())

#Drop blank entries in first row (if any)
merged_data.dropna(inplace=True)
```

2 Model Fitting

Compute Emperical Moments

[311]:
```
# Create a dataframe for the first four empirical moments (mean, variance,␣
↪skewness, kurtosis)
print("Empirical moments for each index's monthly log-returns:\n")
moments_df = pd.DataFrame(columns=['Mean', 'Variance', 'Skewness', 'Kurtosis'],␣
↪index=all_tickers)

# Calculate and print empirical moments for each ticker
for ticker in all_tickers:
    data = merged_data[ticker]
    moments_df.loc[ticker, 'Mean'] = data.mean()
    moments_df.loc[ticker, 'Variance'] = data.var()
    moments_df.loc[ticker, 'Skewness'] = data.skew()
    moments_df.loc[ticker, 'Kurtosis'] = data.kurtosis()

print(moments_df)
```

```
Empirical moments for each index's monthly log-returns:

             Mean  Variance  Skewness  Kurtosis
^GDAXI   0.006703  0.003432 -0.825112  2.748861
```

```
^AEX      0.005083  0.002829 -0.934392  2.595402
^FTSE     0.002842  0.001509  -0.69922  1.200117
^N225     0.002085  0.003134 -0.584054  1.236312
^GSPC      0.00686  0.001819 -0.660288  1.198035
^GSPTSE   0.005302  0.001734  -1.06382  3.594747
^TYX     -0.001292  0.004282 -0.132197   3.96642
```

2.1 Fit GARCH Models

Here, we first fit a Students-t GARCH model to the log-return data for each index, using the arch library.

To do this, we must first decide what order (p) to use for the GARCH terms (σ_t^2), and what order (q) to use for the ARCH Terms (ϵ_t).

Choose (p,q) We will test different pairs of (p, q), and choose the one with the lowest average Akaike Information Criterion (AIC) across all distributions and indices.

Note: We are actually fitting the GARCH model to 100 times the log-return data, as recommended by the library documentation. This is done to improve numerical stability.

[312]:
```python
distributions = ("normal","t","skewt")

combinations = [(i, j) for i in range(1, 5) for j in range(1, 5)]
# i.e. [(1,1),(1,2),...,(1,4),(2,1),(2,2),...,(4,4)]

best_AIC = pd.DataFrame(np.inf,columns=all_tickers, index = distributions)
best_pair = pd.DataFrame(columns=all_tickers, index = distributions)

for dist in distributions:
    for (p,q) in combinations:
        for ticker in all_tickers:
            # as suggested by the documentation, we scale our data to improve␣
↪numerical stability
            model = arch_model(merged_data[ticker]*100, vol="Garch", p=p, q=q,␣
↪dist=dist, rescale=False)
            aic = model.fit(options={'maxiter': 10000, 'ftol':␣
↪1e-8},disp="off").aic

            if best_AIC.loc[dist,ticker] > aic:
                best_AIC.loc[dist,ticker] = aic
                best_pair.loc[dist,ticker] = (p,q)

print(best_pair)
print(round(best_AIC,2))
```

```
         ^GDAXI     ^AEX    ^FTSE    ^N225    ^GSPC  ^GSPTSE     ^TYX
normal   (1, 1)   (1, 1)   (1, 1)   (1, 1)   (1, 1)   (1, 1)   (1, 4)
t        (1, 1)   (1, 1)   (1, 1)   (1, 1)   (1, 1)   (1, 1)   (1, 4)
```

```
skewt   (1, 1)  (1, 1)  (1, 1)  (1, 1)  (1, 1)  (1, 1)  (1, 4)
          ^GDAXI     ^AEX    ^FTSE    ^N225    ^GSPC  ^GSPTSE     ^TYX
normal  2417.25  2309.76  2106.20  2418.14  2151.53  2142.13  2449.53
t       2393.25  2305.33  2102.16  2415.92  2150.67  2125.99  2424.26
skewt   2380.69  2291.11  2086.74  2404.97  2128.18  2109.61  2424.42
```

The above results suggest that GARCH(1,1) models best fit our data, for all distribution/equity index pairs. Also, the skewed Student's-t GARCH(1,1) model has the best fit for all equity indices, as evident by its relatively lower AIC.

Thus, it is best if we set the order of our GARCH and ARCH terms to (1,1), for each index and distribution.

Fit GARCH(1,1) models Next, we will save our normal, Student's-t and skewed Student's-t GARCH(1,1) models, as well as their standarized residuals. We also model the US 30 Yield Treasury Bond yield, to illustate the situation in our introduction.

[313]:
```
fitted_models = {}
params = {}
std_residuals = {}

#Fit each GARCH model, store models in fitted_models dictionary and standarized␣
 ↪residuals in std_residuals dictionary
for dist in distributions:

        # as suggested by the documentation, we scale our data to improve␣
 ↪numerical stability
    fitted_models[dist] = {ticker: arch_model(merged_data[ticker]*100,␣
 ↪vol="Garch", p=1, q=1, dist=dist, rescale=False)
                                        .fit(options={'maxiter': 1000, 'ftol':␣
 ↪1e-8},disp="off")
                                    for ticker in all_tickers}

    std_residuals[dist] = pd.DataFrame([fitted_models[dist][ticker].resid /␣
 ↪fitted_models[dist][ticker].conditional_volatility
                                        for ticker in all_tickers],index =␣
 ↪all_tickers).T

#Creates a dataframe with the fitted parameters from each model, and prints␣
 ↪results
print("Fitted parameters for each model:")
for dist in distributions:
    df = pd.DataFrame(columns=all_tickers)
    for ticker in all_tickers:
        df[ticker] = fitted_models[dist][ticker].params

    print("\nModel: "+dist)
    print(round(df,5))
    params[dist] = df
```

```
Fitted parameters for each model:

Model: normal
            ^GDAXI      ^AEX     ^FTSE      ^N225     ^GSPC   ^GSPTSE      ^TYX
mu         0.96413   0.81644   0.46937    0.30679   0.86002   0.84143  -0.25449
omega      4.48422   3.10260   1.46425   12.15326   1.01616   2.38309   7.06496
alpha[1]   0.19497   0.26727   0.17240    0.11457   0.18891   0.34923   0.25131
beta[1]    0.68466   0.63542   0.73668    0.49666   0.76467   0.56289   0.58163

Model: t
            ^GDAXI      ^AEX     ^FTSE      ^N225      ^GSPC   ^GSPTSE      ^TYX
mu         1.17846   0.90692   0.60052    0.47316    0.96469   0.82941  -0.24311
omega      3.32143   3.45841   1.65368    2.20445    1.00371   1.46103   0.61276
alpha[1]   0.18755   0.25124   0.18675    0.04877    0.19368   0.19025   0.06148
beta[1]    0.72808   0.63266   0.71356    0.87897    0.76326   0.73664   0.92769
nu         5.90685   9.41351   9.05120   12.44338   12.09854   6.68848   5.07259

Model: skewt
            ^GDAXI       ^AEX      ^FTSE      ^N225      ^GSPC   ^GSPTSE      ^TYX
mu         0.86357    0.70533    0.44126    0.29193    0.82432   0.60354  -0.33474
omega      3.49826    2.68825    1.54851    2.87828    0.96162   1.09066   0.73884
alpha[1]   0.17152    0.23363    0.15931    0.06534    0.19289   0.15650   0.07568
beta[1]    0.72828    0.67443    0.73541    0.83926    0.75932   0.78629   0.91121
eta        6.80541   11.15431   13.43454   18.90336   15.70029   8.17215   5.41805
lambda    -0.28906   -0.30059   -0.30482   -0.26259   -0.35255  -0.33997  -0.08834
```

Recall that we scaled our log-returns by a factor of 100 before fitting each model. Hence, our mean returns (mu) above are scaled by a factor of 100, while our baseline variances (omega) are scaled by a factor of 100^2

We note that, in each model, the Nikkei 225 has a low alpha parameter, suggesting a lack of volatility clustering. Additionally, we note that, when compared to the other indices, the Nikkei 225 and S&P 500 log-return distributions demonstrate thinner tails, as demonstrated by the relatively greater degrees of freedom in both the classical and skewed Student's-t GARCH models. We observe that each index's returns are negatively skewed, as evident by the lambda parameter in the skewed Student's-t GARCH models.

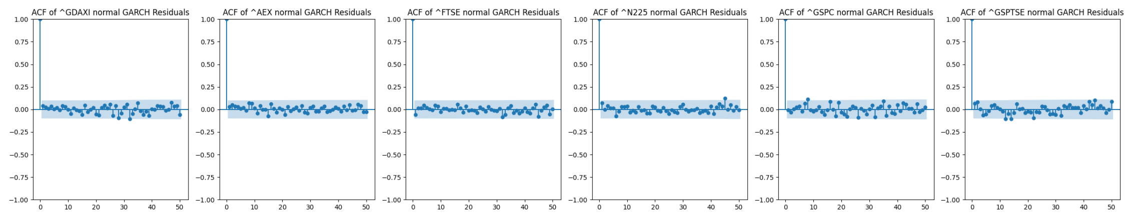
From here on, we will only focus on equity indices, as they are the main focus of this study.

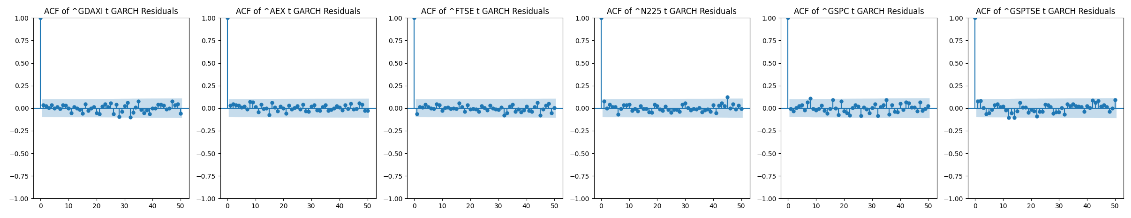
```
[314]: tickers = ["^GDAXI", "^AEX", "^FTSE", "^N225", "^GSPC", "^GSPTSE"]
for dist in distributions:
    std_residuals[dist] = std_residuals[dist][tickers]
    fitted_models_temp = {ticker: fitted_models[dist][ticker] for ticker in␣
 ↪tickers}
    fitted_models[dist] = fitted_models_temp
```

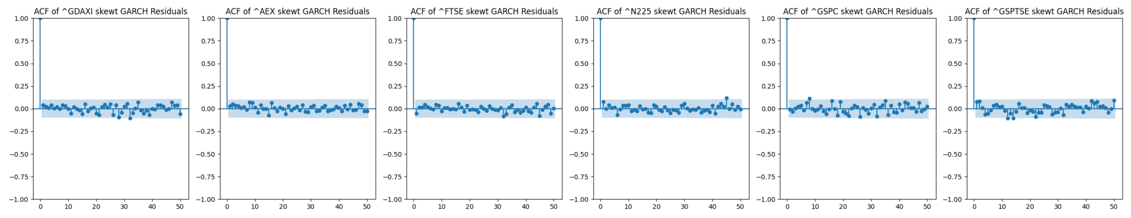
GARCH Goodness of Fit Tests

Plot ACF of Standardized Residuals We plot the ACF of the residuals of each GARCH, and display them side by side for comparision.

```
[315]: for dist in distributions:
           fig, axes = plt.subplots(1, len(tickers), figsize=(5 * len(tickers), 5))
           for i in range(len(tickers)):
               ticker = tickers[i]
               plot_acf(std_residuals[dist][ticker],lags = 50, ax = axes[i])
               axes[i].set_title('ACF of ' + ticker + " " + dist + ' GARCH Residuals')
```





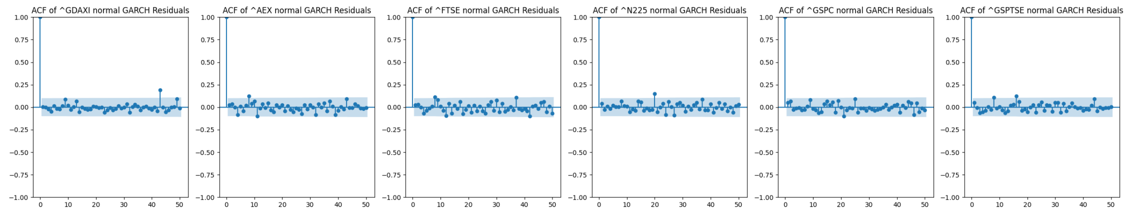


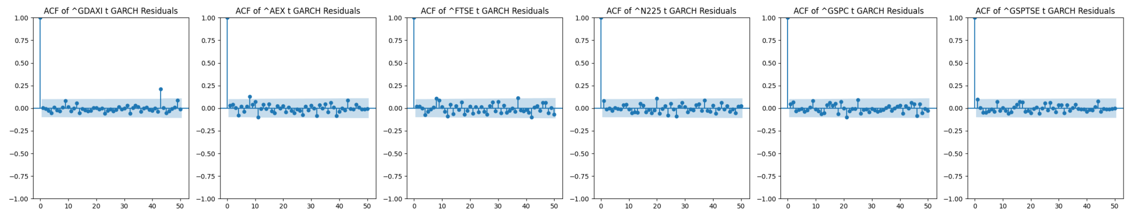
The above results suggest that there is little autocorrelation in the stardarized residuals of our fitted GARCH models.

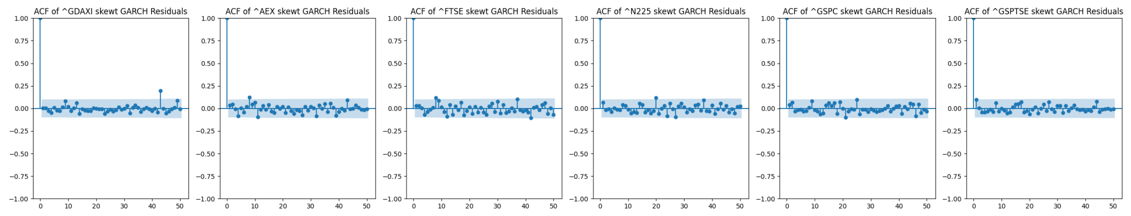
Plot ACF of Squared Standardized Residuals We also plot the ACF of the squared residuals of each GARCH.

```
[316]: for dist in distributions:
           fig, axes = plt.subplots(1, len(tickers), figsize=(5 * len(tickers), 5))
           for i in range(len(tickers)):
               ticker = tickers[i]
               plot_acf((std_residuals[dist][ticker])**2,lags = 50, ax = axes[i])
```

```
        axes[i].set_title('ACF of ' + ticker + " " + dist + ' GARCH Residuals')
```





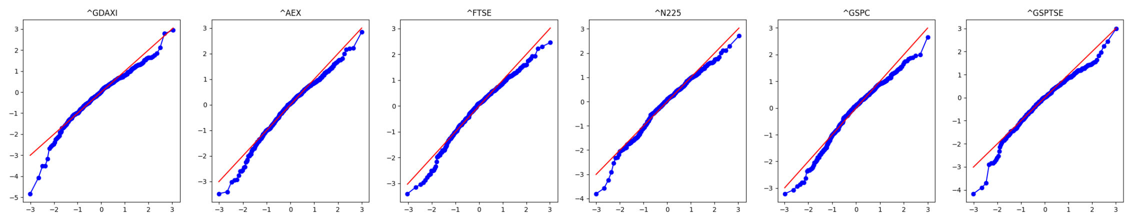


The above results suggest that there is little autocorrelation in the stardarized squared residuals of our fitted GARCH models.

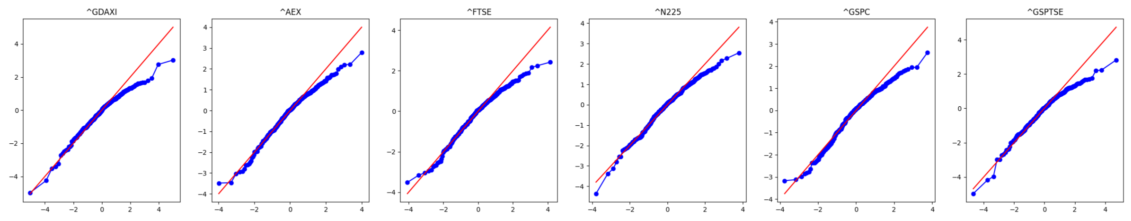
Plot QQ Plot for residuals We also conduct a QQ Plot of the standarized residuals of each GARCH.

[317]:
```
for dist in distributions:
    print('QQ Plots of ' + dist + ' GARCH Residuals')
    num_tickers = len(tickers)
    fig, axes = plt.subplots(1, num_tickers, figsize=(5 * num_tickers, 5))

    n = len(merged_data)

    quantiles = (np.arange(1, n + 1) - 0.5) / n

    for i in range(len(tickers)):
        ticker = tickers[i]
        if dist == 'normal': theoretical_quantiles  =  np.quantile(norm.
↪rvs(0,1,100000), quantiles)

        elif dist == 't':
            nu = params[dist][ticker]['nu']
            theoretical_quantiles  =  np.quantile(t.rvs(nu,0,1,100000),␣
↪quantiles)

        elif dist == 'skewt':
            eta = params[dist][ticker]['eta']
            lambda_param = params[dist][ticker]['lambda']
            skew_t_rvs = SkewStudent().simulate(parameters=[eta, lambda_param])
            theoretical_quantiles  =  np.quantile(skew_t_rvs(10000), quantiles)

        sorted_std_residuals = sorted(std_residuals[dist][ticker])

        axes[i].plot(theoretical_quantiles,sorted_std_residuals,color = "blue",␣
↪marker = 'o')
        axes[i].plot(theoretical_quantiles,theoretical_quantiles,color = "red")
        axes[i].set_title(ticker)

    plt.show()
```

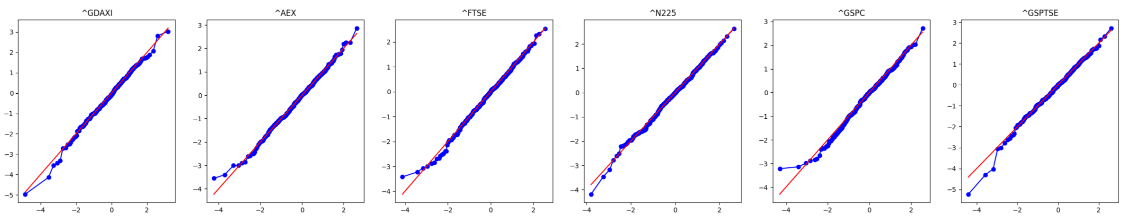
```
QQ Plots of normal GARCH Residuals
```



```
QQ Plots of t GARCH Residuals
```



```
QQ Plots of skewt GARCH Residuals
```



The above results suggest that the skewed Student's-t distribution has the best fit, across all indices.

However, most of the our models predict slightly more extreme negtaive shocks than the data suggests, as evident by bottom left regions in the above QQ Plots.

2.2 Fit Multivariate Model

3 Compute Spearman's Variance-Covariance Matrix

Using the standarized residuals from our Student's t GARCH model, we calculate the spearman correlation & covariance of the standarized residuals.

[318]:
```
spearman_cov = {}

print("Spearman Correlation Matrix of Standarized Residuals")
for dist in distributions:
    temp = pd.DataFrame(index=tickers,columns=tickers)
    for ticker_1 in tickers:
        for ticker_2 in tickers:

            #Rank residuals
            ranked_residuals_1 = pd.Series(std_residuals[dist][ticker_1]).
 ↪rank(method='average')
            ranked_residuals_2 = pd.Series(std_residuals[dist][ticker_2]).
 ↪rank(method='average')

            #Compute Spearman Correlation Coeff./Covariance
            temp.loc[ticker_1,ticker_2] = np.
 ↪corrcoef(ranked_residuals_1,ranked_residuals_2)[0][1]

    spearman_cov[dist] = temp
    print("GARCH Distribution: ",dist)
    print(spearman_cov[dist])
    spearman_cov[dist] = spearman_cov[dist].to_numpy(dtype=float) #convert to␣
 ↪numpy array for later
    print()
```

```
Spearman Correlation Matrix of Standarized Residuals
GARCH Distribution:  normal
```

```
          ^GDAXI      ^AEX     ^FTSE     ^N225     ^GSPC   ^GSPTSE
^GDAXI       1.0  0.834389  0.722828  0.528443  0.715045  0.640025
^AEX    0.834389       1.0  0.748364   0.52752  0.715801  0.643565
^FTSE   0.722828  0.748364       1.0  0.414478  0.684442  0.674762
^N225   0.528443   0.52752  0.414478       1.0  0.511986  0.436771
^GSPC   0.715045  0.715801  0.684442  0.511986       1.0  0.730446
^GSPTSE 0.640025  0.643565  0.674762  0.436771  0.730446       1.0

GARCH Distribution:  t
          ^GDAXI      ^AEX     ^FTSE     ^N225     ^GSPC   ^GSPTSE
^GDAXI       1.0  0.834805   0.72218  0.531475  0.713148  0.639529
^AEX    0.834805       1.0  0.747889  0.531233  0.714845  0.648134
^FTSE    0.72218  0.747889       1.0  0.415437  0.682222   0.67857
^N225   0.531475  0.531233  0.415437       1.0  0.514955  0.443044
^GSPC   0.713148  0.714845  0.682222  0.514955       1.0  0.733162
^GSPTSE 0.639529  0.648134   0.67857  0.443044  0.733162       1.0

GARCH Distribution:  skewt
          ^GDAXI      ^AEX     ^FTSE     ^N225     ^GSPC   ^GSPTSE
^GDAXI       1.0  0.835681  0.724301  0.531427  0.715109  0.641378
^AEX    0.835681       1.0   0.74925  0.532172   0.71675  0.648637
^FTSE   0.724301   0.74925       1.0  0.417106  0.684398  0.678694
^N225   0.531427  0.532172  0.417106       1.0  0.513752  0.445453
^GSPC   0.715109   0.71675  0.684398  0.513752       1.0  0.733292
^GSPTSE 0.641378  0.648637  0.678694  0.445453  0.733292       1.0
```

Fit Student's-t Copula Next, we will fit our Student's-t copula to our normal, Student's-t and skewed Student's-t GARCH marginal residuals. To do this, we will use the Spearman correlation matrices Σ_{COR} from above. We can numerically estimate our fitted degrees of freedom ν, with the constraint that $\nu > 2$, by using scipy's minimize function, applied to our copula's log likelihood function.

[319]:
```python
u,fitted_df = {},{}

def negative_tcopula_log_likelihood(nu,corr,x):
    copula = StudentTCopula(corr=corr,df = nu, k_dim=corr.shape[0])
    return -1*sum(copula.logpdf(row) for row in x)

options = {
    'maxiter': 10000, #Max iterations
    'gtol': 1e-12, #Tolerance for convergence
    'disp': True #Display warnings
}

for dist in distributions:
    t_model = t.fit(std_residuals[dist].to_numpy())
    u[dist] = t.cdf(std_residuals[dist].
 ↪to_numpy(),t_model[0],t_model[1],t_model[2])

    fitted_df[dist] = minimize(negative_tcopula_log_likelihood,10, args =␣
 ↪(spearman_cov[dist],u[dist]),bounds = [(2,1000)],options = options).x

print("Multivariate-t fitted degrees of freedom for standaridized residuals:")
print("Normal:",fitted_df["normal"])
print("Student's-t:",fitted_df["t"])
print("Skewed Student's-t:",fitted_df["skewt"])
```

```
Multivariate-t fitted degrees of freedom for standaridized residuals:
Normal: [11.33194331]
Student's-t: [11.03030989]
Skewed Student's-t: [11.03619766]
```

Following the procedure in "Copula Methods in Finance", we can sample from our Student's t-copula using the following algorithm:

1. Find the Choelsky Decomposition A of R, where R represents our Spearman Correlation Matrix
2. Simulate n i.i.d. $N(0,1)$ random variables z_i, set $z = (z_1, z_2, ..., z_n)$
3. Set $X = Az$
4. Simulate $s \sim \chi^2_\nu$ independently of z
5. Set $x = \sqrt{\frac{\nu}{s}}y$
6. Set $u_i = F_\nu(x_i)$ for $i = 1, 2, ..., n$ where $F_n u$ represents the univariate Student's t c.d.f. with ν degrees of freedom

We can then set $t_i = F_i^{-1}(u_i)$ for $i = 1, 2, ..., n$ where F_i^{-1} represents the marginal inverse c.d.f for t_i

Here, each t_i represents a simulated standardized residual.

Conveniently, the statsmodels StudentTCopula class allows us to sample from our copula directly. We will use this instead to conduct 100,000 samples from our copula. Then, applying the marginal inverse c.d.f's component wise, we get 100,000 simulations for our standardized residuals.

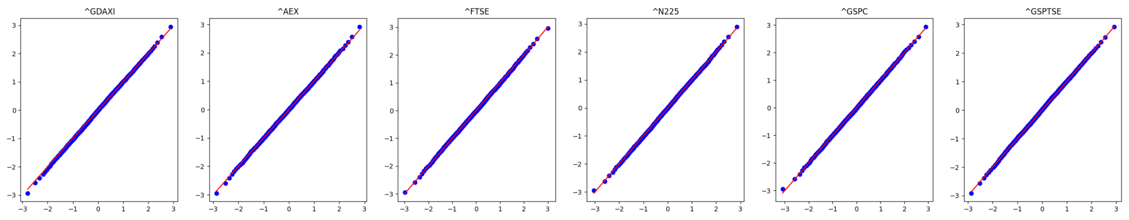
[320]:
```
copulas = {dist: StudentTCopula(corr=spearman_cov[dist],
                                df = fitted_df[dist],
                                k_dim=spearman_cov[dist].shape[0]) for dist in␣
 ↪distributions}



def inverse_marginal(x,dist):
    marginals = []
    for ticker in tickers:
        #We want to forecast standarized residuals
        mu = 0
        omega = 1
        if dist == "normal" :
            marginals.append(norm.ppf(x[:,tickers.index(ticker)],loc=mu,scale =␣
↪np.sqrt(omega)))
        elif dist == "t":
            nu = params[dist][ticker]['nu']
            marginals.append(t.ppf(x[:,tickers.index(ticker)],df = nu,␣
↪loc=mu,scale = np.sqrt(omega)))
        elif dist == "skewt":
            eta = params[dist][ticker]['eta']
            lambda_param = params[dist][ticker]['lambda']
            marginals.append((SkewStudent().ppf(pits = x[:,tickers.
↪index(ticker)],parameters=[eta,lambda_param]))*np.sqrt(omega)+ mu)
    return pd.DataFrame(np.array(marginals).T,columns = tickers)

def get_simulated_t(dist, num_samples = 100000):
    return inverse_marginal(copulas[dist].rvs(num_samples),dist)
```

We will plot QQplots of our marginal samples drawn from our copula, compared to our GARCH residuals. We expect there to be a strong fit, as our copula is fitted to the transformed residuals.

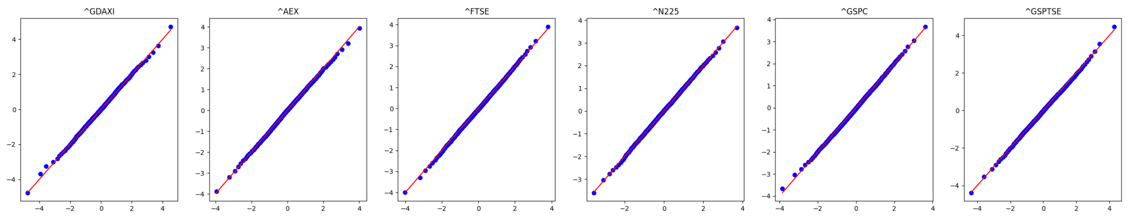
[321]:
```
num_samples = 10000
for dist in distributions:
    fig, axes = plt.subplots(1, num_tickers, figsize=(5 * num_tickers, 5))
    print('QQ Plots of copula marginal simulated residuals vs ' + dist + '␣
↪GARCH Residuals')

    num_tickers = len(tickers)
    n = 300

    quantiles = (np.arange(1, n + 1) - 0.5) / n
    simulated_results = get_simulated_t(dist)

    for ticker in tickers:
        theoretical_quantiles = []

        #We want to forecast standarized residuals
        mu = 0
        omega =  1
        if dist == 'normal': theoretical_quantiles  =  np.quantile(norm.
↪rvs(mu,np.sqrt(omega),num_samples), quantiles)

        elif dist == 't':
            nu = params[dist][ticker]['nu']
            theoretical_quantiles  =  np.quantile(t.rvs(nu,mu,np.
↪sqrt(omega),num_samples), quantiles)
```

```
        elif dist == 'skewt':
            eta = params[dist][ticker]['eta']
            lambda_param = params[dist][ticker]['lambda']
            skew_t_rvs = SkewStudent().simulate(parameters=[eta, lambda_param])
            theoretical_quantiles  =  np.quantile(skew_t_rvs(num_samples)*np.
↪sqrt(omega) + mu, quantiles)

        simulated_quantiles = np.quantile(simulated_results[ticker],quantiles)

        axes[tickers.index(ticker)].
↪plot(theoretical_quantiles,theoretical_quantiles,color = "red")
        axes[tickers.index(ticker)].
↪scatter(theoretical_quantiles,simulated_quantiles,color = "blue", marker =␣
↪'o')
        #axes[i].set_aspect("equal")
        axes[tickers.index(ticker)].set_title(ticker)

    plt.show()
```

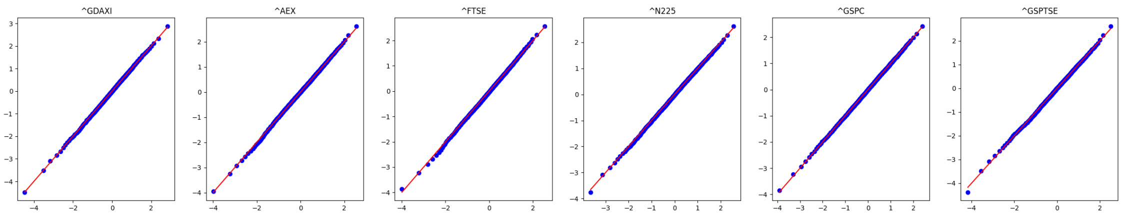
```
QQ Plots of copula marginal simulated residuals vs normal GARCH Residuals
```



```
QQ Plots of copula marginal simulated residuals vs t GARCH Residuals
```



```
QQ Plots of copula marginal simulated residuals vs skewt GARCH Residuals
```



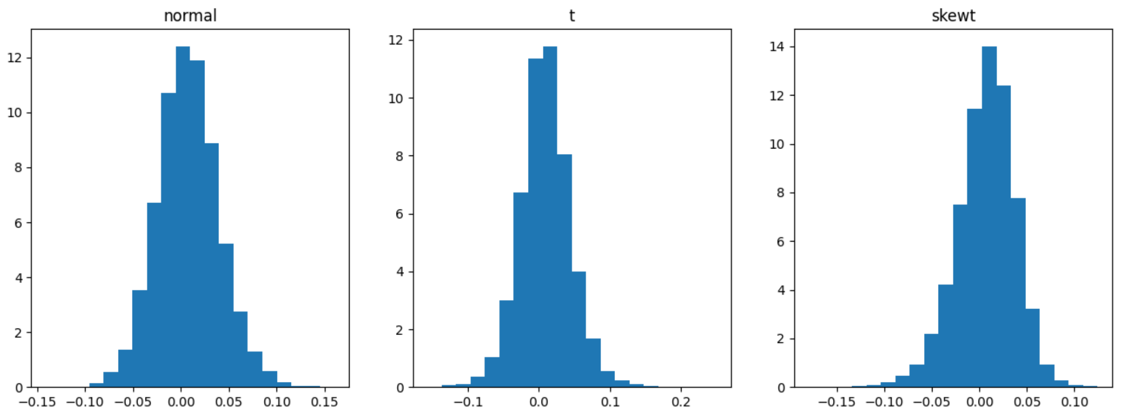
At least on a marginal level, our copula accurately models the standardized residuals from our GARCH models.

We can forecast the one period ahead return as follows:

1. Forecast the conditional volatility of each index using our GARCH models.
2. Multiply our n-simulated standardized residuals by our conditional volatility forecasts - this gives n raw residual simulations for each index.
3. Add our mean log-return to each residual forecast - this yields a one-period ahead forecast for each index, following our dependence structure.

[322]:
```python
forecast_dfs = {}


def compute_forecast(dist, cond_vol_index = -1, num_samples = 10000):
    forecasts = {}
    simulated_results = get_simulated_t(dist,num_samples)
    for ticker in tickers:
        forecasts[ticker] = params[dist][ticker]['mu']/100 +\
                simulated_results[ticker]*\
                fitted_models[dist][ticker].conditional_volatility.
 ↪iloc[cond_vol_index]/100
    return pd.DataFrame(forecasts)

strikes = [0.01,0.02,0.03,0.04,0.05]
simulated_prices = pd.DataFrame(index = strikes, columns =␣
 ↪['normal','t','skewt'])
simulated_prices.index.name = "strike"
fig, axes = plt.subplots(1, 3, figsize=(5 * 3, 5))

for dist in distributions:

    forecasts = compute_forecast(dist,-1 )

    #convert to simple returns for computing payoff
    forecast_dfs[dist] = np.exp(pd.DataFrame(forecasts)) - 1

    strikes = [0.01,0.02,0.03,0.04,0.05]
    #Compute and plot basket returns
    forecast_dfs[dist]["Average"] = forecast_dfs[dist].mean(axis=1)
    axes[distributions.index(dist)].hist(forecast_dfs[dist]["Average"], density␣
 ↪= True, bins = 20)
    axes[distributions.index(dist)].set_title(dist)

    #Compute simulated payoffs
    for k in strikes:
        simulated_prices.loc[k,dist] = np.maximum(forecast_dfs[dist]["Average"]␣
 ↪- k,0).mean()

print("Simulated Mean Returns:")
for dist in distributions:
    print(dist,'\t', forecast_dfs[dist]["Average"].mean())

print("Simulated Mean Option Payoffs:")
print(simulated_prices)

print("\nSimulated Density of Simple Returns:")
plt.show()
```

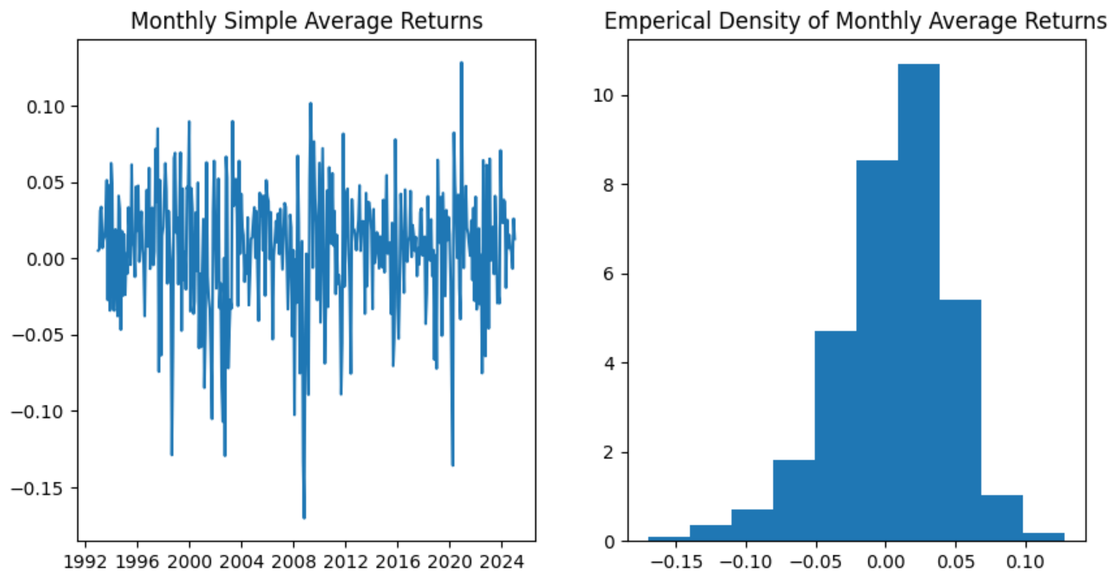
```
Simulated Mean Returns:
normal   0.007806308290388148
t        0.009206141063391031
skewt    0.007200887584390559
Simulated Mean Option Payoffs:
          normal         t     skewt
strike
0.01    0.011707   0.01324  0.010456
0.02    0.007687  0.009054  0.006242
0.03     0.00483   0.00599  0.003398
0.04    0.002912  0.003846  0.001686
0.05    0.001694  0.002412  0.000782

Simulated Density of Simple Returns:
```

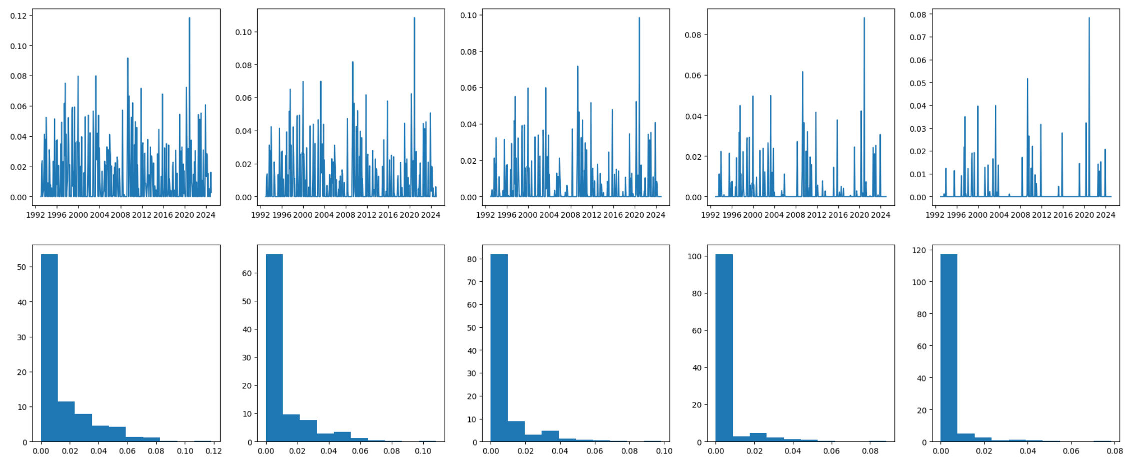


Historical Payoffs Here we compute the historical payoff of our basket option, for various strikes, at each month in our dataset.

```
[323]: merged_data["Simple Average Return"] = np.exp(merged_data[tickers]).
        ↪mean(axis=1) - 1


       print("Emperical Average Simple Return:",str(merged_data["Simple Average␣
        ↪Return"].mean()))
       for k in strikes:
           merged_data["Basket Payoff, strike:",k] = np.maximum(merged_data["Simple␣
        ↪Average Return"] - k,0)

       for k in strikes:
           print("Strike:",k,"Payoff:",merged_data["Basket Payoff, strike:",k].mean())


       fig, axes = plt.subplots(1, 2, figsize=(5 * 2, 5))

       axes[0].plot(merged_data["Simple Average Return"])
       axes[0].set_title("Monthly Simple Average Returns")
       axes[1].hist(merged_data["Simple Average Return"], density  = True)
       axes[1].set_title("Emperical Density of Monthly Average Returns")
       plt.show()

       fig, axes = plt.subplots(2, len(strikes), figsize=(5* len(strikes), 10 ))

       for i in range(len(strikes)):
           k = strikes[i]
           axes[0][i].plot(np.maximum(merged_data["Simple Average Return"] - k, 0))
           axes[1][i].hist(np.maximum(merged_data["Simple Average Return"] - k, 0),␣
        ↪density = True)
```

```
Emperical Average Simple Return: 0.006018745426154646
Strike: 0.01 Payoff: 0.01343324878084908
Strike: 0.02 Payoff: 0.008963616389594812
Strike: 0.03 Payoff: 0.005610366763297805
Strike: 0.04 Payoff: 0.003311825946792608
Strike: 0.05 Payoff: 0.0018801396256648635
```



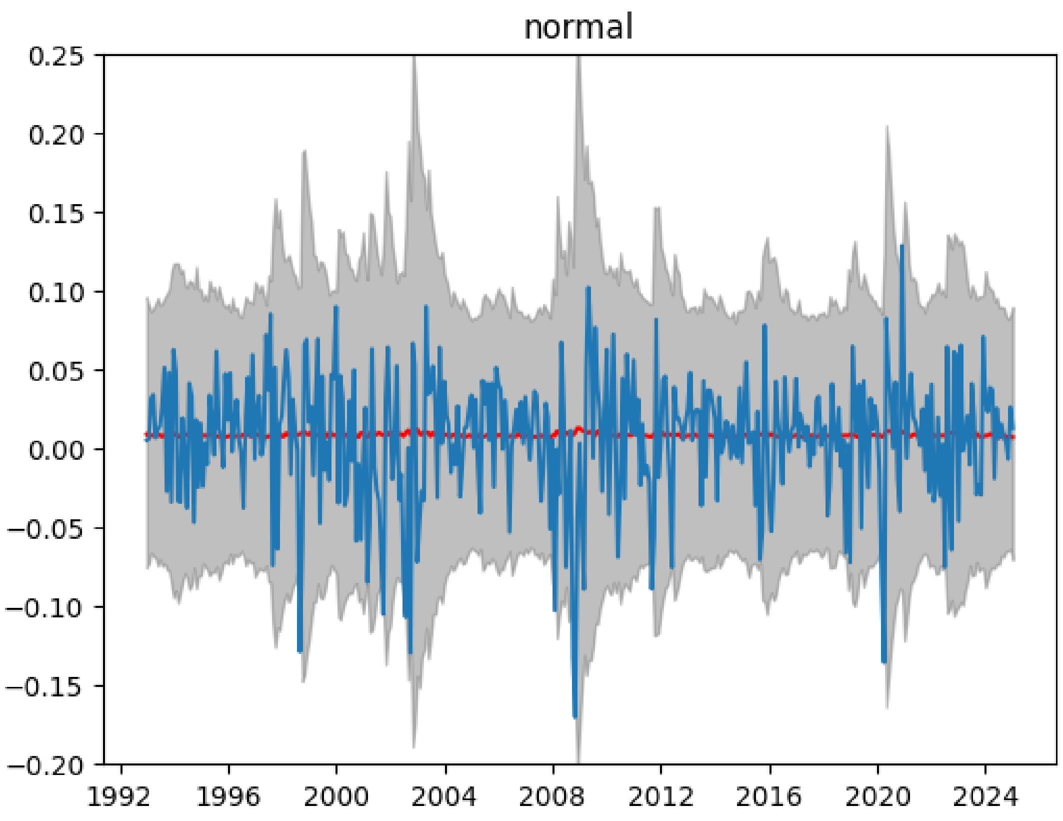


Conduct Historical Simulation Next, we compute the conditional volatility of our log returns. We then compute 10000 simulations at each month, to estimate the basket option payoff at time t. We repeat this process assuming different conditional volatility distributions. We then compare our estimates (red) to the historical payoffs (blue). The shaded region represents the upper and lower 5% quantiles of our forecasted basket returns.

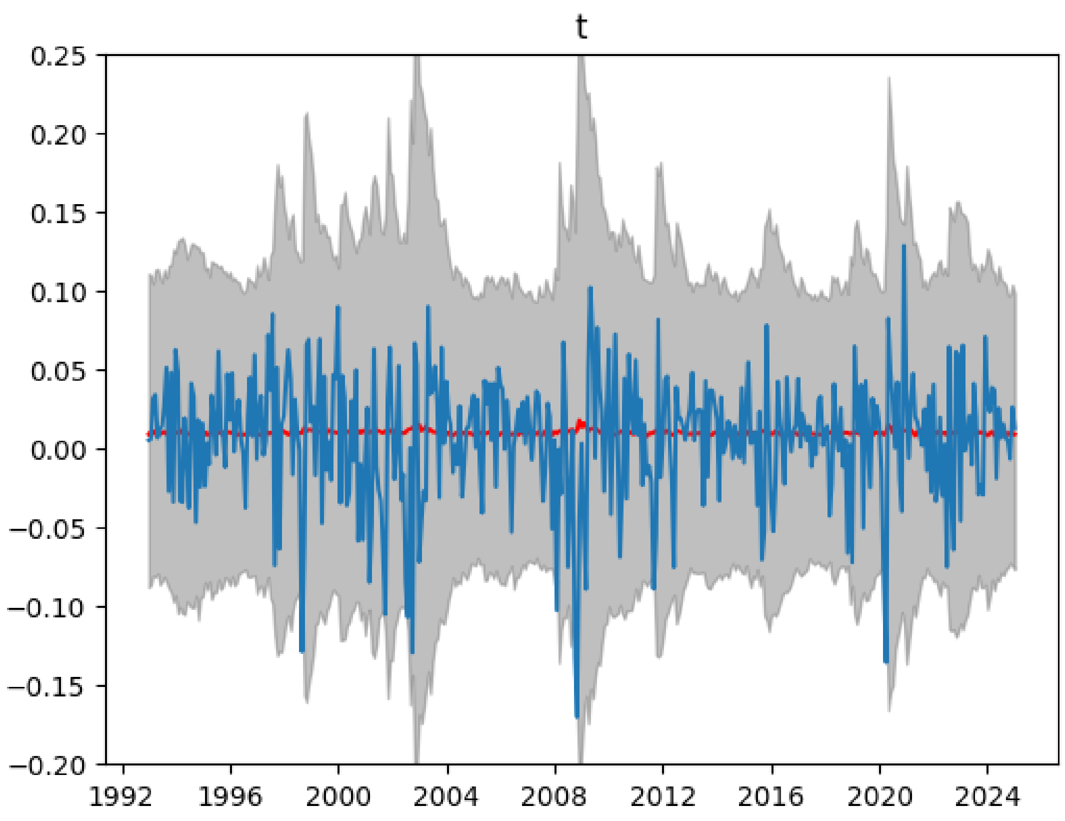
[349]:
```
forecasts = {}
forecasted_avg_return = {}
forecasted_95p = {}
forecasted_5p = {}
error = {}
```

```python
timesteps = len(merged_data["Simple Average Return"])

for dist in distributions:
    forecasts[dist] = []
    forecasted_avg_return[dist] = np.empty(timesteps)
    forecasted_95p[dist] = np.empty(timesteps)
    forecasted_5p[dist] = np.empty(timesteps)

    for i in range(timesteps):
        forecasts[dist].append(np.exp(compute_forecast(dist,i,3000)) - 1)
        forecasted_avg_return[dist][i] = forecasts[dist][i].mean(axis=None)
        forecasted_95p[dist][i] = np.quantile(forecasts[dist][i],0.975)
        forecasted_5p[dist][i] = np.quantile(forecasts[dist][i],0.025)

    outside_upper_bounds = merged_data["Simple Average Return"] >␣
 ↪forecasted_95p[dist]
    outside_lower_bounds = merged_data["Simple Average Return"] <␣
 ↪forecasted_5p[dist]
    percent_outside_bounds = (sum(outside_upper_bounds)␣
 ↪+sum(outside_lower_bounds))/timesteps

    print("Within Bounds: ",end="")
    print(1 - percent_outside_bounds)
    if percent_outside_bounds > 0:
        print("Exceptions:")
        print(merged_data["Simple Average Return"].index[outside_upper_bounds])
        print(merged_data["Simple Average Return"].index[outside_lower_bounds])


    plt.plot(merged_data["Simple Average Return"].
 ↪index,forecasted_avg_return[dist], color = "red")
    plt.plot(merged_data["Simple Average Return"].index,merged_data["Simple␣
 ↪Average Return"])
    plt.fill_between(merged_data["Simple Average Return"].
 ↪index,forecasted_5p[dist],forecasted_95p[dist],color = "gray", alpha = 0.5)
    plt.ylim(-0.2,0.25)
    plt.title(dist)
    plt.show()
```

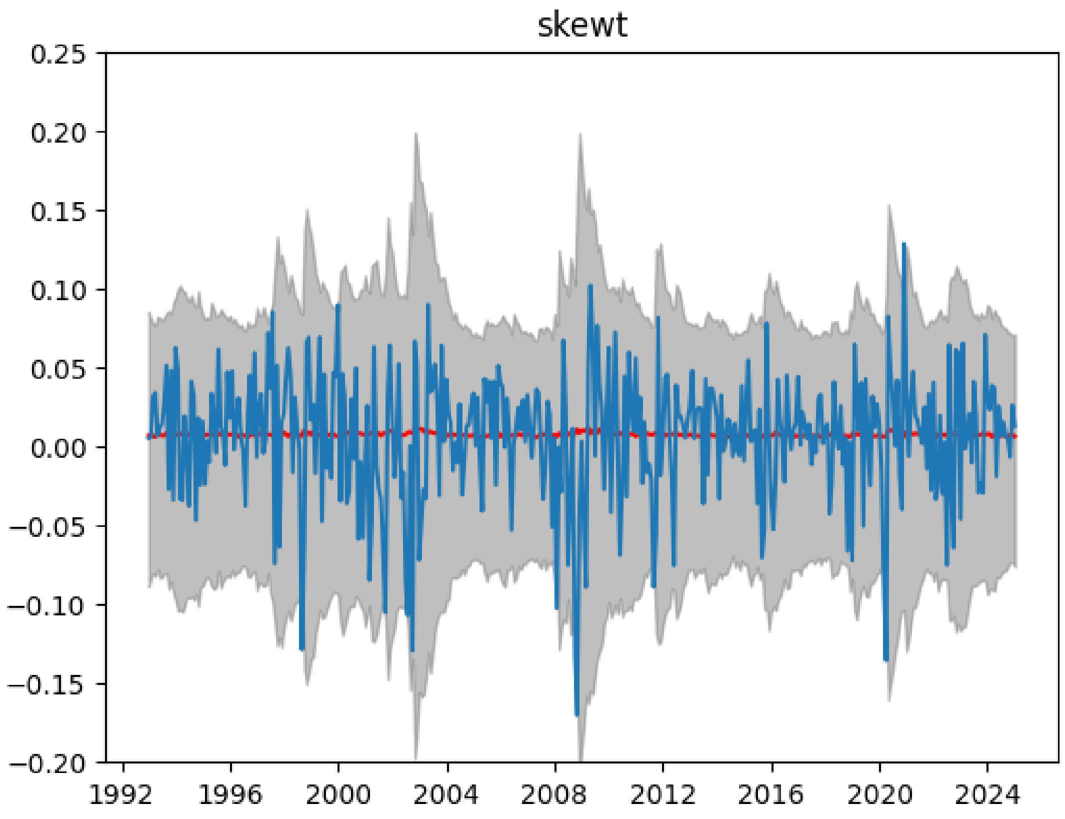
```
Within Bounds: 0.974025974025974
Exceptions:
DatetimeIndex(['2020-11-30'], dtype='datetime64[ns]', name='Date', freq='ME')
DatetimeIndex(['1998-08-31', '2001-09-30', '2002-09-30', '2008-01-31',
               '2008-09-30', '2008-10-31', '2011-08-31', '2020-02-29',
               '2020-03-31'],
              dtype='datetime64[ns]', name='Date', freq=None)
```



```
Within Bounds: 0.9818181818181818
Exceptions:
DatetimeIndex([], dtype='datetime64[ns]', name='Date', freq='ME')
DatetimeIndex(['1998-08-31', '2008-01-31', '2008-09-30', '2008-10-31',
               '2011-08-31', '2020-02-29', '2020-03-31'],
              dtype='datetime64[ns]', name='Date', freq=None)
```



```
Within Bounds: 0.9766233766233766
Exceptions:
DatetimeIndex(['1999-12-31', '2020-11-30'], dtype='datetime64[ns]', name='Date',
freq=None)
DatetimeIndex(['1998-08-31', '2008-01-31', '2008-09-30', '2008-10-31',
               '2011-08-31', '2020-02-29', '2020-03-31'],
              dtype='datetime64[ns]', name='Date', freq=None)
```



Our models fail to accurately model the returns of the basket. This is expected, as we assumed a constant mean for each model, and have created forecasts by computing the average of our simulations, which are centered about the assumed mean. Our forecasts deviate from our assumed mean only slightly, unlike real returns.

However, by excluding the upper and lower 5% quantiles, we can create bounds on our return data, allowing us to forecast, with some confidence, the more extreme scenarios. Caution should be excercised here: our model was not able to predict the start of some of the most extreme shocks, including the 1998 Russian Financial Crisis (August, 1998), the 2008 Global Financial Crisis (September, 2008), and the COVID-19 pandemic (February, 2020)

[360]:
```
#jupyter nbconvert --to pdf "program.ipynb"
```

```
^C
```

4 Maha stuffWe will conduct a goodness of fit test using the squared Mahalanobis distance.

Note that if

$$t = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \\ \vdots \\ t_d \end{pmatrix} \sim t_\nu(\mu, \Sigma)$$

Then we can write

$$t = \mu + \sqrt{\frac{\nu}{W}} X$$

Where $X \sim N_d(0, \Sigma), W \sim \chi^2_\nu$, and W & Z are independent. Recall that the squared Mahalanobis distance for a variable t is given by:

$$D^2 = (t-\mu)^\top \Sigma^{-1} (t-\mu)$$

Substituting $t = \mu + \sqrt{\frac{\nu}{W}} Z$ for X, we get

$$D^2 = \left(\sqrt{\frac{\nu}{W}} Z\right)^\top \Sigma^{-1} \left(\sqrt{\frac{\nu}{W}} Z\right) = \frac{\nu}{W} Z^\top \Sigma^{-1} Z$$

Now, since $Z \sim N_d(0, \Sigma)$, we have that

$$Z^\top \Sigma^{-1} Z \sim \chi^2_d$$

We can let $U = Z^\top \Sigma^{-1} Z \sim \chi^2_d$ so that

$$D^2 = \frac{\nu}{W} U = p \frac{(U/p)}{(W/\nu)}$$

Then,

$$\frac{D^2}{p} = \frac{(U/p)}{(W/\nu)} \sim F(p, \nu)$$

The above QQ Plots show that our Multivariate T distribution closely fits the standaridized residuals, especically when our residuals come from the Student's-t, and skewed Student's-t GARCH models.

[326]:
```python
def Squared_Mahalanobis_distance(x, sigma_inv,mu):
    return np.dot((x-mu),np.dot(sigma_inv,(x-mu)))

print()
for dist in distributions:

    f_samples = f.rvs(len(tickers),fitted_df[dist],size = 10000)
    scale_inv = np.linalg.inv(spearman_cov[dist])
    mu = [fitted_models[(ticker,dist)].params["mu"] for ticker in tickers]
    D_squared_over_p = [Squared_Mahalanobis_distance(row,scale_inv,mu)/␣
↪len(tickers) for row in merged_residuals[dist].T]

    n = len(D_squared_over_p)
    quantiles = (np.arange(1, n + 1) - 0.5) / n


    theoretical_quantiles  =  np.quantile(f_samples, quantiles)
    sample_quantiles = np.quantile(sorted(D_squared_over_p),quantiles)

    plt.plot(theoretical_quantiles,sample_quantiles,color = "blue", marker =␣
↪'o')
    plt.plot(theoretical_quantiles,theoretical_quantiles,color = "red")
    plt.title("QQ Plot for Scaled Squared Mahalanobis Distance vs F(d,nu) -␣
↪underlying GARCH: "+dist)
    plt.show()
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[326], line 10
      7 f_samples = f.rvs(len(tickers),fitted_df[dist],size = 10000)
      8 scale_inv = np.linalg.inv(spearman_cov[dist])
---> 10 mu = [fitted_models[(ticker,dist)].params["mu"] for ticker in tickers]
     11 D_squared_over_p = [Squared_Mahalanobis_distance(row,scale_inv,mu)/␣
 ↪len(tickers) for row in merged_residuals[dist].T]
     13 n = len(D_squared_over_p)

KeyError: ('^GDAXI', 'normal')
```

4.0.1 References

1. https://users.ssc.wisc.edu/~bhansen/papers/ier_94.pdf skewed students t

https://onlinelibrary-wiley-com.libproxy.wlu.ca/doi/pdf/10.1002/9781118673331 reference

ifm https://open.library.ubc.ca/media/stream/pdf/52383/1.0225985/5

arch https://arch.readthedocs.io/en/latest/univariate/generated/arch.univariate.SkewStudent.html

1. Author, *Title of the Book/Article*, Publisher, Year.