Final Bayesian Project 2023

Eyuel Muse

Dario Delgado

JD O’Hea

Introduction

This project involves collecting data on the stock's pricing and returns over a certain period of time. Having a good predictor of volatility is important for trading stocks successfully because it allows traders to make more informed decisions about when to buy and sell stocks. By knowing the level of volatility, traders can determine the level of risk involved in a particular investment and adjust their strategy accordingly. Depending on the trading strategy it can sometimes be even more favourable to trade in highly volatile stock, especially in high frequency trading.

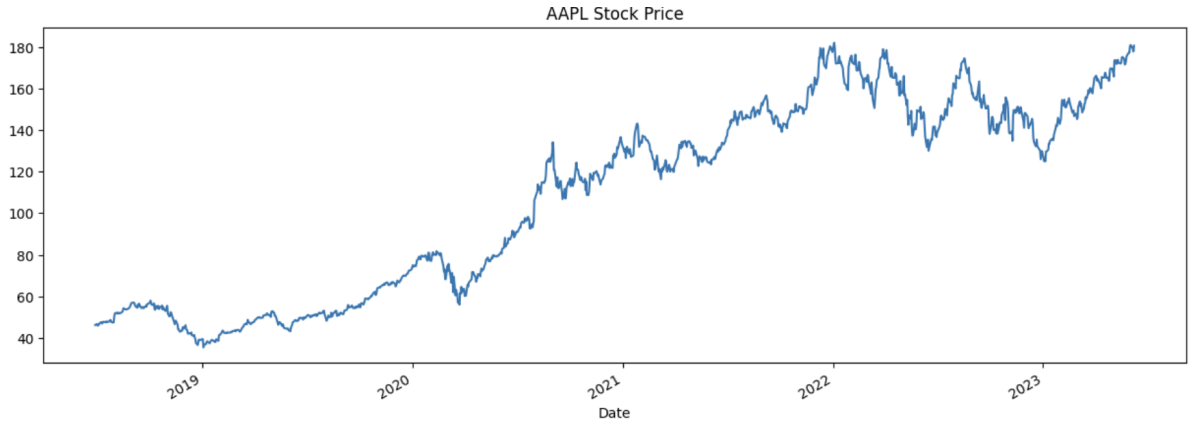
We first set out with the goal of clustering daily volatility of returns. One approach is to employ Gaussian clustering, also known as Gaussian mixture models (GMM). GMM assumes that the data points are generated from a mixture of Gaussian distributions. In the context of stock returns, GMM can be used to identify different clusters or groups of returns with similar volatility characteristics.

Gaussian clustering provides a probabilistic framework for clustering returns based on volatility. It allows for a flexible modelling of the data and can uncover underlying patterns or groups that might not be apparent through other methods.

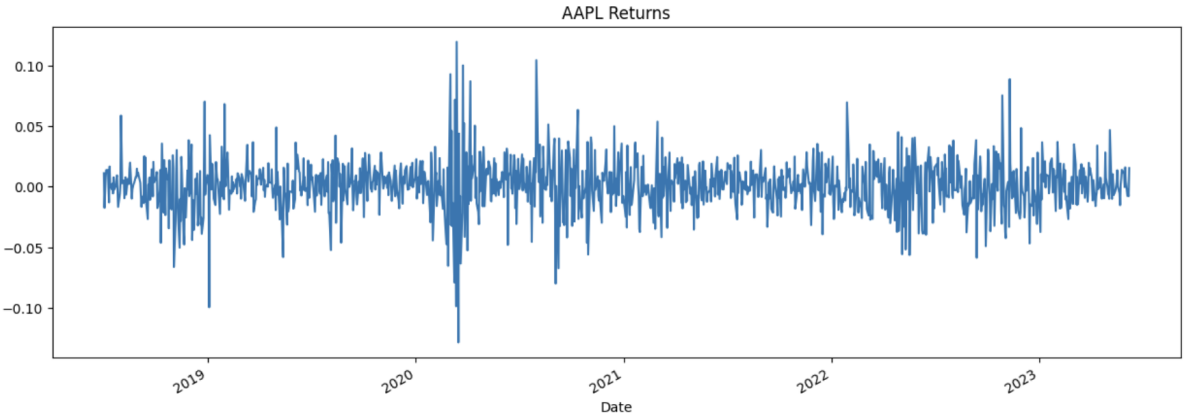
We then go on to use hierarchical linear regression to examine the relationships between variables at different levels of our predefined hierarchies. The overall distribution of volatility is at the top of the hierarchy, and then we use the hierarchical linear regression to learn a distribution of parameters for each cluster that we have obtained from our gaussian mixture model.

The Dataset

Our code uses the `yfinance` library to download daily stock data for Apple Inc. (`AAPL`) from June 29th 2018 to June 9th 2023, 5 years of daily data. The `get_data()` function takes in a `ticker` parameter for the stock symbol, `start_date` and `end_date` for the date range, and an `interval` parameter to specify the frequency of the data (in this case, daily). The data is returned as a Pandas DataFrame object named `df`.



Plotting the returns using the `get_returns()` function we see:



This function takes a Pandas DataFrame object as input. It adds a new column called 'returns' to the DataFrame, which calculates the percentage change in the 'Close' column from one day to the next. The function then returns the modified DataFrame.

Volatility is a statistical measure of the dispersion of returns for a given security or market index. It is often described as the degree of variation of a financial instrument's price over time. A higher volatility means that the price can fluctuate

dramatically over a short period of time, while a lower volatility means that the price is more stable.

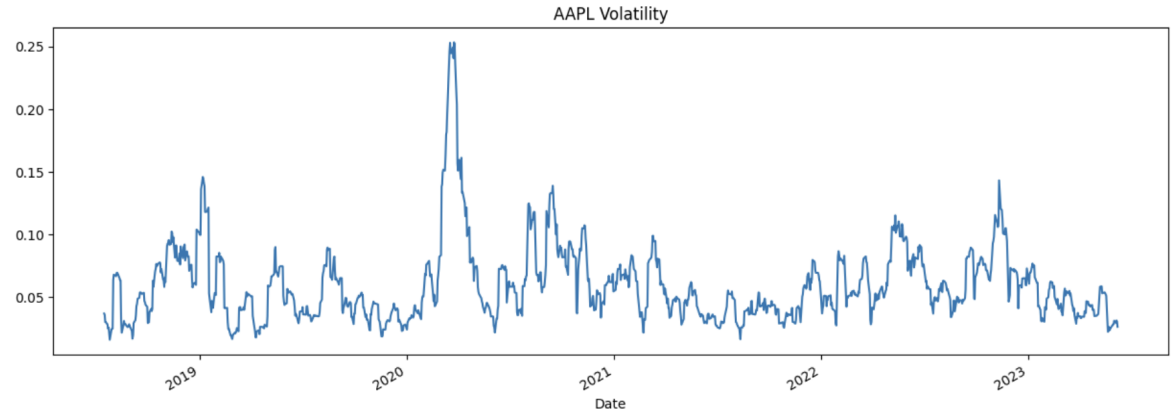
The formula for calculating volatility over a time horizon is:

$$\sigma_T = \sigma\sqrt{T}$$

where σ_T represents volatility over a time horizon, σ is the standard deviation of returns, and T is the number of periods in a time horizon.

In the context of the stock market, volatility refers to the degree of variation or fluctuation in the price of a stock over time. It is an essential aspect of financial markets and plays a crucial role in investment decision-making. Volatility is typically measured using statistical metrics such as standard deviation, variance, or historical price data.

The line of code `df['volatility'] = df['returns'].rolling(10, closed='left').std() * np.sqrt(10)` calculates the volatility of a stock over a period of 10 days based on the daily returns of the stock. The `rolling()` function creates a rolling window of 10 days, and `std()` calculates the standard deviation of the returns within that window. The `np.sqrt(10)` term scales the volatility to an annualised value based on the assumption that there are 252 trading days in a year. The resulting volatility values are stored in a new column called `'volatility'` in the `df` DataFrame.



PyMC Gaussian Clustering of Volatility Periods

PyMC is a Python library for probabilistic programming. Probabilistic programming allows users to specify probabilistic models in code, PyMC allows users to specify models using a probabilistic programming language that closely mirrors the

mathematical notation used in Bayesian statistics, making it easier to reason about uncertainty and make probabilistic predictions.

In the context of the provided code, PyMC is used to cluster stock data into different volatility categories based on previous stock prices. Specifically, the PyMC clustering implementation aims to classify the volatility of a stock on a given day based on the previous 10 days of stock prices.

The PyMC code consists of four main parts: specifying the prior distributions, specifying the likelihood distribution, sampling from the posterior distribution, and interpreting the results.

First, the prior distributions for the model parameters are specified. This includes the prior distributions for the means and standard deviations of the clusters, as well as the prior distribution for the weights of the clusters. These prior distributions represent the user's initial beliefs about the values of these parameters before observing any data.

Next, the likelihood distribution for the observed data is specified. In this case, a Normal mixture model is used to model the data, where each cluster represents a different volatility category. The user specifies the weights, means, and standard deviations of the clusters, and the observed data is used to estimate the posterior distribution of these parameters.

After specifying the prior and likelihood distributions, the user samples from the posterior distribution of the model using the `pm.sample()` function. This function takes two arguments: the number of samples to draw from the posterior distribution (`2` in this case), and the number of warm-up samples to discard (`tune=10` in this case). The resulting samples are used to estimate the posterior distribution of the model parameters, which can be used to make probabilistic predictions about future stock prices.

Interpreting the results of the PyMC model involves analysing the posterior distributions of the model parameters. This includes examining the posterior means and standard deviations of the cluster means and standard deviations, as well as the posterior distribution of the cluster weights. These results can be used to understand the different volatility categories and make more informed decisions about when to buy and sell stocks.

Overall, PyMC is a powerful tool for probabilistic programming and Bayesian modelling. The provided code demonstrates how PyMC can be used to cluster stock data into different volatility categories, but the library can be used for a wide range of other applications as well.

Below is a detailed explanation of every line of code used in this clustering implementation:

```
with pm.Model() as model:
```

This line of code initiates the PyMC model.

```
k = 3
```

The variable `k` is set to 3, representing the number of clusters we want to classify our data into.

```
mus = pm.Normal('mus', mu=mus, sd=10, shape=k)
```

This line of code specifies the prior distribution of the means for the clusters using a normal distribution. The `'mus'` argument represents the name of the variable, `mu` represents the mean of the variable, `sd` represents the standard deviation of the variable, and `shape` represents the number of clusters.

```
sigmas = pm.HalfNormal('sigmas', sd=10, shape=k)
```

This line of code specifies the prior distribution of the standard deviations for the clusters using a half-normal distribution. The `'sigmas'` argument represents the name of the variable, `sd` represents the standard deviation of the variable, and `shape` represents the number of clusters.

```
weights = pm.Dirichlet('weights', a=np.ones(k))
```

This line of code specifies the prior distribution of the weights for the clusters using a Dirichlet distribution. The `'weights'` argument represents the name of the variable, `a` represents the concentration parameter of the Dirichlet distribution, and `np.ones(k)` represents an array of ones with a length of `k`.

```
likelihood = pm.NormalMixture('likelihood', w=weights, mu=mus, sd=sigmas, observed=val
ues)
```

This line of code specifies the likelihood distribution for the observed data. The `'likelihood'` argument represents the name of the variable, `w` represents the weights of the clusters, `mu` represents the means of the clusters, `sd` represents the standard deviations of the clusters, and `observed` represents the observed data.

```
trace = pm.sample(2000, tune=1000)
```

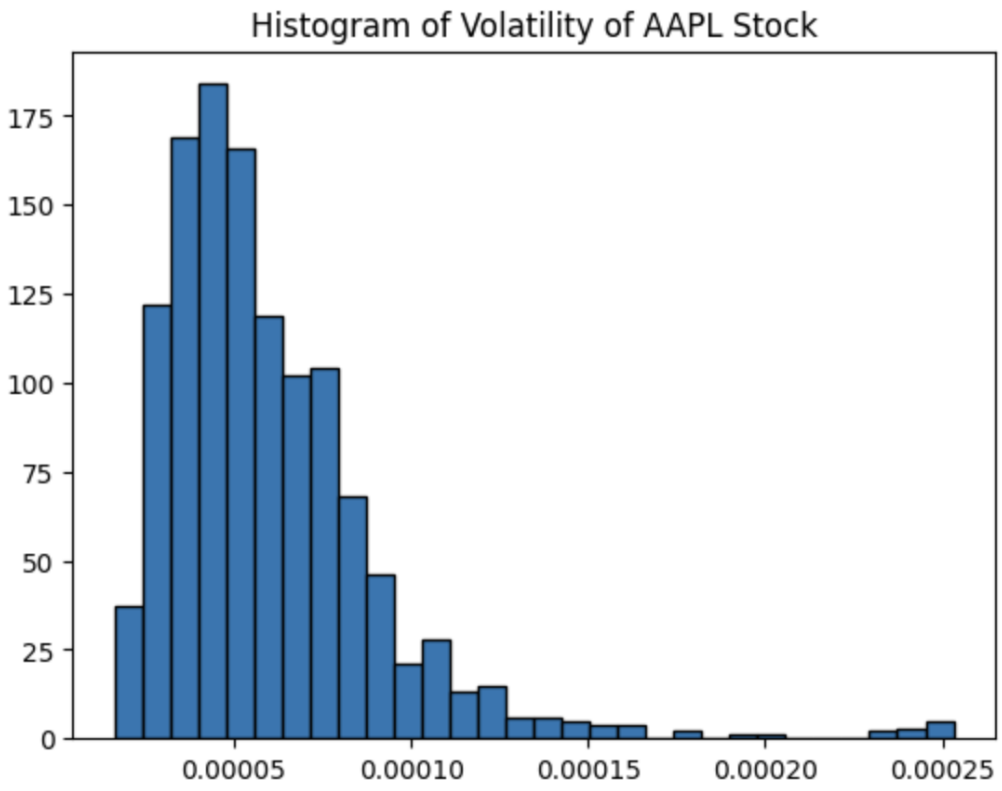
This line of code samples from the posterior distribution of the model. The 2000 argument represents the number of samples to draw from the posterior distribution, and `tune=1000` represents the number of warm-up samples to discard.

We chose a low value for samples and tuning because the clustering is an arb

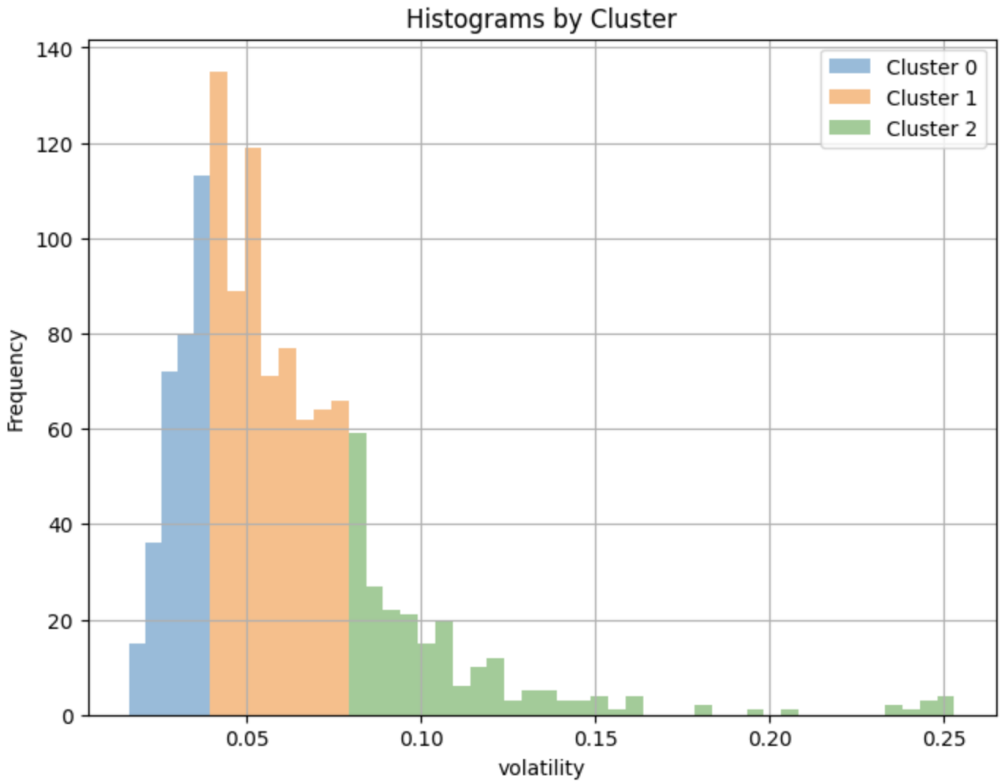
Overall, this code is used to cluster stock data into different volatility categories based on previous stock prices.

The Results from PyMC Gaussian Mixture Clustering

Here is a plot of the distribution of daily volatility measure before clustering:

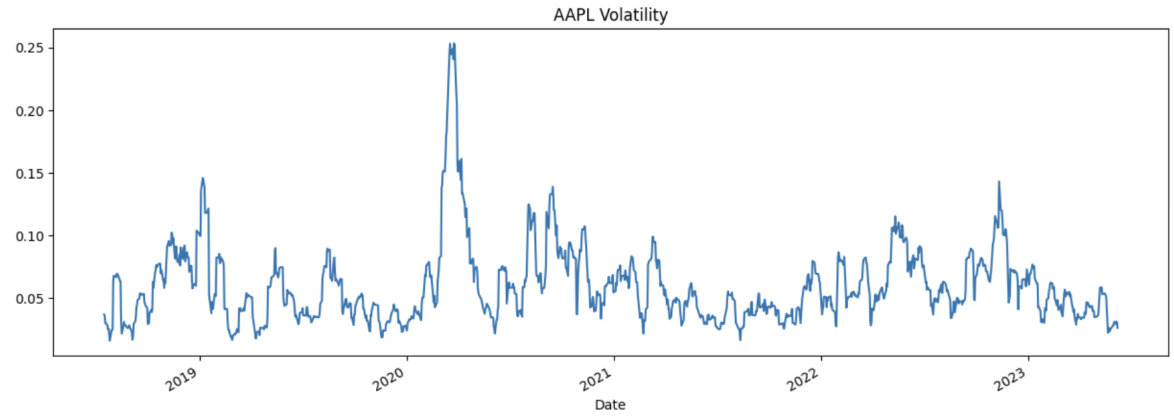


After applying our gaussian mixture model in PyMc we get the following clusters:



If we plot each time stamp with their respective cluster we get:

If we look to the original volatility plot of AAPL we can clearly see that the gaussian is identifying the areas of high volatility and plotting (in green).



Next Steps

In the previous section we explained how we did the clustering of the different volatility periods using bayesian methods. The end goal of doing that is what we will see in this section.

Now that we have each time stamp classified as a period of low, medium or higher volatility, we will try to fit a linear regression using bayesian hierarchical model. We chose to use a hierarchical model because our resulting model will contain posterior distributions for each cluster of volatility and one for the overall distribution of volatilities for that time period.

Bayesian Hierarchical Linear Regression

Bayesian hierarchical linear regression is a statistical method used to analyse data that has a hierarchical or nested structure. In this method, the data is partitioned into groups or clusters, and each group is assumed to have its own set of regression coefficients. The coefficients are then modelled as random variables with distributions that are themselves modelled as random variables. This nesting of random variables leads to a hierarchical structure.

In the context of linear regression, the goal of Bayesian hierarchical linear regression is to estimate the regression coefficients for each group or cluster, as well as the hyper-parameters that determine the distribution of these coefficients across groups. The hyper-parameters are often referred to as "global" parameters, as they describe the distribution of the coefficients across all groups. The coefficients themselves are

often referred to as "local" parameters, as they describe the coefficients for each individual group.

The advantage of Bayesian hierarchical linear regression over traditional linear regression is that it allows for more flexibility in modelling the data. By allowing the coefficients to vary across groups, the method can account for differences in the relationship between the predictor variables and the response variable across different groups. This can be particularly useful when the data has a hierarchical or nested structure, such as when the data is collected from individuals who are nested within groups or clusters.

One of the key features of Bayesian hierarchical linear regression is the use of prior distributions to specify the distribution of the regression coefficients and hyper-parameters. These prior distributions allow the modeller (us) to incorporate prior beliefs or knowledge about the parameters into the analysis. The prior distributions also help to regularise the estimates of the coefficients and hyper-parameters, which can improve the accuracy of the estimates when the data is limited or noisy.

The code

```
with pm.Model() as hierarchical_model:
```

This line of code initialises a PyMC3 model object called `hierarchical_model`, which will contain all of the model components and data for the analysis.

```
α_μ_tmp = pm.Normal('α_μ_tmp', mu=0.5, sd=0.1)
```

This line of code defines a normal prior distribution for the global mean of the `α` coefficients. The `μ` parameter of the prior distribution is set to 0.5, which means that the model expects the mean of the `α` coefficients to be around 0.5. The `sd` parameter is set to 0.1, which provides some uncertainty around this prior belief.

```
α_σ_tmp = pm.HalfNormal('α_σ_tmp', .05)
```

This line of code defines a half-normal prior distribution for the standard deviation of the `α` coefficients.

```
β_μ = pm.Normal('β_μ', mu=0.5, sd=0.1)
```

This line of code defines a normal prior distribution for the global mean of the `β` coefficients.

```
β_σ = pm.HalfNormal('β_σ', .05)
```

This line of code defines a half-normal prior distribution for the standard deviation of the `β` coefficients.

```
α_tmp = pm.Normal('α_tmp', mu=α_μ_tmp, sd=α_σ_tmp, shape=n_clusters)
```

This line of code defines a normal prior distribution for the `α` coefficients specific to each Cluster. The `mu` parameter is set to the global mean `α_μ_tmp`, and the `sd` parameter is set to the global standard deviation `α_σ_tmp`. The `shape` parameter specifies that there will be `n_clusters` different `α` coefficients, one for each Cluster.

```
β = pm.Normal('β', mu=β_μ, sd=β_σ, shape=n_clusters)
```

This line of code defines a normal prior distribution for the `β` coefficients specific to each Cluster. The `mu` and `sd` parameters are set to the same values as the `β_μ` and `β_σ` priors, and the `shape` parameter specifies that there will be `n_clusters` different `β` coefficients, one for each Cluster

```
eps = pm.HalfCauchy('eps', .5)
```

This line of code defines a half-Cauchy prior distribution for the model error.

```
return_est = α_tmp[train_cluster_idx] + β[train_cluster_idx]
```

This line of code calculates the estimated volatility for each observation in the dataset, based on the `α` and `β` coefficients specific to the Cluster for that observation. The `train_cluster_idx` variable is an index that specifies the Cluster for each observation.

```
return_like = pm.Normal('return_like', mu=return_est, sd=eps, observed=df.volatility)
```

This line of code specifies the likelihood function for the model. It assumes that the observed `volatility` data for each observation is normally distributed around the estimated return for that observation, with a standard deviation of `eps`. The `observed` parameter specifies that the `volatility` data is the observed data that we are trying to model.

```
with hierarchical_model:
    hierarchical_trace = pm.sample(2000, tune=1000, target_accept=.9)
```

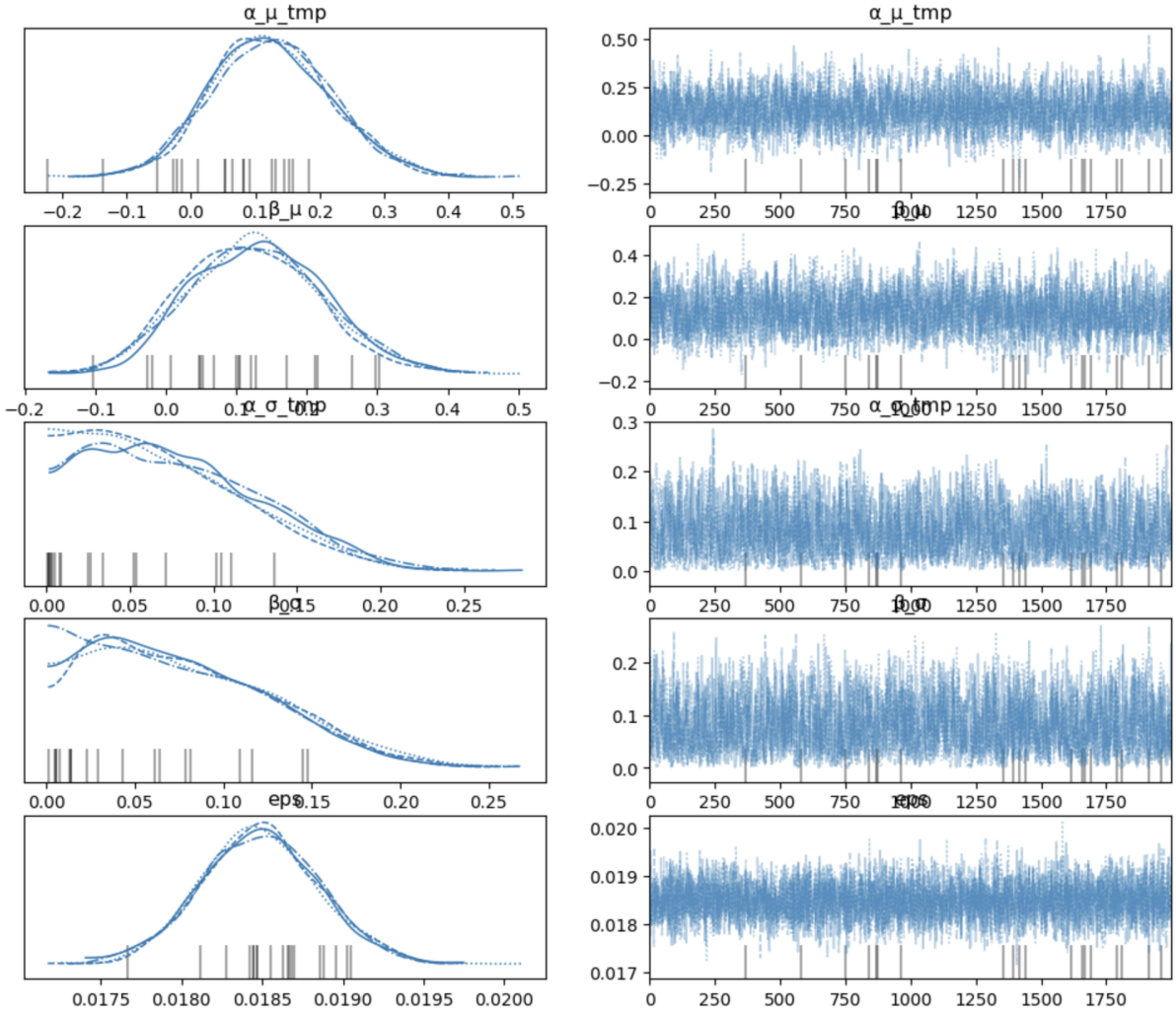
This line of code runs the NUTS (the default for the PyMC3 library) algorithm to obtain samples from the posterior distribution of the model parameters. The `sample` function takes three arguments: the number of samples to draw (`2000`), the number of tuning steps to take before starting to draw samples (`1000`), and the target acceptance rate for the Metropolis-Hastings algorithm (`.9`).

```
pm.traceplot(hierarchical_trace, var_names=['α_μ_tmp', 'β_μ', 'α_σ_tmp', 'β_σ', 'ep
s'])
```

This line of code generates trace plots of the posterior distributions of the model parameters specified in the `var_names` argument.

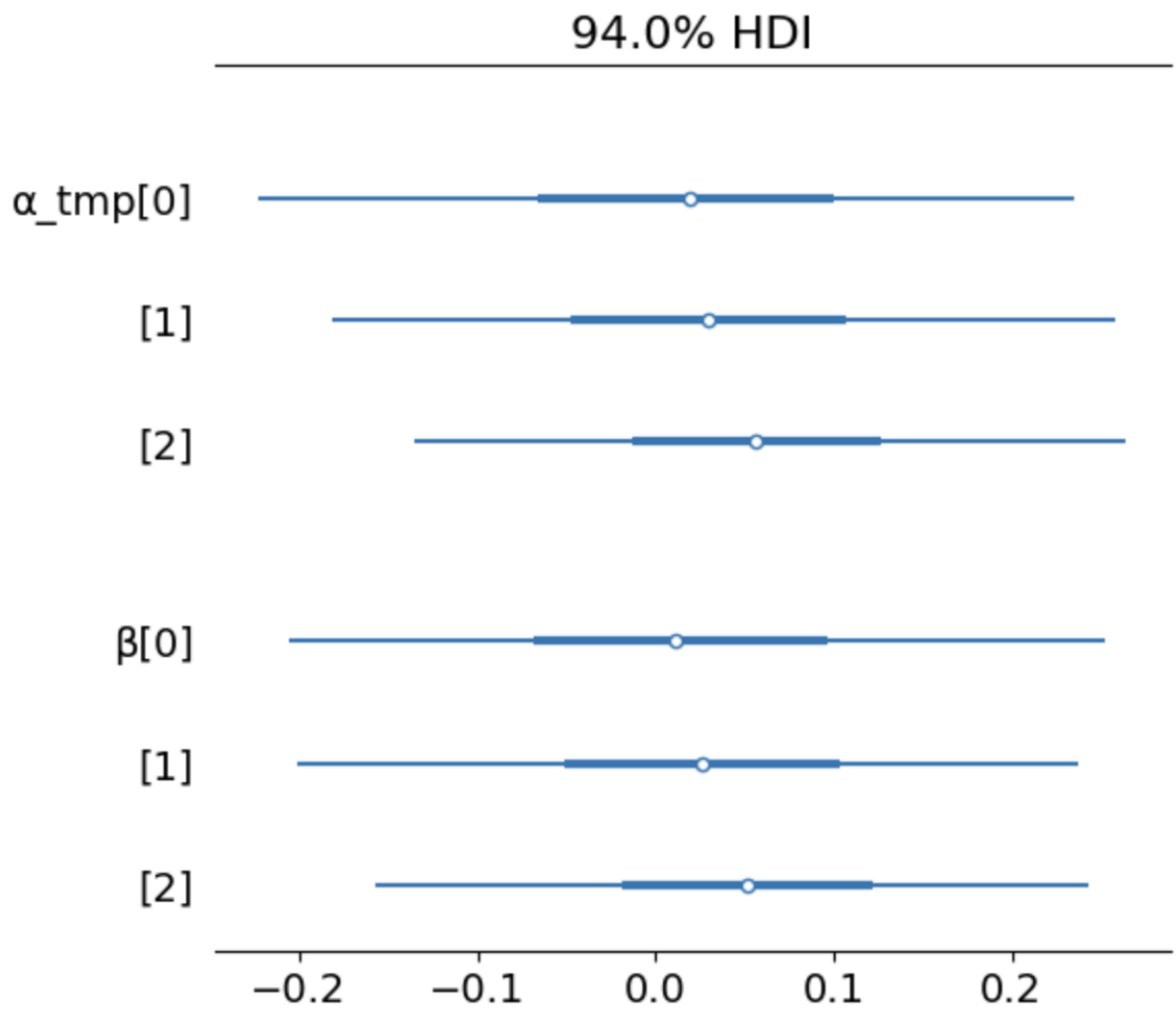
Results

In the Following chart we see the results of the model for each one of the variables.



As we can see in the above plot that the chain do converge for all of the different variables, which is good news.

Having uncertainty quantification of some of our estimates is one of the powerful things about Bayesian modelling. We have got a Bayesian credible interval for the volatility of different the different clusters.



From what we can see above, there is a lot of uncertainty in our estimates. However, we can see the for both, intercept and slop, if we look a the mean estimates, we can see they agree with what we see in the data. Cluster 0 being the one with the least volatility and Cluster 2 the one with the highest

10 Cents on NUTS

NUTS is a powerful sampling algorithm that can be used to draw samples from complex high-dimensional distributions. However, it can be computationally expensive, especially for large datasets or models with many parameters. In addition, it is important to check for convergence and autocorrelation in the samples drawn by the sampler to ensure that it has explored the posterior distribution sufficiently. Trace plots and other diagnostic plots can be used to check for these issues.

The "No U-Turn" part of the algorithm refers to the fact that the simulation is stopped when the trajectory of the particle begins to turn back on itself, indicating that it has explored the posterior distribution sufficiently. This prevents the sampler from getting stuck in high-density regions of the posterior distribution.

The basic idea behind NUTS is to simulate the trajectory of a particle moving around in a potential energy landscape defined by the posterior distribution of a model. The particle's position and momentum are updated at each step of the simulation using the Hamiltonian equations of motion. The path length and step size are determined adaptively during the simulation, so that the particle explores the posterior distribution efficiently.

The No-U-Turn Sampler (NUTS) is a Markov Chain Monte Carlo (MCMC) algorithm used to draw samples from the posterior distribution of a model. It is a variant of the Hamiltonian Monte Carlo (HMC) algorithm that uses a recursive algorithm to automatically determine the optimal step size and path length for each iteration of the sampler.

Conclusions

In conclusion, this study provides a comprehensive overview of Bayesian analysis and its applications in finance and statistics. The topics covered range from volatility and PyMC clustering to Bayesian hierarchical linear regression and the NUTS algorithm.

By understanding the principles and methods, practitioners can gain a powerful tool for making probabilistic predictions about future outcomes, with the ability to incorporate prior beliefs and knowledge about a system.