**Project Scope**

The goal of this project is to create and compare different models to forecast the volatility of the US market. The intuition on which this project is based upon is that conditional variance present auto-regressive properties and can be modelled using GARCH models. Most GARCH models make primarily linear assumptions about the auto-regressive patterns. This begs the question if we can introduce non-linear relationships for more flexibility.

The problem of this project is a regression, the dataset is labeled.

**Summary of methodology**

The first step in this study is to forecast the variance of a time series based on the series past results, but the problem is that we do not directly observe the time-series variance.

To solve this problem, we need a tiny bit of probability theory and a loose assumption on our data. In fact, we can prove that the variance of our target variable equals the expectation of the squared mean-transformed variable.

This information allows us to conclude that if we train a model to minimize MSE on the squared transformed variable, that model becomes an estimator for the conditional variance, we’ll use this idea to create our label. The features, on the other hand, will be the lagged returns, as the conditional variance is auto-regressive.

One limitation of using linear regression models for variance prediction is that they may produce negative predictions, which is not ideal. However, Decision Trees and Random Forests can be useful in this context since they always produce non-negative predictions when trained on squared data.

This is because squared data is always non-negative, and Decision Trees are designed to work with such data. In contrast, a raw linear regression model may not always produce non-negative predictions.

It's worth noting that Gradient Boosted Trees may not be the best choice for variance prediction since they involve a weighted sum of Decision Trees. A single negative weight could result in negative predictions, which is not desirable. Therefore, it's important to consider the limitations of different models and choose the most appropriate one for the task at hand.

Fortunately, many popular Boosting libraries allow for the definition of custom loss functions, which can help address this issue. By defining a custom loss function that enforces non-negativity of predictions, we can use Gradient Boosted Trees for variance prediction while still ensuring that our predictions are valid.

With this approach, we can take advantage of the power of Gradient Boosting while also mitigating the risk of negative predictions. It's important to carefully consider the choice of loss function and ensure that it is appropriate for the specific problem at hand.

A way to solve this potential negativity in the output of the Gradient Boosting algorithm, is to impose exponentiation to our distribution, doing so we’ll simply need to implement a log-likelihood loss function and calculate gradient and hessian.

CODE

This is part simply gives an intuition of what each line of code does, a more exhaustive explanation is given directly on Jupyter.

[2-7] (**Data description**)

First is downloading the data. I got all my data from Yahoo Finance, in fact I used the historic data of the S&P500 as a representation of the US market, from 1984-10-01 to 2022-10-01. I used the ‘Close’ price to calculate the daily returns.

[8-9]

Later I created a test dataset, a validation data set and a test dataset. The first one has 9378 data, the other two 100 each. Notice that I am performing a z-Normalization. This will make the subsequent GARCH model more robust to degenerate scaling of the time-series.

The first graph shows the whole spectrum of data I will be working with.

[10-15]

The following piece of code allows us to find the optimal lag, in fact we know that conditional variance is auto-regressive, but we don’t know to which extent. So basically, I am creating random forests with a lag up to 10 and select the one that minimizes MAE. My result is that the model that best perform on my Validation dataset is the random forest with Lag 2. This implies that going forward I will be working with models with a Lag of 2.

In the following line, we create the new train dataset on which we are going to train the models to forecast the variance. It includes the square value and the lagged value up to 2, so we have the square of the value at time t, then the value at time t-1 and t-2.

[16-19]

Here I performed hyperparameter tuning using GridSearchCV to find the best hyperparameters for the Random Forest Regressor model. I defined a range of values for each of the hyperparameters to be tested and assigned them to a parameter grid.

Then, I created an instance of the GridSearchCV class with the specified parameters, and fit the model using the training data. After fitting the model, I obtained the best parameters for the model from the results of the grid search.

Finally, I created a new instance of the Random Forest Regressor model with the best hyperparameters obtained from the grid search and fit the model using the training data.

[20]

To generate forecasts, we can use Monte Carlo simulation by sampling from the predicted distribution of conditional variances. This is necessary because our model is limited to predicting conditional variance only one step ahead.

To estimate a 90% forecast interval, we can assume that the noise is Gaussian and use the predicted variance distribution to construct the interval. Specifically, we can simulate multiple future paths by drawing samples from the predicted variance distribution, and then combine these paths to estimate the 90% interval.

Overall, this code generates samples using Monte Carlo simulation to estimate the variance predicted by the forest model and then calculates the confidence interval for each time point based on the distribution of the samples.

[21]

Now remember that we said that to use the gradient boosting we needed to use a log-likelihood loss function, here is the given code for a custom loss function that can be used for Gradient Boosting to predict non-negative variances.

This loss function is defined in terms of the predicted variance y\_pred and the true variance y\_true, which are obtained from the input data. It computes the negative log-likelihood of the predicted variance given the true variance, assuming that the noise is Gaussian.

The negative log-likelihood is returned as the loss value, with a tag of "loglike" to identify it. The third argument (False) indicates that the loss value should not be automatically transformed by the Booster.

It's important to note that since boosting minimizes the loss function, the negative log-likelihood is used instead of the log-likelihood to maximize the likelihood. The same applies to the gradient and Hessian used in the optimization process.

Then we have a code for the gradient and Hessian computation for the custom loss function.

This function computes the gradient and Hessian of the negative log-likelihood with respect to the predicted variance y\_pred. It uses the true variance y\_true from the input data and assumes that the noise is Gaussian.

To compute the gradient and Hessian, the function first computes the exponential of the predicted variance (exp\_pred). Then, it applies the chain rule to compute the gradient and Hessian of the negative log-likelihood with respect to the predicted variance.

The gradient and Hessian are returned as negative values, which is required for the optimization algorithm used by the Booster.

[22]

The code provided is an implementation of training and evaluating a LightGBM model with a custom loss function for predicting conditional variance.

[23]

Here, we are fitting an ARCH model to the training data and then using it to generate variance forecasts for the test set. We use the arch\_model function from the arch library to specify the model and then call the fit method to train it. We set the update\_freq parameter to 2 to print the optimization progress every 2 iterations.

After fitting the model, we call the forecast method to generate forecasts for the variance of the test set. We set the horizon parameter to the length of the test set to indicate that we want to generate forecasts for all the remaining observations. We also set the reindex parameter to False to avoid reindexing the data.

To generate samples from the fitted model, we use the forecast method again but this time set the simulations parameter to the desired number of simulations. This generates simulations number of Monte Carlo samples for each point in the forecasted variance series. We then calculate the standard deviation, lower and upper quantiles of the samples to obtain the forecast interval.

In addition, we also generate a forecast interval using a kernel density estimate (KDE) of the training data. This is done using the gaussian\_kde function from the scipy.stats library to estimate the density of the training data and then resampling from it using the resample method. We set the number of resamples to 50000\*len(test) to obtain a sufficient number of samples for accurate quantile estimation. Finally, we calculate the lower and upper quantiles of the samples to obtain the forecast interval.

[24]

This code is creating a plot with four subplots, each showing the forecasted values of a different model for a given time series. The four models being compared are Random Forest ARCH, Gradient Boosting ARCH, GARCH(2,2), and i.i.d. Kernel Density.

For each model, the code generates forecasted values and forecast intervals using different methods, such as simulation or quantiles, and calculates the log-likelihood of the test set given the model's forecasted values.

The resulting plot shows the last 50 observations of the training set, the test set, and the 90% forecast interval for each model. It also displays the log-likelihood of the test set given the model's forecasted values.

**Project results/ Conclusion**

How we can see, all three of my model outperformed the kernel density benchmark. The best performing model was the GARCH(2,2), but there is not a significant distance from the Random Forest and Gradient Boosting ARCH models results. We must keep in mind though that a single evaluation doesn't allow any generalizing conclusions.

Even though I consider the results of my project to be satisfying, I would like to point out possible improvements that for several constraints I was not able to implement.

Possible enhancement could be a better choice of conditional distributions. All our models, except the kernel density benchmark, worked with Gaussian assumptions. In for financial time-series, this might obviously be a sub-optimal choice. If, for example, we wanted to account for heavy conditional tails, a Student's T-distribution would be better suited.

On top of that, for computational reasons when implementing the grid-search for the random forest, I fed the model a limited number of possible values for the hyperparameters.

A picture containing calendar

Description automatically generated