Predicting Financial Market Volatility

Jack St. John
Virginia Tech

Kyle Weisbaum
Virginia Tech

Abstract

Financial markets exhibit a behavior known as "volatility clustering." Periods of high volatility tend to be followed by more periods of high volatility. The behavior makes the volatility of financial markets amenable to prediction using time-series models. This paper attempts to improve on the predictive ability of the GARCH model using a gradient boosted autoregressive model and a recurrent neural network. Currently, the GARCH model outperforms both of the other methods, but we are still figuring out the details with regards to hyperparameter tuning.

1 Introduction

To model volatility we assume returns follow a process of the form

$$R_t = \alpha + \epsilon_t$$

When specifying this functional form, we make no assumptions about the specific distribution of ϵ, only that it has mean zero and variance σ_t^2. After squaring and taking expectations we get:

$$E\left[(R_t - \alpha)^2\right] = \sigma_t^2$$

In general, we're interested in predicting values of σ_t using its previous values, which may be estimated using previous squared deviations in returns.

2 Methods

To forecast volatility we used a GARCH type model, a gradient boosted autoregressive model and a recurrent neural network.

2.1 GARCH

The GARCH model specifies an ARMA(p,q) process specified for σ_t^2. It takes the form.

$$R_t = a + a_1 R_{t-1} + \epsilon_t, \qquad \epsilon_t \sim N(0, \sigma_t^2)$$

$$\sigma_t^2 = \alpha + \sum_{i=1}^{p} \rho_i \epsilon_{t-i}^2 + \sum_{i=1}^{q} \gamma_i \sigma_{t-i}^2$$

These models remain extensively used for risk management at major financial firms due to their relative simplicity and remarkable robustness. The ARMA process allows the model to pick up on both short-term dependencies via the AR term, and long-term dependencies in the time series via the MA term. GARCH(1,1) was used in the project. The simple specification is standard and usually gives the best results.

The GARCH enjoys substantial popularity because it is good for prediction and because in addition to the fact that you can use it as a model, it defines a stochastic process. So after you fit the model, you can then use it to run monte-carlo simulations to assess financial risk. GARCH models are often used over log-normal models because they capture the excess kurtosis of financial data series.

2.2 Boosted Autoregressive Model

The Boosted Autoregressive model is similar to the GARCH, but with only AR terms and no MA terms. We then apply gradient boosting to see if we can improve the predictive ability.

Gradient boosting first considers a linear regression model. It makes a tree by splitting the residuals from that model based on the predictors, then it fits more linear regression models to predict the subsets of the residuals. It repeats this process a number of times specified by the user, building a large tree. Predictions may then be made by summing the results of the leaves in the tree.

In general the process is as follows:

1. Fit linear model $y = f(X)$. Get residuals based on fitted values $\epsilon = y - \hat{y}$
2. Split residuals using some decision rule based on X. $\epsilon_1 \cup \epsilon_2 = \epsilon$
3. Fit linear models $\epsilon_1 = f(X)$ and $\epsilon_2 = f(X)$
4. Repeat

The features that we let the boosting algorithm use are the previous 10 values of the squared returns.

2.3 Recurrent Neural Network

A recurrent neural network is a type of neural network that takes feedback from a hidden layer as input for a later iteration of a prior layer. This feedback feature separates recurrent neural networks from others as a notably great candidate for modelling time series data. Conceptually, the network used in this project functions as a set of long short-term memory network (LSTM) layers. Any given layer has input, forget and output gates to manage the information flow into subsequent layers. The forget gate determines what information to forget from memory, while the input gate determines what new information to include. Utilizing LSTM helps us avoid running into a vanishing gradient problem, which is often faced during training. We've specifically implemented four LSTM layers and four separate dropout layers to prevent overfitting. We plan to continue editing the structure of this network in attempt to decrease the overall sample MSE and boost performance. The ability of the RNN to have “long memory" should make it comparable to the GARCH which can capture long-term dependence with the MA term. A big theoretical advantage over the GARCH should be the fact that the activation function allows the RNN to capture non-linear forms of dependence in the data, while the GARCH is a strictly linear model.

The following equations represent the input, forget, and output gates in an LSTM respectively:

$$i_t = \sigma(w_i[h_{t-1}, x_t] + b_i)$$
$$f_t = \sigma(w_f[h_{t-1}, x_t] + b_f)$$
$$o_t = \sigma(w_o[h_{t-1}, x_t] + b_o)$$

$i_t =$ input gate, $f_t =$ forget gate, $o_t =$ output gate, $w_x =$ weight for the respective gate(x) neurons

$h_{t-1} =$ output of the previous LSTM layer, $x_t =$ input at the current timestamp

$b_x =$ biases for the respective gates(x)

Lastly, The equations for the cell state, candidate cell state and final output:

$$\tilde{c_t} = \tanh(w_c[h_{t-1}, x_t] + b_c)$$

$$c_t = f_t * c_{t-1} + i_t * \tilde{c_t}$$
$$h_t = o_t * \tanh(c^t)$$

$c_t =$ cell memory at t, $\tilde{c_t} =$ candidate for cell state at t, $h_t =$ output of the LSTM layer

3 Results

The models were estimated on the five day sum of daily squared returns of the TLT long-term US Treasury bond ETF. Except for the RNN, the models were estimated on the 450 observations proceeding each individual out of sample observation.

The RNN was estimated on the training set, but was not re-fit for every observation. The RNN just took too long to fit. This likely explains it's sub-par performance compared to the other models, but the one we did do a re-fitting for every observation it still under-performed.

We couldn't really adjust the GARCH to predict in the same manner. The MA term depends on previous period errors, so it makes that rather difficult. Re-fitting the model for every observation then predicting forward one is more natural with the GARCH.

Despite the fact that the GARCH did the best, they're really all quite good. This is MSE, so the difference between the RMSEs, which is in the original units, is really quite low.

GARCH: 11.277
Boosted Autoregressive: 11.866
Recursive Neural Net: 14.78

Also, we tried using data on smaller time-scales, in which the volatility clustering effect should be stronger, but no significant change in performance was noted.

3.1 Model tuning and alterations

3.1.1 Boosted Autoregressive Model

The XGBoost package has quite a few hyperparameters.

α and λ act as regularization parameters.

The "max depth" parameter is self explanatory. It sets a maximum for how large the trees can get.

The "num estimators" parameters denotes the number of trees formed.

The "colsample_bytree" hyperparameter acts like a dropout parameter. It makes it so that the algorithm uses only a subset of the data when building trees.

To tune these hyperparameters we use grid-search cross-validation which took awhile to run, but led to fairly large improvements in performance. Oddly, the best set of hyperparameters involved having a fairly large number of trees of depth 1. I'm not sure why exactly this is the case, but it does predict better out of sample than other combinations of parameters. The model substantially outperformed a basic autoregressive model.

The output from grid-search is below:

```
XGBRegressor(alpha=0, base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=0.5000000000000001, gamma=0,
             gpu_id=-1, importance_type='gain', interaction_constraints='',
             learning_rate=0.1, max_delta_step=0, max_depth=1,
             min_child_weight=1, missing=nan, monotone_constraints='()',
             n_estimators=80, n_jobs=0, num_parallel_tree=1,
             objective='reg:squarederror', random_state=0, reg_alpha=0,
             reg_lambda=0, scale_pos_weight=1, subsample=1, tree_method='exact',
             validate_parameters=1, verbosity=None)
```

3.1.2 Recurrent Neural Network

The first thing tested and altered was the dropout rate in the dropout layers. We were worried we were over-fitting after realizing that the model failed to pick up on a lot of the variability spikes present in the testing data. By increasing the drop rate from 20% to 38 % we saw a decrease in MSE.

Next, we analyzed how many memory layers we were using, as well as what the unit size was for said layers. We discovered that cutting back the LSTM layer count from 4 to 2 reduced the MSE and sped up computation.

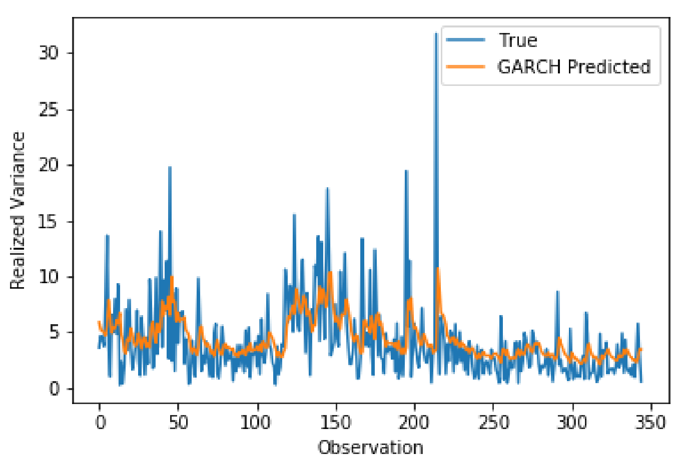
We experimented with epoch size, trying a number of different values ranging from 100 to 2500. As expected, more epochs led to better fitting, but more computation time. Although it makes sense to maximize accuracy, we found that the increased accuracy between 1000 and 2500 epochs became negligible for our use case.

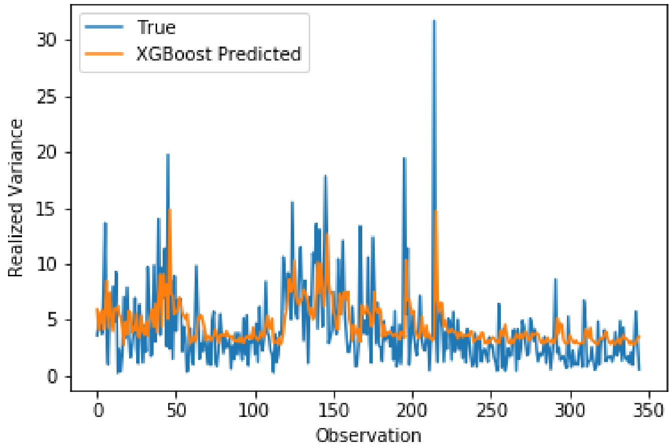
Lastly, e spent time analyzing and testing various lag lengths, ranging from just 1 period, to 50. The greater lags significantly slowed down the model, and smoothed many of the predictions. We found a strong relationship between lags and dropout. The lower dropout rates made the model more sensitive, as did lower lag rates. Our best run with the RNN by the end of the project produced a sample MSE of just under the orignial, at 14.78 (See Figures section). Although the RNN was much less effective than both of the other models, now appreciate the complexity and flexibility associated with this model. It was also necessary to not re-fit the model for every observation, the other models have the computational simplicity to allow that, while each RNN takes a long time to fit.

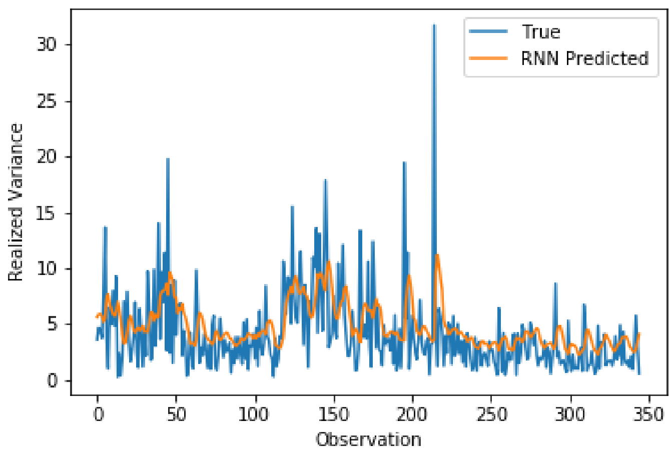
One of the more effective runs, shown below, included 5 lags, 500 epochs, 3 LSTM layers, 10 units per layer and a dropout rate of 70% at each layer. In general, The more simple the model was and the less sensitive it was to the previous observations, the better it did. The RNN without very high dropout rates would tend to over-predict after spikes. Were we to continue the project trying to understand the RNN better would be a priority. It's likely better at predicting time-series with more complex behavior. The volatility series has a lot of noise and the RNN seems to get “tricked" in a sense by the noise, while the ARBoost and GARCH models are less sensitive. With a lot of dropout regularization, the RNN started to look and perform more like the GARCH which you can tell by looking at the figures below.

3.2 Figures

The following plots demonstrate the accuracy of the predictions for each of the approaches after altering parameters and model implementation.







Conclusions & Broader Impact

Being able to anticipate volatility, and by extension manage risk is an essential function for financial institutions. The failure of risk management functions in banks has led to catastrophe in the past. Improving our understanding of financial risk, though most directly beneficial to financial institutions, indirectly benefits all of us.

Note on Additional Results

Apologies if it seems like there's not much in terms of additional results from the milestone report. We really had most of what we wanted to do done by the milestone. All that was left was to tune the hyperparameters. We managed to significantly improve the performance of the boosted AR model, while improving the RNN remained elusive.

Contributions

Kyle did the coding for the GARCH model and some of the RNN. Jack did some of the RNN and the boosted AR model. Both contributed to the final.

References

[1] Haykin, S. 2009. Neural Networks and Learning Machines. New Jersey: Prentice-Hall, Inc.

[2] Li, C. A Gentle Introduction to Gradient Boosting, Northeastern University.

[3] Hochreiter, S. and Schmidhuber, J. 1997. Long Short-term Memory, Neural Computation.

[4] Vania Orva Nur Laily et al 2018 J. Phys.: Conf. Ser. 1025 012103.

[5] Schmidhuber, J. and Cummins, F. 2000. Leanring to Forget: Continual Prediction with LSTM, Neural Computation.