Optiver Realized Volatility Prediction

COMP9417

Major Work

T2 2022

Members:

Kevin Fine (z5308356), Hugo Giles (z5309502)

01/08/22

Kevin Fine	z5308356	01/08/2022
Hugo Giles	z5309502	01/08/2022

1. Introduction

Market-makers are the backbone of financial markets. Their role is to provide liquidity for a wide variety of financial instruments, on several exchanges - so that market participants can constantly access the market at the fairest price possible. Since market-makers take on a lot of risk, it is important for them to be able to model both accurately and quickly the fair value of any financial instrument. Within options market-making, one of the key drivers of fair value, is the volatility of the underlying product. In general, high volatility is associated with periods of market turbulence and large price swings, while low volatility describes more calm and quiet markets. If one can build a model that can accurately predict the volatility of an underlying, market-making firms can better generate fairer options prices for market participants. Within this paper, we harness the dataset provided in the *"Optiver Realized Volatility Prediction"* Kaggle competition to explore how machine learning can be effectively used to predict volatility

I. Data

The dataset provided in the competition contained real stock market data relevant to the practical execution of trades in financial markets. In particular, it included order book snapshots and executed trade data.

Order book features included:

- stock_id - ID code for the stock. Not all stock IDs exist in every time bucket. Parquet coerces this column to the categorical data type when loaded; you may wish to convert it to int8.
- time_id - ID code for the time bucket. Time IDs are not necessarily sequential but are consistent across all stocks.
- seconds_in_bucket - Number of seconds from the start of the bucket, always starting from 0.
- bid_price[1/2] - Normalized prices of the most/second most competitive buy level.
- ask_price[1/2] - Normalized prices of the most/second most competitive sell level.
- bid_size[1/2] - The number of shares on the most/second most competitive buy level.
- ask_size[1/2] - The number of shares on the most/second most competitive sell level.

Trade features included:

- stock_id - Same as above.
- time_id - Same as above.
- seconds_in_bucket - Same as above. Note that since trade and book data are taken from the same time window and trade data is more sparse in general, this field is not necessarily starting from 0.
- price - The average price of executed transactions happening in one second. Prices have been normalized and the average has been weighted by the number of shares traded in each transaction.
- size - The sum number of shares traded.
- order_count - The number of unique trade orders taking place.

In the competition, competitors trained their data on the whole dataset and were then evaluated on a test training set run by Optiver, this was to involve any malpractice (i.e. training their data on the test dataset). However, in order to simulate this for ourselves we split our dataset into an 80-20% split with the 20% being an 'unseen' dataset which we couldn't model our data on. We split the data based on time_id with all time_id's below 25683 as training and all time_id's above as test. We did not need to worry about a randomised shuffle for train and test, as Optiver has provided the data with the time_id's not ordered and with no correlation between each other.

II. Target

The goal of the machine learning algorithm within this paper is to accurately predict the realized volatility for the **next** ten-minute bucket (denoted as the 'target volatility') which proceeded previous 'ten-minute' buckets (time_ids). Realized volatility, **σ**, is defined as the square root of sum of squared log returns.

$$\boldsymbol{\sigma} = \sqrt{\sum_t \mathrm{r}^2{}_{t-1,t}}$$

Kaggle submissions were evaluated on minimising the root mean square percentage error:

$$\boldsymbol{RMPSE} = \sqrt{\frac{1}{n}\sum_t ((y_i - \hat{y}_i)/y_i)^2}$$

2. Exploratory Analysis and Feature Engineering

In this section of the paper, we discuss how exploratory data analysis and feature engineering contributed to our final set of models. Analysis was conducted in Jupyter whilst all implementation was produced using Python 3.10.

I. Correlation of Realized Volatility

Since the dataset provided trade and order book data on a list of 127 different stocks, we thought the simplest approach was to first was compute the volatility of all stocks. We computed this across all time buckets (where each time_id represented the first ten minutes of volatility at a point in time) to see how much it correlated with (on average), the target volatility (of the next ten minutes). We thought that this could act as our benchmark model.

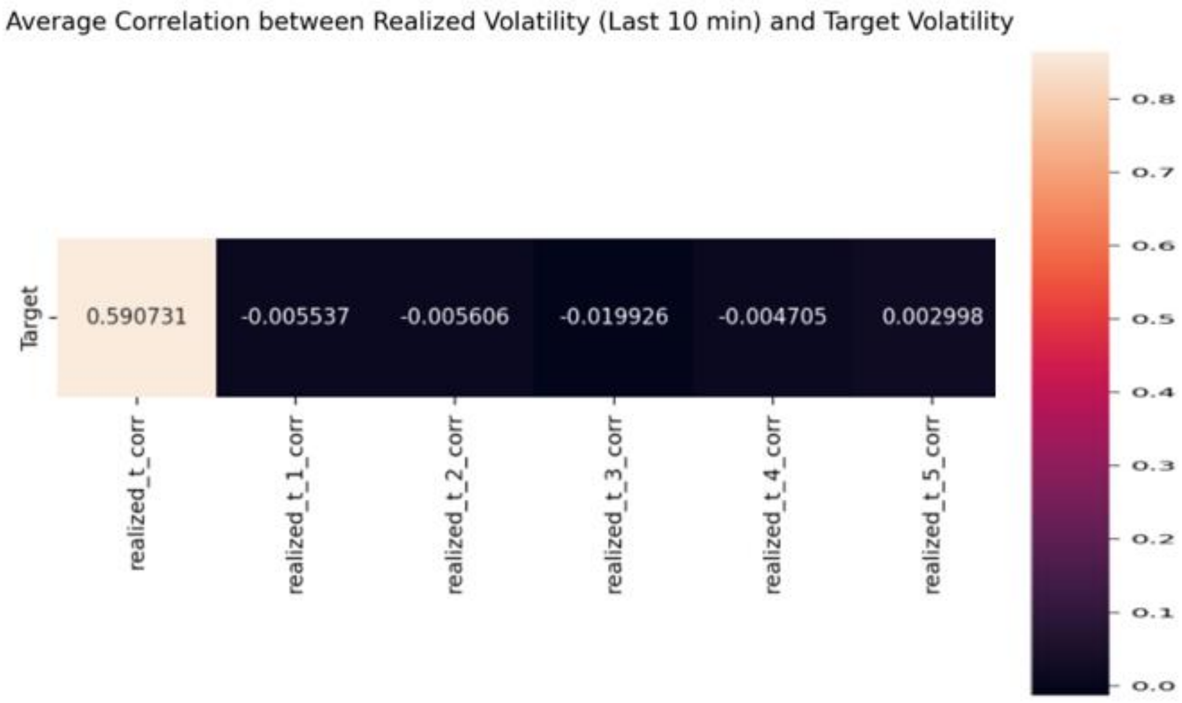


Fig 2.1

In Fig **2.1,** we can see that the realized volatility of the first ten minutes at time t has a moderately positive correlation (0.59) with the volatility of the target (next 10 minutes at time t). However, we can clearly see for other time_ids (t_1, t_2, t_3, t_4, t_5), there is **zero** relationship between the realized volatility at time t and the target volatility at time t_k which confirms that all time_id's are independent.

Through some further exploration, we found that the maximum average correlation between our realized volatility and target volatility occurs when we subset the latter half of the realized volatility calculation. That is, we take the correlation between the last 5 minutes of realized volatility, with the next ten minutes of target volatility. We can see a large improvement in the correlation below (Fig **2.2**) , indicating a strong positive relationship.

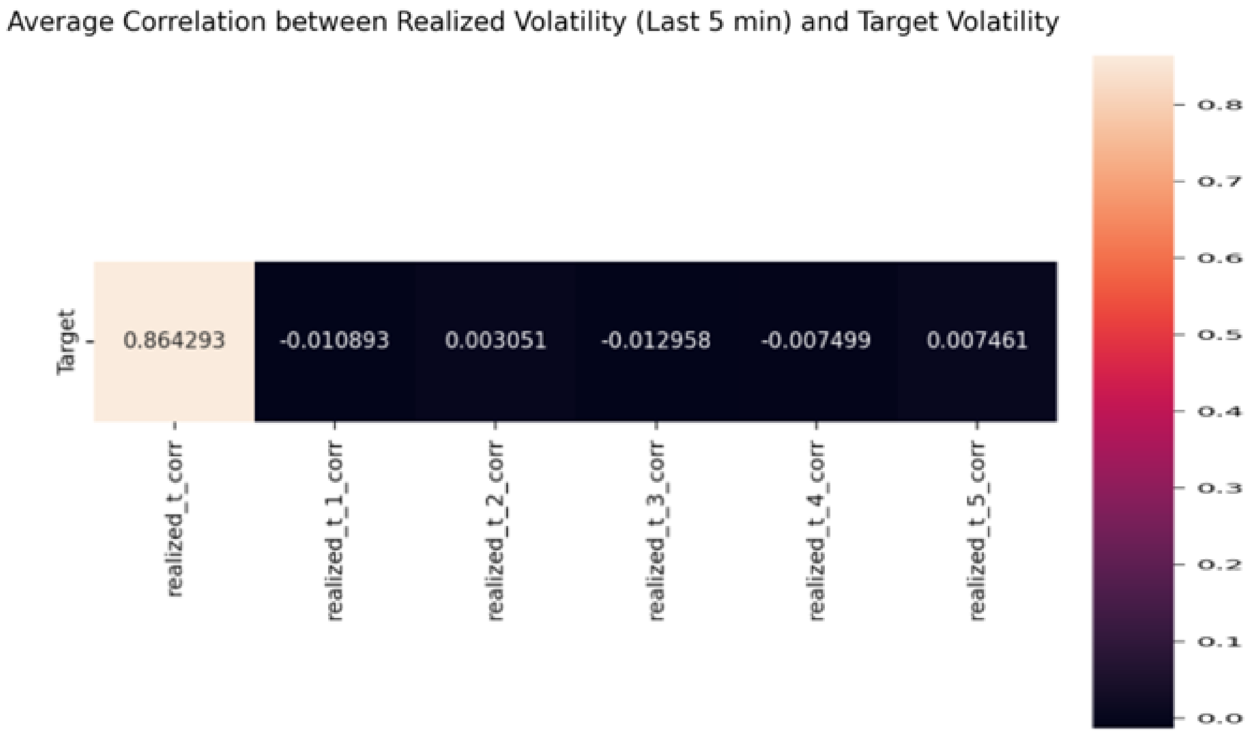


Fig 2.2

Sticking with this simple model, we ran a univariate linear regression with the last 5 minutes of realized volatility as our predictor, and the next 10 minutes of volatility as the response. Interestingly, we obtained an R^2 value of 0.75 which indicated that the simple model has a decent fit, but as expected – the RMSPE value of 0.33 indicated that the model has lots of room to improve (see Appendix 1).

II. Feature Engineering

Our feature engineering process is depicted in the table on the next page. In order to get a model with better fit, we decided to engineer more relevant predictors than just the realized volatility of the last 5 minutes.

Book Data

Feature Name	Computation	Description
Weighted Average Price (WAP)	$\frac{bid\_price * \ ask\_size + ask\_price * \ bid\_size}{ask\_size + bid\_size}$	For each seconds_in_bucket within a time_id, we compute the current volume weighted average price. We do this at the most competitive bid/ask and the second-most competitive bid/ask. We also compute the average of the two. We aggregate the time_id by the mean.
Weighted Average Price (WAP) Spread	$WAP_1 - WAP_2$	For each seconds_in_bucket within a time_id, we compute the current volume weighted average spread. We do this at the most competitive bid/ask and the second-most competitive bid/ask. We aggregate the time_id by the mean.
Price Spread	$ask\_price - \ bid\_price$	For each seconds_in_bucket within a time_id, we compute the current price spread. If a stock is more volatile and market-makers cannot determine a fair value with great confidence, the price spread will be higher. If a stock's volatility is **higher**, we expect the price spread to be **bigger**. We aggregate the time_id by the mean.
Total Volume	$ask\_size + \ bid\_size$	For each seconds_in_bucket within a time_id, we compute the total volume. We aggregate the time_id by the mean.

Trade Data

Feature Name	Computation	Description
Trade Volume	$current\_trade\_vol$	For each respective ten-minute bucket (time_id), we compute the current volume (sum of all shares traded). We aggregate the time_id by the sum.
Relative Trade Volume	$\frac{current\_trade\_vol}{avg\_trade\_vol}$	For each respective ten-minute bucket (time_id), we compute the current volume (sum of all shares traded) and divide it by the average trade volume over all time buckets. The **bigger** the relative trade volume, the **higher** the volatility. We aggregate the time_id by the sum. A Relative Trade Volume of **3.0x** indicates a stock is currently trading 3x as much volume compared to its average ten-minute interval.
Percent Range	$current\_percent\_range$ $\text{current\_percent\_range} = \frac{(\text{max\_price} - \text{min\_price})}{\text{median\_price}}$	For each respective ten-minute bucket (time_id), we compute the current percentage range the stock is trading in (relative to its current median for that time bucket). The **bigger** the percent range, the **higher the volatility.** We aggregate the time_id by the sum.
Size per Order	$\frac{current\_trade\_vol}{current\_order\_count}$	For each respective ten-minute bucket (time_id), we compute the current volume (sum of all shares traded) and divide it by the current order count. We aggregate the time_id by the sum.

Relative Percent Range	$$\frac{current\_percent\_range}{avg\_percent\_range}$$ $$\text{current\_percent\_range} = \frac{(\text{max\_price} - \text{min\_price})}{\text{median\_price}}$$ $$\text{avg\_percent\_range} = \frac{1}{n}\sum_t \frac{(\text{max\_price} - \text{min\_price})}{\text{median\_price}}$$	For each respective ten-minute bucket (time_id), we compute the current percentage range the stock is trading in (relative to its current median for that time bucket) and divide it by the average percent range for all time buckets. The **bigger** the relative percent range, the **higher the volatility.** A Relative Percent Range of **3.0x** indicates a stock is currently trading 3x its percent range compared to its average ten-minute interval.
Relative Trade Volume	$$\frac{current\_trade\_vol}{avg\_trade\_vol}$$	For each respective ten-minute bucket (time_id), we compute the current volume (sum of all shares traded) and divide it by the average trade volume over all time buckets. The **bigger** the relative trade volume, the **higher** the volatility. A Relative Trade Volume of **3.0x** indicates a stock is currently trading 3x as much volume compared to its average ten-minute interval.

3. Model Selection

In terms of model selection, we mainly considered Gradient Boosting Decision Trees (GBDT). In deciding a model to use, we aimed to choose one that performed efficiently and accurately (in predicting volatility). Overall, Gradient Boosting Decision Trees was highly efficient in feature selection – especially during trial phases when we didn't know what features to select. Additionally, there were several parameters we could specify to change the accuracy and efficiency, depending on what was required.

I. Gradient Boosting Decision Tree's (GBDT)

In conventional Gradient Boosting Decision Tree's (GBDT), there is a big trade-off between accuracy and efficiency. This is because implementations of GBDT need to, for every feature, scan all the data instances to estimate the information gain of all the possible split points. Therefore, their computational complexities are proportional to both the number of features and the number of instances. This makes traditional GDBT implementations very time consuming when handling big data.

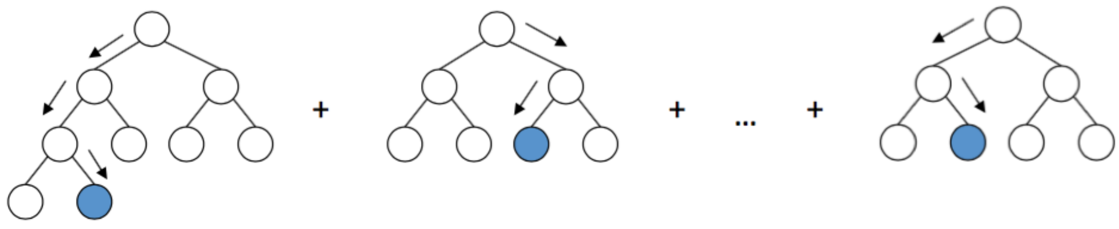


Figure 1: Basic Gradient Boosting Tree Process

In older implementations of GBDT, a simple algorithm to find split points required pre-sorted feature values and enumerations across all possible split points. One could only imagine how computationally intensive this algorithm is. The LightGBM model is a more modern implementation of GBDT and harnesses a histogram-based algorithm which buckets features into discrete bins and uses these bins to construct feature histograms during training. The algorithm is described on the next page.

We can follow the algorithm outlined in the original LightGBM paper [1],

Algorithm 1 Histogram-based Algorithm

```
1:  Input: I: training data, d: max depth
2:  Input: m: feature dimension
3:  nodeSet ←{0} ≫ Tree nodes in current level
4:  rowSet ← {{0, 1, 2, ...}} ≫ Data indices in tree nodes
7:  for i = 1 to d do
8:      for node in nodeSet do
9:          usedRows ← rowSet[node]
10:         for k = 1 to m do
11:             H ← new Histogram()≫ Build Histogram
12:         for j in usedRows do
13:             bin ← I.f[k][j].bin
14:             H[bin].y ← H[bin].y + I.y[j]
15:             H[bin].n ← H[bin].n + 1
16:             Find the best split on Histogram H...
17:
16:     Update rowSet and nodeSet according to the best split points...
```

II. Complexity of GBDT

If we then begin to examine the complexity of the histogram-based algorithm it starts to become clear how traditional GBDT can become so computationally intense. For histogram building – we can see that the complexity is *O(#instances x #features)* whilst for split point finding it is *O(#bins x #features)*. Since #bins is usually much smaller than #instances, histogram building will dominate the computations. Therefore, if one wanted to speed up the GBDT algorithm, they would have to reduce *#instances* or *#features* substantially.

III. LightGBM

LightGBM is a gradient boosting framework that is an extension of traditional GBDT. LightGBM unpacks the computational complexities addressed in the GBDT histogram implementation and aims to solve the complexity issue using two novel techniques: Gradient-based One Side Sampling (GOSS) and Exclusive Feature Bundling (EFB).

Gradient-based One Side Sampling

In GBDT, we notice that data instances with different gradients play different roles in the computation of information gain. According to the definition of information gain, instances with larger gradients (magnitudes) will contribute more to the information gain calculation. Therefore, to eliminate computational complexities, GOSS focuses on keeping instances with large gradients, and randomly dropping instances with small gradients (down sampling) – which is proven to lead to a more accurate gain estimation than uniform sampling. This reduces the #instances component as illustrated within the complexity analysis **(II)**.

We can follow the algorithm outlined in the original LightGBM paper [1],

Algorithm 2 Gradient-based One Side Sampling

```
1:  Input: I: training data, d: iterations
2:  Input: a: sampling ratio of large gradient data
3:  Input: b: sampling ratio of small gradient data
4:  Input: loss: loss function. L: weak learner
5:  models ← {}, fact ← (1-a)/b,
6:  topN ← a x len(I),  randN ← b x len(I) ,
7:  for i = 1 to d do
8:        preds ← models.predict(I)
9:        g ← loss(I, preds), w ←{1, 1,...}
10:       sorted ← GetSortedIndices(abs(g))
11:       topSet ← sorted[1:topN]
12:       randSet ← RandomPick(sorted[topN:len(I)], randN)
13:       usedSet ← topSet + randSet
14:       w[randSet] x = fact ≫ Assign weight fact to the small gradient data
15:       newModel ← L(I[usedSet], -g[usedSet], w[usedSet])
16:       models.append(newModel)
```

Exclusive Feature Bundling (EFB)

In traditional machine learning problems, we usually find that we are dealing with gigantic datasets. Having gigantic datasets can lead to large feature spaces which results in intense computations. However, in practise a lot of the feature space is exclusive in the sense that many features take non-zero values simultaneously. The aim of EFB is to design an efficient algorithm that bundles these mutually exclusive features together which is a graph colouring problem in disguise. In the algorithm, we take features as vertices and add edges for every two features if they are not mutually exclusive. This severely reduces the #features component as illustrated within the complexity analysis.

We can follow the algorithm outlined in the original LightGBM paper [1],

Algorithm 3 Greedy Bundling

```
1:  Input: F: features, K: max conflict count
2:  Construct Graph G
3:  searchOrder ←G.sortByDegree()
4:  bundles ←{}, bundlesConflict ←{}
7:  for i in searchOrder do
8:        needNew ← True
9:        for j = 1 to len(bundles) do
10:             cnt ← ConflictCnt(bundles[j], F[i])
11:       if cnt + bundlesConflict[i] ≤ K = 1 then
12:             bundles[j].add(F[i]), needNew ← False
13:             break
14:       if needNew then
15:             Add F[i] as a new bundle to bundles
16:  Output: bundles
```

After bundling, a similar version to the histogram-based algorithm is run, but we create new bins and bin ranges depending on bundles that were created in algorithm 3. We are effectively merging our exclusive features.

We can follow the algorithm outlined in the original LightGBM paper [1],

Algorithm 4 Merge Exclusive Features

```
1:  Input: numData: number of data
2:  Input: F: One bundle of exclusive features
3:  binRanges ←{0}, totalBin ←{0}
4:  for f in F do
8:        totalBin += f.numBin
9:        binRanges.append(totalBin)
10: newBin ← new Bin(numData)
11: for i=1 to numData do
12:       newBin[i] ←0
13:       for j=1 to len(F) do
14:            if F[j].bin[i] ≠0 then
15:                 newBin[i] ← F[j].bin[i] + binRanges[j]
16: Output: newBin, binRanges
```

In summation, we call this altered GBDT algorithm with Gradient-based One Side Sampling **(GOSS)** and Exclusive Feature Bundling **(EFB)**, LightGBM. Experiments on multiple public datasets within the original paper show that LightGBM can accelerate the training process by up to over 20 times while achieving almost the same accuracy as traditional GBDT algorithms.

4. Results

As mentioned earlier, our benchmark model was a simple linear model with realised volatility as the only predictor. We took this simplicity and extended it to our LightGBM framework with only our realised volatility predictors from both the book and trade datasets. This resulted in an RMSPE value of 0.306778. We then adjusted our model to utilise our created features, and we noticed a decrease in our final RMSPE value to 0.297537. Following this, we further adjusted our model to include the stock id as a categorical variable - this resulted in a large decrease in our RMSPE value to 0.240839. Unsurprisingly, we found that stock_id became the most important feature out of our whole feature space which is illustrated in this diagram (Appendix 2). We can compare this to our old model without the stock_id as a feature which is illustrated in this diagram (Appendix 3).

Lastly, we also tried our final dataset with the whole ten-minute period which we found improved the RMSPE value slightly to 0.239743. Although the last 5 minutes of data generally had strong correlations with the target, we believe using the last 5 minutes of data increased the standard error of predictions through overfitting. This is the reason why we received lower RMSPE's when we considered the whole ten-minute period.

LGBM Parameters

We used a 'feature_fraction' of 0.8 which helps improve training by splitting the columns of the training set again in an 80-20% ratio. Further a 'bagging_fraction' of 0.8 has a similar affect but with rows. These help to reduce the risk of overfitting our training dataset.

We also adapted our model to experiment with different parameter values for 'boosting_type' and 'learning_rate'.

		Gradient Boosting Method					
		GBDT	Time Taken	**GOSS**	Time Taken	**DART**	Time Taken
Learning rate	0.05	0.240761	23.2s	0.238788	27.1s	0.240004	>60 mins
	0.01	0.239743	51.4s	0.238438	1m 1.2s	*	>60 mins
	0.001	0.239881	5m 8.2s	**0.238282**	**6m 52.6s**	*	>60 mins

* *we stopped the model at the 2 hour mark.*

I. GBDT vs DART vs GOSS

Within LightGBM, we can run different types of gradient boosting methods which is specified by different parameters: GBDT, DART and GOSS. As depicted in the table above, different gradient boosting methods produce different results for the RMSPE and take different amounts of time to run.

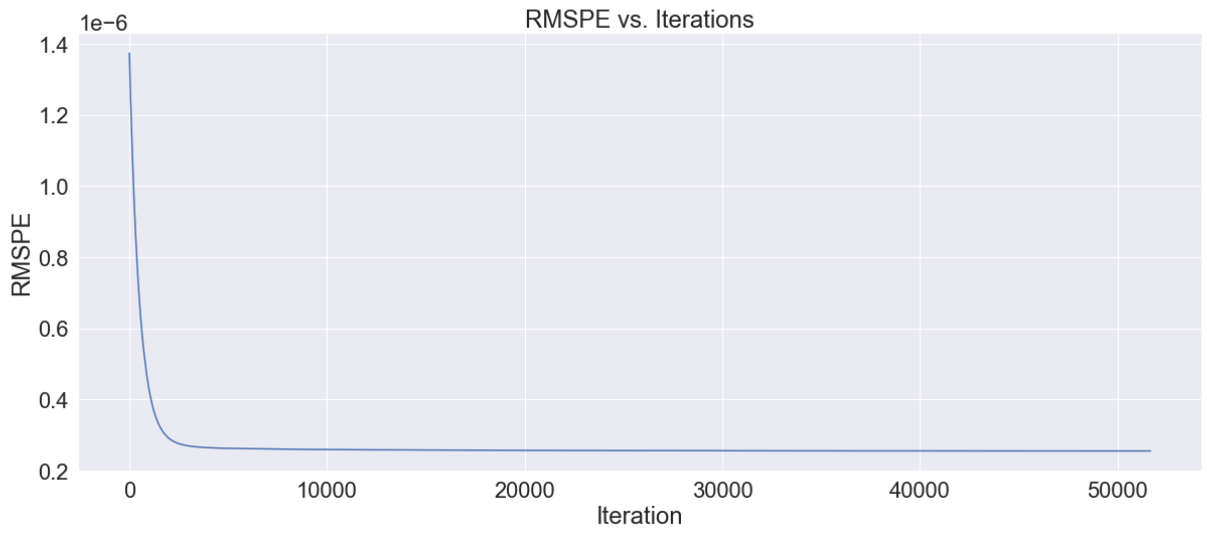
The GBDT parameter refers to the original gradient boosting methodology that was discussed earlier in the paper (3I). Within this method, we sequentially build trees that learn how to fit the residuals of previous trees. More specifically, our first tree learns how to fit the target variable. Our second tree learns how to fit the residuals between the fitted and true values of the first tree. And our third tree learns how to fit the residuals of the second tree, with the pattern extended to the number of trees that one specifies. As previously discussed, GBDT is a computationally intensive method as the algorithm finds the best split points in each tree node. In general, GBDT is regarded as reliable but inefficient to use on large datasets.

The GOSS parameter refers to the gradient boosting techniques employed by LightGBM also discussed earlier in the paper (3III). Recall that GOSS is a newer implementation of GBDT that is based off a sampling method to reduce computations. More specifically, for data instances where gradients are small this generally indicates that the data is well-trained. However, for gradients that are larger, this indicates that the data should be retrained. Hence this is where the name Gradient-Based One Side Sampling comes from as we keep all data with large gradients and randomly sample data with smaller gradients. This makes computations much smaller and more manageable, which is why GOSS converges much faster than the other parameters.

The DART parameter refers to a different prominent technique where we drop trees at random to prevent over-specialisation of base learners. This helps to improve performance by making it more common for dropouts and thus harder for later iterations to specialise on specific samples. This can help to remove some occasional bias on later iterations and thus improves performance. In our results we found Dart to be extremely long to run, in an ideal situation we could leave the model running for longer.

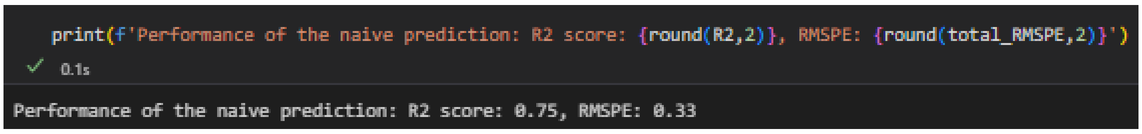
II. Final Model Choice (GOSS)

After considering the three different parameters, we decided that GOSS was the best gradient boosting method for us to use. GBDT was a good start for us, but it wasn't as effective. On the other hand, DART was too computationally intensive to produce effective results in a reasonable time. Therefore, we chose GOSS, as it was not only efficient – but highly accurate. We achieved a RMSPE of 0.238282 and the time taken for the algorithm to run was only 6 min 52.6 seconds, which was parameterized by a learning rate of 0.001. Below is a plot of the convergence of the GOSS algorithm, which is seen to converge very quickly.



Appendices

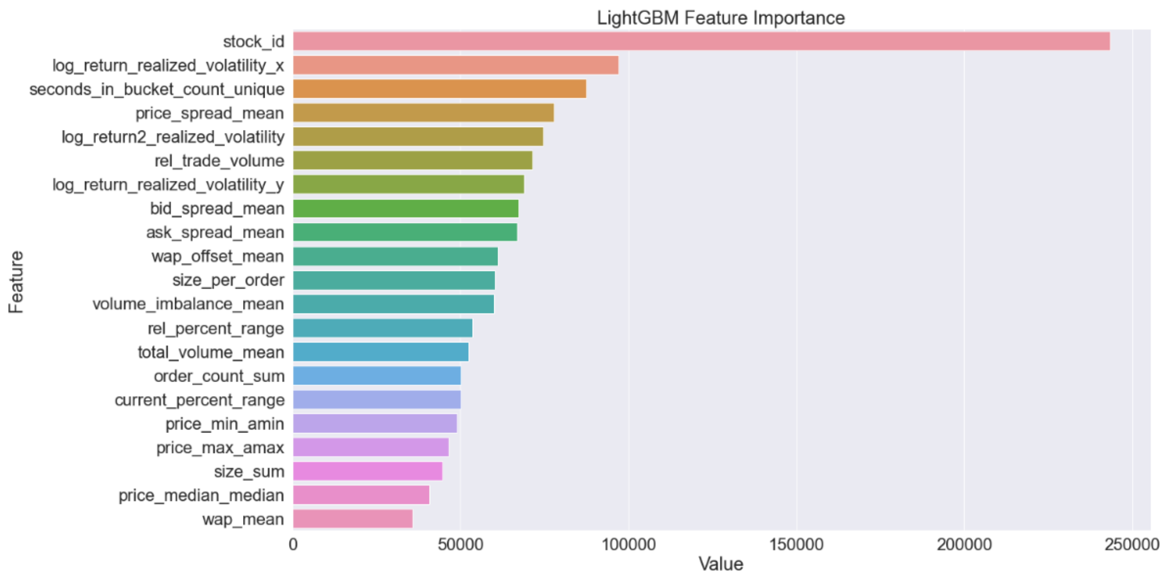
Appendix 1:



```
print(f'Performance of the naive prediction: R2 score: {round(R2,2)}, RMSPE: {round(total_RMSPE,2)}')
✓ 0.1s
Performance of the naive prediction: R2 score: 0.75, RMSPE: 0.33
```

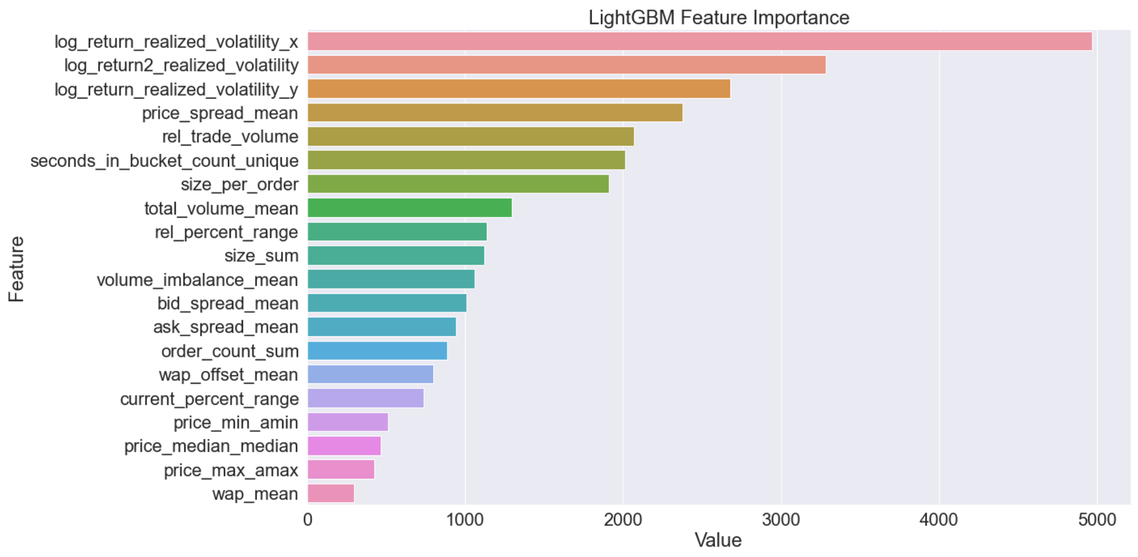
Performance of Simple Linear Model with one predictor - Realized Volatility (Last 5 min)

Appendix 2:



Graph of Feature Importance (including stock_id)

Appendix 3:



Graph of Feature Importance (excluding stock_id)

References

[1] Ke, G, Meng, Q, Finley, T, Wang, T, Chen, W, Ma, W, Ye, Q & Liu, T-Y n.d., *LightGBM: A Highly Efficient Gradient Boosting Decision Tree*, viewed 20 July 2022, <https://proceedings.neurips.cc/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf>.

[2] Brownlee, J 2020, *How to Develop a Light Gradient Boosted Machine (LightGBM) Ensemble*, Machine Learning Mastery, viewed 19 July 2022, <https://machinelearningmastery.com/light-gradient-boosted-machine-lightgbm-ensemble/>.

[3] Carr, P, Wu, L & Zhang, Z 2019, 'Using Machine Learning to Predict Realized Variance', *arXiv:1909.10035 [q-fin]*, viewed 16 July 2022, <https://arxiv.org/abs/1909.10035>.

[4] Bahmani, M 2020, *Understanding LightGBM Parameters (and How to Tune Them)*, neptune.ai, viewed 30 July 2022, <https://neptune.ai/blog/lightgbm-parameters-guide>.

[5] *LightGBM (Light Gradient Boosting Machine)* 2020, GeeksforGeeks, viewed 25 July 2022, <https://www.geeksforgeeks.org/lightgbm-light-gradient-boosting-machine/>.

[6] LIU, J n.d., *Introduction to financial concepts and data*, kaggle.com, viewed 11 July 2022, <https://www.kaggle.com/code/jiashenliu/introduction-to-financial-concepts-and-data?scriptVersionId=67183666#Competition-data>.

[7] Pranjal 2016, *Which algorithm takes the crown: Light GBM vs XGBOOST?*, Analytics Vidhya, viewed 22 July 2022, <https://www.analyticsvidhya.com/blog/2017/06/which-algorithm-takes-the-crown-light-gbm-vs-xgboost/>.

[8]Rashmi, K & Gilad-Bachrach, R 2015, *DART: Dropouts meet Multiple Additive Regression Trees*, 7 May, viewed 21 July 2022, <http://proceedings.mlr.press/v38/korlakaivinayak15.pdf>.