Fractional Brownian Motion

June 24, 2019

陳君彥, b04703091 b04703091@ntu.edu.tw
劉育嘉, b06902008 b06902008@ntu.edu.tw
黃柏豪, b06902124 b06902124@ntu.edu.tw

Division of Work

陳君彥, b04703091:

- Data initial processing and calculation tool creation.
- Initial creation and testing of XG-boost, LightGBM, and Random Forest Models

劉育嘉, b06902008:

- Indepth testing of tree based methods
- Testing and Creation of select feature models.

黃柏豪, b06902124:

- Creation and testing of neural network based models.
- Creation and testing of blending based models.

1 Introduction

The goal of this project is to reverse learn model parameters used to simulate a fractional Brownian Motion [1] simulation. The parameters used to run this simulation were alpha, mesh size, and penetration rate. We used 2 methods of evaluation, one with "Weighted Mean Absolute Error" (WMAE), and one with "Normalized Absolute Error" (NAE)

$$WMAE(Y,\hat{Y}) = \frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} \sum_{j=1}^{3} w_j \mid y_{ij} - \hat{y}_{ij} \mid, NAE(Y,\hat{Y}) = \frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} \sum_{j=1}^{3} \frac{\mid y_{ij} - \hat{y}_{ij} \mid}{y_{ij}}$$

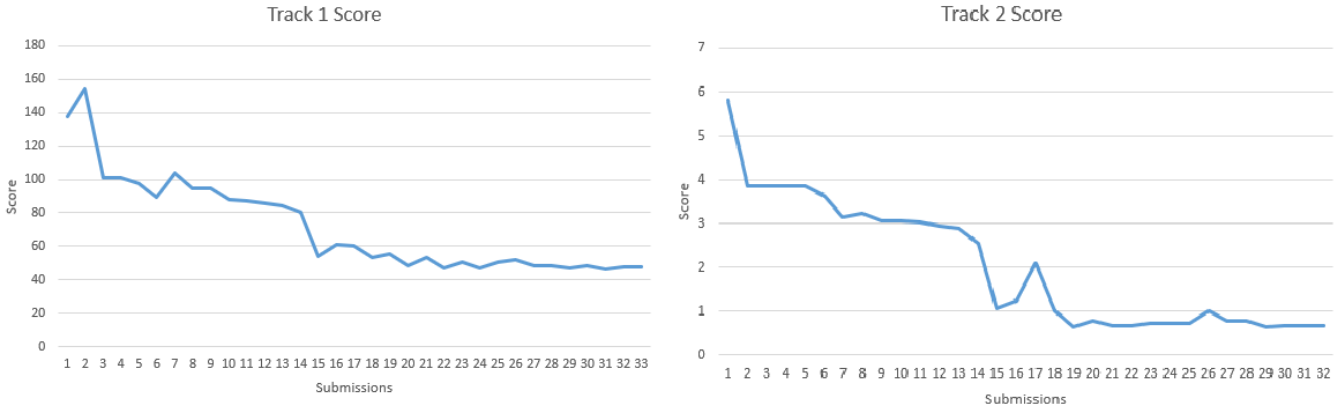

Figure 1: Submission scores of our models.

2 Features

2.1 Original Features

Our training dataset consists of 47,500 simulations with 10,000 features each. The first 5,000 features represented the mean-square displacements (MSD) of our particles ordered from time = [1,5000]. The second 5,000 features were 50 sets of velocity auto-correlations (VAC) calculated in different methods, ordered by the early VAC being more representative of instataneous velocity, with later VAC being closer to average velocity. Each sets consists of 100 calculation using the respective VAC, with time intervals from t = [1,100].

Our testing sets consists of 2,500 simulations, half of which is tested in the public score, and half hidden in the private score.

2.1.1 Feature Importance

In the interest of reducing the time need to train our models, we took a look at the feature importance determined by our earlier models and selected those that had an importance higher than a given threshold. Through several rounds of testing, we found 10^{-4} to perform the best. This allowed us to greatly increase the training iteration count and deepen our original model, while giving us increased accuracy with much lower training times.

Something of particular to note is the high reliance of a select few features for each parameter. Due to the lack of professional knowledge in this field and the lack of raw data, we could do little to take advantage of this except to reduce the dimensionality. However we expect that further studies can be improve by generating more features that are centered around these specific features.

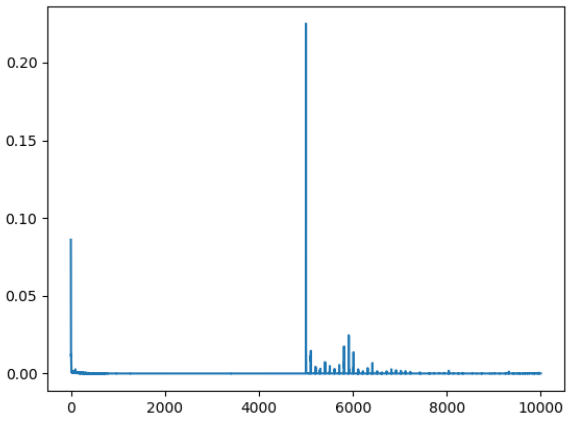

Figure 2: Feature importance of penetration rate.

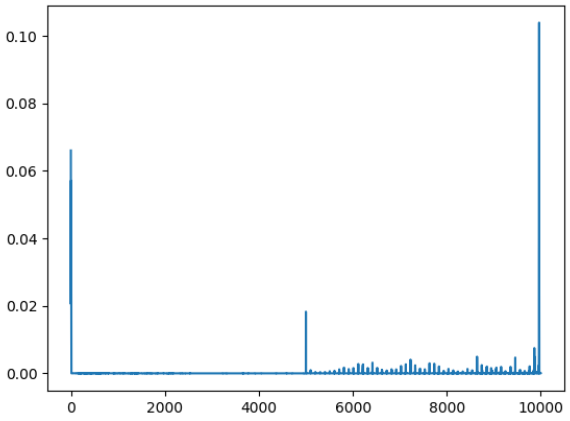

Figure 3: Feature importance of mesh size.

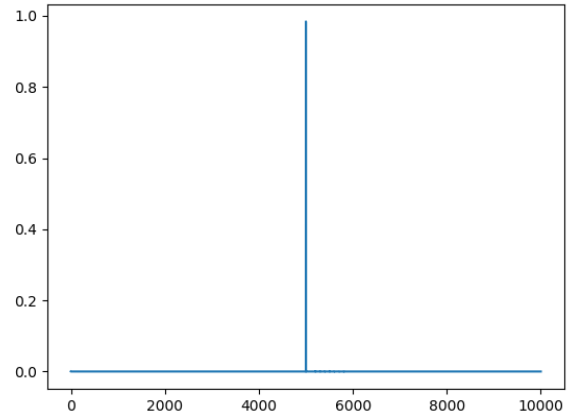

Figure 4: Feature importance of alpha.

2.2 Additional Features

To gather more physics based features, we also extrapolated some of our own data.

2.2.1 Average and MinMax

We were interested in having our model be less overfitting and took feature based methods to reduce the feature count. We tried taking the average of every 10 features, which would result in the

mean displacement or velocity over a larger time period. We also tried taking the minimum and maximum of each 10 features, which in essence was a form of data pooling.

2.2.2 Square Error on Linear Regression

We assume a particle without external force will go straight forward with constant speed, so the force generated by collision may have something to do with the difference between a linear line and its real value. Therefore, we performed linear regression and computed the variance of those 5000 time intervals, substituting them for the original 5000 features.

2.2.3 Alpha feedback

Our models performed exceptionally well on predicting alpha, and so we decided to feedback the results of our third parameter prediction back in as a feature for our other two predictions, giving a slight boost to performance.

3 Individual Model Experiments

3.1 Tree based models

3.1.1 Random Forest

We used the models provided in the python package Sci-kit Learn [2] for our random forest, with bootstrap enable, estimator count of 50, and a max depth of 9. The following shows the results of our experiments.

Tree count	Penetration	Mesh Size	Alpha
50	65.72	31.92	1.96
100	64.74	30.69	1.81
200	64.06	29.93	1.74
300	61.02	27.20	1.54
500	58.67	25.10	1.40
1250	55.07	22.40	1.23

Table 1: Performance on track 1 for random forest (Validation)

As seen in Table 1, we achieve moderate performance by simply throwing in the original features into a random forest model. The accuracy gradually increased with each additional tree count, though the time required to train them started to get out of hand. Each tree required around 1.5 minute to train when parallelized, meaning training each model for all three features required over 3 hours for each 50 step increment. Hence we tried it out with a reduced feature set selected using the previously trained model's feature importances.

Tree count	Alpha	Mesh Size	Penetration
50	67.31	30.69	3.13
100	67.42	30.60	3.12
200	67.35	30.65	3.11
300	67.35	30.64	3.11

Table 2: Performance on track 1 for random forest with reduced input. (Validation)

However, as seen in Table 2, the reduced feature set performed poorly, though it did require very little computational power. Yet it failed to beat even the base RF model with 300 trees. We gave up and tried a different approach, reducing the dataset by picking averages of every 10 features.

Tree count	Alpha	Mesh Size	Penetration
50	64.57	23.98	2.37
100	64.56	23.94	2.36
200	64.56	23.86	2.36
300	64.60	23.85	2.35

Table 3: Performance on track 1 for random forest with averaged features. (Validation)

The results as shown in Table 3 shows that this performed better than the feature selection model, and slighly better than the original model, with a combined score of 90.81 on track 1 and 2.29 on track 2. However, we notice that stop in decrese of the validation error, and recognized that further training would do the model no good.

3.1.2 XGBoost

We tried the very popular XGBoost [3] often used in machine learning competitions. XGBoost is a gradient boosting tree model that has an emphasize on reduced tree depth and higher parallelism. This allows us to train a model much faster and allows easier batching on the massive dataset.

Notes	Track 1	Track 2
base	137.23	6.22
+ filtered data & alpha feedback	48.86	0.66
+ auto tree method	43.49	0.60
+ tweedie regularizer	41.70	0.38
+ boost round = 5	40.85	0.33

Table 4: Performance of xgboost with each additional parameter tuning (Validation).

As seen in table 4, xgboost relies heavily on custom tuned parameter, with the untuned model performing much worse than others. After some tuning and using filtered data, our model provided a very respectable result on the validation set. After submitting the prediction, we found out that the final rounds of tuning had led to the model being slightly overfitted, with the submission score being worse than the previous round. We thus stopped our experiments on xgboost here and moved on to a different model.

3.1.3 LightGBM

LightGBM [4] is a very new contender to the scene, with a focus on limiting leaf count rather than xgboost's method of limiting tree depth.

In our instant, lightgbm performs much better with base parameters, with learning rate set to 0.01. As seen in Table 5, it is already in clear competition with xgboost. We also went through the same process of using the predicted alpha to feed back into the model, as well changing to use 'dart' as a boosting method with more iterations. It can be seen from Table 6 that the submission score is also dropping at a steady rate, but then shows signs of overfitting with the last submission, we then started performing regularization to combat the issue.

Notes	Track 1	Track 2
base	56.80	1.52
+ alpha feedback	55.87	1.20
+ 1000 iterations + dart	38.25	1.07
+ 2500 iterations	30.84	0.36

Table 5: Validation performance of lightgbm with each additional parameter tuning.

We tried using the square error on linear regression as a feature, and averaging them out, cutting our feature set down to one tenth of the size, and as seen in Table 7, shows that the overfitting issue has been somewhat allieviated.

Notes	Track 1	Track 2
base	60.76	2.94
+ alpha feedback	59.87	2.10
+ 1000 iterations + dart	48.23	0.92
+ 2500 iterations	50.48	0.86

Table 6: Submission performance of lightgbm with each additional parameter tuning.

Notes	Validation	Submission
500 iterations	45.52	49.17
1000 iterations	43.27	48.13

Table 7: Submission performance of lightgbm with regularization.

3.2 Neural network models

We used the neural network model found in keras [5] to predict our three parameters. As the number of features is too big for a neural network, the features were first passed through an autoencoder of size 5000-2048-1024 and train with 'linear' activation on a 5000-2048-512-64-32-1 model.

```
def encoder(x,
    activation_list = ['relu', 'relu'],
    layer_list = (2048, 1024), # dimension
        for each layer
    name = 'temp', # to specify model file
    use_old = False): # load existing model
        or not
```

However, our model (loss='mse', optimizer = 'adam', batch size = 64, epochs = 10), as shown in Table 8, performed poorly alone.

Notes	Track 1	Track 2
In-Sample Error	120.19	3.45
Out-of-Sample Error	121.47	3.52

Table 8: Performance of the neural network model.

We attempted to improve it by reducing the layer count, while changing the activation method to use 'relu' and arrived at the results shown in table 9. While not the best, we deemed it good enough to be used in our final method.

Notes	Track 1	Track 2
In-Sample Error	98.15	1.82
Out-of-Sample Error	100.15	1.85

Table 9: Performance of the improved neural network model.

3.3 Blending

Our last methods involved the blending of our previously successful models. The first one we tried as the mixing of xgboost into our neural network. We already showed that xgboost provides a very good performance that can be extended upon. We used the autoencoded features as our inputs with weights initialized to 0, and xgboost predictions as a second input with weights initialized to 1, and the same layers as used before. As shown in table 10, this is our best results so far, though we suspect it may be somewhat a lucky guess due to the slight overfitting that occurs.

We tried to replicate it further, using a uniform blending of our top three models, xgboost, NN + xgboost, and lightgbm. As shown in table 11, the blending has made our model much more stable, providing better performance while being less overfitted than our previous model.

Notes	Track 1	Track 2
In-Sample Error	35.56	0.33
Out-of-Sample Error	44.79	0.45
Submission	47.71	0.71

Table 10: Performance of xgboost + NN

Notes	Track 1	Track 2
Validation	39.63	0.42
Submission	46.87	0.65

Table 11: Performance of Uniform Blending Model

4 Conclusion

In this paper, we tested a multitude of methods to predict the parameters used to generate a fractional Brownian Motion simulation. While our testing used two methods of evaluation, we noticed that none of our models performed exceptionally well with either evaluation methods. Based on the final competition entry, our submissions on both track achieved the same rank, and we determine that our models were not specific to either evaluation method.

We took approachs to reduce the feature size of each entry, such as filtering features based on importance as determined by preliminary models, as well as mathematical equations to reduce the dimensionality. This reduction allowed us to train the models much further than was previously feasible. We also generated our own features, like MSE on linear regression or feeding back our predicted alpha. These features were ran through multiple different models that provided different insights.

First, our pure Random Forest Models performed moderately, and showed a clear sign of the error trending downwards with each iteration. It's downfall is the massively long time it took to train each iteration, making it unfeasible for such a large dataset.

We took a look at improved tree models, such as the gradient boosted xgboost and lightgbm. These models showed the benefit of tree based models of being quite performant, while also being much faster to train than the random forest models. This allowed us to test many more parameters and feature inputs than before, giving us a better heading to work off of.

Next, we also tried neural network based models. Alone, our NN model performed moderately. The problem with neural network being that it is rather hard to improve iteratively. One set of parameters or settings may work better than another, but a combination of them may perform even worse than either. We tried giving our neural network a jumpstart by passing in the predicted results of our xgboost model. This gave improved accuracy, and showed that there was more information that could be extracted from the xgboost. This model however, was also suspicious of being incredibly

overfitting, with the high accuracy possibly due to luck.

Our final model is the uniform blending of the three models, NN + xgboost, xgboost, and lightgbm. This stabilized our results, leading to a lower validation accuracy but a higher test accuracy. Yet blending can only strengthen the performance of the input models, and itself does not contribute much to the prediction. Thus, this method also depends on having a good, or at least moderately performing prediction model to be effective when blending. This also increases the model complexity several folds, being a combination of so many already complex models. This increases VC dimension, meaning that it is even more important that the data size is large enough to sufficiently prevent overfitting. We arrive at the conclusion that the blending of multiple successful models is one of the best way to achieve a good and stable way of creating a machine learning model, provided you have a good base, and sufficient data to work off with.

References

[1] Wikipedia contributors, "fractional Brownian Motion," 2019, [Online; accessed 22-June-2019]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Fractional_Brownian_motion&oldid=895902440

[2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[3] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 785–794. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939785

[4] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. USA: Curran Associates Inc., 2017, pp. 3149–3157. [Online]. Available: http://dl.acm.org/citation.cfm?id=3294996.3295074

[5] F. Chollet *et al.*, "Keras," https://keras.io, 2015.