Embarrassingly parallel time-series modeling with Lambda

Ryan Cox, Griffin McCauley, and Douglas Ziman
University of Virginia
rsc6k@virginia.edu, kzj5qw@virginia.edu, dez6rq@virginia.edu

Abstract

Time-series forecasting is of tremendous importance to many businesses and organizations. It can be applied to determine projected product demand levels or stock prices. While the methods for forecasting such trends are typically relatively simple (exponential smoothing, ARIMA models, random forest, etc.), forecasts often need to be executed across many different categories of products, stocks, or data units at the same time. Such a scenario motivates the use of serverless computing as a means of effectively parallelizing and scaling time-series forecasting. This research seeks to apply serverless computing to parallelize time-series forecasting for stock price prediction and build a data pipeline to support distributed ETL processes.

1 Introduction and Background

One of the principle challenges in today's economy is forecasting based on time-series data. The 21st century has seen a massive increase in the amount of data readily available for analysis across industries. From product inventory to energy demand, applications of forecasting are persistent in nearly every discipline. One of the most well known forecasting problems is the challenge of predicting stock price movements.

With stock markets dating back to the 1600s, the problem of forecasting stock price is not a new one. However, it is of great consequence. Much of today's modern economy involves publicly traded companies and many individuals partake in investing, whether from a retail or institutional perspective. With the massive influx of computational tools and information, stock price prediction has evolved from individual analyst opinions to quantitative models rooted in big data. Recent research has shown that machine learning models like random forests can be successful in predicting stock direction [1] and price [2] [3]. However, while the methods in question can yield promising results, the data in question is often wide and long, thereby slowing traditional computer systems.

Stock price prediction is far from today's only big data problem. As such, many applications and tools have arisen to allow businesses and researchers alike to execute big data models more effectively. One such tool is AWS Lambda. This serverless, event-driven computation platform requires minimal implementation overhead, yet provides elastic scaling capabilities. This is an ideal framework for parallelizing stock price prediction, tuning, and training. With thousands of stocks available for prediction, sequential, looped computational patterns will not suffice in today's world. Serverless computing offers an opportunity to apply the computational tools to stock price prediction in an efficient manner.

2 Goals

The primary goal of this project was to create a set of random forest models using embarrassingly parallelized functions through AWS Lambda. Specifically, random forest models were particularly suited to this task due to their naturally parallel training structure.

3 Hypotheses

AWS Lambda can be used as a means of paralellizing stock prediction computational model tuning and training. Further, AWS Lambda's parallelization of these stock prediction workloads can drastically improve runtimes over traditional, local computing methods.

4 Design and Implementation

A random forest creates a series of decision trees on bootstrapped samples. While there are often thousands of trees in these models, in theory, each tree could be trained on a separate core and then recombined. This approach can involve high memory overhead and caching costs, but, when building thousands of trees, it is still common to use many cores in order to speed up training.

In the case of this project, there is not a single model to construct, but 100 (one per stock). To build 100 models, each with 750 trees or more on hundreds of MB of data, running on a single core would take a prohibitively long time. In fact, even using an EC2 instance with many cores to run this type of model would be inefficient and potentially expensive depending on how reproducible the analysis needs to be. AWS Lambda could help solve this problem by potentially giving temporary access to many computing cores for a brief period of a few minutes.

4.1 Set-up

There were three main problems that needed to be resolved in order to get the project working:

1. Data storage needed to be cheap and scalable.
2. Lambda instances needed access to certain Python libraries like pandas and Sklearn.
3. Lambda instances needed to be generalizable and scalable with any outputs recorded in permanent storage.

AWS S3 was chosen for data storage due to its flexibility, cost, and pre-built connectivity to Lambda. Setting up S3 was remarkably easy and proved beneficial. Its undefined size allowed buckets to expand as data was processed and outputs were generated.

While there are a number of libraries and packages built into the Lambda environment automatically, accessing other essential libraries such as pandas and Sklearn requires the use of Lambda layers. Although AWS does curate their own repository of layers for a few prominent libraries, since each function can have at most five layers that cannot exceed 250MB in total, the best approach for this research was to manually create custom layers. This allows for the greatest control and flexibility regarding which libraries (and dependencies) are included. To achieve this, one must first download the built distribution wheels for the desired libraries from a repository such as the Python Package Index (PyPI) and unpack the wheels locally. Then, the contents of the folders from each wheel should all be added to a single folder named "python." This folder can then be zipped and uploaded as a custom Lambda layer on AWS. Now, any Lambda function with this layer will have access to the libraries built into that layer.

A small amount of data processing was needed initially before running any models in order to generate a few columns, remove others, and separate the files into test/train/validate splits. To achieve the data processing, a `for` loop was used to iterate over the stocks (one iteration per stock name, totalling 100 iterations) with each iteration initializing a new Lambda instance, passing the data to said Lambda instance, and performing the processing functions. Finally, the data was passed back to a specified directory and saved for future use. This eliminated overhead data processing in future training runs.

4.2 Model design

Generating a random forest model that has been tuned correctly often means building many models to test the hyperparameters. The approach taken in this research was as follows:

1. Define a grid of potential parameters. This research optimized the number of features and the minimum leaf node size. Together these form a matrix of potential hyperparameter pairs, each of which must be tested to find the optimal model.
2. Iterate over the hyperparameter matrix, training a model on the train dataset each time. Record the validate MSE for each set of hyperparameters.

Hyperparameter	Values
Maximum Features	5,7,9
Minimum Sample Split	5,15,25

3. Select the best model hyperparameters (lowest MSE).
4. Retrain a new model with a larger number of trees using the train and validate datasets as the training data.
5. Test using the final testing holdout set.
6. Store the final model for further use. Repeat the process for each stock.

4.3 Implementation

In order to implement the desired model design, this research constructed eight Lambda functions. Each would support various components of the analysis:

1. Processing - Performs the data cleaning and processing on the raw data files.
2. Parampreds - Trains a random forest regression model with a single specified set of hyperparameters on a given stock's training data and produces predictions for the validation set.
3. Predmses - Aggregates the predictions from all trees that were trained on the same stock data with the same set of hyperparameters, and computes an overall MSE score for the ensembled prediction.
4. Minmse - Determines the set of hyperparameters that produced the lowest MSE for a specific stock.
5. Trainfull - Trains a larger random forest on the training and validation data for each stock using the optimally determined hyperparameters and generates predictions using the test set.

6. Testfull - Evaluates the new model's predictions for a stock's test data by calculating the MSE score.

7. Parent - Asynchronously invokes the functions above for each stock (and each permutation of hyperparameters).

8. Timingmetrics - Assesses the internal runtime metrics produced by these executions.

Each of these Lambdas was configured to have 3008MB of memory and a 15 minute timeout limit. They were also given full access permissions to both S3 and Lambda via an IAM role to get (put) data from (to) S3 and to invoke other Lambda functions. Lastly, they were equipped with layers that contained the necessary dependencies to run pandas and Sklearn.

All data and models used in the analysis were stored in a tiered folder structure across 100 S3 buckets which could be accessed, manipulated, and updated directly from the Lambda functions. In order to reduce the volume of concurrent read and writes to the same S3 buckets, the research established an infrastructure whereby the processed data, trained models, and generated predictions for each stock were stored in separate buckets.

The system's overall workflow consisted of a series of high volume asynchronous Lambda invocation blocks that read from S3, performed a specific operation, and wrote their output to a new folder in S3. This process is elegantly displayed in 1.

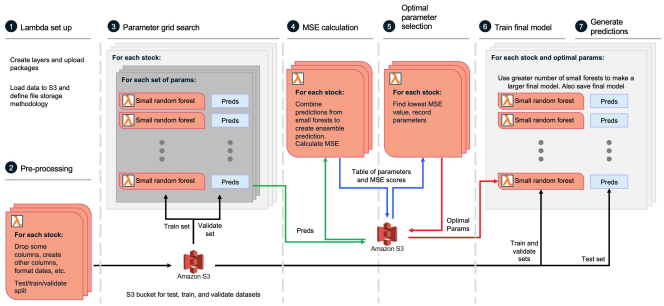

Figure 1: Overview of research architecture

Beginning with the raw data for each stock, the original data was transformed into a format that could serve as an input to a random forest regression model. This required the creation of 10 lag variables for the target variable of TYP-PRICE and the removal of all extraneous feature columns and rows containing NAs. Following this basic cleaning, the data was then split into training, validation, and testing sets and stored in folders within the appropriate stock's bucket.

Once the data for each stock had been parsed and converted into the desired form, it could be used to train the distributed random forest regression models. Since partitions of random forest models can be trained independently from one another and then aggregated later during the inference phase, this provided an incredible opportunity to leverage the highly scalable and parallelizable nature of Lambda functions to perform distributed hyperparameter tuning. For each stock and for each permutation of hyperparameters listed in the table in Section 4.2, this research trained a random forest regression model with 300 estimators. This process could be easily parallelized using Lambda since each function could be given a specific stock and set of hyperparameters as inputs. It should be noted that the memory utilization and computation time constraints imposed by Lambda made it necessary to further break these forests of size 300 into three smaller groves of size 100 each. The additional layer of parallelization now enabled each of the functions to satisfy Lambda's size and runtime requirements. This configuration resulted in 2700 asynchronous executors in parallel, and produced three validation prediction files for each hyperparameter permutation for each stock. With these predictions generated, the independence property of random forests could be leveraged to produce a single aggregated prediction and associated MSE score for each hyperparameter permutation.

Combining the validation predictions and computing the MSE for each set of hyperparameters simply required averaging the individual prediction values from the distributed groves. Since the prediction files stored were just a few MB in size, a single Lambda function was able to perform this process for all hyperparameter configurations of a given stock. Thus, this functional step only required an execution concurrency of 100. Even though this component of the workflow was quite swift, the overall runtime could be expedited further by parallelizing this step and having a different asynchronous Lambda function be invoked for each hyperparameter permutation. This would reduce the time required for this step by 9x. This is relatively trivial compared to the longest running elements of the full procedure. Therefore, it seemed prudent to identify the optimal set of parameters that produced the lowest MSE.

By iteratively reading in the previously calculated and stored MSEs as rows of a new dataframe, the pipeline could locate the minimum value and save the hyperparameters that produced it in just a few lines of code. This was the fastest and simplest function in the workflow chain and only needed to be asynchronously invoked once per stock. With the best hyperparameter set determined for each stock, the final phase of the analysis was entered: training, storing, and evaluating a final, optimized, larger, distributed random forest.

Similar to the methodology employed to perform the training and prediction generation during the hyperparameter grid search, here a Lambda executor fetched the saved optimal hyperparameters from S3, trained a new random forest regression model using both the training and validation sets, and calculated predictions for the test set. Since this was the final model, both the model and predictions were stored in S3 as .sav and .csv files, respectively. With full models being

saved in addition to the predictions, a similar scheme as before was implemented to reduce the chances of hitting either the runtime or memory constraints of Lambda. Since the final forest was designed to consist of 750 trees, training was broken down into thirty groves of 25 trees each. This ensured that no errors would be thrown and the increased concurrency acted to reduce the time necessary for this step as well. Under this strategy, 3000 executors were asynchronously invoked to produce thirty groves of size 25 along with their associated test predictions for each of the 100 stocks in the dataset. Now, all that was left was to combine these distributed predictions and compute an overall MSE for the model.

The prediction aggregation process required iteratively pulling and averaging the thirty independent prediction files. This average was then used to calculate the model's MSE score for the test set.

All of the function calls were generated within `for` loops of a single parent function. By using this parent Lambda to asynchronously invoke one Lambda executor for each stock (and each permutation of hyperparameters) when performing the embarrassingly parallelizable tasks of data processing and model training, the research leveraged the full benefits of this serverless computing environment and optimized the overall runtimes. Since each function stored its runtime in an S3 bucket as a .txt file, the empirical efficiency gains of successful parallelization could be easily compared.

These full runtimes, however, also include the time required to read in the data from and write the results of each function to S3 which is known to have relatively high latency, especially when concurrent operations are being performed. To more accurately measure each functions' runtime metrics, the research designed and developed timing infrastructure such that, not only was the overall execution time captured, but it was also broken down into time spent on invocation, computation, and I/O. By tracking each of these performance metrics on a more granular level, a clearer picture of the functions' behaviors was present. It also provided more insights into how the pipeline could be further improved.

Through the model implementation described above and the establishment of runtime tracking infrastructure, the research was able to successfully design, construct, deploy, and evaluate how serverless computing platforms such as AWS Lambda could be leveraged to perform massively distributed random forest optimization and training for time-series forecasting. While the primary metric this research focused on assessing and improving was the overall runtime, additional results and commentary are also provided on the breakdown between invocation, computation, and I/O times. Further, the comparison between Lambda and other frameworks such as EC2 instances both in terms of runtimes and monetary cost is also outlined below.

5 Results

The primary results of this research focus primarily on runtime and monetary efficiency. While the main intent was to devise a system of optimizing and training a massively distributed random forest in as short a time as feasibly possible, it is important to contextualize the runtime results by evaluating the financial ramifications of the system design. This section will provide an overview of the empirically determined runtime metrics related to invocation, computation, and I/O for each function. It will then move to explaining these findings within the context of the greater AWS computing ecosystem.

With regards to the absolute runtime of the analysis, in a perfectly parallelized setting, one would expect to see a total runtime for each step equal to $\max_i runtime_i$ rather than $\sum_{i=1}^{101} runtime_i$ which is what would occur if everything was run sequentially. The empirical versus sequential runtime performance analysis for our implementation is summarized in the tables below.

Function	Invocation (s)	Computation (s)	I/O (s)
Processing	3.255980	35.520818	24.381985
Parampreds	84.437543	1577.005723	12.372995
Predmses	3.540197	0.346063	9.922652
Minmse	3.467081	0.055044	1.461748
Trainfull	92.360568	453.238807	19.269423
Testfull	3.372392	0.546637	13.328561
Total runtimes	190.433761	2066.713092	80.737364

Table 1: Empirical runtime analysis breakdown for each function

Function	Invocation (s)	Computation (s)	I/O (s)
Processing	3.255980	3.221591e+03	1800.507082
Parampreds	84.437543	7.487812e+05	6380.858350
Predmses	3.540197	1.493793e+02	4489.880415
Minmse	3.467081	2.392879e+00	77.485839
Trainfull	92.360568	3.027070e+05	10599.520189
Testfull	3.372392	3.722374e+01	1083.174426
Total runtimes	190.433761	1.054899e+06	24431.426302

Table 2: Sequential runtime analysis breakdown for each function

Based on these tabular results, the dramatic runtime efficiency gain induced by the parallelized infrastructure is apparent. This research's entire process was able to successfully complete in 38.96 minutes. If it had been run in a fully sequentially manner with the same computing resources, this would have taken nearly 300 hours, thus representing a 99% improvement. While this improvement is quite staggering, it is not particularly surprising given that the longest running computational steps performed by the Parampreds and Trainfull functions were invoked with the highest concurrency

levels of 2700 and 3000, respectively. Since the concurrency limit was only set to 1000, these two massively parallel and asynchronous function invocations also experienced meaningful throttling which delayed their overall completion by a factor of almost 3x. Therefore, with a higher concurrency limit, these times could be reduced further, and, based on the current invocation runtime results, the entire runtime for the process could be limited to 16.14 minutes if fully parallelized without throttling.

In addition to the aggregate runtime comparisons, it is also interesting to note the distribution of execution times spent across the three components of invocation, computation, and I/O for each function. As can be seen in Table 1, the internal breakdown between these three aspects varies dramatically across the different functions. While the computationally intensive and highly parallelized tasks contained within Parampreds and Trainfull require the longest amount of time to invoke and perform computations, their I/O costs are actually lower than some of the other functions both in absolute and relative terms. For example, Processing is a fairly computationally simple function. Yet, because it involves reading, manipulating, and writing 100MBs files to and from S3, it experiences the highest latency I/O delays. Even some of the smaller functions such as Predmses also demonstrate interesting distributions that are highly consolidated towards I/O since they require reading and writing thousands of smaller files to and from S3 for all of the stocks (and hyperparameter configurations). While much can be gleaned from these tables alone, the runtime distribution can also be visualized across all executors invoked for a given function to better understand the potential influence of stragglers. The kernel density plots below capture these relationships succinctly.

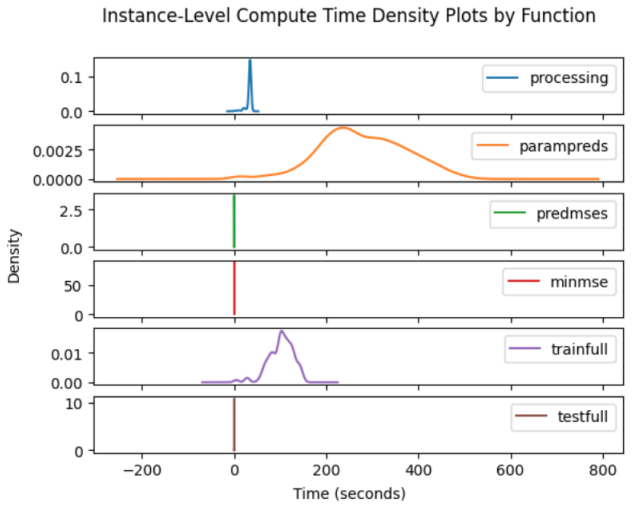

Figure 2: Kernel density plots for executor computation durations

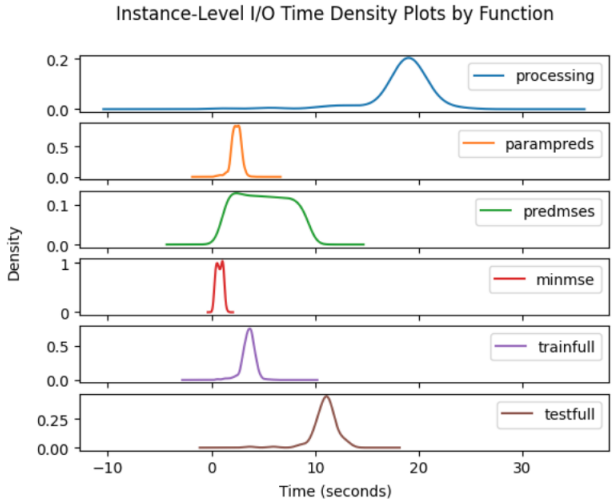

Figure 3: Kernel density plots for executor I/O durations

From these plots, it is readily apparent which functions require the longest amount of time to complete their given tasks. However, they also offer additional insights into the performance spread across the various executors. While most of the distributions are relatively mound shaped and symmetrical with small standard deviations, a few of the functions do exhibit noteworthy behavior. In particular, the computation time for Parampreds' executors varies greatly due to the different hyperparameter configurations. The I/O time for the Predmses' invocations has a fascinating plateau profile due to the manner in which it traverses all hyperparameter permutations for a given stock. Overall, these graphics offer a more granular understanding of the behavior of each of these functions and allow for quick comparisons between them.

5.1 AWS Ecosystem Runtime and Cost Comparisons

While the above results speak to the runtime analysis with a specific focus on this research's pipeline, it is important to compare this hyper-parallelized framework to other computing options. For example, a direct comparison can be drawn to potential EC2 configurations within AWS. 3 and 4 show architectural configuration, performance and cost for various AWS computing environments completing the same tasks outlined in this research. It should be noted that the EC2 instance calculations were scaled based on a single stock run on a 4 core, 16 GB machine which ran in 25.48 minutes. This computation repeated 100 times in a row leads to a run time of 43.08 hours. Scaling up cores then results in an efficiency gain assuming proper optimal parallelization is configured.

Architecture	Instance name	# of CPUs	Memory
Lambda	NA	2	3GB
EC2	t4g.xlarge	4	16GiB
EC2	m6g.4xlarge	16	64GB
EC2	c6a.48xlarge	192	384 GiB

Table 3: Single instance architecture details

Architecture	Instance name	Runtime	Cost
Lambda	NA	38.96 minutes	$53.89
EC2	t4g.xlarge	43.08 hours	$5.79
EC2	m6g.4xlarge	10.77 hours	$6.634
EC2	c6a.48xlarge	53.85 minutes	$6.59

Table 4: Single instance performance and cost details

4 shows that Lambda accomplishes a 99% runtime improvement over the t4g.xlarge EC2 instance. Lambda even maintains a 15 minute improvement over the best EC2 instance, the c6a.48xlarge. These results come with heavy caveats but the assumptions hold that the order of magnitude improvement is considerable. These runtime savings do not even factor in the more user friendly Lambda infrastructure which can be far easier to maintain than the EC2 overhead. Clearly, these results show that Lambda possesses tremendous runtime benefits over traditional serverful computing options when applied for embarrassingly parallel modeling.

The runtime benefits do appear to come at financial expense, however. Cost information from this research and AWS's pricing calculator (4) shows that Lambda is nearly 10X more expensive than serverful options. At present, this represents a considerable drawback to this implementation but the research in question did not optimize for cost. This offers a future consideration and opportunity for exploration. Lowering the cloud cost for the Lambda implementation might certainly be possible with further development and configuration. This is something the researchers in this project will look to prioritize in the future.

6 Conclusion

This research clearly established AWS Lambda as an effective means of improving runtimes for big data time-series forecasting. From a usability perspective, this research showed that AWS Lambda is capable as an ETL pipeline mechanism that can streamline big data preprocessing. This research successfully built infrastructure to support big data modeling using highly paralellized processing, model tuning, and prediction functions. Further, by building embarrasingly parallel random forests, this research was able to find a several order of magnitude runtime improvement for AWS Lambda over sequential computing methods. This method while efficient from a runtime perspective still has limitations from a cost standpoint that need to be further optimized in future work. Moving forward, the application of Lambda in parallelizable big data modeling is of tremendous interest and warrants more exploration. High parallelization and better resource management stand out as two key opportunities for future work. Regardless, the results of this research lay the foundation that AWS Lambda can be successfully deployed with significant runtime benefits.

7 Metadata

The data of the project can be found at:

https://www.kaggle.com/datasets/debashis74017/stock-market-data-nifty-50-stocks-1-min-data

The code of the project can be found at:

https://github.com/Griffin-McCauley/DS5110FinalProject

The presentation of the project can be found at:

https://darden-virginia.zoom.us/rec/share/sAEL0XA6vufwKLqaYvapDNtNuwWWz1cZKivLobVFkSev1P3iGUr17jmfEj1cncyQOlmvNGhuGh?startTime=1683036623000

References

[1] Luckyson Khaidem, Snehanshu Saha, and Sudeepa Roy Dey. Predicting the direction of stock market prices using random forest, 2016.

[2] Subba Rao Polamuri, Kudipudi Srinivas, and A Krishna Mohan. Stock market prices prediction using random forest and extra tree regression. *Int. J. Recent Technol. Eng*, 8(1):1224–1228, 2019.

[3] Perry Sadorsky. A random forests approach to predicting clean energy stock prices. *Journal of Risk and Financial Management*, 14(2), 2021.