Serverless Computing for Big Data Time-Series Forecasting

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 the project is to create a set of random forest models using embarrassingly parallelized functions through AWS Lambda. Specifically, random forest models are 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 runtime 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 1000 trees or more and 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 can 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 since 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 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. We optimize 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.

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. Test using the final testing holdout set.

5. Store the final model for further use. Repeat the process for each stock.

4.3 Preliminary Implementation & Results

In order to implement the desired model design, we constructed five Lambda functions which would support various components of our analysis:

1. Performing the data cleaning and processing on the raw data files.

2. Training a random forest regression model with a single specified set of hyperparameters on a given stock's training and validation data. (Note that there is no optimization or hyperparameter tuning being conducted yet (see 7).)

3. Generating and evaluating predictions for a stock's test data using the newly trained model.

4. Asynchronously invoking the functions above for each stock.

5. Assessing the runtime and MSE 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 our analysis are stored in a tiered folder structure in a single S3 bucket which can be accessed, manipulated, and updated directly from the Lambda functions.

By using a parent Lambda to asynchronously invoke one Lambda executor per stock when performing the embarrassingly parallelizable tasks of data processing and model training, we were able to leverage the full benefits of this serverless computing environment and optimize the overall runtimes. Since we programmed each function to store its runtime to our S3 bucket as a .txt file, we are able to effectively compare the empirical efficiency gains of our successful parallelization.

In a perfectly parallelized setting, we would expect to see a total runtime 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 runtime performance analysis for our current implementation is as follows:

- Data processing took 64.956 seconds, representing a 98.85% reduction from the sequential runtime of 5670.190 seconds (94.503 minutes)
- Model training took 241.292 seconds (4.022 minutes), representing a 98.66% reduction from the sequential runtime of 18005.951 seconds (300.100 minutes or 5.002 hours)

These runtimes, however, also include the time required to store the results of each function in S3 which is known to have high latency, especially when concurrent writes are being performed to the same bucket and folder. In our analysis, we found that S3 storage time alone took 11.671 and 3.188 seconds during the processing and training tasks, respectively. While these values are not terribly detrimental to the overall performance benefits of our approach, the 11.671 seconds spent on I/O during data processing does constitute 17.97% of the total runtime, and, therefore, may be an area for further investigation and optimization down the line.

5 Full Scale Baseline Results

At this point in the research, full scale runtime testing on parallelized Lambda functions has not been fully completed (see 7). However, in order to establish a baseline against which to compare, model tuning and training was done on a 4-core, 16 GB machine for a single stock dataset. The NESTLE stock information was preprocessed into 10 time-lag variables and a random forest was tuned to predict 5 minute ahead stock price. The model was tuned using a hyper-parameter grid of 5 feature values and 3 min sample values (15 combinations) over 100 estimators or trees. Then, the model was retrained with best combination of hyper-parameters and 500 trees in the forest. The 4-core, 16 GB CPU ran this entire modeling process in 1,436.53 seconds or 23.94 minutes. It is important to realize that this training was for a single dataset amongst 100 datasets and was done with a relatively sparse variable space. Executing 100 sequential models in this manner would take approximately 40 hours based on this baseline. Clearly, this immense runtime motivates the analysis in this study. Preprocessing was also baselined on the 4-core, 16 GB machine and this took 30 seconds for the single NESTLE file. Following similar logic, 100 datasets would require nearly 50 minutes to preprocess. Again, this lengthy sequential process clearly motivates the use of parallelized Lambda functions to improve runtime efficiency.

6 Preliminary Conclusion

At this point in the project, we have set up the underlying infrastructure for preprocessing and modeling our data. Further, we have run baseline testing to establish worst-case performance metrics against which to compare. Finally, we have acclimated ourselves to the AWS environment and serverless computing more broadly through detailed research and experimentation across the platform.

7 To-Do

The goal of the final phase of the project is to use Lambda functions to fully parallelize the optimization process. The challenges we face are that Lambda functions have limited memory and only 15 minutes of processing time before timing out. As such, large models will need to be trained on multiple Lambda instances and then combined. This is easy with random forests as smaller models built on the same dataset can simply be concatenated to make larger forests.

However, Lambdas cannot pass information to each other, so all data must be stored separately in a new S3 bucket and then reaccessed by a new function. In order to build all the many forests needed for optimization, then the large models needed for the final product, we will need to establish a complex system of calling multiple Lambda instances within nested loops. In high level pseudo code:

```
for STOCK in stock_list:

    for params in parameter_grid:
        data = STOCK_training_data.csv

        for n in number_of_functions:
            Launch_lambda_function[
                # Function will train a random
                forest with small number of trees
                # Will then save forest in S3

                # Will also save .txt file with
                amount of time needed to run
            ]

# Once all optimize functions conclude...
```

```
for STOCK in stock_list:
    data_directory = S3/STOCK

    # Load all optimized models
    models_list = load[data_directory/*]

    # Combine models (simple concat) by parameter
    # giving a single model for each parameter set
    for param in parameter_grid:
        param_model = concat(subset from model_list)

        #generate MSE from validate set
        mse = param_model.mse(validate)

    save_params = params_with_lowest_mse

# Finally, retrain the model using one parameter set.
# Use similar structure to above, with many models
# concatenated after training is complete. However,
# use a larger number of trees and a larger dataset.
```

8 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

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.