Distributed Stock Prediction System

Josiah Buxton
Department of Computer Science
University of Colorado Boulder
Boulder, Colorado 80303
Email: Josiah.Buxton@colorado.edu

Christopher Godley
Department of Computer Science
University of Colorado Boulder
Boulder, Colorado 80303
Email: Christopher.Godley@colorado.edu

Dixit Patel
Department of Computer Science
University of Colorado Boulder
Boulder, Colorado 80303
Email: Dixit.Patel@colorado.edu

***Abstract*—With the rise of machine learning techniques and distributed systems, real time stock prediction has become a feasible area of study. Analyzing the time series data and coming to accurate future predictions for a particular stock is a very difficult problem due to the amount of variables that act on the stock market. This study proposes loading a previously programmed stock prediction system onto a distributed framework. Hadoop was used for the distributed file system and Spark was used for forming input feature vectors and training each of the models. We compared computational time of these jobs to the jobs running on different environments. We've found similar runtime performance between a single computer and a distributed cluster of 4 datanodes when training a maximum of 200 stocks.**

I. Introduction

The main objective of this study is to distribute a stock market prediction system onto a customized Hadoop cluster. Due to advances in machine learning techniques, accessibility to big data, and the reduction in price of computing resources, predicting trends and patterns in large datasets have become increasingly easier. Predicting trends lends itself very well to the domain of inferring the daily price of different stocks. Each stock's price is affected by many variables such as politics/law, the economy, weather trends, social media, etc. The interaction between these variables makes the input data very noisy and difficult to predict accurately. In order to retrieve and format the input data into feature vectors that will be fed into different machine learning models, extensive computing resources will be needed. Also, in order to generate a well balanced stock portfolio, thousands of stocks need to be analyzed and vast amounts of data (input, feature vectors, analysis, etc.) need to be stored and easily accessible. A distributed system will address all of these needs.

A. Distributed File System

A major requirement for building data-intensive applications is to have a robust distributed file system that can provide easy access to data along-with fault tolerance and high availability. We use Apache Hadoop framework that provides the Hadoop File System [8] and MapReduce [2] programming as our underlying storage and processing model. We also aim to utilize the power of container technology to easily deploy and manage our application on the file system. With a container orchestration tool like Kubernetes[1], we can automate the process of deploying and scaling the application as determined by our Load Balancer.

B. Load Balancing

A cluster is a type of distributed system loosely defined as a set of networked computers that get queued and assigned jobs by a scheduler. In homogeneous clusters, all nodes on the network are identical, using the same hardware and software to complete their tasks. Traditional load balancing on these systems can attempt to optimize throughput or energy efficiency based on job size, execution time, etc. Heterogeneous clusters provide a more complex scheduling and balancing problem, as these clusters may contain a multitude of systems with different hardware or operating systems. Since the stock prediction model should ultimately distribute data fetching, preprocessing, and the machine learning algorithms, there will be a large degree of variance to the computing resources required. The disparity between jobs will allow optimizations such as resource balancing to be much more impactful than on homogeneous systems. Smaller jobs may be scheduled to nodes with fewer resources and vice versa. Since heterogeneous clusters can optimize to a wider variety of machines, we expect homogeneous performance to be efficient as well. Additionally, a cluster load balancer that can tie into a distributed file system containing program resources will be capable of scheduling based on the locality of data.

II. Related Work

A. Distributed Stock Prediction Systems

Stock market prediction using machine learning concepts is a very heavily studied field. However, there are much fewer papers that implement stock prediction on distributed systems. First, we will discuss key points from relevant papers in machine learning techniques for stock time series analysis. Then we will introduce papers that apply these techniques to a distributed framework.

Before creating the machine learning models, feature vectors must be built in order to train the models effectively. Another large area of research is efficiently creating small feature vector datasets that train very accurate models. There have been attempts at applying feature vector selection methods specifically to the stock prediction domain. One particular study used union, intersection, and multi-intersection approaches to filter out 80% of the unrepresentative variables [10]. Filtering the feature vector dataset is essential in the application of stock price prediction because it drastically

reduces the computational time training daily models for each stock.

One technique that is notable in the field of stock market forecasting is Ticknor's implementation of a Bayesian regularized artificial neural network [9]. In this model, Ticknor combines Bayesian probabilistic statistics with a standard neural network in order to significantly reduce the amount of features needed to train a model. Ticknor compared his model with current machine learning models at Goldman Sachs Group Inc. and Microsoft Corp. to determine the model's effectiveness. Stock prediction has also been achieved through the use of decision trees and SVM's, where the data was preprocessed using distributed algorithms in order to achieve a computational speedup [11].

The majority of distributed stock forecasting studies implement a variation of Google's MapReduce programming model called Hadoop [3] [5]. One study found success in computational speedup of training neural networks by distributing the feedforward and backpropagation algorithms, but only when they utilized a cluster of over 30 nodes [3]. There have also been implementations of the distribution of training machine learning models over an array of GPU resources but it was found that there wasn't a significant speedup in computational time (probably due to small datasets and small neural networks) [4]. As most success has been seen with utilizing the Hadoop framework, we will try to do the same with our stock price prediction system. The stock prediction system is already developed to the point where it can be fed parameters for a model and a stock ticker symbol and a model will be trained and future predictions will be made. We will **map** the generation of all the stocks' machine learning models to key/value pairs that can be fed into the MapReduce programming model. The keys will represent serial numbers for a unique stock and machine learning model. The values will represent the stock ticker symbol and the parameters for the particular machine learning model. We will then **reduce** by iterating over each of the the ticker symbols/parameters and generating their respective models on different nodes in the cluster which will be returned as the output for a particular key [2].

B. Load Balancers

Research into load balancing on clusters stretches all the way back to the '80s. Despite being quite a mature subject, there is still a lot of room for novel research to be done as both the hardware and software has come a long way. In 2001, Pinheiro et al. [7] demonstrated three methods of handling load balancing for power and performance. The first is an application level algorithm, which must acknowledge the shortcomings in accessing low level system or network data. The second was an OS level algorithm, which solves the problem of accessing the low level system information but does not have all the benefits of the application level algorithm. The third was a natural negotiation of the two algorithms. To attain the best results, we will produce a system that can interact at these two levels, as the necessity of system level information is evident, but the flexibility of the application level is just as important.

The power utilization versus performance relationship presented in that paper is also quite a large topic in measuring the effectiveness of our algorithms. Ultimately, only the user can prioritize power utilization over performance, or the other way around. We will be prioritizing performance in our implementation, attempting to maximize the throughput of our cluster. In our specific application, time is the most sensitive component. Stock predictions are computed for an advantage on trading, and being successful relies heavily on maintaining that competitive edge.

To best test our stock prediction cluster, we would run a variety of baseline scheduling algorithms to compare against. In the Mahalle et al. paper from 2013 [6], they identify two main scheduling algorithms to act as benchmarks. The round robin algorithm provides a great baseline to compare against, as it's a simple passing of jobs around the cluster in an ordered fashion. Equally spreading the load is another rudimentary method to schedule the jobs across the cluster. Each node will be queued such that the load will be maintained across the entire cluster evenly. This does add complexity, however, as it requires the balancer to monitor the jobs. These methods would be the first to implement and compare against the default Spark scheduler, Yarn, as well as the built in Kubernetes scheduler. The ultimate goal would be to use these as our baseline measurements for our algorithms based on using both application data and system data. The final stage will be creating a feature vector for training an ML model. The feature vector will consist of the same kinds of information as the conventional algorithms, and it will be interesting to see how the performance compares.

III. Implementation

A. Stock Prediction System

The stock prediction system itself is a collection of python scripts and a document based storage system. A sub-collection of scripts are meant to be run in a sequential order everyday to retrieve real-time stock price data from an API hosted by AlphaVantage. There are a variety of attributes being retrieved including: price, simple moving average, exponential moving average, sector data, etc. All of this data is stored into pickled files where each file pertains to a particular stock.

Another script generates a feature vector when given a stock and a series of parameters. These parameters determine what types of data to retrieve for a stock, the prediction range for how many days out you'd like to train a model to predict on, sentiment analysis on relevant news articles, etc. The script outputs a uniquely identified training vector that can then be inputted into different machine learning models. Currently, we've developed linear regression and multi-layer perceptron regression models but we would like to add probabilistic models in the future.

When the model scripts are run with a stock passed as an argument, a unique model is stored into a dictionary database that is then pickled into a file. All of the parameters for the

feature vector and machine learning model are stored in this database so that they can be retrained programmatically.

Another script uses the model database and the stored stock data in order to generate predictions on all of the unique models. It then stores all of these predictions into a pandas dataframe that is pickled upon termination of the script. There is a final shell script that provides an interface to call all of the other scripts that make up the system. This interface was used when we implemented the Spark programming model on the distributed file system.

Stock Prediction System Scripts:

- analysis.py - Performs different analysis techniques on the outputted pandas dataframe from prediction_machine.py
- daily_update.bat - Runs the stock_price_data and prediction_machine scripts (bat file scheduled in Windows OS to run everyday)
- feature_vector.py - Given a stock and parameters, generates a complete training feature vector for ML models
- linear_regression.py - Given a stock and feature_vector, generates a linear regression model to fit the data
- mlp_regression.py - Given a stock and feature_vector, generates a multi-layer perceptron regression model to fit the data
- model_db.py - Class for storing unique ML models into a database
- news_scraper.py - Given a stock and parameters, performs semantic analysis on relevant news articles
- prediction_machine.py - Generates predictions for the current stock price data on all of the unique ML models in the database
- spi.py - Stock prediction system interface
- stock_price_data.py - Retrieves relevant stock price data for all stocks from AlphaVantage and stores them into a file based database

To the right is an example of the output of one of the linear regression models for Apple's stock. It shows the actual price as black points and the blue line depicts the predictions of the linear regression model after it has been trained with data from the feature vectorizor python script.

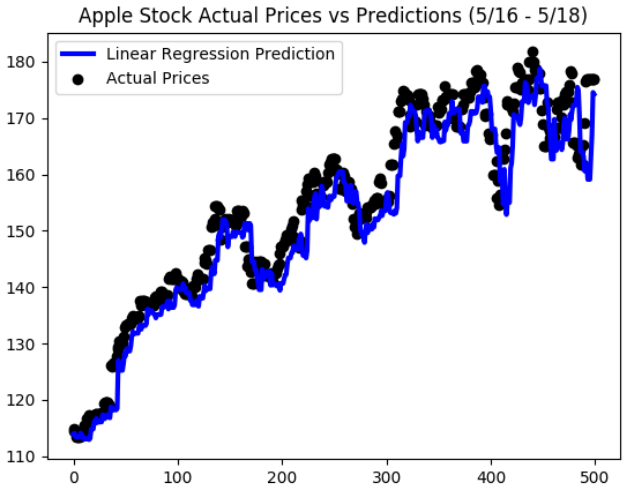

Fig. 1. Example Linear Regression Output

B. Hadoop Distributed File System

The distributed file system is setup currently on a hadoop cluster of five different nodes. There are single namenode/-yarnmaster nodes and three datanodes within the cluster. Each of the nodes are t2.large EC2 instances which contain an Intel Xeon family processor with 2 vCPU's and 8GB of RAM. They were spawned with 100GB of allocated memory and are running a minimalized Centos 6.5 distribution of linux.

A configuration directory was built during this project to store scripts and configuration files that allow the cluster to be programmatically generated. All of the functions that can be performed on the cluster are located in the cluster_manager script at the root of the configuration directory. This script provides the following functions with the associated command line arguments:

- -i - Initializes the cluster of AWS nodes by spawning hadoop dfs/yarn startup scripts on all available nodes in the local ssh config file. Copies over updated files on the local disk of namenode to the HDFS.
- -k - Kills the cluster of AWS nodes by spawning hadoop dfs/yarn shutdown scripts on all of the available nodes in the local ssh config file.
- -c - Copies over the configuration directory to all of the available nodes in the local ssh config file and runs a series of scripts that will install dependencies and copy over configuration files to appropriate directories (Should only be when virgin resources are available and the cluster needs to be initialized. Note: The public/private IP addresses of the virgin resources need to be stored in an init script to configure communication correctly between nodes).
- -m - Performs a mapreduce job on the cluster (Currently not implemented).
- -s - Performs a spark job on the cluster (Trains all of the stocks located in a model_db dataframe by formatting raw data from the AlphaVantage API and feeding the formatted data into linear regression models).
- -t - Helper function that copies a file/directory over to all of the nodes in the cluster (source and dest file locations are passed as command line arguments).
- -r - Helper function that runs a command on all of the nodes in the cluster (command is passed as a command line argument to this script).

C. Spark

Spark and all of its dependencies are installed with -c option of the cluster_manager script. We are using python3.6 and the sci-kit learn package for our machine learning models. The

pyspark job that is performed on the cluster is invoked by the -s option and runs a script that is located on the yarnmaster. The script generates RDD's by parallelizing python dictionaries of stock data that are keyed into by a specific metric (SMA, EMA, MACD, price, etc.) and then a timestamp. Note that each RDD contains all of the data for a particular stock. The RDD's are transformed into formatted stock data that can be fed into linear regression models. The output of the previous transformation becomes the input to the next transformation that generates trained linear regression models. The RDD's are then collected and stored on the distributed file system.

Most of the configuration settings were set to the default values but some needed to be changed based on our specific application. These configuration settings are available in the spark-env.sh and spark-defaults.template files in the configuration directory.

D. Virtual Machine Management

The test environment is a crucial part of determining the success of any project. As a reminder, the cluster may be managed at the application level by the use of the cluster_manager script. The important thing to note is that this script does not manage the cluster at a hardware/machine level. Baseline testing for Spark was done on EC2. To accomplish our load balancing goals, we need access to both sides of the system. A complementary "spawn_vm" script was created to control the spawning of discrete computing resources for management at the hardware level.

Running spawn_vm with various command line arguments will allow the creation and management of specified virtual machines. These are KVM based virtual clients that are configured via the cloud-config utility [REF]. By the use of cloud-config, we have total control over all aspects of the spawned client's attributes, including hardware resources, networking protocols, and any runtime scripts that may be relevant. By use of cloud-config and the KVM hypervisor, we can spawn however many homogeneous or heterogeneous computing resources as desired, hardware permitting. Cloud-config manages the configuration of a VM by the means of an init.iso file. This file is generated after editing both a user-data and meta-data file as desired. The user-data file is where boot commands, run commands, system info, ssh keys and user information is placed, among other attributes of the system. The meta-data file only contains the instance-id and the local-hostname.

Using the spawn vm utility, a user may create or destroy as many VMs as desired. Virtual machines may be managed as a whole, or individually. The process of creating a virtual machine starts by generating a unique identifier for each VM, which may be utilized by a user to allow individual or batch control of the cluster. Following this, a unique ssh key pair is generated for each new client. This is the main method of interaction once a VM is alive. Now the cloud-config's init.iso information is generated by first utilizing config templates and the "sed" command to populate unique user-data and meta-data files for each VM. Using the genisoimage command, unique init.iso's are produced for each VM as well. At this point, the only remaining task before installing the new client is to provide it a unique storage disk. These disks are all folders which are cloned from a template directory. The "virt-install" command will finalize the installation. This is when hardware resources are specified, as well as placing the init.iso into the virtual CDROM, specifying the desired networking interface (a virtual bridge again) and disk path. The following details the functions implemented by the spawn_vm script:

- -c - Creates an atomic-host VM using the method described above
- –spark-create - The first time this function is called, it spawns 5 VMs: yarnmaster, namenode, datanode1, datanode2, and datanode3. On subsequent calls, only datanodes are spawned.
- -n - Specifies the number of atomic-hosts or datanodes to spawn (default 1)
- –ip - Performs an arp-scan on the virtual bridge the VMs are networked on.
- –net-config - Updates iptables and manages routing for all nodes in cluster via /.ssh/config and /etc/hosts files. These files are generated with layered networking, such that regardless whether the VMs are on the same virtual bridge, the same host, or on separate machines, a single 'ssh yarnmaster' command will always find the appropriate destination.
- –start - Starts VM (supports node and group granularity)
- –stop - Stops VM (supports node and group granularity)
- –ssh - ssh onto desired VM
- –scp - copy files to desired VM (supports node and group granularity)

E. Computing Resources

For the task of running Kubernetes to assist in the management of our container based cluster, we chose to run Fedora Atomic on our simulated hosts. This OS was built with Kubernetes in mind, and is by result lightweight and perfectly suited for our needs. Virtual machines running our pre-configured Kubernetes instances on the Fedora Atomic hosts will be allocated containers from a job queue, which is the queue for all stock prediction applications to be placed into. The atomic-host VMs created with the -c flag support this cluster.

To establish our virtual machine based node management, we began by duplicating the work done on EC2 by constructing a Spark cluster via the –spark-create flag. The purpose of this was to both provide way to verify the functionality of our system, and to explore the hardware limitations we'd be working with. The EC2 cluster was built with CentOS-6.5 clients, and since the Fedora Atomic OS does not contain a package manager, we decided to mirror the EC2 cluster by using CentOS on our VM solution as well. This would ultimately be a fateful decision, as these images are much heftier than the atomic-hosts, and our resources are already limited.

Physical resources that we utilize for this project include Amazon AWS EC2 services as well as personal hardware. We have a few high end desktop type servers available amongst us that act as the innate heterogeneity of our available computing resources. These machines will all have the preconfigured virtual machines installed on them, spawned through the virtual machine management script, spawn_vm. The reason we wish to utilize the virtual machines alongside the containers is that the containers represent the logical cluster, while the virtual machines represent the simulated physical hardware hosting the cluster. Utilizing a single machine with all of its resources available to the containers would not reflect the performance across heterogeneous computing resources. With enough space on a homogeneous system, such as EC2, we may use the virt-installer to modify the resources to simulate further heterogeneous nodes.

IV. Experimental Setup

Initially, we chose to distribute the training of linear regression models on formatted stock price data over the cluster. We found that training a linear regression model is not very time consuming so we created a similar pyspark job that formatted raw stock price data and then trained the model with the formatted data.

To test the distributed systems, we performed the synthesized pyspark job on different stock batches and collected the time to completion. In order to perform the job, the pyspark job needed to be supplied with a model database, which is a dictionary that contains all of the unique models generated in the past with the associated parameters for the model and feature vectorizor. We created model databases that contained different sized stock batches and used these to generate our different data points. We performed this pyspark job on both the AWS cluster and the simulated VM cluster.

We created another script to generate our baseline metric which was supposed to be a similar task run on a single computer. We passed the generated model databases to this script and it formatted stock data, generated updated linear regression models, and saved the models to disk.

V. Results

From the experiment described above, we collected a total of 8 datapoints due to time constraints. Below are two different depictions of this dataset.

We expected to see much different data than what was collected. We were expecting to see a speedup in the AWS cluster as we increased the number of stocks and scaled the system. There are a number of reasons why we did not see the results we were expecting. The first major potential flaw is in the pyspark algorithm which initially loads in the stock prediction data from disk on the yarnmaster before it parallelizes it. It could be the case that the RDD's generated are being sent to each of the nodes from the yarnmaster over TCP/IP. This is different than specifying locations of stock price data on the distributed file system, so each node can generate RDD's based on what's available on their local disk.

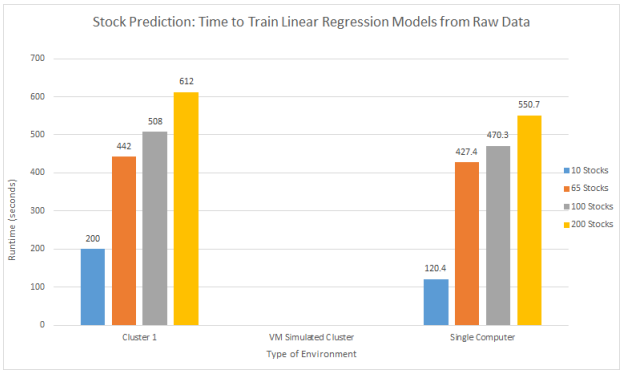

Fig. 2. Depiction of the data collected as a bar graph. Each series represents a different number of stocks. The x-axis is associated with which environment and the y-axis is the seconds it took for the model to be trained from the raw stock data.

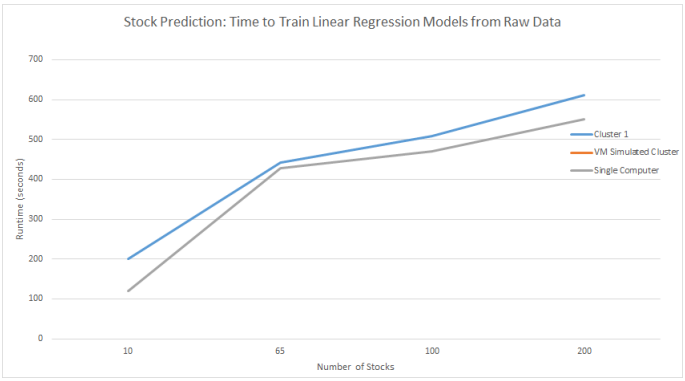

Fig. 3. Depiction of the data collected as a line graph. Each line series represents a particular environment the experiment was performed in. The x-axis is associated with the number of stocks and the y-axis is the seconds it took for the model to be trained from the raw stock data.

Another potential flaw in our experiment could be the partitioning of jobs. As we scaled the system, we noticed that a large portion of our stocks had a very small amount of stock price data associated with it. For a standalone single computing system, formatting feature vectors and generating a linear regression model is fast when working with small amounts of data. This problem was probably perpetuated by the design decision to format a maximum of 500 feature vectors (500 days). We made this decision because of time constraints. We would like to look into intelligent algorithms of partitioning the jobs in the future and increase the maximum amount of feature vectors to generate to see a speedup in the pyspark job.

It is important to note as well that we saw an enormous speedup after we adjusted our pyspark job to format the raw stock data although it is not presented in the data. Knowing this, it makes us believe we could see a speedup if we added more computationally/bandwidth intensive tasks to the algorithm, such as making predictions on all of the trained models and/or downloading the stock price data from the AlphaVantage API to the hdfs. This will be done in the future along with adding semantic analysis to the stock prediction

system. We also plan on training different types of machine learning models on each of these stocks in the future, so training of the models and cross-validation may be more computationally intensive tasks we wish to distribute.

We were unable to get measurements for the VM cluster as the pyspark job was not fully implemented.

VI. Conclusion

Our completed system maintains control at both the application and machine levels. We've addressed how to potentially improve the stock prediction model at the application level in our results, such as analyzing points of bottleneck in the model and retooling the model to better parallelize its tasks. To improve the throughput of our stock model at the machine level there are also many things we need to address. The machine level management was set up and configured, but scheduling and balancing tasks did not make it to the implementation stage. The next step would be to complete verification of the VM-based Spark model and begin experimenting with Yarn optimizations. Spark has official integration with Kubernetes, so we could load the model onto containers managed by Kubernetes running on our atomic-host VMs. At this point, we could then configure the cluster to run with either Yarn or Kubernetes as the application scheduler.

Despite the shortcomings, the stock prediction model was succesfully implemented on a distributed system. We've created an effective API for creating and managing our cluster at any level necessary. Our results were not what we wanted, but they are very much part of the process and we were left with many more ideas than we started out with. A smaller stock prediction model may run well enough on individual hardware, but to complete predictions on a full stock portfolio will require a more efficient distribution, and more intelligent scheduling and balancing.

References

[1] Brendan Burns et al. "Borg, Omega, and Kubernetes". In: *ACM Queue* 14 (2016), pp. 70–93. URL: http://queue.acm.org/detail.cfm?id=2898444.

[2] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[3] A. K. Dubey, V. Jain, and A. P. Mittal. "Stock market prediction using Hadoop Map-Reduce ecosystem". In: *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. 2015, pp. 616–621.

[4] J. Jenq. "Parallel Implementation of Moving Averages and Stock Market Prediction". In: (2012).

[5] Emad A. Mohammed, Behrouz H. Far, and Christopher Naugler. "Applications of the MapReduce programming framework to clinical big data analysis: current landscape and future trends". In: *BioData Mining* 7.1 (2014), p. 22. ISSN: 1756-0381. DOI: 10.1186/1756-0381-7-22. URL: https://doi.org/10.1186/1756-0381-7-22.

[6] Sandip Patel et al. "Article: CloudAnalyst : A Survey of Load Balancing Policies". In: *International Journal of Computer Applications* 117.21 (2015). Full text available, pp. 21–24.

[7] Eduardo Pinheiro et al. "Load Balancing and Unbalancing for Power and Performance in Cluster-Based Systems". In: *In Workshop on Compilers and Operating Systems for Low Power*. 2001.

[8] Konstantin Shvachko et al. "The Hadoop Distributed File System". In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. ISBN: 978-1-4244-7152-2. DOI: 10.1109/MSST.2010.5496972. URL: http://dx.doi.org/10.1109/MSST.2010.5496972.

[9] Jonathan L. Ticknor. "A Bayesian regularized artificial neural network for stock market forecasting". In: *Expert Syst. Appl.* 40 (2013), pp. 5501–5506.

[10] Chih-Fong Tsai and Yu-Chieh Hsiao. "Combining Multiple Feature Selection Methods for Stock Prediction: Union, Intersection, and Multi-intersection Approaches". In: *Decis. Support Syst.* 50.1 (Dec. 2010), pp. 258–269. ISSN: 0167-9236. DOI: 10.1016/j.dss.2010.08.028. URL: http://dx.doi.org/10.1016/j.dss.2010.08.028.

[11] D. Wang, X. Liu, and M. Wang. "A DT-SVM Strategy for Stock Futures Prediction with Big Data". In: *2013 IEEE 16th International Conference on Computational Science and Engineering*. 2013, pp. 1005–1012. DOI: 10.1109/CSE.2013.147.