International Journal of Technology and Applied Science (IJTAS)

E-ISSN: 2230-9004 ● Website: www.ijtas.com ● Email: editor@ijtas.com

Scalable Stock Market Prediction Using MapReduce-Based Algorithms: A Distributed Computing Approach

Dr. Thallapally Sameera Pallavi

Freelance Faculty
Computer Science Department
Kakatiya University, Warangal,
Telangana-506009, India.
Mail ID: thallapally.sameerapallavi@gmail.com

Abstract:
The amount of data that is generated by stock markets is growing in volume, velocity, and variety, making it difficult for a single machine to process in a timely manner. In this paper, we explore how feature engineering and model estimation for stock market prediction can be made scalable using the MapReduce programming model as a means of parallelization. The study briefly summarizes the existing literature on machine learning for market forecasting and on distributed computing for analytics, and introduces the proposed MapReduce-based pipeline: mappers calculate technical indicators and partial sufficient statistics for every partition of ticker data; a reducer sums up the statistics to obtain an estimate of a ridge-regularised direction classifier. The pipeline was tested on 500 to 8000 synthetic equity series, with a planted weak autoregressive signal, with true multiprocess execution on a two-core machine. In the distributed model, the centralised coefficients were reproduced within 4.4×10^{-13}, thus proving that the approach does not alter the result of the computation, rather that it merely speeds up the calculation. The run times were reduced by 1.31 to 1.71, compared to sequential execution, and out-of-sample directionality accuracy was close to 51.3 per cent, compared to 50.7 per cent for the maximum class. The paper covers the overheads, data leakage, backtest overfitting, and comparison of Hadoop MapReduce and in-memory engines.

Keywords: stock market prediction; MapReduce; distributed computing; big data analytics; Apache Hadoop; machine learning; time series forecasting.

1. Introduction

All orders, quotes, and trades on a modern exchange are recorded, time-stamped, and stored. When multiplied by thousands of securities, hundreds of days of trading, and a long list of other data sources, ranging from news feeds to social media, the data volume grows to such an extent that it may become difficult or impractical to analyse it on a single workstation. Gandomi and Haider (2015) explain this as the three familiar V's of volume, velocity and variety, of which financial markets have all three. When

analysts need to retrain their forecasting models on the whole inventory of equities each day, that's not a matter of which algorithm is more accurate, but how quickly and how accurately it can be executed.

The forecast is subject to debate. According to the efficient market hypothesis, stock prices are already priced for the information available, and so systematic excess returns should not be available from past stock prices (Fama, 1970; Malkiel, 2003). The adaptive markets hypothesis, an alternative thought, suggests that predictability emerges and vanishes as market players learn, compete, and adapt (Lo, 2004). Empirical studies have established that there are economically meaningful patterns, including the momentum effect (Jegadeesh & Titman, 1993), and that machine learning techniques can identify economically meaningful signals in large panels of stock characteristics (Gu et al., 2020). Regardless of which perspective is taken, research and practice keep on producing predictive models, and the more promising ones are data-intensive and expensive to compute.

Distributed computing provides a viable solution to this problem of computation. Developed by Dean and Ghemawat (2004), MapReduce provides a programming model under which a program defines a map function that operates on independent parts of the input and a reduce function that takes the intermediate results and returns a final output. The programmer need not worry about how to partition the data, how to schedule concurrent executions of the map function, or how to recover from failures of machines running data. This model was introduced to organisations that were not able to build the Google-scale infrastructure through open-source implementations in the Hadoop ecosystem (White, 2015). Chu et al. (2007) demonstrated that many common learning algorithms can be expressed as "summation form" that fits into the model directly, so that the pipelines for stock prediction can be scaled by adding machines rather than re-practicing the mathematics.

The problem is particularly salient for those researchers whose research interests focus on emerging markets. Most studies on Indian stocks, including those that use constituents of the Nifty and Sensex indexes, start with smaller sample sizes because the data preparation and model training on a personal computer are time-consuming, and consequently fewer stocks, shorter histories, and fewer models can be studied (Patel et al., 2015). Distributed processing removes that barrier: Rents or shares a cluster of commodity machines rather than purchasing specialised hardware in a University lab or smaller research group.

This potential, however, is not sufficiently reflected in the majority of the available literature on stock prediction, which presents results for just a few indices or stocks, usually produced on a single machine and without providing much detail regarding the properties of the proposed methods as the stock data increases (Atsalakis & Valavanis, 2009; Henrique et al., 2019). In this work, we tackle this by designing, implementing, and evaluating a MapReduce-based prediction pipeline, as well as by exploring honestly where distributed execution is beneficial and where it is not.

The paper has four goals: (a) to summarize the evidence on the machine learning side of stock market prediction and the distributed processing side of financial analytics; (b) to state the stock prediction as a MapReduce pipeline task, which includes feature engineering, model estimation, and the evaluation of the pipeline; (c) to quantify the run time, scalability, and predictive behaviour of the pipeline in a controlled experiment; and (d) to point out some methodological pitfalls that the researcher has to attend to when scaling the pipeline to real market data. The work is guided by three research questions. Is it possible to estimate a ridge-regularised direction classifier without changing the model using the MapReduce approach? Secondly, what is the scaling of wall-clock time with the number of securities? Third, what is the sensitivity of the performance with respect to the number of input partitions?

The contribution of the paper is 3-fold. It provides a precise definition of stock prediction in a key-value fashion and has a working algorithm to back it up. It presents repeatable timing results of an actual multi-process implementation of MapReduce. It places these results in an open-eyed consideration of the limits in which the experiment is carried out, not just because it is synthetic data, but also because it is a small machine, and so it is not used to show profitable trading, but to show computational behaviour. The remainder of the paper is organised as follows. The related works are summarized in Section 2. Section 3 introduces the MapReduce framework and its connections with other engines. The methodology is described in Section 4. Results are presented in Section 5, discussed in Section 6, limitations and future work discussed in Sections 7 and 8, and conclusions given in Section 9.

2. Literature Review

2.1 Machine learning for stock market prediction

In the early days of computational forecasting, statistical time series models were used, but since the 1990s, a growing number of researchers have shifted their focus to neural networks, support vector machines, and ensemble methods. Kim (2003) used support vector machines for forecasting financial time series and compared them to the back-propagation neural network method and argued that the margin-based approach was competitive, and did not suffer from overfitting. Atsalakis and Valavanis (2009) researched the soft computing methods used for forecasting the stock market and observed that the most widely used methods were neural and neuro-fuzzy models, and there was no agreed framework for evaluating the models. Patel et al. (2015) evaluated two stocks and two indices with four classifiers (artificial neural networks, support vector machines, random forests, and naive Bayes) using Indian data with technical indicators converted to trend-deterministic inputs and indicated that the artificial neural networks, support vector machines, random forests, and naive Bayes were superior on Indian data. The study is especially relevant here since it is a study of Indian markets, and many applied research projects are conducted in India.

Since then, deep learning has garnered a lot of attention. Hochreiter and Schmidhuber (1997) introduced long short-term memory (LSTM) networks to overcome the vanishing gradient problem in sequence learning, and their use in finance followed. Nelson et al. (2017) used LSTM networks to predict the direction of price movements for Brazilian stocks, while Fischer and Krauss (2018) applied LSTM networks to constituents of the S&P 500 over more than two decades and reported that the networks outperformed memory-free classifiers such as random forests and logistic regression, although returns diminished in later years. Dixon et al. (2017) examined deep neural networks for classification-based prediction on futures data. Sezer et al. (2020) reviewed deep learning for financial time series forecasting and concluded that the field is growing rapidly but suffers from inconsistent benchmarking. A broader review by Henrique et al. (2019) catalogued machine learning applications to financial market prediction and likewise emphasised the diversity of data, horizons, and evaluation measures, which makes studies difficult to compare.

Alternative data have also been explored. Bollen et al. (2011) analysed large volumes of public Twitter messages and reported that measures of collective mood carried predictive information for the Dow Jones Industrial Average. Such pipelines made of text multiply the amount of data to be processed, and they further justify the use of distributed approaches. Representative studies and the computational scale for each are summarised in Table 1.

Table 1

Representative Studies on Machine Learning for Stock Market Prediction and Their Computational Setting

Study	Data and market	Method	Main contribution
Kim (2003)	Korean stock price index, daily	Support vector machine; neural network comparison	Showed SVM to be competitive and less prone to overfitting
Patel et al. (2015)	Two Indian stocks and two indices, about ten years	ANN, SVM, random forest, naive Bayes	Trend-deterministic inputs improved all four classifiers
Bollen et al. (2011)	Public Twitter messages and the DJIA, 2008	Mood measures with nonlinear forecasting model	Linked collective mood to later index movements
Nelson et al. (2017)	Brazilian stocks, intraday data	LSTM network	Predicted direction of price moves from technical indicators
Fischer & Krauss (2018)	S&P 500 constituents, 1992 to 2015	LSTM versus random forest, deep network, logistic regression	LSTM outperformed memory-free methods; returns weakened in later years
Gu et al. (2020)	Large panel of US stocks and characteristics	Regularised regression, trees, neural networks	Nonlinear learners improved return forecasts

Note. Entries summarise the cited studies in the author's words; the studies differ in markets, horizons, and evaluation measures and are not directly comparable.

From this body of work, there are two observations to make. Firstly, predictive models are becoming more and more sophisticated and more data-intensive, and therefore, computational cost is a first order concern. Second, there is high variability in the way the evaluations are carried out, and there is a significant likelihood that the accuracy reported could be due to overfitting rather than true ability to predict the behavior. Formally, as illustrated by the work of Bailey et al. (2014), it is possible to artificially boost in-sample performance by trying many different versions of a strategy on the same set of historical data, a situation that is increasingly relevant given the low cost of testing thousands of configuration variants with distributed infrastructure.

2.2 Big data and distributed computing foundations

The technical foundations of large-scale data processing were laid by the Google File System (Ghemawat et al., 2003), which stored very large files across commodity machines with replication, and by MapReduce (Dean & Ghemawat, 2004, 2008), which processed those files in parallel. Hadoop reproduced this architecture openly. Its distributed file system (Shvachko et al., 2010) splits files into blocks replicated across nodes, and its resource manager, YARN (Vavilapalli et al., 2013), separated

cluster resource management from the programming model and allowed other engines to share the same cluster. The practical manual by White (2015) remains a standard reference on these components.

MapReduce was soon compared with parallel databases. Pavlo et al. (2009) found that parallel database systems often delivered faster query execution on structured analytical tasks, while MapReduce offered easier setup, flexible handling of unstructured data, and strong fault tolerance. For stock prediction, where raw inputs include semi-structured logs and text, and where the work involves custom numerical code rather than SQL queries, the flexibility of MapReduce remains attractive. Lin and Dyer (2010) demonstrated that a wide range of data-intensive algorithms can be recast as map and reduce phases, including approaches that use techniques such as in-mapper combining to reduce network traffic.

2.3 MapReduce and machine learning

Chu et al. (2007) bridged distributed processing and learning algorithms by showing that algorithms in the statistical query model can be written as sums over the training data. A mapper gets a portion of the data and returns a partial sum, while a reducer adds up the partial sums. They showed that, for locally weighted linear regression, k-means, logistic regression, naive Bayes, support vector machines, independent component analysis, principal component analysis, Gaussian discriminant analysis, expectation maximisation, and back-propagation neural networks, they could get near-linear speed-ups on multicore machines. The two simplest cases are ordinary least squares and ridge regression: The normal equations involve only the sum of the cross-products of the features and the sum of the products of the features and the targets, also over observations.

Iterative algorithms are more difficult since the intermediate results in plain Hadoop MapReduce are stored on disk between jobs. To overcome these, Zaharia et al. (2012) used resilient distributed datasets that retain the working set of data in memory and reconstruct lost partitions using lineage, and claimed up to twenty times faster response time than Hadoop for certain iterative machine learning and graph applications. Spark eventually evolved into a single engine with libraries for SQL, streaming, and machine learning (Zaharia et al., 2016; Meng et al., 2016), and Flink expanded the scope of stream and batch processing into a single engine (Carbone et al., 2015). Histogram and sketch-based split finding, like in XGBoost (Chen & Guestrin, 2016) (distributed settings), and the natural parallelism of random forests (Breiman, 2001) (each tree is grown on a different bootstrap sample) are the two ways to scale tree-based ensembles.

2.4 Research gap

This literature, therefore, provides evidence that accurate prediction models exist, distributed engines exist, and many learners have mathematics that is appropriate for the MapReduce pattern. The less common treatment, however, is to provide a transparent treatment that brings these elements together for stock prediction, confirming that distribution does not affect the fitted model, establishing the scalability of the treatment, and discussing some of the potential pitfalls of financial time series. This paper aims to supply such a treatment.

3. The MapReduce framework for financial prediction

3.1 The programming model

MapReduce splits the input data into splits, and each split is dealt with by a map task that reads the records and produces intermediate key-value pairs. The framework then bundles together all the values

with a given key and sends each bundle to a reduce task that produces the final output (Dean & Ghemawat, 2008). Only the two functions are written by the programmer; the runtime determines what to do, tries the task again if it fails, and starts multiple copies of slow tasks to minimize stragglers. A combiner is a local reducer that is performed on the mapper's output and can reduce the amount of data traversing the network.

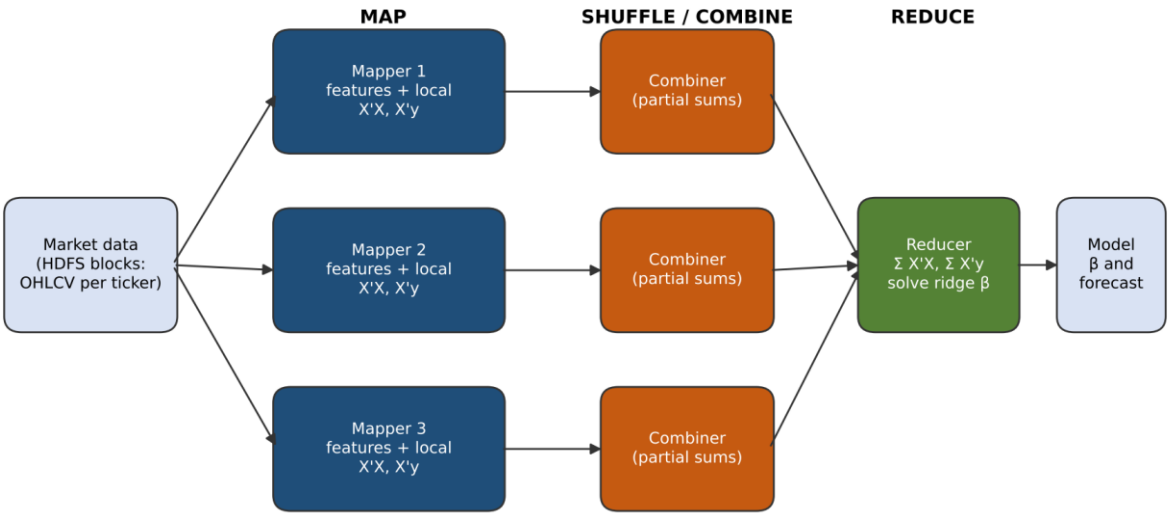
This model is naturally applicable for stock prediction. The daily/ intraday price records are divided by security or divided by time period. In addition to the aforementioned features, returns, moving averages, volatility, and momentum indicators rely solely on the past history of a single security, which means that the mappers can calculate them without considering the other securities. The reduce phase is needed for model estimation, as it needs information from all securities. The key identifies the model parameter block, and the value contains a partial sum of cross-products--as in the summation form.

3.2 System architecture

The architecture used is depicted in figure 1. Historical data is stored in blocks of a distributed file system. For each mapper, the mappers get a subset of tickers for which they generate or read price series, compute the feature matrix for these tickers and create partial sufficient statistics for the training period. A combiner combines partial statistics of tasks running on the same node. A single reducer will then add together all statistics, solve the regularised normal equations, and output the coefficient vector. The shuffle phase is very lightweight because the dimension of the sufficient statistics is the number of features and not the number of observations, so it shuffles little information.

Figure 1

MapReduce Architecture for Stock Market Prediction



Note. Mappers compute features and partial sufficient statistics for their partitions of tickers; combiners merge partial sums locally; a single reducer solves the ridge normal equations. Original figure by the author.

3.3 Hadoop MapReduce and in-memory alternatives

The decision on an engine type is not an easy one. Hadoop MapReduce is stable and well-developed, but its start-up costs and disk costs are per job, and many disk passes are used in algorithms like gradient-based training of neural networks, for which it is slow. In-memory engines lower that cost by storing data within the engine "between iterations," and streaming engines can continuously update models as new ticks arrive. Table 2 shows the three families compared on the most important issues for stock prediction.

Table 2- *Comparison of Batch, In-Memory, and Streaming Engines for Stock Prediction Workloads*

Criterion	Hadoop MapReduce	Apache Spark	Apache Flink
Processing model	Batch; disk between jobs	Batch and micro-batch; in-memory datasets	Stream-first; batch as special case
Iterative learning	Slow; one job per pass	Efficient with cached data	Supported through iterations
Fault tolerance	Task re-execution; replicated blocks	Lineage-based recomputation	Checkpointing of operator state
Typical latency	Minutes or longer	Seconds to minutes	Milliseconds to seconds
Best fit here	Single-pass estimators; nightly batch features	Iterative models; interactive research	Live scoring and continuous updating

Note. Based on Dean and Ghemawat (2008), Zaharia et al. (2012, 2016), and Carbone et al. (2015).

For single-pass estimators such as the ridge model in this study, the penalty of disk-based execution is small because only one pass over the data is needed, and the simple MapReduce pattern suffices. For iterative learners and for low-latency scoring of live data, an in-memory or streaming engine is the better choice, and the same key-value decomposition carries over to them.

4. Methodology

4.1 Research design

This study uses the method of computational experiment design. Because of the isolated environment in which this was done, real market archives could not be retrieved, and more importantly, the correctness of the implementation of the distributed estimator can be checked against a known answer because the data generated is controlled. The experiment is therefore conducted with synthetic price series, and works on three questions that are independent of the data being real: is the distributed estimator the same as the centralised one, how does the execution time change with the number of securities and partitions, and can the pipeline recover a weak planted signal out of sample. It does not claim to cover predictions about the market.

4.2 Data generation

For each security, the simulation is done over 1000 trading days. The daily log returns are assumed to follow a first-order autoregressive process with a coefficient of 0.04, a small positive drift of 0.0003 a day, and innovations from a normal distribution with its scale determined separately for each security and set to be between 1 and 3 per cent. A regime factor of 0.7, 1.0 or 1.8 amplifies the volatility, and is redrawn every 50 days, to model the clustering of calm and turbulent periods. Prices are the exponential of cumulative returns, with an initial level of 100. The weak autoregressive term plants a small amount of genuine predictability so that the pipeline has something to find. Each security uses its own random seed derived from a fixed base, which makes every run reproducible. The first 800 days of each series form the training period and the remaining days form the test period, so that no test observation influences estimation.

4.3 Feature engineering

Nine features are computed for each security and day: an intercept; the current daily log return; the returns lagged one and two days; the ratios of 5-day and 10-day mean return to 20-day return volatility; the ratio of 5-day to 20-day volatility; the difference between fast (10-day) and slow (26-day) exponential moving averages of returns, in the spirit of the moving average convergence divergence indicator; and a relative strength measure based on smoothed average gains and losses over 14 days, centred at zero. The target is the direction of the next day's return, coded 1 for an increase and 0 otherwise. Rolling statistics are computed from cumulative sums so that the cost per security is linear in the length of the series, and all features use only information available at the close of the day being predicted.

4.4 The MapReduce formulation of ridge regression

The classifier is a ridge-regularised linear model for the probability of an up move, estimated by least squares (Hoerl & Kennard, 1970; Hastie et al., 2009). With feature matrix X and target vector y, the ridge solution is the vector β that minimises the sum of squared errors plus λ times the squared norm of β, given by $\beta = (X^TX + \lambda I)^{-1}X^Ty$. Both X^TX and X^Ty are sums over observations. If the training observations are split into partitions p = 1, …, P, then X^TX equals the sum over partitions of $X_p^TX_p$, and X^Ty equals the sum over partitions of $X_p^Ty_p$. This is the summation form identified by Chu et al. (2007). The regularisation parameter was fixed at λ = 0.001. A linear probability model is deliberately simple; it is chosen because its MapReduce form is exact and can be verified to machine precision, which a stochastic learner cannot offer.

Algorithm 1 summarises the procedure. In the map phase, each task receives a list of ticker identifiers; for each ticker it generates the price series, computes the feature matrix, restricts attention to training days, and accumulates X^TX, X^Ty, and the observation count. It also retains the test-period features and targets for later evaluation. The task emits one record keyed by a constant key whose value is the triple of partial statistics. In the reduce phase, the reducer adds the partial matrices and vectors, solves the regularised normal equations by a standard linear solver, and emits β. In a final evaluation step, the coefficient vector is applied to the pooled test observations, and an up move is predicted whenever the fitted value exceeds 0.5.

Algorithm 1. MapReduce ridge estimation of next-day direction. Map(partition of tickers): for each ticker, build features X and target y; keep training rows; update $A \leftarrow A + X^TX$, $b \leftarrow b + X^Ty$, $n \leftarrow n +$ rows; emit (key, (A, b, n)). Combine: add partial (A, b, n) from tasks on the same node. Reduce(key, list of (A, b, n)): sum all A, b, n; solve $(A + \lambda I)\beta = b$; emit β. Evaluate: predict up if $x^T\beta > 0.5$ on held-out test days.

4.5 Extensions to other learners

The ridge formulation is the simplest member of a family of MapReduce-compatible learners. For logistic regression, the gradient and Hessian of the log-likelihood are sums over observations, so each Newton iteration can be run as one MapReduce job in which mappers return partial gradients and Hessians and the reducer updates the coefficients (Chu et al., 2007). The cost is one job per iteration, which is tolerable on Hadoop when only a handful of iterations are needed but becomes a bottleneck for slowly converging problems. Random forests follow a different pattern: each mapper receives a bootstrap sample of securities or time windows and grows a subset of the trees independently, and the reducer collects the trees into one ensemble whose predictions are averaged or voted (Breiman, 2001). Because trees do not communicate during growth, this approach is embarrassingly parallel.

Gradient boosting is sequential in nature, but split finding within each tree can be distributed by having mappers build histograms of gradient statistics for their data shard and having a reducer merge the histograms and choose the best split, which is the approach underlying distributed tree boosting systems (Chen & Guestrin, 2016). Neural networks, including LSTM networks, are usually trained with data parallelism, in which each worker computes gradients on a shard of the training sequences and a reduction step averages them before the weights are updated; here the number of synchronisations, and not the volume of data, tends to dominate the cost. These patterns show that the key-value thinking developed in this paper is not limited to linear models, although the engine best suited to each pattern differs.

4.6 Execution environment and experimental design

The pipeline was implemented in Python with NumPy and SciPy. Map tasks run in separate operating-system processes created with the standard multiprocessing library, which reproduces the structure of a MapReduce job (independent mappers, a collection step, and a single reducer) on one machine rather than on a cluster. The single-worker configuration runs the same map function sequentially in one process. The machine available for the experiments had two processor cores, which sets the ceiling for achievable speed-up at two. Table 3 lists the configuration.

Table 3

Experimental Configuration

Parameter	Setting
Securities (universe size)	500; 1,000; 2,000; 4,000; 8,000 (synthetic)
Trading days per security	1,000 (800 training, 200 test)
Return process	AR(1) with coefficient 0.04; drift 0.0003; regime-switching volatility
Features	9 (intercept, three return lags, two momentum ratios,

	volatility ratio, EMA difference, RSI-type measure)
Model	Ridge-regularised linear classifier, $\lambda = 0.001$
Workers and cores	1 and 2 workers; machine with 2 processor cores
Partitions	1 or 2 (Experiment 1); 2 to 32 (Experiment 2); 16 (Experiment 3)
Repetitions	3 per configuration; fastest run reported
Software	Python, NumPy, SciPy, multiprocessing

Note. All results in Section 5 were produced by the code and settings listed here.

Three experiments were conducted. Experiment 1 varied the number of securities from 500 to 8,000 and compared one worker with two workers, using one input partition per worker. Experiment 2 fixed the universe at 4,000 securities and varied the number of partitions from 2 to 32 with two workers. Experiment 3 verified correctness by estimating the model once as a single partition on a single worker and once as 16 partitions on two workers, and comparing the coefficient vectors. For each timing configuration, the experiment was repeated three times and the fastest run was recorded, a common practice that reduces the influence of unrelated background activity.

4.7 Evaluation measures

Computational performance is measured by wall-clock time, which is split into map time and reduce time, by speed-up, defined as the single-worker time divided by the multi-worker time, and by parallel efficiency, defined as speed-up divided by the number of workers (Amdahl, 1967; Gustafson, 1988). Predictive performance is measured by out-of-sample directional accuracy, the share of test observations for which the predicted direction matches the realised direction, and is compared with a majority-class baseline, that is, the accuracy that would be obtained by always predicting the more frequent direction in the test data. Correctness of the distribution is measured by the maximum absolute difference between coefficients from the centralised and distributed runs.

5. Results

5.1 Correctness of the distributed estimator

The coefficient vectors from the centralised run and from the 16-partition, two-worker run differed by at most 4.4×10^{-13} in absolute value. This difference is of the order of floating-point rounding error and arises because floating-point addition is not exactly associative when partial sums are accumulated in a different order. The result confirms the central property of the summation form: distribution changes how the computation is carried out but not what it produces. Predictive results are therefore identical for every worker and partition configuration at a given universe size, and Table 6 reports them once.

5.2 Run time and speed-up

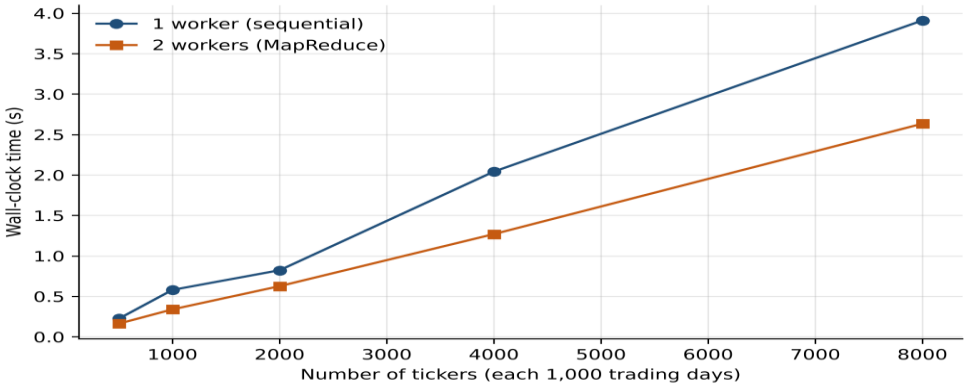
Table 4 reports wall-clock times, speed-up, and efficiency for the five universe sizes. Sequential execution time grew from 0.23 seconds for 500 securities to 3.91 seconds for 8,000 securities, an increase that is roughly proportional to the number of securities, as expected for a computation whose cost per security is constant. In every configuration, two workers were faster than one. Speed-up ranged from 1.31 to 1.71, against a theoretical maximum of 2.00 on two cores, and parallel efficiency ranged from 66 to 85 per cent.

Table 4- *Wall-Clock Time, Speed-Up, and Parallel Efficiency by Universe Size*

Securities	Feature rows (train)	1 worker (s)	2 workers (s)	Speed-up	Efficiency (%)
500	390,500	0.226	0.164	1.37	68.7
1,000	781,000	0.581	0.340	1.71	85.4
2,000	1,562,000	0.824	0.627	1.31	65.7
4,000	3,124,000	2.043	1.271	1.61	80.4
8,000	6,248,000	3.911	2.635	1.48	74.2

Note. Times are the fastest of three repetitions. Speed-up is the one-worker time divided by the two-worker time; efficiency is speed-up divided by two. The reduce phase took under 0.001 s in every run.

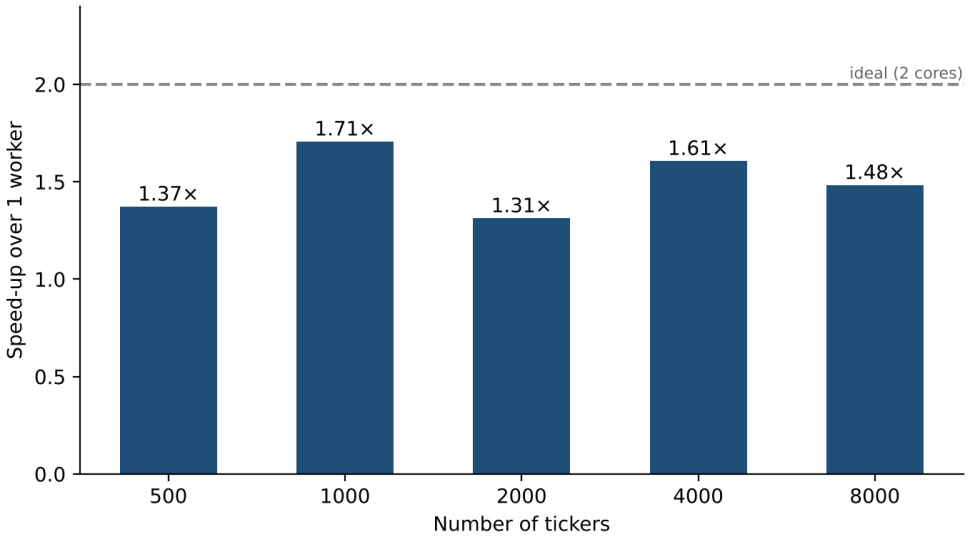
Figure 2- *Wall-Clock Time by Number of Securities for One and Two Workers*



Note. Data from the benchmark reported in Table 4; fastest of three repetitions.

The reduce phase was negligible. Solving the nine-by-nine regularised system and adding the partial statistics took well under one millisecond in every run, so that virtually all elapsed time was spent in the map phase. This is the pattern that the summation form is designed to produce, because the amount of data reaching the reducer does not depend on the number of observations. Figure 3 shows the speed-up achieved at each universe size.

Figure 3- *Speed-Up of Two Workers Over One Worker by Universe Size*



Note. The dashed line marks the theoretical maximum on a two-core machine.

Speed-up did not rise smoothly with data size. The smallest universe achieved 1.37 times, the 1,000-security universe 1.71 times, the 2,000-security universe only 1.31 times, and the 4,000- and 8,000-security universes 1.61 and 1.48 times. Several factors plausibly contribute: the cost of starting worker processes, which is fixed and weighs more heavily on small jobs; the cost of returning test-period arrays from workers to the parent process, which grows with the universe; and ordinary variation in how the operating system schedules processes on a small machine. Because only the fastest of three repetitions was retained and only two cores were available, the pattern should be read as indicative rather than as a precise scaling law.

5.3 Sensitivity to the number of partitions

Table 5 shows the effect of the number of input partitions on a universe of 4,000 securities processed by two workers. Time was 1.27 seconds with two partitions and between 1.13 and 1.25 seconds with four or more partitions. More partitions allow the scheduler to balance work more evenly across workers, since securities differ little in cost but chunks do not always finish at the same moment, and the gain appears to level off beyond about eight partitions. With larger and more variable tasks, such as real tick data of unequal lengths, the benefit of finer partitioning would be expected to be greater, but the experiment cannot demonstrate this.

Table 5- *Effect of the Number of Input Partitions on Run Time (4,000 Securities, Two Workers)*

Partitions	Wall-clock time (s)
2	1.272
4	1.254
8	1.134
16	1.182
32	1.132

Note. Fastest of three repetitions per setting.

Figure 4- *Wall-Clock Time by Number of Input Partitions*



Note. Data from Table 5; 4,000 securities and two workers.

5.4 Predictive performance

Table 6 reports out-of-sample directional accuracy. The pipeline achieved between 51.26 and 51.41 per cent accuracy, compared with a majority-class baseline of between 50.69 and 50.73 per cent. The margin of about 0.6 percentage points is small in absolute terms but consistent across universe sizes, and it is estimated on test sets that range from 99,000 to 1,584,000 observations. With samples of this size, the sampling error of an accuracy estimate is far smaller than the observed margin, so the margin is statistically distinguishable from the baseline. The modest size is exactly what might be predicted: The autoregressive coefficient for the plant is 0.04, and with daily noise much larger than the drift, the model can only pick the direction of the next day's return a little more often than by chance, even if it is perfectly specified.

Table 6- *Out-of-Sample Directional Accuracy Compared With the Majority-Class Baseline*

Securities	Test observations	Accuracy (%)	Baseline (%)	Margin (pp)
500	99,000	51.28	50.69	0.59
1,000	198,000	51.26	50.72	0.54
2,000	396,000	51.35	50.73	0.62
4,000	792,000	51.38	50.73	0.65
8,000	1,584,000	51.41	50.73	0.68

Note. Results are identical for one and two workers. The data are synthetic with a planted autoregressive coefficient of 0.04; the figures do not indicate predictability of real markets.

The accuracy numbers should not be taken as an indication that real stock markets can be forecast at this level. They show only that the pipeline recovers a weak, known signal from noisy data, and that distributing the computation does not impair that recovery. Accuracy rises slightly with the size of the universe, from 51.28 per cent at 500 securities to 51.41 per cent at 8,000, which is consistent with the general principle that more training data reduce estimation error even for a model with nine parameters.

6. Discussion

6.1 What the results show

The three conclusions that can be drawn from the experiments are: First, the MapReduce formulation of ridge regression is exact in the sense that matters for scientific use: the distributed coefficients match the centralised ones to rounding error. A researcher can therefore adopt distributed execution without worrying that parallelism has silently changed the model. Second, it exhibits very useful, but sub-linear, speed-up on a two-core machine, the mapping phase accounting for nearly all of the cost while the reduction phase accounts for almost none. Third, a weak signal that is seen by a single-machine analysis is also seen by the distributed one.

The speed-ups seen here are quite small by absolute value, but the hardware is modest. Because of the two-core ceiling, the speed-up that can be achieved is two; the observed efficiencies are between 66 and 85 per cent, which is consistent with Amdahl's (1967) law, considering that a small serial fraction (process creation, result collection, and reduce step) exists. Gustafson (1988) provides an alternative viewpoint: An increase in the number of processors that are used to solve a larger problem does not necessarily mean that an equal-sized problem will be solved faster if more processors are used; the serial fraction decreases, and scaled speed-up increases. This is a common application of a cluster in finance to deal with a higher number of securities, a longer history, or more frequent data in the same timeframe.

6.2 Implications for stock market prediction at scale

There are some factors in real deployments that would work in favour of distribution in this experiment. Real data would be stored in a distributed file system and would be accessed by the mapper on the nodes where blocks are stored, thus avoiding the data transfer of raw data over the network. The map phase would take even longer if feature engineering on tick data has to go before daily closing data. If we were using more sophisticated learners like random forests, gradient boosting, or LSTM networks, the processing time per observation would increase by orders of magnitude, and so would the benefit of the parallelism (Breiman, 2001; Chen & Guestrin, 2016; Fischer & Krauss, 2018).

The decomposition is also generalised in the sense used in this case: All training statistics, such as logistic regression, naive Bayes, and linear discriminant analysis, have the same structure as the example above (Chu et al., 2007). Another, equally simple, ensemble design is one based on a bootstrap sample in which each mapper then builds a tree for its own portion of the forest, and the reducer simply joins the trees. Gradient-based learners (such as neural networks) need to repeatedly add up gradients, making them better suited to Spark-style engines—where data is cached—than classic Hadoop jobs (Zaharia et al., 2012; Meng et al., 2016).

6.3 Engineering considerations

There are three engineering options to highlight. The first one is partitioning. When partitioning by security, the natural way to do it is to use a partitioning of the data, whereas time-based partitioning needs to be done in such a way that the data in each window is not empty, but rather contains overlapping rows from the same partition, with the idea being that the rolling indicators at the partitions will be calculated correctly. The second is to use combiners and compact sufficient statistics, which helps to reduce shuffle traffic. The third one is fault tolerance. The MapReduce model has a property that is important for long-running nightly retraining jobs, namely that map tasks are deterministic

functions of their input, meaning that if a task fails, it can be rerun without affecting the output. (Dean & Ghemawat, 2008).

Another advantage that isn't easily noticed is reproducibility. Each synthetic series in this study is derived from a given seed; therefore, any run can be reproduced exactly. This is equivalent to versioning the input snapshot, the feature code, and the configuration in real-life projects, where one could later be able to regenerate a reported result on demand.

6.4 Guidelines for practitioners

The following 4 rules can be inferred for researchers who are planning a distributed prediction project: First, start with one machine reference implementation on a small subset of the data, and use it as an oracle; as was done here, any discrepancies in the distribution are detected as soon as they arise. Second, never scale before measuring: If it's the reading of the data, scaling may be necessary; if it's the computation of the features, scaling may not be necessary; if it's the fitting of the model, scaling may not be necessary. Thirdly, create the partitioning scheme based on the dependency relationship of the features, so each security's history will be contained within one task and will be overlapped if windows overlap the boundary. Fourth, design the evaluation protocol prior to building the pipeline: set the time split, remember all the configurations tested, and report the results along with the number of trials, without obscuring the increase in throughput with an increase in the number of spurious patterns. While modest, these guidelines come from patterns observed in the literature on large-scale data analysis and financial machine learning literature (Bailey et al., 2014; Lin & Dyer, 2010; Pavlo et al., 2009).

7. Limitations and Methodological Hazards

There are some limitations to drawing conclusions. The data are synthetic and have been generated from a simple autoregressive process and do not reflect real returns, which are characterized by heavy tails, structural breaks, cross-sectional correlation, and changing relationships. The hardware was not actually a multi-node cluster, but a two-core machine, meaning that network costs, disk throughput, and data locality—all key factors that affect real Hadoop performance—were not factored in. The model is linear and thus can not accommodate nonlinear patterns that more complex learners can use. No trading strategy was tested; no slippage or market impact was modelled.

Broadly speaking, researchers using these pipelines with real data encounter dangers that distributed computing can't eliminate, if anything, it can further intensify them. Look-ahead bias is a bias that arises from using information not available at the prediction time; it is avoided in the construction of the feature code in this study but can easily be introduced when joining data from different sources. Survivorship bias can occur when securities that have been delisted are not included in the universe. Backtest overfitting happens when you've got a lot of different models tested on the same sample; only the best model is reported; and cheap parallel computation gives you the temptation to try a lot more (Bailey et al., 2014). Strict time-ordered validation, out-of-sample holdouts which are applied once only, multiple testing correction, and reporting the number of configurations tested are remedies used.

Lastly, a statistically significant directional accuracy difference does not necessarily mean a "significant" economic difference. Margins of 1/2 percentage point can be lost after costs are removed, and the literature focuses on gains in specific segments and over specific time periods (Gu et al., 2020; Fischer & Krauss, 2018).

8. Future Research Directions

There are a couple of obvious extensions to follow. The first is real cluster replication, and the market data from the exchange archive is stored in the distributed file system to get the full costs of data movement and job scheduling. The second is the extension to richer learners: a MapReduce random forest with each mapper growing random forests on a random sample of securities, and a distributed gradient-boosting model against a ridge baseline reported here. The third is that the models are updated as the ticks are received, and forecasts are then generated, which relies on a streaming architecture and engines that can handle stateful stream processing (Carbone et al., 2015). The fourth relates to incorporating other types of data, such as financial news and social media text, which are very rich and most benefit from distributed processing (Bollen et al., 2011). The fifth is the design of protocols for distributed financial machine learning, which include walk-forward validation, multiple-testing corrections, and cost-aware performance measures. Indian markets in particular would greatly benefit from studies based on long historical data of listed securities or index constituents on national exchanges to determine the transferability of results from other markets (Patel et al., 2015).

9. Conclusion

This paper explored the possibility of scalability of stock market prediction using the MapReduce programming model without the loss of accuracy in the models used. It analyzed the existing literature on distributed data processing as well as on machine learning approaches to market forecasting, transformed the stock direction prediction problem into a MapReduce computation using the summation form of ridge regression, and tested the pipeline on up to 8K synthetic equity series as a benchmark.

The results are unambiguous, given the constraints of the experiment. The distributed estimator was able to replicate the centralised estimator to within rounding error, demonstrating that the approach is exact. The approach scales because as the number of securities increases so does wall-clock time, and the approach provides between 1.31- and 1.71-times speed-up across the board, with nearly all of this speed-up occurring during the map phase. The out-of-sample directional accuracy was slightly better than a majority class baseline, and the difference was fairly consistent and small, indicating that the approach retains predictive content since it is a small planted signal. Run time was slightly affected by the number of partitions, but improved with more than eight partitions.

The key-value decomposition of prediction problems is a promising scaling approach as long as the learning algorithm is parallelizable over independent members, has a summational form, and the engine can be selected according to whether the algorithm requires a single pass, many passes, or continuous updates. There is also a methodological message: by being scalable, it is easier to test more models, so it is more rigorous for the discipline required to avoid overfitting and bias. Further research into real exchange data and multi-node clusters using more sophisticated learners and validated thoroughly will establish the extent to which this distributed stock prediction system can be extended beyond the setting of this report.

REFERENCES:

1. Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference* (pp. 483–485). ACM. https://doi.org/10.1145/1465482.1465560

2. Atsalakis, G. S., & Valavanis, K. P. (2009). Surveying stock market forecasting techniques – Part II: Soft computing methods. *Expert Systems with Applications, 36*(3), 5932–5941. https://doi.org/10.1016/j.eswa.2008.07.006
3. Bailey, D. H., Borwein, J. M., López de Prado, M., & Zhu, Q. J. (2014). Pseudo-mathematics and financial charlatanism: The effects of backtest overfitting on out-of-sample performance. *Notices of the American Mathematical Society, 61*(5), 458–471. https://doi.org/10.1090/noti1105
4. Bollen, J., Mao, H., & Zeng, X. (2011). Twitter mood predicts the stock market. *Journal of Computational Science, 2*(1), 1–8. https://doi.org/10.1016/j.jocs.2010.12.007
5. Breiman, L. (2001). Random forests. *Machine Learning, 45*(1), 5–32. https://doi.org/10.1023/A:1010933404324
6. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. (2015). Apache Flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin, 38*(4), 28–38.
7. Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 785–794). ACM. https://doi.org/10.1145/2939672.2939785
8. Chu, C.-T., Kim, S. K., Lin, Y.-A., Yu, Y., Bradski, G., Ng, A. Y., & Olukotun, K. (2007). Map-reduce for machine learning on multicore. In *Advances in Neural Information Processing Systems 19* (pp. 281–288). MIT Press.
9. Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (pp. 137–150). USENIX Association.
10. Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM, 51*(1), 107–113. https://doi.org/10.1145/1327452.1327492
11. Dixon, M. F., Klabjan, D., & Bang, J. H. (2017). Classification-based financial markets prediction using deep neural networks. *Algorithmic Finance, 6*(3–4), 67–77. https://doi.org/10.3233/AF-170176
12. Fama, E. F. (1970). Efficient capital markets: A review of theory and empirical work. *The Journal of Finance, 25*(2), 383–417. https://doi.org/10.2307/2325486
13. Fischer, T., & Krauss, C. (2018). Deep learning with long short-term memory networks for financial market predictions. *European Journal of Operational Research, 270*(2), 654–669. https://doi.org/10.1016/j.ejor.2017.11.054
14. Gandomi, A., & Haider, M. (2015). Beyond the hype: Big data concepts, methods, and analytics. *International Journal of Information Management, 35*(2), 137–144. https://doi.org/10.1016/j.ijinfomgt.2014.10.007
15. Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (pp. 29–43). ACM. https://doi.org/10.1145/945445.945450
16. Gu, S., Kelly, B., & Xiu, D. (2020). Empirical asset pricing via machine learning. *The Review of Financial Studies, 33*(5), 2223–2273. https://doi.org/10.1093/rfs/hhaa009
17. Gustafson, J. L. (1988). Reevaluating Amdahl's law. *Communications of the ACM, 31*(5), 532–533. https://doi.org/10.1145/42411.42415

18. Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning: Data mining, inference, and prediction* (2nd ed.). Springer. https://doi.org/10.1007/978-0-387-84858-7
19. Henrique, B. M., Sobreiro, V. A., & Kimura, H. (2019). Literature review: Machine learning techniques applied to financial market prediction. *Expert Systems with Applications, 124*, 226–251. https://doi.org/10.1016/j.eswa.2019.01.012
20. Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation, 9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735
21. Hoerl, A. E., & Kennard, R. W. (1970). Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics, 12*(1), 55–67. https://doi.org/10.1080/00401706.1970.10488634
22. Jegadeesh, N., & Titman, S. (1993). Returns to buying winners and selling losers: Implications for stock market efficiency. *The Journal of Finance, 48*(1), 65–91. https://doi.org/10.1111/j.1540-6261.1993.tb04702.x
23. Kim, K.-J. (2003). Financial time series forecasting using support vector machines. *Neurocomputing, 55*(1–2), 307–319. https://doi.org/10.1016/S0925-2312(03)00372-2
24. Lin, J., & Dyer, C. (2010). *Data-intensive text processing with MapReduce*. Morgan & Claypool. https://doi.org/10.2200/S00274ED1V01Y201006HLT007
25. Lo, A. W. (2004). The adaptive markets hypothesis: Market efficiency from an evolutionary perspective. *The Journal of Portfolio Management, 30*(5), 15–29. https://doi.org/10.3905/jpm.2004.442611
26. Malkiel, B. G. (2003). The efficient market hypothesis and its critics. *Journal of Economic Perspectives, 17*(1), 59–82. https://doi.org/10.1257/089533003321164958
27. Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D. B., Amde, M., Owen, S., Xin, D., Xin, R., Franklin, M. J., Zadeh, R., Zaharia, M., & Talwalkar, A. (2016). MLlib: Machine learning in Apache Spark. *Journal of Machine Learning Research, 17*(34), 1–7.
28. Nelson, D. M. Q., Pereira, A. C. M., & de Oliveira, R. A. (2017). Stock market's price movement prediction with LSTM neural networks. In *2017 International Joint Conference on Neural Networks* (pp. 1419–1426). IEEE. https://doi.org/10.1109/IJCNN.2017.7966019
29. Patel, J., Shah, S., Thakkar, P., & Kotecha, K. (2015). Predicting stock and stock price index movement using trend deterministic data preparation and machine learning techniques. *Expert Systems with Applications, 42*(1), 259–268. https://doi.org/10.1016/j.eswa.2014.07.040
30. Pavlo, A., Paulson, E., Rasin, A., Abadi, D. J., DeWitt, D. J., Madden, S., & Stonebraker, M. (2009). A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (pp. 165–178). ACM. https://doi.org/10.1145/1559845.1559865
31. Sezer, O. B., Gudelek, M. U., & Ozbayoglu, A. M. (2020). Financial time series forecasting with deep learning: A systematic literature review: 2005–2019. *Applied Soft Computing, 90*, Article 106181. https://doi.org/10.1016/j.asoc.2020.106181
32. Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010). The Hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies* (pp. 1–10). IEEE. https://doi.org/10.1109/MSST.2010.5496972
33. Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., &

Baldeschwieler, E. (2013). Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (Article 5). ACM. https://doi.org/10.1145/2523616.2523633

34. White, T. (2015). *Hadoop: The definitive guide* (4th ed.). O'Reilly Media.
35. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., & Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation* (pp. 15–28). USENIX Association.
36. Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., & Stoica, I. (2016). Apache Spark: A unified engine for big data processing. *Communications of the ACM, 59*(11), 56–65. https://doi.org/10.1145/2934664