García-Gil *et al. Big Data Analytics* (2017) 2:1
DOI 10.1186/s41044-016-0020-2

Big Data Analytics

REVIEW Open Access

A comparison on scalability for batch big data processing on Apache Spark and Apache Flink

Diego García-Gil[1*], Sergio Ramírez-Gallego[1], Salvador García[1,2] and Francisco Herrera[1,2]

*Correspondence:
djgarcia@decsai.ugr.es
[1]Department of Computer Science and Artificial Intelligence, CITIC-UGR (Research Center on Information and Communications Technology), University of Granada, Calle Periodista Daniel Saucedo Aranda, 18071 Granada, Spain
Full list of author information is available at the end of the article

Abstract
The large amounts of data have created a need for new frameworks for processing. The MapReduce model is a framework for processing and generating large-scale datasets with parallel and distributed algorithms. Apache Spark is a fast and general engine for large-scale data processing based on the MapReduce model. The main feature of Spark is the in-memory computation. Recently a novel framework called Apache Flink has emerged, focused on distributed stream and batch data processing. In this paper we perform a comparative study on the scalability of these two frameworks using the corresponding Machine Learning libraries for batch data processing. Additionally we analyze the performance of the two Machine Learning libraries that Spark currently has, MLlib and ML. For the experiments, the same algorithms and the same dataset are being used. Experimental results show that Spark MLlib has better perfomance and overall lower runtimes than Flink.

Keywords: Big data, Spark, Flink, MapReduce, Machine learning

Introduction

With the always growing amount of data, the need for frameworks to store and process this data is increasing. In 2014 IDC predicted that by 2020, the digital universe will be 10 times as big as it was in 2013, totaling an astonishing 44 zettabytes [1]. Big Data is not only a huge amount of data, but a new paradigm and set of technologies that can store and process this data. In this context, a set of new frameworks focused on storing and processing huge volumes of data have emerged.

MapReduce [2] and its open-source version Apache Hadoop [3, 4] were the first distributed programming techniques to face Big Data storing and processing. Since then, several distributed tools have emerged as consequence of the spread of Big Data. Apache Spark [5, 6] is one of these new frameworks, designed as a fast and general engine for large-scale data processing based on in-memory computation. Apache Flink [7] is a novel and recent framework for distributed stream and batch data processing that is getting a lot of attention because of its streaming orientation.

Most of these frameworks have their own Machine Learning (ML) library for Big Data processing. The first one was Mahout [8] (as part of Apache Hadoop [3]), followed by MLlib [9] which is part of Spark project [5]. Flink also has its own ML library that, while it

is not as powerful or complete as Spark's MLlib, it is starting to include some classic ML algorithms.

In this paper, we present a comparative study between the ML libraries of these two powerful and promising frameworks, Apache Spark and Apache Flink. Our main goal is to show the differences and similarities in performance between these two frameworks for batch data processing. For the experiments, we use two algorithms present in both ML libraries, Support Vector Machines (SVM) and Linear Regression (LR), on the same dataset. Additionally, we have implemented a feature selection algorithm to compare the different functioning of each framework.

Background

In this section, we describe the MapReduce framework and two extensions of it, Apache Spark and Apache Flink.

MapReduce

MapReduce is a framework that has supposed a revolution since Google introduced it in 2003 [2]. This framework processes and generates large datasets in a parallel and distributed way. It is based on the Divide and Conquer algorithm. Briefly explained, the framework splits the input data and distributes it across the cluster, then the same operation is performed on each split in parallel. Finally, the results are aggregated and returned to the master node. The framework manages all the task scheduling, monitoring and re-executing in case of failed tasks.

The MapReduce model is composed of two phases: Map and Reduce. Before the Map operation, the master node splits the dataset and distributes it across the computing nodes. Then the Map operation is performed to every key-value pair to the node local data. This produces a set of intermediate key-value pairs. Once all Map tasks have finished, the results are grouped by key and redistributed so that all pairs belonging to one key are in the same node. Finally, they are processed in parallel.

The Map function takes data structured in <key, value> pairs as input and outputs a set of intermediate <key, value> pairs:

$$Map(< key1, value1 >) \rightarrow list(< key2, value2 >) \tag{1}$$

The result is grouped by key and distributed across the cluster. The Reduce phase applies a function to each list value, producing a single output value:

$$Reduce(< key2, list(value2) >) \rightarrow < key2, value3 > \tag{2}$$

Apache Hadoop [3, 4] has become the most popular open-source framework for large-scale data storing and processing based on the MapReduce model. Despite its popularity and performance, Hadoop presents some important limitations [10]:

- Intensive disk-usage
- Low inter-communication capability
- Inadequacy for in-memory computation
- Poor perfomance for online and iterative computing

Apache Spark

Apache Spark [5, 6] is a framework aimed at performing fast distributed computing on Big Data by using in-memory primitives. This platform allows user programs to load data into memory and query it repeatedly, making it a well suited tool for online and iterative processing (especially for ML algorithms). It was developed motivated by the limitations in the MapReduce/Hadoop paradigm [4, 10], which forces to follow a linear dataflow that make an intensive disk-usage.

Spark is based on distributed data structures called Resilient Distributed Datasets (RDDs) [11]. Operations on RDDs automatically place tasks into partitions, maintaining the locality of persisted data. Beyond this, RDDs are an immutable and versatile tool that let programmers persist intermediate results into memory or disk for reusability purposes, and customize the partitioning to optimize data placement. RDDs are also fault-tolerant by nature. The lazy operations performed on each RDD are tracked using a "lineage", so that each RDD can be reconstructed at any moment in case of data loss.

In addition to Spark Core, some additional projects have been developed to complement the functionality provided by the core. All these sub-projects (built on top of the core) are described in the following:

- Spark SQL: introduces DataFrames, which is a new data structure for structured (and semi-structured) data. DataFrames offers us the possibility of introducing SQL queries in the Spark programs. It provides SQL language support, with command-line interfaces and ODBC/JDBC controllers.
- Spark Streaming: allows us to use the Spark's API in streaming environments by using mini-batches of data which are quickly processed. This design enables the same set of batch code (formed by RDD transformations) to be used in streaming analytics with almost no change. Spark Streaming can work with several data sources like HDFS, Flume or Kafka.
- Machine Learning library (MLlib) [12]: is formed by common learning algorithms and statistic utilities. Among its main functionalities includes: classification, regression, clustering, collaborative filtering, optimization, and dimensionality reduction. This library has been especially designed to simplify ML pipelines in large-scale environments. In the latest versions of Spark, the MLlib library has been divided into two packages, MLlib, build on top of RDDs, and ML, build on top of DataFrames for constructing pipelines.
- Spark GraphX: is the graph processing system in Spark. Thanks to this engine, users can view, transform and join interchangeably both graphs and collections. It also allows expressing the graph computation using the Pregel abstraction [13].

Apache Flink

Apache Flink [7] is a recent open-source framework for distributed stream and batch data processing. It is focused on working with lots of data with very low data latency and high fault tolerance on distributed systems. Flink's core feature is its ability to process data streams in real time.

Apache Flink offers a high fault tolerance mechanism to consistently recover the state of data streaming applications. This mechanism is generating consistent snapshots of the

distributed data stream and operator state. In case of failure, the system can fall back to these snapshots.

It also supports both stream and batch data processing with his two main APIs: DataStream and DataSet. These APIs are built on top of the underlying stream processing engine.

Apache Flink has four big libraries built on those main APIs:

- Gelly: is the graph processing system in Flink. It contains methods and utilities for the development of graph analysis applications.
- FlinkML: this library aims to provide a set of scalable ML algorithms and an intuitive API. It contains algorithms for supervised learning, unsupervised learning, data preprocessing, recommendation and other utilities.
- Table API and SQL: is a SQL-like expression language for relational stream and batch processing that can be embedded in Flink's data APIs.
- FlinkCEP: is the complex event processing library. It allows to detect complex events patterns in streams.

Although Flink is a new platform, it is constantly evolving with new additions and it has already been adopted as a real-time process framework in many big companies, such as: ResearchData, Bouygues Telecom, Zalando and Otto Group.

Spark vs. Flink: main differences and similarities

In this section, we present the main differences and similarities in the engines of both platforms in order to explain which are the best scenarios for one platform or the other. Afterwards, we highlight the main differences between three ML algorithms implemented in both platforms: Distributed Information Theoretic Feature Selection (DITFS), SVM and LR.

Comparison between engines

The first remarkable difference between both engines lies in the way each tool ingests streams of data. Whereas Flink is a native streaming processing framework that can work on batch data, Spark was originally designed to work with static data through its RDDs. Spark uses micro-batching to deal with streams. This technique divides incoming data and processess small parts one at a time. The main advantage of this scheme is that the structure chosen by Spark, called DStream, is a simple queue of RDDs. This approach allows users to switch between streaming and batch as both have the same API. However, micro-batching may not perform quick enough in systems that requires very low latency. Nevertheless, Flink fits perfectly well in those systems as it natively uses streams for all type of workloads.

Unlike Hadoop MapReduce, Spark and Flink have support for data re-utilization and iterations. Spark keeps data in memory across iterations through an explicit caching. However, Spark plans its executions as acyclic graph plans, which implies that it needs to schedule and run the same set of instructions in each iteration. In contrast, Flink implements a thoroughly iterative processing in its engine based on cyclic data flows (one iteration, one schedule). Additionally, it offers delta iterations to leverage operations that only changes part of data.

Till the advent of Tungsten optimization project, Spark mainly used the JVM's heap memory to manage all its memory [14]. Although it is straightforward solution, it may suffers from overflow memory problems and garbage collect pauses. Thanks to this novel project, these problems started to disappear. Through DataFrames, Spark is now able to to manage its own memory stack and to exploit the memory hierarchy available in modern computers (L1 and L2 CPU caches). Flink's designers, however, had these facts into consideration from the initial point [15]. The Flink team thus proposed to maintain a self-controlled memory stack, with its own type extraction and serialization strategy in binary format. The advantage derived from these tunes are: less memory errors, less garbage collection pressure, and a better space data representation, among others.

About optimization, both frameworks have mechanisms that analyze the code submitted by the user and yields the best pipeline code for a given execution graph. Spark through the DataFrames API and Flink as first citizen. For instance, in Flink a join operation can be planned as a complete shuffling of two sets, or as a broadcast of the smallest one. Spark also offers a manual optimization, which allows the user to control partitioning and memory caching.

The rest of matters about easiness of coding and tuning, variety of operators, etc. have been omitted from this comparison as these factors do not affect the performance of executions.

A thorough comparison between algorithm implementations

Here, we present the implementation details of three ML algorithms implemented in Spark and Flink. Firstly, a feature selection algorithm implemented by us in both platforms is reviewed. Secondly, the native implementation of SVM in both platforms is analyzed. And lastly, the same process is applied for the native implementation of LR.

Distributed information theoretic feature selection

For comparison purposes, we have implemented in both platforms a feature selection framework based on information theory. This framework was proposed by Brown et al. [16] in order to ensemble multiple information theoretic criteria into a single greedy algorithm. Through some independence assumptions, it allows to transform many criteria as linear combinations of Shannon entropy terms: mutual information (MI) and conditional mutual information (CMI). Some relevant algorithms like minimum Redundancy Maximum Relevance or Information Gain, among others, are included in the framework. The main objective of the algorithm is to assess features based on a simple score, and to select those more relevant according to a ranking. The generic framework proposed by Brown et al. [16] to score features can be formulated as:

$$J = I(X_i; Y) - \beta \sum_{X_j \in S} I\left(X_j; X_i\right) + \gamma \sum_{X_j \in S} I\left(X_j; X_i | Y\right), \tag{3}$$

where the first term represents the relevance (MI) between the candidate input features X_i and the class Y, the second one the redundancy (MI) between the features already selected (in the set S) and the candidate ones, and the third one the conditional redundancy (CMI) between both sets and the class. γ represents a weight factor for CMI and β the same for MI.

Brown's version was re-designed for a better performance in distributed environments. The main changes accomplished by us are described below:

- Column-wise transformation: most of feature selection methods performs computations by columns. It implies that a previous transformation of data to a columnar format may improve the performance of further computations, for example, when computing relevance or redundancy. Accordingly, the first step in our program is aimed at transforming the original set into columns where each new instance contains the values for each feature and partition in the original set.
- Persistence of important information: some pre-computed data like the transformed input or the initial relevances are cached in memory in order to avoid re-computing them in next phases. As this information is computed once at the start, its persistence can speed up significantly the performance of the algorithm.
- Broadcast of variables: in order to avoid moving transformed data in each iteration, we persist this set and only broadcast those columns (feature) involved in the current iteration. For example, in the first iteration the class feature is broadcasted to compute the initial relevance values in each partition.

In the Flink implementation a bulk iteration process has been used to cope with the greedy process. In the Spark version, the typical iterative process with caching and repeated tasks has been implemented. Flink code can be found in the following GitHub repository: https://github.com/sramirez/flink-infotheoretic-feature-selection. The Spark code was gathered into a package and uploaded to the Spark's third-party package repository: https://spark-packages.org/package/sramirez/spark-infotheoretic-feature-selection.

Linear support vector machines

Both Spark and Flink implements SVMs classifiers using a linear optimizer. Briefly, the minimization problem to be solved is the following:

$$\min_{\mathbf{w}\in\mathbb{R}^d} \frac{\lambda}{2}\|\mathbf{w}\|^2 + \frac{1}{n}\sum_{i=1}^{n} l_i\left(\mathbf{w}^T\mathbf{x}_i\right) \tag{4}$$

where $\mathbf{w}$ is the weight vector, $\mathbf{x}_i \in \mathbb{R}^d$ the data instances, λ the regularization constant, and l_i the convex loss functions. For both versions, the default regularizer is l_2-norm and the loss function is the hinge-loss: $l_i = \max\left(0, 1 - y_i\mathbf{w}^T\mathbf{x}_i\right)$

The Communication-efficient distributed dual Coordinate Ascent algorithm (CoCoA) [17] and the stochastic dual coordinate ascent (SDCA) algorithms are used in Flink to solve the previously defined minimization problem. CoCoA consists of several iterations of SDCA on each partition, and a final phase of aggregation of partial results. The result is a final gradient state, which is replicated across all nodes and used in further steps.

In Spark a distributed Stochastic Gradient Descent[1] (SGD) solution is adopted [12]. In SGD a sample of data (called mini batches) are used to compute subgradients in each phase. Only the partial results from each worker are sent across the network in order to update the global gradient.

Linear regression

Linear least squares is another simple linear method implemented in Spark. Despite it was designed for regression, its output can be adapted for binary classification problems. Linear least squares follows the same minimization formula described for SVMs (see Eq. 4) and the same optimization method (based on SGD), however, it uses squared loss (described below) and no regularization method: $l_i = \frac{1}{2}\left(\mathbf{w}^T\mathbf{x}_i - y_i\right)^2$

The Flink version for this algorithm is quite similar to the one created by Spark's developers. It uses SGD to approximate the gradient solutions. However, Flink only offers squared loss whereas Spark offers many alternatives, like hinge or logistic loss.

Experimental results

This section describes the experiments carried out to show the performance of Spark and Flink using three ML algorithms over the same huge dataset. We carried out the comparative study using SVM, LR and DITFS algorithm.

The dataset used for the experiments is the ECBDL14 dataset. This dataset was used at the ML competition of the Evolutionary Computation for Big Data and Big Learning held on July 14, 2014, under the international conference GECCO-2014. It consists of 631 characteristics (including both numerical and categorical attributes) and 32 million instances. It is a binary classification problem where the class distribution is highly imbalanced: 2 % of positive instances. For this problem, two pre-processing algorithms were applied. First, the Random OverSampling (ROS) algorithm used in [18] was applied in order to replicate the minority class instances from the original dataset until the number of instances for both classes was equalized, summing a total of 65 millions instances. Finally, for DITFS algorithm, the dataset has been discretized using the Minimum Description Length Principle (MDLP) discretizer [19].

The original dataset has been sampled randomly using five differents rates in order to measure the scalability performance of both frameworks: 10, 30, 50, 75 and 100 % of the pre-processed dataset is used. Due to a current Flink limitation, we have employed a subset of 150 features of each ECBDL14 dataset sample for the SVM learning algorithm.

Table 1 gives a brief summary of these datasets. For each one, the number of examples (Instances), the total number of features (Feats.), the total number of values (Total), and the number of classes (CL) are shown.

We have established 100 iterations, a step size of 0.01 and a regularization parameter of 0.01 for the SVM. For the LR, 100 iterations and a step size of 0.00001 are used. Finally, for DITFS 10 features are selected using minimum Redundancy Maximum Relevance algorithm [20].

Table 1 Summary description for ECBDL14 dataset

Dataset	Instances	Feats.	Total	CL
ECBDL14-10	6 500 391	631	4 101 746 721	2
ECBDL14-30	19 501 174	631	12 305 240 794	2
ECBDL14-50	32 501 957	631	20 508 734 867	2
ECBDL14-75	48 752 935	631	30 763 101 985	2
ECBDL14-100	65 003 913	631	41 017 469 103	2

Table 2 SVM learning time in seconds

Dataset	Spark MLlib	Flink	Difference
ECBDL14-10	42	111	69
ECBDL14-30	61	196	135
ECBDL14-50	103	302	199
ECBDL14-75	123	456	333
ECBDL14-100	174	783	609

As an evaluation criteria, we have employed the overall learning runtime (in seconds) for SVM and Linear Regression, as well as the overall runtime for DITFS.

For all experiments we have used a cluster composed of 9 computing nodes and one master node. The computing nodes hold the following characteristics: 2 processors x Intel Xeon CPU E5-2630 v3, 8 cores per processor, 2.40 GHz, 20 MB cache, 2 x 2TB HDD, 128 GB RAM. Regarding the software, we have used the following configuration: Hadoop 2.6.0-cdh5.5.1 from Cloudera's open-source Apache Hadoop distribution, Apache Spark and MLlib 1.6.0, 279 cores (31 cores/node), 900 GB RAM (100 GB/node) and Apache Flink 1.0.3, 270 TaskManagers (30 TaskManagers/core), 100 GB RAM/node.

Table 2 shows the learning runtime values obtained by SVM with 100 iterations, using the reduced version of the datasets with 150 features. Currently SVM is not present in the Spark ML library, so we omit that experiment. As we can see, Spark scales much better than Flink. The time difference between Spark and Flink increases with the size of the dataset, being 2.5x slower at the beginning, and 4.5x with the complete dataset.

Table 3 compares the learning runtime values obtained by LR with 100 iterations. The time difference between Spark MLlib and Spark ML can be explained by internally transforming the dataset from DataFrame to RDD in order to use the same implementation of the algorithm present in MLlib. Spark ML is around 8x times faster than Flink. Spark MLlib version have shown to perform specially better compared to Flink.

Table 4 compares the runtime values obtained by DITFS algorithm selecting the top 10 features of the discretized datset. As stated previously, the differences between Spark MLlib and Spark ML can be explained with the internal transformation between DataFrame and RDD. We observe that Flink is around 10x times slower than Spark for 10, 30 and 50 % of the dataset, 8x times slower for 75 %, and 4x times slower for the complete dataset.

In Fig. 1 we can see the scalability of the three algorithms compared side to side.

Table 3 LR learning time in seconds

Dataset	Spark MLlib	Spark ML	Flink
ECBDL14-10	3	26	181
ECBDL14-30	5	63	815
ECBDL14-50	6	173	1314
ECBDL14-75	8	260	1878
ECBDL14-100	12	415	2566

Table 4 DITFS runtime in seconds

Dataset	Spark MLlib	Spark ML	Flink
ECBDL14-10	44	55	487
ECBDL14-30	111	143	1891
ECBDL14-50	317	441	3240
ECBDL14-75	590	783	4928
ECBDL14-100	1696	2159	6615

Conclusions

In this paper, we have performed a comparative study for batch data processing of the scalability of two popular frameworks for processing and storing Big Data, Apache Spark and Apache Flink. We have tested these two frameworks using SVM and LR as learning algorithms, present in their respective ML libraries. We have also implemented and tested a feature selection algorithm in both platforms. Apache Spark have shown to be the framework with better scalability and overall faster runtimes. Although the differences between Spark's MLlib and Spark ML are minimal, MLlib performs slightly better than Spark ML. These differences can be explained with the internal transformations from DataFrame to RDD in order to use the same implementations of the algorithms present in MLlib.

Flink is a novel framework while Spark is becoming the reference tool in the Big Data environment. Spark has had several improvements in performance over the different releases, while Flink has just hit its first stable version. Although some of the Apache Spark improvements are already present by design in Apache Flink, Spark is much refined than Flink as we can see in the results.

Apache Flink has a great potential and a long way still to go. With the necessary improvements, it can become a reference tool for distributed data streaming analytics. It is pending a study on data streaming, the theoretical strengh of Apache Flink.

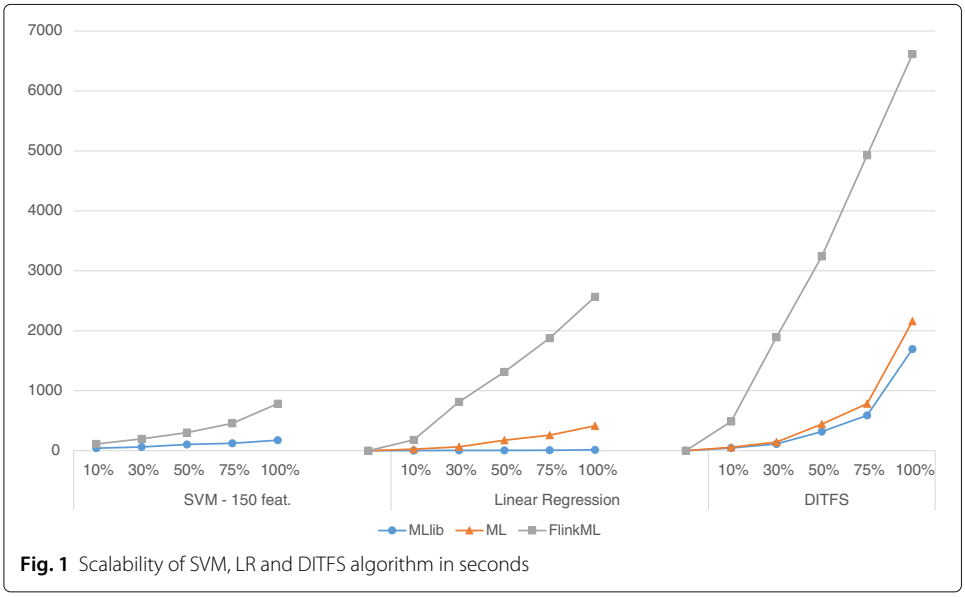

Fig. 1 Scalability of SVM, LR and DITFS algorithm in seconds

Endnote

[1] https://en.wikipedia.org/wiki/Stochastic_gradient_descent.

Acknowledgements
Not applicable.

Funding
This work is supported by the Spanish National Research Project TIN2014-57251-P, and the Andalusian Research Plan P11-TIC-7765. S. Ramirez-Gallego holds a FPU scholarship from the Spanish Ministry of Education and Science (FPU13/00047).

Availability of data and materials
ECBDL14 dataset is freely available in [21].

Authors' contributions
DG and SR carried out the comparative study and drafted the manuscript. SG and FH conceived of the study, participated in its design and coordination and helped to draft the manuscript. All authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Consent for publication
Not applicable.

Ethics approval and consent to participate
Not applicable.

Author details
[1]Department of Computer Science and Artificial Intelligence, CITIC-UGR (Research Center on Information and Communications Technology), University of Granada, Calle Periodista Daniel Saucedo Aranda, 18071 Granada, Spain.
[2]Faculty of Computing and Information Technology, King Abdulaziz University, North Jeddah, Saudi Arabia.

Received: 13 August 2016 Accepted: 21 October 2016
Published online: 01 March 2017

References

1. IDC. The Digital Universe of Opportunities. http://www.emc.com/infographics/digital-universe-2014.htm. Accessed 14 July 2016.
2. Dean J, Ghemawat S. Mapreduce: Simplified data processing on large clusters. In: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6. OSDI'04. Berkeley: USENIX Association; 2004. p. 10–10.
3. Apache Hadoop Project. Apache Hadoop. http://hadoop.apache.org. Accessed 14 July 2016.
4. White T. Hadoop: The Definitive Guide. Sebastopol: O'Reilly Media, Inc; 2012.
5. Hamstra M, Karau H, Zaharia M, Konwinski A, Wendell P. Learning Spark: lightning-fast big data analytics. Sebastopol: O'Reilly Media; 2015.
6. Spark A. Apache Spark: lightning-fast cluster computing. http://spark.apache.org. Accessed 14 July 2016.
7. Flink A. Apache Flink. http://flink.apache.org. Accessed 14 July 2016.
8. Apache Mahout Project. Apache Mahout. http://mahout.apache.org. Accessed 14 July 2016.
9. MLlib. Machine Learning Library (MLlib) for Spark. http://spark.apache.org/docs/latest/mllib-guide.html. Accessed 14 July 2016.
10. Lin JJ. Mapreduce is good enough? if all you have is a hammer, throw away everything that's not a nail! Big Data. 2012;1(1):28–37.
11. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. NSDI'12. Berkeley: USENIX Association; 2012. p. 2–2.
12. Meng X, Bradley J, Yavuz B, Sparks E, Venkataraman S, Liu D, Freeman J, Tsai D, Amde M, Owen S, Xin D, Xin R, Franklin MJ, Zadeh R, Zaharia M, Talwalkar A. Mllib: Machine learning in apache spark. J Mach Learn Res. 2016;17(34):1–7.
13. Malewicz G, Austern MH, Bik AJC, Dehnert JC, Horn I, Leiser N, Czajkowski G. Pregel: A system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. SIGMOD '10. New York: ACM; 2010. p. 135–46. doi:10.1145/1807167.1807184.
14. Apache Spark Project. Project Tungsten (Apache Spark). https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html Accessed 14 July 2016.
15. Apache Flink Project. Peeking Into Apache Flink's Engine Room. https://flink.apache.org/news/2015/03/13/peeking-into-Apache-Flinks-Engine-Room.html. Accessed 14 July 2016.
16. Brown G, Pocock A, Zhao MJ, Luján M. Conditional likelihood maximisation: A unifying framework for information theoretic feature selection. J Mach Learn Res. 2012;13:27–66.

17. Jaggi M, Smith V, Takác M, Terhorst J, Krishnan S, Hofmann T, Jordan MI. Communication-efficient distributed dual coordinate ascent. CoRR. 2014:3068–76. abs/1409.1458.
18. del Río S, López V, Benítez JM, Herrera F. On the use of mapreduce for imbalanced big data using random forest. Inf Sci. 2014;285:112–37.
19. Ramírez-Gallego S, García S, Mouriño-Talín H, Martínez-Rego D. Distributed entropy minimization discretizer for big data analysis under apache spark. In: Trustcom/BigDataSE/ISPA, 2015 IEEE; 2015. p. 33–40. doi:10.1109/Trustcom.2015.559.
20. Ding C, Peng H. Minimum redundancy feature selection from microarray gene expression data. J Bioinforma Comput Biol. 2005;3(02):185–205.
21. Evolutionary Computation for Big Data and Big Learning Workshop. http://cruncher.ncl.ac.uk/bdcomp/. Accessed 14 July 2016.