Big Data Analytics: A Tutorial on Information Process Fusion with MapReduce

Sergio Ramírez[a,*], Alberto Fernández[a], Salvador García[a], Francisco Herrera[a]

[a]*Department of Computer Science and Artificial Intelligence, University of Granada, Granada, Spain*

Abstract

TODO

Keywords: Big Data Analytics, MapReduce, Information Fusion, Spark, Machine Learning

1. Introduction

TODO

In order to address all these objectives, this paper is organized as follows. First, Section 2 presents an introduction on the MapReduce programming framework, also stressing some alternatives for Big Data processing. Section 3 includes an overview on those technologies currently available to address Big Data problems from a distributed perspective. Section 4 presents the core of this paper, analyzing the different design options for developing Big Data analytics algorithms regarding how the partial data and models are aggregated. Then, we show a case study in Section 5 to contrast the capabilities regarding scalability of the different approaches previously introduced. Finally, Section 6 summarizes and concludes this paper.

2. MapReduce as Information Process Fusion

TODO

2.1. MR

The rapid growth and influx of data from private and public sectors has popularized the notion of "Big data [1]". The surge in Big Data has led to the development of custom paradigms and algorithms that are able to extract significant value and insight in different areas such as medical, health care, business, management and so on [2, 3, 4].

The MapReduce execution environment [5] is the most common framework used in this scenario. Being a privative tool, its open source counterpart, known as Hadoop, has been traditionally used in academia

*Corresponding author. Tel:+34-958-240598; Fax: +34-958-243317

Email addresses: `sramirez@decsai.ugr.es` (Sergio Ramírez), `alberto@decsai.ugr.es` (Alberto Fernández), `salvagl@decsai.ugr.es` (Salvador García), `herrera@decsai.ugr.es` (Francisco Herrera)

research [6]. It has been designed to allow distributed computations in a transparent way for the programmer, also providing a fault-tolerant execution scheme. To take advantage of this scheme, any algorithm must be divided into two main stages: Map and Reduce. The first one is devoted to split the data for processing, whereas the second collects and aggregates the results.

Additionally, the MapReduce model is defined with respect to an essential data structure: the <key,value> pair. The processed data, the intermediate and final results work in terms of <key,value> pairs. To summarize its procedure, Figure 1 illustrates a typical MapReduce program with its *Map* and *Reduce* steps.

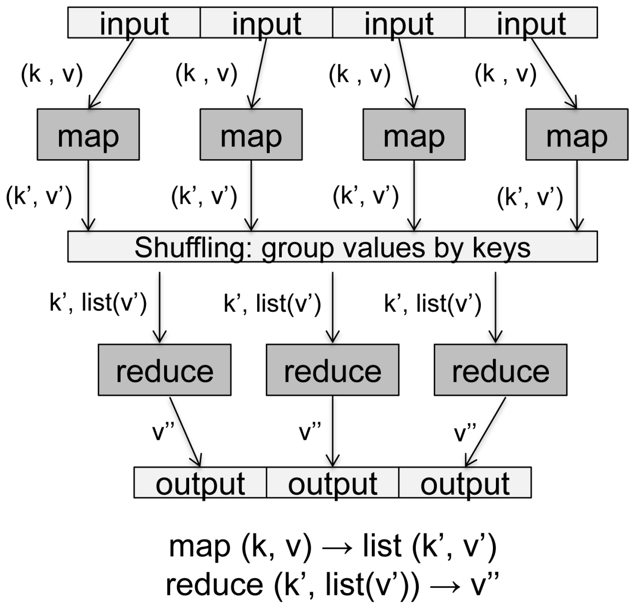

Figure 1: The MapReduce programming model.

Map function first reads data and transforms them into a key-value format. Transformations in this phase may apply any sequence of operations on each record before sending the tuples across the network. Output keys are then shuffled and grouped by key value so that coincident keys are grouped together to form a list of values. Keys are then partitioned and sent to the Reducers according to some key-based scheme previously defined. Finally, the Reducers perform some kind of aggregation on the lists to eventually generate a single value for each pair. As an optimization, the reducer is also used as a combiner on the map outputs. This improvement reduces the total amount of data sent across the network by combining each word generated in the Map phase into a single pair.

From another perspective, MapReduce, concretely the Reduce stage, can be seen as a information and/or model fusion process that aggregates partial results to obtain a more coarse-grained outcome. Although the Reduce phase may be skipped in some jobs in order to perform a straightforward parallelization of tasks, it is not the case in most of use cases in Big Data.

For example, Word Count in MapReduce comes to be one of the most widespread examples to illustrate the intrinsic fusion process performed behind this paradigm. WordCount example is intended to count the number of occurrences per word in a set of input text files. In WordCount, each mapper reads a set of blocks formed by lines, and splits them into words. It then emits a key-value pair with the word as key, and 1 as value. Afterwards, each reducer sums the scores for each word, and outputs a single key-value pair with the word and sum.

For example, consider the string "knock knock who is there?". A single mapper would receive and split this sentence as words, and then, it would form the initial pairs as: (knock,1), (knock,1), (who,1), (is,1), (there,1) (punctuations signs are obviated).

In reducers, the keys are grouped together and the count values for identical keys (words) are added. In this case only one pair of similar keys 'knock' would be aggregated so that the output pairs would be as follows: (knock,2), (who,1), (is,1), (there,1). As result, the user would receive the final number of hits for each word.

[7, 8, 9, 10].

2.2. Alternatives

Distributed processing frameworks in Big Data are responsible of ingesting, filtering, transforming, learning, querying, and exporting large amounts of data. Yet there exist many implementations for this concept, most of modern distributed frameworks follows a Single Instruction Multiple Datasets (SIMD) [**?**] scheme in order to execute the same sequence of instructions simultaneously on a distributed set of data partitions. Apart from defining the main scheme of execution, these frameworks also cope with other problems, such as: restarting failed processes, job scheduling, network synchronization, load-balancing, etc. In this section we focus on those models more relevant for current implementations in distributed computing.

2.2.1. Directed Acyclic Graph (DAG) Parallel Processing

All DAG-based distributed frameworks for Big Data [**?**], like Spark, organize their jobs by splitting them into a smaller set of atomic tasks. In this model vertices correspond with parallel tasks, whereas edges are associated with exchange of information. As shown in Figure **??**, vertices can have multiple connections between inputs and outputs, which imply that the same task can be run in different data and the same data in different partitions. Data flows are physically supported by shared memory, pipes, or disks. Instructions are duplicated and sent from the master to the slave nodes for a parallel execution. Notice that MapReduce can be deemed as an specific implementation of DAG-based processing, with only two functions as vertices.

2.2.2. Bulk Synchronous Parallel (BSP) Processing

Bulk Synchronous Parallel (BSP) [**?**] systems are formed by a series of connected supersteps, implemented as directed graphs. In this scheme input data is the starting point. From here to the end, a set of Supersteps are applied on partitioned data in order to obtain the final output. As mentioned before, each Superstep correspond with an independent graph associated with a subtask to be solved. Once all compounding subtasks end, bulk synchronization of all outputs is committed. At this point vertices may send messages to the next Superstep, or receive some from previous steps, and also to modify its state and outgoing edges.

3. Big Data Technologies for Analytics

Nowadays, the volume of data currently managed by our storage systems have surpassed the processing capacity of traditional systems [4], and this applies to data mining as well. Distributed computing has been widely used by data scientists before the advent of Big Data. Many standard and time-consuming algorithms were replaced by more rapid distributed versions that make affordable the learning process. Nevertheless, for most of current massive problems, a distributed approach becomes mandatory nowadays since no batch architecture is able to address such magnitudes.

Beyond High Performance Computing (HPC) solutions, new large-scale processing platforms are intended to bring closer distributed processing to practitioners and experts by hiding the technical nuances derived from these environments. Novel and complex designs are required to create and maintain these platforms, which generalizes the utilization of distributed computing for standard users.

As a result of the fast evolving of Big Data environment, a myriad of tools, paradigms and techniques have surged to tackle different use cases in industry and science. However, because of the myriad of tools, it is often difficult for practitioners and experts to analyze and select the right tool for each goal. In this section we present and analyze some alternatives for distributed processing with the objective of providing the necessary knowledge that helps users to decide which alternative better fits their requirements. We also outline the software libraries that gives support to the distributed learning task in these platforms.

3.1. Apache Hadoop

Undoubtedly Hadoop MapReduce may be deemed as the primary platform in the Big Data space. After the presentation of MapReduce by Google designers [11], Hadoop MapReduce was grown by the community, and became the most used and powerful open-source implementation of MapReduce. Nowadays leading companies such as Yahoo has scaled from 100-node Hadoop clusters to 42K nodes and hundreds of petabytes [12] thanks to the outstanding performance of Hadoop.

The main idea behind Hadoop was to create a common framework which can process large-scale data on a cluster of commodity hardware, without incurring in a high cost in developing (in contrast to HPC solutions) and execution time. Hadoop MapReduce was originally composed by two elements: the first one was a distributed storage system called Hadoop Distributed File System (HDFS), whereas the second one was a data processing framework that allows to run MapReduce-like jobs. Apart from these goals, Hadoop implements primitives to address cluster scalability, failure recovery, and resource scheduling, among others.

But Hadoop is today more than a single technology, but a complete software stack and ecosystem formed by several top-level components that address diverse purposes. For instance, Apache Giraph for graph processing or Apache Hive for data warehousing. The common factor is that all of them rely on Hadoop, and are tightly linked to this technology. Some projects are actually Apache top-level projects [13], whereas others are continuously evolving or being created.

HDFS [14] can be deemed as the main module of Apache Hadoop. It supports distributed storage for large-scale data through the use of distributed files, which themselves are composed by fixed-size data blocks. These blocks or partitions are equally distributed among the data nodes in order to balance as much as possible the overall disk usage in the cluster. HDFS also allows replication of blocks across different nodes and racks. In HDFS, the first block is ensured to be placed in the same processing node, whereas the other two replicas are sent to different racks to prevent abrupt ends due to inter-rack issues.

HDFS was thought to work with several storage formats. It offers several APIs to read/write registers. Some relevant APIs are: InputFormat (to read customizable registers), or RecordWriter (to write record-shaped elements). Users can also developed their own storage format, and to compress data according to their requirements. Persistence in Hadoop is mainly performed in disk. However, there are some novel advances to optimize persistence by introducing some memory usage. For instance, in Apache Hadoop version 3.0 was introduced the option of memory usage as temporary storage.

Although **MapReduce** [11] is the native processing solution in Apache Hadoop, today it supports multiple alternatives with different processing schemes. All these solutions have in common that use a set of data nodes to run tasks on the local data blocks, and one master node (or more) to coordinate these tasks. For instance, **Apache Tez** [15] is a processing engine that transforms processing jobs into direct acyclic graphs (DAGs). Thanks to Tez, users can run any arbitrary complex DAG of jobs in HDFS. Tez thus efficiently solves interactive and iterative processes, like those present in machine learning processes. Its most relevant contribution is that Tez translate any complex job to a single MapReduce phase. Furthermore, it does not need to store intermediate files and reuses idle resources, which highly boost the overall performance.

Hadoop MapReduce evolves to a more general component, called **Yet Another Resource Negotiator (YARN)** [16], which provides extra management and maintenance services relied to other components in the past. YARN also acts as a facade for different types of distributed processing engines based on HDFS, such as Spark, Flink or Storm. In short, YARN was intended as a generic purpose system that separates the responsibilities of resource management (performed by YARN), and running management (performed by top-level applications).

Among the full set of advantages claimed by YARN, we can highlight its capacity to run several application on the same cluster without the necessity of moving data. In fact, YARN allows reusing resources across alike applications in parallel, which improves the overall usage of resources.

3.1.1. Apache Mahout

Since the magnitude of learning problems has been growing exponentially, data scientists demands rapid tools that efficiently extract knowledge from large-scale data. This problem has been solved by MapReduce and other platforms by providing scalable algorithms and miscellaneous utilities in form of machine learning libraries. These libraries are compatible with the main Hadoop engine, and use as input the data stored in the storage components.

Apache Mahout [17] was the main contribution from Apache Hadoop to this field. Although it can be deemed as mainly obsolete nowadays, Mahout is considered as the first attempt to fill the gap of scalable machine learning support for Big Data. Mahout comprises several algorithms for plenty of tasks, such as: classification, clustering, pattern-mining, etc. Among a long list of golden algorithms in Mahout, we can highlight Random Forest or Naïve Bayes.

The most recent version (0.13.0) provides three new major features: novel support for Apache Spark and Flink, a vector math experimentation for R, and GPU support based on large matrix multiplications. Although Mahout was originally designed for Hadoop, some algorithms have been implemented on Spark as a consequence of the latter one's popularity. Mahout is also able to run on top of Flink, being only compatible for static processing though.

3.2. Spark

Apache Spark Framework [18] was born in 2010 with the publication of Resilient Distributed Datasets (RDD) structures [19], the keystone behind Spark. Although Spark has a close relationship with Hadoop Ecosystem, it provides specific support for every step in the Big Data stack, such as its own processing engine, and machine learning library.

Apache Spark [20] is defined as a distributed computing platform which can process large volume data sets in memory with a very fast response time due to its memory-intensive scheme. It was originally thought to tackle problems deemed as unsuitable for previous disk-based engines like Hadoop. Continued use of disk is replaced in Spark by memory-based operators that efficiently deal with iterative and interactive problems (prone to multiple I/O operations).

The heart of Spark is formed by **Resilient Distributed Datasets (RDD)**, which transparently controls how data are distributed and transformed across the cluster. Users just need to define some high-level functions that will be applied and managed by RDDs. These elements are created whenever data are read from any source, or as a result of a transformation. RDDs consist of a collection of data partitions distributed across several data nodes. A wide range of operations are provided for transforming RDDs, such as: filtering, grouping, set operations, among others. Furthermore RDDs are also highly versatile as they allows users to customize partitioning for an optimized data placement, or to preserve data in several formats and contexts.

In Spark, fault tolerance is solved by annotating operations in a structure called lineage. Spark transformations annotated in the lineage are only performed whenever a trigger I/O operations appears in the log. In case of failure, Spark re-computes the affected brach in the lineage log. Although replication is normally skipped, Spark allows to spill data in local disk in case the memory capacity is not sufficient.

Spark developers provided another high-level abstraction, called **DataFrames**, which introduces the concept of formal schema in RDDs. DataFrames are distributed and structured collections of data organized by named columns. They can be seen as a table in a relational database or a dataframe in R, or Python (Pandas). As a plus, relational query plans built by DataFrames are optimized by the Spark's Catalyst optimizer throughout the previously defined schema. Also thanks to the scheme, Spark is able to understand data and remove costly Java serialization actions.

A compromise between structure awareness and the optimization benefits of Catalyst is achieved by the novel Dataset API. **Datasets** are strongly typed collections of objects connected to a relational schema. Among the benefits of Datasets, we can find compile-time type safety, which means applications can be sanitized before running. Furthermore, Datasets provide encoders for free to directly convert JVM objects to the binary tabular Tungsten format. These efficient in-memory format improves memory usage, and allows to directly apply operations on serialized data. Datasets are intended to be the single interface in future Spark for handling data.

3.2.1. MLlib

MLlib project [21] was born in 2012 as an extra component of Spark. It was released and open-sourced in 2013 under the Apache 2.0 license. From its inception, the number of contributions and people involved in the project have been growing steadily. Apart from official API, Spark provides a community package index [22] (Spark Packages) to assemble all open source algorithms that work with MLlib.

MLlib is a Spark library geared towards offering distributed machine learning support to Spark engine. This library includes several out-of-the-box algorithms for alike tasks, such as: classification, clustering, regression, recommendation, even data preprocessing. Apart from distributed implementations of standard algorithms, MLlib offers:

- Common Utilities: for distributed linear algebra, statistical analysis, internal format for model export, data generators, etc.

- Algorithmic optimizations: from the long list of optimizations included, we can highlight some: decisions trees, which borrow some ideas from PLANET project [23] (parallelized learning both within trees and across them); or generalized linear models, which benefit from employing fast C++-based linear algebra for internal computations.

- Pipeline API: as the learning process in large-scale datasets is tedious and expensive, MLlib includes an internal package (*spark.ml*) that provides an uniform high-level API to create complex multi-stage pipelines that connect several and alike components (preprocessing, learning, evaluation, etc.). *spark.ml* allows model selection or hyper-parameter tuning, and different validations strategies like k-fold cross validation.
- Spark integration: MLlib is perfectly integrated with other Spark components. Spark GraphX has several graph-based implementations in MLlib, like LDA. Likewise, several algorithms for online learning are available in Spark Streaming, such as online k-Means. In any case, most of component in the Spark stack are prepared to effortlessly cooperate with MLlib.

3.3. Flink

Apache Flink [24] is a distributed processing component focused on streaming processing, which was designed to solve problems derived from micro-batch models (Spark Streaming). Flink also supports batch data processing with programming abstractions in Java and Scala, though it is treated as a special case of streaming processing. In Flink, every job is implemented as a stream computation, and every task is executed as cyclic data flow with several iterations.

Flink provides two operators for iterations [25], namely, standard and delta iterator. In standard iterator, Flink only works with a single partial solution, whereas delta iterator utilizes two worksets: the next entry set to process and the solution set. Among the set of advantages provided by iterators is the reduction of data to be computed and sent between nodes [26]. According to the authors, new iterators are specially designed to tackle machine learning and data mining problems.

Apart from iterators, Flink leverages from a PACT optimizer which analyzes the code and the data access conflicts to reorder operators and create semantically equivalent execution plans [27, 28]. Physical optimization is then applied on plans to boost data transport and operators' execution on nodes. Finally, PACT optimizer selects the most resource-efficient plan, regarding network and storage.

Furthermore, Flink provides a complex 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.

FlinkML is aimed at providing a set of scalable ML algorithms and an intuitive API to Flink users. Until now, FlinkML provides few alternatives for some fields in machine learning: SVM with CoCoA [29], or Multiple Linear regression for supervised learning, k-NN join for unsupervised learning, scalers and polynomial features for preprocessing, Alternating Least Squares for recommendation, and other utilities for validation and outlier selection, among others. FlinkML also allows users to build complex analysis pipelines via chaining operations (like in MLlib). FlinkML pipelines are inspired by the design introduced by sklearn in [30].

4. Big Data Analytics on Fusion Process

TODO

4.1. Direct fusion of models: ensemble approach

TODO

4.2. Approximate fusion of models: aggregation of partial systems

TODO

4.3. Exact fusion for scalable models: distributed data and models' partition

TODO

5. Practical Study on XXX

TODO

6. Concluding Remarks

TODO

Acknowledgments

This work have been partially supported by the Spanish Ministry of Science and Technology under projects TIN2014-57251-P and TIN2015-68454-R; and the Foundation BBVA project 75/2016 BigDaP-TOOLS.

[1] A. Fernández, S. Río, V. López, A. Bawakid, M. J. del Jesus, J. Benítez, F. Herrera, Big Data with cloud computing: An information sciencesight on the computing environment, MapReduce and programming framework, WIREs Data Mining and Knowledge Discovery 4 (5) (2014) 380–409.

[2] K. Kambatla, G. Kollias, V. Kumar, A. Grama, Trends in Big Data analytics, Journal of Parallel and Distributed Computing 74 (7) (2014) 2561–2573.

[3] C. P. Chen, C.-Y. Zhang, Data-intensive applications, challenges, techniques and technologies: A survey on Big Data, Information Sciences 275 (2014) 314–347.

[4] X. Wu, X. Zhu, G.-Q. Wu, W. Ding, Data mining with big data, Knowledge and Data Engineering, IEEE Transactions on 26 (1) (2014) 97–107.

[5] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107–113.

[6] T. White, Hadoop: The Definitive Guide, 4th Edition, O'Reilly Media, 2015.

[7] Q. Zhang, B. Wu, J. Yang, Parallelization of ontology construction and fusion based on mapreduce, 2014, pp. 439–443.

[8] S. del Río, V. López, J. Benítez, F. Herrera, On the use of mapreduce for imbalanced big data using random forest, Information Sciences 285 (2014) 112 – 137.

[9] J. Meng, R. Li, J. Zhang, Parallel information fusion method for microarray data analysis, 2015, pp. 1539–1544.

[10] S. del Río, V. López, J. Benítez, F. Herrera, A mapreduce approach to address big data classification problems based on the fusion of linguistic fuzzy rules, International Journal of Computational Intelligence Systems 8 (3) (2015) 422–437.

[11] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, in: OSDI04: PROCEEDINGS OF THE 6TH CONFERENCE ON SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION, USENIX Association, 2004.

[12] D. Harris, The history of Hadoop: From 4 nodes to the future of data, `https://gigaom.com/2013/03/04/the-history-of-hadoop-from-4-nodes-to-the-future-of-data/` (2013).

[13] A. S. Foundation, Apache Project Directory, `https://projects.apache.org/` (2017).

[14] H. D. F. System, Hadoop Distributed File System, `https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html` (2017).

[15] A. Tez, Apache Tez, `https://tez.apache.org/` (2017).

[16] A. YARN, Apache YARN, `https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html` (2017).

[17] A. Mahout, Apache Mahout, `https://mahout.apache.org/` (2017).

[18] A. Spark, Apache Spark: Lightning-fast cluster computing, `http://spark.apache.org/` (2016).

[19] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, 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, 2012, pp. 2–2.

[20] M. Hamstra, H. Karau, M. Zaharia, A. Konwinski, P. Wendell, Learning Spark: Lightning-Fast Big Data Analytics, O'Reilly Media, Incorporated, 2015.

[21] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, A. Talwalkar, Mllib: Machine learning in apache spark, Journal of Machine Learning Research 17 (34) (2016) 1–7.

[22] S. Packages, 3rd Party Spark Packages, `https://spark-packages.org/` (2017).

[23] B. Panda, J. S. Herbach, S. Basu, R. J. Bayardo, Planet: Massively parallel learning of tree ensembles with mapreduce, in: Proceedings of the 35th International Conference on Very Large Data Bases (VLDB-2009), 2009.

[24] A. Flink, Apache Flink, `http://flink.apache.org/` (2016).

[25] 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` (2016).

[26] S. Ewen, K. Tzoumas, M. Kaufmann, V. Markl, Spinning fast iterative data flows, PVLDB 5 (11) (2012) 1268–1279.

[27] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinlnder, M. J. Sax, S. Schelter, M. Hger, K. Tzoumas, D. Warneke, The stratosphere platform for big data analytics, International Journal on Very Large Databases 23 (6) (2014) 939–964.

[28] F. Hueske, M. Peters, M. Sax, A. Rheinlnder, R. Bergmann, A. Krettek, K. Tzoumas, Opening the black boxes in data flow optimization, PVLDB 5 (2012) 1256–1267.

[29] M. Jaggi, V. Smith, M. Takác, J. Terhorst, S. Krishnan, T. Hofmann, M. I. Jordan, Communication-efficient distributed

dual coordinate ascent, CoRR abs/1409.1458.
URL `http://arxiv.org/abs/1409.1458`

[30] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, G. Varoquaux, API design for machine learning software: experiences from the scikit-learn project, in: ECML PKDD Workshop: Languages for Data Mining and Machine Learning, 2013, pp. 108–122.