Implementing Dictionary Learning in Apache Flink, Or: How I Learned to Relax and Love Iterations

Geoffrey Mon*, Milad Makkie†, Xiang Li‡, Tianming Liu†, and Shannon Quinn†§
Department of Computer Science
University of Georgia
Athens, GA 30602 USA
Email: *geoffreytmon@gmail.com, †{milad,tliu,squinn}@cs.uga.edu, ‡xiang_li@dfci.harvard.edu
§Corresponding author

***Abstract*—The authors evaluate the use of Apache Flink, a novel data analysis framework offering optimizations over competitors such as Apache Spark, in order to use a rank-1 dictionary learning (r1DL) algorithm to decompose fMRI data. We first expand the functionality of the Flink Python API in order to accommodate the implementation of rank-1 dictionary learning, a model for decomposing a large matrix. Iterative algorithms, aggregators, and other features are added to the incomplete Python API, and the experiences and lessons learned are described. Using these features, we port an existing implementation of r1DL from using the Python API of Apache Spark to using the Python API of Apache Flink. In preliminary testing, this implementation suggests performance boosts over Spark for large input files, meriting further research. We conclude that Flink is likely a feasible tool for the application of dictionary learning to decompose fMRI data, and we continue to evaluate and apply it.**

Keywords*—Data science, open source software, iterative algorithms, distributed computing, machine learning.

I. Introduction

Iterative algorithms, which repeatedly apply a step function until a termination criterion is reached, are difficult to scale properly because of the magnitude of inter-node communication required. Despite this, they are becoming more and more prevalent and promising, especially in the field of machine learning. They have been used for tasks such as graph analytics [1], gradient learning [2], and dimensionality reduction [3].

Existing established data analysis frameworks such as Apache Spark support iterative algorithms, but leave much to be desired. Spark is an open source framework based around resilient distributed datasets, a construct for efficient and fault-tolerant data computation [4] [5] [6]. It has been a significant development in performance over previous solutions such as Hadoop MapReduce [7], and has been extended for graph processing [8], machine learning [9], streaming data processing [10], and many other applications.

Unfortunately, Spark does not support native iterations, which means the its engine does not directly handle iterative algorithms. In order to implement an iterative algorithm, a loop in the Spark job file needs to repeatedly instruct Spark to execute the step function and manually check the termination criterion, significantly increasing overhead for large-scale iterative jobs [11].

A. Apache Flink

Apache Flink (formerly Stratosphere) is a novel JVM-based data analysis framework with promising iteration-friendly features [12] [13].

Using Flink is similar to using Spark: the user writes a program (called an execution plan) in Java, Scala, or other JVM-compatible language using the Flink API to define data sources, operations, and data sinks. This program is compiled into a JAR that is submitted to Flink, processed by a jobmanager (master) process, and executed by taskmanager (worker) processes. Flink offers many of the same transformations used by Spark.

The defining feature of Flink is its pipeline-based architecture specializing in operations on real-time streaming data. While Spark is based on batch processing architecture, Flink relies on real-time streaming over a pipeline-based architecture [14]. This means Flink treats streaming data more efficiently as what it is (a stream), unlike Spark which splits a stream into small batch ("micro-batch") jobs [10] [15]. Flink also treats batch data as finite streams of data using the same engine [14].

Flink also features an optimizing engine. It will choose the best execution strategy for certain types of transformations, such as the joining of two data sets [12]. It also chains operations by running consecutive operations in the same thread to reduce overhead [16]. Spark offers no such automatic optimizations and requires manual tweaking by the user [11].

However, the most important feature Flink offers for iterative algorithms is native iterations [17]: the iterative step function only needs to be scheduled once (rather than multiple times as in Spark) and the repetition and termination criterion are all handled by the Flink engine, significantly reducing overhead over Spark.

We applied Flink and these distinctive benefits to existing applications in order to measure its performance benefit and feasibility for future use.

B. Flink Python API

Flink does have limited support for one non-JVM language, Python. Because of Flink's young age (its first major version 1.0.0 was released in March 2016), the Python API is a relatively new feature that lacks access to some features of

JVM-based Flink. The Python API was first added to the Flink codebase, about one and a half years ago, on March 3, 2015 under bug FLINK-671 by project committer Chesnay Schepler [18]. Similar to using Java or Scala, a Python plan file describes the data sources, operations, and data sinks. The Python bindings serialize the plan specifications from the Python plan and send them to the Flink Python API module, which deserializes them and executes the equivalent Java functions. Custom functions defined in Python are applied by spawned Python processes onto data streamed to and from Java.

Because of this design, accessor methods for Flink features must be separately implemented in the Python API, which also has to deal with the differences between Java and Python. One of the most significant differences is that Python is dynamically typed, while Java is statically typed. Operations in Flink expect to operate on statically typed variables, so the Python API packs data into byte arrays (`byte[]`) before they entering Java, and unpacks them into their original type as they return to Python [19].

The design, context, and age of the Python API contribute to its incomplete feature set. Before this project began, such missing features otherwise accessible from Java included iterations, aggregations (which find the sum, minimum, or maximum of part of a dataset), and utility methods such as `zip_with_index` (which indexes a dataset in order). Some of these features, especially the use of iterative algorithms, were necessary for our project; we attempted to implement accessors for these features in the Python API.

C. Rank-1 Dictionary Learning

Dictionary learning is a learning method that seeks to find a sparse representation of a large data set.

We use our novel distributed rank-1 dictionary learning (r1DL) model. Using the power of distributed computing, the model decomposes the a large input matrix S by iteratively estimating multiple rank-1 basis vectors u and their loading coefficient vectors v [20]. This algorithm is highly iterative, which means it can take advantage of Flink's native iterations.

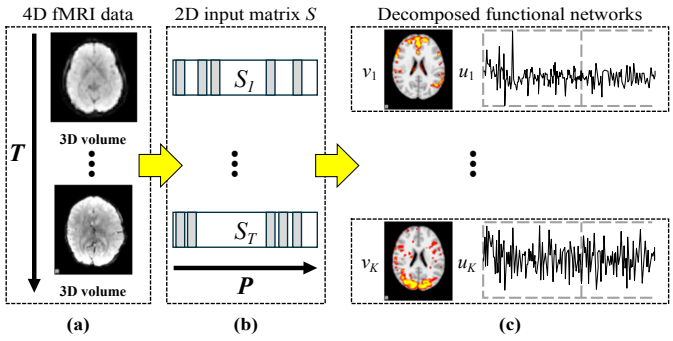

Fig. 1. Illustration of the r1DL model applied on our use case (fMRI data). (a) 4D fMRI data represented as a series of 3D volume images. (b) Converting 4D data into 2D input matrix S. (c) Spatial ($v_1 \ldots v_K$) and temporal ($u_1 \ldots u_K$) patterns of the decomposed functional networks from S using r1DL.

One category of large matrices that can be decomposed by dictionary learning is functional magnetic resonance imaging (fMRI) data, which can be used to infer groups of connected neurons that indicate specialized function. Dictionary learning has shown good performance and promise for extracting functional networks from different types of fMRI signals [21]. However, the computational work is very heavy [22]. We have used the Python API of Apache Spark in prior work to implement this r1DL algorithm for analyzing fMRI data [20]; this implementation is able to decompose the entire set of functional networks in a fast, efficient, and distributed way. However, we struggled with Spark's relatively high-level distributed primitives that eagerly executed some of the operations, missing opportunities to cache and optimize intermediate results and minimize the overall number of computations.

Because of the promising Flink architecture and its complete library of lazily-executed primitives, we sought to port the existing Spark program to use the Flink Python API, in order to measure the performance improvements and feasibility of using Flink. This task would also establish Flink as a useful open source tool for neurobiology, a field built on large-scale open source data analysis tools [23].

This paper describes the experiences, lessons learned, and results of addressing the missing features in the Python API in order to successfully port and evaluate a Flink version of r1DL for use in decomposing fMRI data.

II. Procedure

A. Aggregations

Aggregations are prepackaged group reduce functions that are applied to tuple-based data sets. For example, a sum aggregation would group reduce a data set by adding all of the values in a certain field in each tuple. Because of the relative simplicity and prominent appearance in basic example plans of this feature, aggregations were seen as a basic task to tackle in order to orient ourselves with the internals of Flink.

Initially, we focused on utilizing the existing Java aggregation functionality by adding methods to the Python API that specified those existing Java functions to be run on the data set. However, problems soon arose. In order to account for the dynamically typed nature of the Python language, data used with Python API do not retain their type when they reach Java, but instead take on a generic type by becoming byte arrays. However, because of the statically typed nature of Java, aggregations used in Java are defined for specific data types. These aggregations refused the generic typed data from the Python API.

Ultimately, we decided to address this problem by implementing aggregations as pre-defined Python-side group reduce functions. Group reduce functions were already implemented in the Python API prior to our work and served as a basis for creating identically-functioning aggregations in the Python API that had access to the actual typed data. The patch for this functionality was merged on July 15, 2016 [24].

B. `zip_with_index`

The `zip_with_index` method adds an index to each element of a data set to form a data set of tuples. For example,

$$\text{zip\_with\_index}(\{1, 9, 8, 4\}) = \{(0, 1), (1, 9), (2, 8), (3, 4)\}$$

This method is important for our r1DL algorithm because the rows of the input matrix of data need to be enumerated in order to preserve order and perform row-wise multiplications and other matrix transformations.

Because data from Python takes on a generic byte array type, we cannot edit specific fields of tuples from Java. Therefore, `zip_with_index` was implemented as a Python method using existing Python methods. The patch for `zip_with_index` was merged on June 22, 2016 [25].

C. Iterations

Our initial attempt to implementation iterations in Python was a simple Python method that deferred to Java iterations. However, this basic implementation did not work: a basic iteration program that iterated each element of a data set of integers would hang. After consultation with Flink committer and Python API author Chesnay Schepler, we discovered that one internal data iterator function would hang if evaluated after already returning false. This problem was solved by wrapping the function: after that function has previously returned false for that iterator, all future evaluations will return false. At this point, the iterations worked for basic programs.

However, after running our completed r1DL Flink script, a cryptic `ClassCastException` arose from a deserializer function handling Python-Java communication. We tried many different workarounds, such as moving around or adding identity operations, in order to change the properties of the data sets involved and work around the problem, but none of these options have worked. To seek help from the Flink mailing list, we made a stripped-down script that was focused on what we thought was causing the error. After the Flink mailing list could not reproduce our issue using that script, we realized that our issue could persist in Flink's its internal caches. Restarting Flink between each error allowed us to assemble another Flink script that always reproduced the issue. Using this script, Schepler discovered that the root of the problem was the chaining optimization, which was disrupting the types of data processed in the Flink engine, hence the `ClassCastException`. Chaining was selectively disabled for iteration operations.

Another problem arose: the deserializing functions suffered from a race condition where invalid characters would disrupt the data streams used to communicate between Python and Java. This was directly caused by the reuse of a data set in the r1DL plan file. Data channels were being used by different operations at the same time because the Python API assumed that each stream would only be subject to use by one operation. The solution was to append a random integer to each data stream so that names would not collide.

At this point, we found that iterations worked, but the termination criterion, a data set derived from the iterative data set used to determine the end of a series of iterations, had no effect. Similar to the previous `ClassCastException` issue, the termination criterion problem turned out to be caused by chaining on the parent set of the termination criterion data set, which disrupted the reference that the iteration API expected to use to access it. Disabling chaining for the parent set resolved the issue.

The final problem was that the calculated v vector featured large long integer components, rather than floating-point numbers. v is finalized by filling in unused components with zeroes, but the plan file used an integer zero literal (`0`) rather than a floating-point zero literal (`0.0`). Because the Python API must deal with the dynamically typed nature of Python against the statically typed nature of Java and because each tuple in a data set is expected to have the same types in each field, the API inferred from the integer zero literals that the vector component value field of v was made up of integers, despite the floating-point values present. The problem was resolved by replacing the integer zero literal with the floating-point zero literal.

Although there are aspects of iterations that are not yet supported in the Python API, such as delta iterations (iterations that only operate on a subset of the data set) and iterative aggregators (aggregations used to track statistics during iterations), the current implementation is sufficient for our use and for many other iterative algorithms.

D. r1DL Script Conversion

As we tackled the iterations implementation, we also converted our existing Apache Spark script for r1DL to use Flink. This was not trivial, mainly because Spark allows direct access to data sets, but Flink's Python API does not. This means that Spark programs, including our original implementation of r1DL, can make use of external libraries such as `numpy` and `scipy` by running those library routines directly onto data sets between operations, but our Flink program could not. We changed the logic substantially to fit this new environment. For example, while Spark r1DL could use data constructs that preserved element order, Flink r1DL used data sets of tuples containing position information. And instead of using `numpy` dot product methods, we used join operations between data sets.

When we discovered issues with the Python API iterations, we still wanted to ensure that the logic of our script would work when we completed iterations. A Java version (that does not use the problematic and incomplete Python API) was created to demonstrate the usability of the program, and functioned as expected. However, because of the inherent performance differences between Java and Python, a comparison between a Python Spark script and Java Flink program would not be fair.

When we completed our implementation of Python API iterations, our Python r1DL plan was able to produce results equivalent to the results from our original Spark implementation using the same parameters and data.

III. Results

In order to benchmark the expected improvements between Spark and Flink r1DL, we set up a virtual cluster (the "resource-limited cluster") of three nodes, each with four virtual CPUs, 8192 MB RAM, and 30 GB disk storage. On this

TABLE I. SUMMARY OF THE INPUT MATRIX FILES USED TO TEST FLINK AND SPARK R1DL ON RESOURCE-LIMITED CLUSTER

File	Size (B)	Dimensions	No. elements
small	4741	100×5	500
medium	123205300	11×895780	9853580
large	246380721	22×895780	19707160

cluster, three input files of varying sizes described in Table I (4.7K "small file", 118M "medium file", and 235M "large file") were processed using both Flink and Spark r1DL, and the run time for each job was recorded.

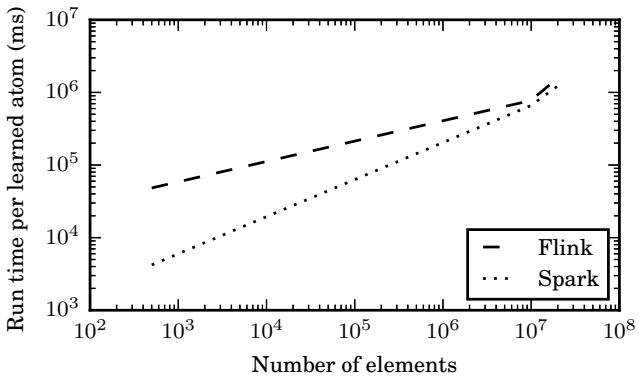

Fig. 2. Average r1DL run time per dictionary atom learned using Flink and Spark on resource-limited cluster with varying input file sizes

As seen in Fig. 2, on the resource-limited cluster, Flink has significantly worse run times than Spark for the small file, but as the file size increases, the difference becomes smaller. We expected Flink to be slower for small files because of the overhead of its optimizing engine, but faster for very large files because of the benefits of its optimizations. In this case, the run times seem to converge. However, this convergence may be caused by the resource limitations of the resource-limited cluster. During testing, we planned to process input files of 2G or more in size, but our cluster was unable to handle the load and repeatedly ran out of RAM and disk space. Multiple trials for each combination of framework and input file were also planned, but curtailed by restricted computing resources and time. We cannot be sure that Flink will become significantly faster for larger files without running tests on such larger input files with a more capable cluster.

Despite all of this, Flink still shows enormous promise given its proximity to Spark in low-resource situations. We are currently exploring further benchmarking using Amazon EMR clusters, which now support Apache Flink [26]. EMR clusters can provide much more computing power than the local resource-limited cluster, which would allow us to use more diverse and large input file sizes in more timed trials.

IV. CONCLUSION AND FUTURE WORK

We began this endeavor with the goal of investigating use of Flink for machine learning algorithms applied to neurobiology data.

To achieve this goal, we have successfully implemented aggregations, the `zip_with_index` utility method, and iterations in the Python API of Apache Flink. The former two features have already been merged into Flink, and the authors are finalizing the iterations implementation in order to make it available for all. We have also successfully ported an implementation of rank-1 dictionary from Apache Spark to Flink. Preliminary testing shows that Flink r1DL could offer significant performance gains over Spark r1DL for large data sets. We are currently conducting more testing with Flink r1DL on these and other-sized data sets to pinpoint the benefits of using Flink.

Throughout the course of this project, we encountered important traits of Python and the Python API that affect how the API works and is implemented, such as how the API handles type or communicates with Java. This experience will facilitate further usage and modification of Flink. We have also contributed features to Flink, an open source tool, in order to make it more capable for more people, including researchers of many fields across the world, for many years to come.

We know that Flink is a very promising tool and we are further evaluating its usage and performance.

ACKNOWLEDGMENTS

The authors would like to thank Chesnay Schepler, a committer for Flink and the main author of the Flink Python API, for his generous and priceless assistance, and for his code review of our modifications of Flink.

REFERENCES

[1] S.-I. Amari, "Natural gradient learning for over-and under-complete bases in ica," *Neural Computation*, vol. 11, no. 8, pp. 1875–1883, 1999.

[2] L. Van Der Maaten, E. Postma, and J. Van den Herik, "Dimensionality reduction: a comparative," *J Mach Learn Res*, vol. 10, pp. 66–71, 2009.

[3] M. Gashler, D. Ventura, and T. R. Martinez, "Iterative non-linear dimensionality reduction with manifold sculpting." in *NIPS*, vol. 8, 2007, pp. 513–520.

[4] M. Zaharia, M. Chowdhury *et al.*, "Spark: cluster computing with working sets." *HotCloud*, vol. 10, pp. 10–10, 2010.

[5] ——, "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*. USENIX Association, 2012, pp. 2–2.

[6] Apache Software Foundation. (2016) Apache Spark - lightning-fast cluster computing. [Online]. Available: http://spark.apache.org/

[7] J. Shi, Y. Qiu *et al.*, "Clash of the titans: MapReduce vs. Spark for large scale data analytics," *Proceedings of the VLDB Endowment*, vol. 8, no. 13, pp. 2110–2121, 2015.

[8] J. E. Gonzalez, R. S. Xin *et al.*, "GraphX: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 599–613.

[9] X. Meng, J. Bradley *et al.*, "MLlib: Machine learning in Apache Spark," *JMLR*, vol. 17, no. 34, pp. 1–7, 2016.

[10] M. Zaharia, T. Das *et al.*, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*. USENIX Association, 2012, pp. 10–10.

[11] S. Baltagi, "Flink vs. Spark," 2015, Flink Forward. [Online]. Available: http://www.slideshare.net/sbaltagi/flink-vs-spark

[12] A. Alexandrov, R. Bergmann *et al.*, "The Stratosphere platform for big data analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, 2014. [Online]. Available: http://dx.doi.org/10.1007/s00778-014-0357-y

[13] Apache Software Foundation. (2016) Apache Flink: Scalable batch and stream data processing. [Online]. Available: http://flink.apache.org/

[14] P. Carbone, A. Katsifodimos *et al.*, "Apache Flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 38, no. 4, pp. 28–38, 2015.

[15] M. A. Lopez, A. Lobato, and O. Duarte, "A performance comparison of open-source stream processing platforms," in *IEEE Global Communications Conference (Globecom), Washington, USA*, 2016.

[16] Apache Software Foundation. Apache Flink 1.1.3 documentation: Concepts. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.1/concepts/concepts.html

[17] S. Ewen, K. Tzoumas *et al.*, "Spinning fast iterative data flows," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1268–1279, 2012.

[18] C. Schepler, R. Metzger *et al.* Python interface for new API (Map/Reduce). [Online]. Available: https://issues.apache.org/jira/browse/FLINK-671

[19] C. Schepler, A. Krettek, and M. Michels. Rework operator distribution. [Online]. Available: https://issues.apache.org/jira/browse/FLINK-1927

[20] X. Li, M. Makkie *et al.*, "Scalable fast rank-1 dictionary learning for fMRI big data analysis," in *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2016.

[21] J. Lv, X. Jiang *et al.*, "Holistic atlases of functional networks and interactions reveal reciprocal organizational architecture of cortical function," *IEEE Trans. Biomed. Eng.*, vol. 62, no. 4, pp. 1120–1131, April 2015.

[22] H. Lee, A. Battle *et al.*, "Efficient sparse coding algorithms," in *Advances in neural information processing systems*, 2006, pp. 801–808.

[23] J. Freeman, "Open source tools for large-scale neuroscience," *Current Opinion in Neurobiology*, vol. 32, pp. 156 – 163, 2015, large-Scale Recording Technology (32). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0959438815000756

[24] G. Mon and C. Schepler. Add Aggregation support to Python API. [Online]. Available: https://issues.apache.org/jira/browse/FLINK-4017

[25] S. Quinn, G. Mon, and C. Schepler. zipWithIndex in Python API. [Online]. Available: https://issues.apache.org/jira/browse/FLINK-3626

[26] C. Foster. Use Apache Flink on Amazon EMR. [Online]. Available: https://aws.amazon.com/blogs/big-data/use-apache-flink-on-amazon-emr/