TECHNICAL UNIVERSITY OF CRETE
ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT
SOFTWARE TECHNOLOGY AND NETWORK APPLICATIONS LAB

Outlier Detection Using Spark Streaming

by

Kyriakos Psarakis

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DIPLOMA OF
ELECTRICAL AND COMPUTER ENGINEERING

December 2017

Abstract

Data is continuously being generated from sources such as machines, network traffic, sensor networks, etc. Timely and accurate detection of outliers in massive data streams has important applications such as in preventing machine failures, intrusion detection, and financial fraud detection. In this thesis, we implement an outlier detection algorithm [1] inside the Spark Streaming environment that, makes only one pass over the data while utilizing limited storage. We chose the Spark Streaming environment because it offers scalable, high-throughput, fault-tolerant stream processing of live data streams. The algorithm adapts ideas from matrix sketching to maintain a set of few orthogonal vectors that form a good approximate basis for all the observed data. Using this constructed orthogonal basis, outliers in new incoming data are detected based on a simple reconstruction error test. Additionally, we have implemented two methods for updating the orthogonal vectors one deterministic and one randomized to further speedup the algorithm with a small cost to accuracy.

Acknowledgements

First of all I would like to thank my supervisor Prof. Antonios Deligiannakis for his guidance throughout this work.

The members of the committee for giving their time to evaluate this work.

And especially my friends and family for their support.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Outlier Detection

In the past decade we faced a major influx in data being generated by various sources such as network traffic, the Internet of things, social media, sensor networks and so on. Consequently, the need to detect data points that represent an anomaly is of paramount importance, especially in critical applications like intrusion detection, fraud detection, machine failure prevention and many others. However, if the detection of those anomalies is not made in a real time fashion many of those critical applications would fail.

There are three major categories of outliers :

- Point: A single instance of data is anomalous if it's too far off from the rest.
- Contextual: The abnormality is context specific.
- Collective: A set of data instances collectively helps in detecting anomalies.

"An outlier is an observation which deviates so much from the other observations as to arouse suspicions that it was generated by a different mechanism"

- D. M. Hawkins Identification of outliers 1980

1.2 Data Streams

Data Streams are sequences of data elements made available over time. Those data are generated continuously by numerous data sources, which typically send in the data records simultaneously, and in small sizes. The before mentioned data could be processed sequentially and incrementally on a record-by-record basis or over sliding time windows, and used for a wide variety of analytics.

There are many examples where streaming data processing is useful. A few of them are projected here as listed in Amazons AWS website :

- Sensors in transportation vehicles, industrial equipment, and farm machinery send data to a streaming application. The application monitors performance, detects any potential defects in advance, and places a spare part order automatically preventing equipment down time.
- A financial institution tracks changes in the stock market in real time, computes value-at-risk, and automatically re-balances portfolios based on stock price movements.
- A real-estate website tracks a subset of data from consumers mobile devices and makes real-time property recommendations of properties to visit based on their geo-location.
- A media publisher streams billions of clickstream records from its online properties, aggregates and enriches the data with demographic information about users, and optimizes content placement on its site, delivering relevancy and better experience to its audience.
- An online gaming company collects streaming data about player-game interactions, and feeds the data into its gaming platform. It then analyzes the data in real-time, offers incentives and dynamic experiences to engage its players.

An other very popular solution for big data analytics is Batch processing. Batch processing can be used to compute arbitrary queries over different sets of data. It usually computes results that are derived from all the data it encompasses, and enables deep analysis of big data sets. On the contrary, stream processing requires ingesting a sequence of data, and incrementally updating metrics, reports, and summary statistics in response to each arriving data record. It is better suited for real-time monitoring and response functions. Moreover, its fair to state that stream processing requires latency in the order of milliseconds in order to achieve near real-time performance, which in some cases could be proven a difficult task.

Furthermore, some challenges arise by working with Streaming Data. The processing requires two layers: a storage layer and a processing layer. The storage layer needs to support record ordering and strong consistency to enable fast, inexpensive, and re-playable reads and writes of large streams of data. The processing layer is responsible for consuming data from the storage layer, running computations on that data, and then notifying the storage layer to delete data that is no longer needed. The user also has to plan for scalability, data durability, and fault tolerance in both the storage and processing layers.

Chapter 2

Apache Spark

2.1 The Spark Framework

Apache Spark [4] alongside with Hadoop and Storm is one of the most popular frameworks for large scale data prosessing. The concept was firstly conceived at UC Berkeley's AMPLab in 2009, then it was donated to the Apache Software Foundation and became a Top Level Apache project in February 2014. In addition, Spark powers a stack of libraries including SQL and DataFrames, MLlib for machine learning, GraphX, and Spark Streaming. Giving the programmer tools in order to solve various problems from different scientific areas. Furthermore, it is proven that Spark is at least 10 times faster than Hadoop MapReduce. Spark achieves it by utilizing its advanced DAG execution engine that supports acyclic data flow and in-memory computing.

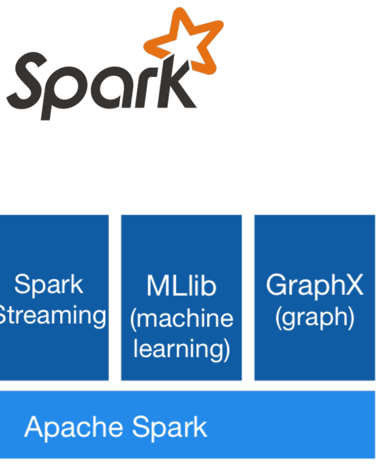


Figure 2.1: Spark

2.1.1 Spark Core

Spark Core [4] is the base of the whole project. It provides distributed task dispatching, scheduling, and basic I/O functionalities. Spark uses a specialized fundamental data structure known as Resilient Distributed Datasets (RDD) that is a logical collection of data partitioned across machines. RDDs can be created in two ways: one is by referencing datasets in external storage systems and second, is by applying transformations on existing RDDs. At high level, when any action is called on the RDD, Spark creates a Directed Acyclic Graph (DAG) and submits it to the DAG scheduler. The DAG scheduler divides operators into stages of tasks. A stage is comprised of tasks based on partitions of the input data. The DAG scheduler pipelines operators together. The final result of a DAG scheduler is a set of stages. At high level, there are two transformations that can be applied onto the RDDs, namely narrow transformation and wide transformation. Wide transformations basically result in stage boundaries. Narrow transformations do not require the data to be shuffled across the partitions. In addition, they will be grouped (pipelined) together into a single stage. The DAG scheduler will then submit the stages into the task scheduler. The number of tasks submitted depends on the number of partitions. On the contrary wide transformation requires the data to be shuffled. An other feature that Spark utilizes, is lazy evaluation, which in Spark means that the execution will not start until an action is triggered. In Spark, the picture of lazy evaluation comes when Spark transformations occur. Transformations are lazy in nature, which means that, when we call some operation on an RDD, it does not execute immediately. Spark maintains the record of which operation is being called through the DAG. We can think Sparks RDD as the data, that we built up through transformations. Since transformations are lazy in nature, we can execute an operation at any time by calling an action on an RDD. In Spark, driver program loads the code to the cluster. When the code executes after every operation, the task will be time and memory consuming, since each time data goes to the cluster for evaluation.

There are some benefits of Lazy evaluation in Apache Spark:

- Increases Manageability: By lazy evaluation, users can organize their Apache Spark program into smaller operations. It reduces the number of passes on data by grouping operations.
- Saves Computation and increases Speed: Spark Lazy Evaluation plays a key role in saving calculation overhead. Since only necessary values get compute. It saves the trip between driver and cluster, thus speeds up the process.
- Reduces Complexities: The two main complexities of any operation are time and space complexity. Apache Sparks lazy evaluation can overcome both. Since it does not execute every operation as a result, time gets saved. The action is triggered only when the data is required so, it reduces overhead.
- Optimization: It provides optimization by reducing the number of queries.

One of the most important capabilities in Spark is persisting (or caching) a dataset in memory across operations. When an RDD is persisted, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset or datasets derived from it. This allows future actions to be much faster. Caching is a key tool for iterative algorithms and fast interactive use. Sparks cache is fault-tolerant if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it. In addition, each persisted RDD can be stored using a different storage level.

2.1.2 Spark Streaming

Spark Streaming [5] is one of the primary libraries of Spark, it leverages Spark Core's fast scheduling capability to perform streaming analytics. It works by ingesting data in batches and performing RDD transformations on those batches of data. This design enables the same set of application code written for batch analytics to be used in streaming analytics. However, this convenience comes with the penalty of latency equal to the batch duration. The use of batches makes Spark Streaming unique, the reason being, that the other streaming frameworks use event by event processing. Spark Streaming can consume data from various sources like HDFS, Kafka, Flume, Twitter, ZeroMQ, Kinesis, TCP/IP sockets, it even has an API that allows construction of custom sources and with that it a versatile solution.

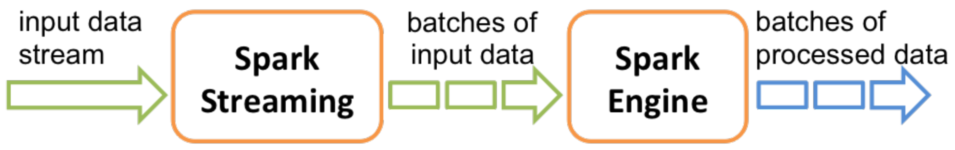


Figure 2.2: Spark Streaming

Discretized Stream or DStream is the basic abstraction provided by Spark Streaming. It represents a continuous stream of data, either the input data stream received from source, or the processed data stream generated by transforming the input stream.

Internally, a DStream is represented by a continuous series of RDDs. Any operation applied on a DStream translates to operations on the underlying RDDs. The RDD transformations are computed by the Spark engine.

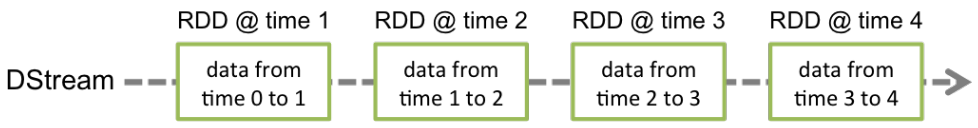


Figure 2.3: Discretized Streams

2.2 Cluster Architecture

Spark applications run as independent sets of processes on a cluster [6], coordinated by the SparkContext object in the main program (Driver). Specifically, to run on a cluster, the SparkContext can connect to several types of cluster managers, which allocate resources across applications. Once connected, Spark acquires executors on nodes in the cluster (Slaves), which are processes that run computations and store data for your application. Next, it sends your application code to the executors. Finally, SparkContext sends tasks to the executors to run. Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads. This has the benefit of isolating applications from each other, on both the scheduling side and executor side. However, it also means that data cannot be shared across different Spark applications without writing it to an external storage system. Spark is agnostic to the underlying cluster manager. As long as it can acquire executor processes, and these communicate with each other, it is relatively easy to run it even on a cluster manager that also supports other applications. The driver program must listen for and accept incoming connections from its executors throughout its lifetime. As such, the driver program must be network addressable from the worker nodes. Moreover, as we explained in 2.1.1 the final result of a DAG scheduler is a set of stages. Those Stages are passed on to the Task Scheduler to be launched via the cluster manager. The task scheduler doesn't know about dependencies of the stages. Furthermore the tasks are executed on the Slaves.

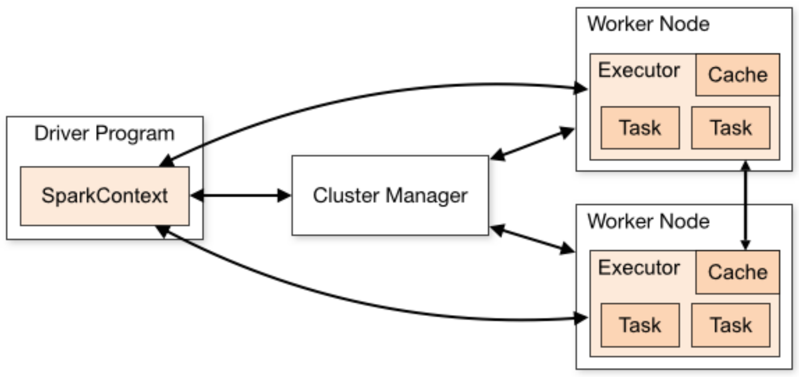


Figure 2.4: Cluster Overview

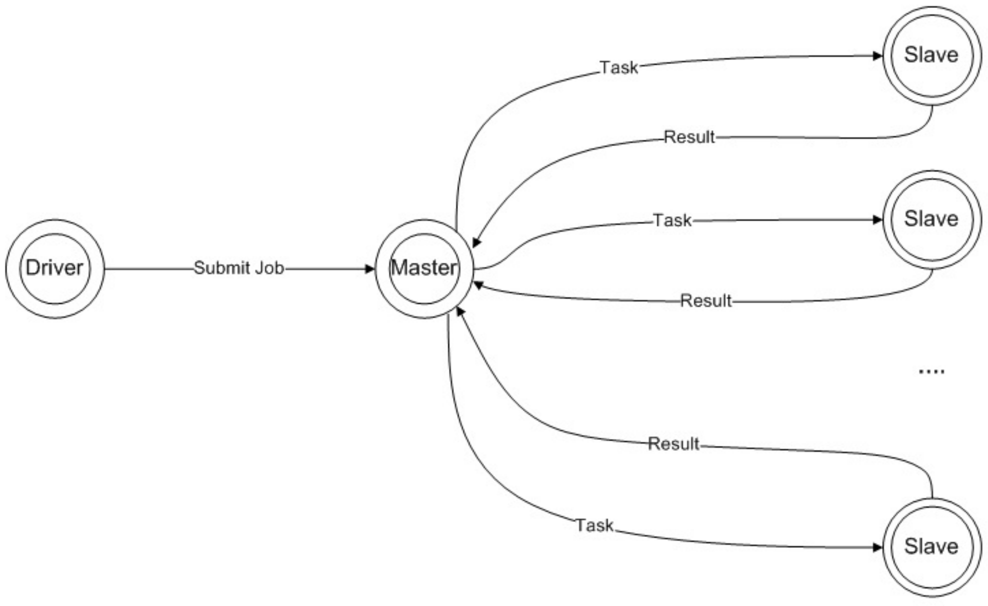


Figure 2.5: Cluster Node Architecture

Chapter 3

Outlier Detection in Data Streams Using Randomized Matrix Sketching

3.1 Problem Formulation

In their work Hao Huang and Shiva Kasiviswanathan [1] create two streaming anomaly detection approaches. These approaches are based in subspace-based anomaly detection, operate by first constructing a low-rank matrix approximation of the input and then the projection of a new data point onto this low-rank matrix is used for deciding whether the point is anomalous or not. This general idea can be utilized to construct a simple anomaly detection framework in a streaming fashion. At time t,let us assume that there is a low-rank matrix U that can linearly represent well all the identified non-anomalous datapoints till time $t-1$. For a new data point y arriving at time t, if there does not exist a good representation of y using U , then y does not lie close to the space of non-anomalous data points, and y could be an anomaly. After identifying the non-anomalous points, the low-rank matrix is updated to capture the insights from these non-anomalous points.
In their paper, they use streaming matrix sketching, to efficiently store and update a low-rank matrix (with orthogonal columns) that can linearly represent well over time the identified non-anomalous data points. For matrix sketching, they build upon and improve an idea proposed by Liberty [7] (Frequent Directions). But, since in the current problem setting more than one point could arrive at each timestep, they extend Frequent Directions to efficiently deal with this scenario with a speedup over Frequent Directions that

is almost linear to the number of data points handled at each timestep. In addition, a second approach further improves this computational efficiency by combining sketching with ideas from the theory of randomized low-rank matrix approximations. The computational efficiency gains of the randomized approach over the deterministic approach come at a cost of a small loss in the sketching performance.
Moreover, theoretical analysis of both approaches to show that, under some reasonable assumptions, they attain almost the same performance as a global approach that uses the entire data history at every timestep. The latter requires repeated and costly singular value decompositions over an ever increasing sized data matrix, while the proposed algorithms operate in a true streaming setting utilizing limited storage. These results are obtained by generalizing the analysis of Frequent Directions from [7–9] and by carefully combining it with recent spectral results in matrix perturbation and randomized low-rank matrix approximation theories.

The proposed anomaly detection algorithms have the following features:

- Identifying anomalies in close to real time, ensuring that the detection keeps up with the rate of data collection.
- Require only one pass for each data point.
- Are space-efficient and require only a small amount of bookkeeping space.
- Operate in an unsupervised setting, but regular model updates allow them to still easily adapt to unseen normal patterns in the data.

Additionally, their algorithms are significantly more time and space efficient, compared to other popular scalable anomaly detection algorithms.

3.1.1 Streaming Outlier Detection

It is assumed that the data arrives in batches in a streaming fashion. Each data point has a time stamp that indicates when it arrives.
Let $\{Y_t \in \Re^{m\times n_t},\ t = 1,\ 2,\ \dots\}$ denote a sequence of streaming data matrices, where Y_t represents the data points arriving at time t. Here m is the size of the feature space, and $n_t \geq 1$ is the number of data points arriving at time t. Y_t is normalized such that each column has a unit L_2-norm. Under this setup, the goal of streaming anomaly detection is to identify anomalous data points in Y_t at every time t.

The algorithm is based on maintaining a low-rank matrix with orthogonal columns, at every time t, that can linearly reconstruct well the entire prior (till time $t-1$) non-anomalous datapoints that the algorithm has identified. In other words, it identifies a small set of (orthogonal) basis vectors that can represent well all the prior non-anomalous data points. At time t, a new point $y_i \in Y_t$ is marked as anomalous if it cannot be reconstructed well linearly using these basis vectors.
Let $N_{[t-1]} = [N_1, \dots, N_{t-1}]$ be the set of all datapoints (columns) in $Y_{[t-1]} = [Y_1, \dots, Y_{t-1}]$ that the algorithm has identified as non-anomalous, with N_i denoting the set of datapoints in Y_i identified as non-anomalous. Consider the rank-k approximation of $N_{[t-1]}$.

$$N_{[t-1]_k} \;=\; \mathrm{SVD}_k(N_{[t-1]}) = U_{(t-1)_k}\, \Sigma_{U_{(t-1)_k}} V^T_{U_{(t-1)_k}}$$

First observation, is that $U_{(t-1)_k}$ is a good rank-k matrix to linearly represent all the points in $N_{[t-1]}$. This follows from the observation that by setting $X = \Sigma_{U_{(t-1)_k}} V^T_{U_{(t-1)_k}}$

$$\sum_{\mathbf{y}_i \in N_{[t-1]}} \min_{\mathbf{x}_j} \|\mathbf{y}_j - U_{(t-1)_k}\mathbf{x}_j\|^2 = \min_X \|N_{[t-1]} - U_{(t-1)_k}X\|_F^2 \leq \|N_{[t-1]} - N_{[t-1]_k}\|_F^2$$

In many practical scenarios, most of the mass from $N_{[t-1]}$ would be in its top k singular values, resulting in $\|N_{[t-1]} - N_{[t-1]_k}\|_F$ being small.

We can now use $U_{(t-1)_k}$ to detect anomalies in Y_t by following a simple approach. Since $U_{(t-1)_k}$ is a good basis to linearly reconstruct all the observed non-anomalous points in $Y_{[t-1]}$, we can use it to test whether a point $y_i \in Y_t$ is close to space of non-anomalous points or not. This can be easily achieved by solving the following simple least-squares problem:

$$\min_{\mathbf{x}} \|\mathbf{y}_i - U_{(t-1)_k}\mathbf{x}\|$$

As the columns of $U_{(t-1)_k}$ are orthogonal to each other, this least-squares problem has a simple closed-form solution

$$\mathbf{x}^* = (U^T_{(t-1)_k} U_{(t-1)_k})^{-1} U^T_{(t-1)_k}\mathbf{y}_i$$

$$\implies \mathbf{x}^* = U^T_{(t-1)_k}\mathbf{y}_i$$

The objective value of $\mathbf{x}^*$ is used as the anomaly score to decide if $\mathbf{y}_i$ is anomalous or not, with larger objective value denoting anomalies. In other words, the anomaly score for $\mathbf{y}_i$ is :

$$\boxed{\|(\mathbb{I}_m - U_{(t-1)_k} U^T_{(t-1)_k})\mathbf{y}_i\|} \tag{3.1}$$

Note that this anomaly score is exactly the length of the orthogonal projection of $\mathbf{y}_i$ onto the orthogonal complement $U_{(t-1)_k}$.

3.2 AnomDetect

Algorithm 1 AnomDetect (algorithm for detecting anomalies at time t)

```
Input: $Y_t \in \Re^{m \times n_t},\ U_{(t-1)_k} \in \Re^{m \times k},\ \zeta \in \Re$

 1: $N_t \leftarrow [],\ \bar{N}_t \leftarrow []$
 2: for $\forall\ \mathbf{y}_i \in\ Y_t$ do
 3:     $a_i \leftarrow \|(\mathbb{I}_m - U_{(t-1)_k} U_{(t-1)_k}^T \mathbf{y}_i\|$
 4:     if $a_i \leq \zeta$ then
 5:         $N_t \leftarrow [N_t, \mathbf{y}_i]$
 6:     end if
 7:     if $a_i > \zeta$ then
 8:         $\bar{N}_t \leftarrow [\bar{N}_t, \mathbf{y}_i]$
 9:     end if
10: end for
11: $U_{t_k} \leftarrow Update(N_t)$
12: return : $N_t,\ \bar{N}_t,\ U_{t_k}$
```

In Algorithm AnomDetect, a simple prototype procedure for anomaly detection is presented based on maintaining the top-k left singular vectors of the streaming data. Since we have normalized all input points ($\mathbf{y}_i$'s) to have unit L_2-length, the objective values (a_i's) for all points are in the same scale. The Algorithm AnomDetect alternates between an anomaly detection and singular vector updating step. In the anomaly detection step, the past basis matrix is used to detect anomalies among the new incoming points by thresholding on the objective value of the least-squares problem 3.1. N_t is set of non-anomalous points in Y_t identified by the algorithm at time t.

The main challenge is in updating the singular vectors. To start off, an inefficient (baseline) approach based on global SVD updates is presented, and then ideas from matrix sketching and randomized low-rank matrix approximations are used to speedup the updating without any significant loss in the detection quality.

At the beginning of the algorithm (when t = 1), in order to initialize U_{tk} , a small training set of non-anomalous data is needed to construct N_1. Alternatively, if obtaining the said training set is not possible, a small set of

data could be collected and then any good unsupervised anomaly detection scheme could be applied to label the data. Consequently, N_1 would be constructed through it. Since this occurs only in the initialization process, it does not heavily influence the algorithm.
The Threshold (ζ) is chosen based on the distribution of the anomaly scores or if instead it is assumed that the percentage of anomalies among all samples is known a priori ($p\%$). At each timestep of the streaming process, the top $p\%$ data-points are marked as anomalies based on their anomaly score.

3.2.1 Concept Drift

An important feature of many real-world data streams, is concept drift, which means that the characteristics of the data can change over time. Algorithms for handling concept drift need to employ regular model updates as new data arrives. In the above algorithm, concept-drift is well captured, as the underlying model is updated at each time t with all the identified non-anomalous points till time $t - 1$.

3.3 Updating the left-singular vectors

3.3.1 Baseline

The simplest way of correctly updating the singular vectors is to simply regenerate them from the globally collected sample set $N_{[t]} = [N_{[t-1]}, N_t]$.
A more mature approach for incrementally and correctly generating the singular vectors of a matrix(with addition of new columns) is based on the following lemma from [10] is outlined in Algorithm GlobalUpdate.

$$\begin{aligned} R &= [P, Q], \\ SVD(P) &= U_P \Sigma_P V_P^T, \\ F &= [\Sigma_P, U_P^T Q], \\ SVD(F) &= U_F \Sigma_F V_F^T \\ U_R &= U_P U_F, \\ \Sigma_R &= \Sigma_F, \end{aligned}$$

Algorithm 2 GlobalUpdate (global update of the left singular vectors at time t)

Input: $U_{t-1},\ \Sigma_{t-1}\ and\ N_t \in \Re^{m \times n_t}$

```
1: $F \leftarrow [\Sigma_{t-1}, U_{t-1}^T N_t]$
2: $U_F \Sigma_F V_F^T \leftarrow SVD(F)$
3: $U_t \leftarrow U_{t-1} U_F$
4: $\Sigma_t \leftarrow \Sigma_F$
5: $U_{tk} \leftarrow [\mathbf{u}_1, \dots, \mathbf{u}_k]$
6: return : $U_t,\ \Sigma_t,\ U_{t_k}$
```

At time t, Algorithm GlobalUpdate takes $O(mn_{[t]})$ space and $O(\min(m^2 n_{[t]}, mn_{[t]}^2))$ time, where $n_{[t]}$ denotes the number of columns (data points) in the matrix $N_{[t]}$. It is obvious, that a significant disadvantage of Algorithm GlobalUpdate is, that both its computational and memory requirement increases with time. This problem is overcomed by using randomized matrix sketching, by that it is proved that while gaining in computational efficiency, the sketching approach still produces a good approximation to the top-k left singular vectors of $N_{[t]}$ at every time t.

3.3.2 Randomised Matrix Sketching

The algorithm for the Matrix Sketching is derived from Liberty's algorithm Frequent Directions [7] with Ghashami's low-rank matrix approximation extension [11] and some adjustments in order to add all the n_t columns simultaneously instead of one at a time.

Frequent Directions

The inputs to the algorithm are an input data matrix $Z \in \Re^{m \times n_t}$ and a sketch matrix $S \in \Re^{m \times l}$. In each iteration, one column of Z is processed by the algorithm and the algorithm iteratively updates the matrix S such that for any unit vector $x \in \Re^m$, $\|Z^T x\|^2 - \|S^T x\|^2 \leq 2\|Z\|_F^2/l$. In other words, the sketched matrix has the guarantee that for any direction it is close to the input matrix.
The computation time for this algorithm is dominated by a SVD computation in each iteration, which gives it a total running time of $O(mnl^2)$ with $m \geq l$.

Frequent Directions with low-rank matrix approximation

This algorithm instead of S, it returns S_k, the rank-k approximation of S, their main result shows that $\|Z_k\|_F^2 - \|S_k\|_F^2 \leq k/(l-k)\|Z - Z_k\|_F^2$.

Updating Sketch with multiple columns

In contrast to Frequent Directions, where the sketch is updated after addition of every new column, it is desired that the sketch could be updated after addition of $n_t \geq 1$ columns. In the current problem setup, at timestep t with $n_t \geq 1$ new columns, using Frequent Directions for sketching would take $O(mnt^2)$ time. However, by an elegant trick of adding all the n_t columns simultaneously and performing a low-rank SVD, the running time is reduced to $O(\max\{mn_tl, ml^2\})$ with $m \geq l$, without any loss in the sketching performance and resulting in a running time $O(\min\{n_t, l\})$ times better than the running time $O(mnt^2)$ of Frequent Directions.

Deterministic Matrix Sketching Update

These three major ideas form the basis for the first sketching algorithm DetUpdate.

Algorithm 3 DetUpdate (deterministic streaming update of the left singular vectors at time t)

```
Input: $N_t \in \Re^{m \times n_t},\ k \leq l \ and \ B_{[t-1]} \in \Re^{m \times l}$
1: $D_t \leftarrow [B_{[t-1]}, N_t]$
2: $U_{t_l} \Sigma_{t_l} V_{t_l}^T \leftarrow SVD(D_t)$
3: $U_{t_k} \leftarrow [\mathbf{u}_1, \ldots, \mathbf{u}_k]$
4: $\Sigma_{t_l}^{(trunc)} \leftarrow diag(\sqrt{\sigma_{t_1}^2 - \sigma_{t_l}^2},\ \ldots,\ \sqrt{\sigma_{t_{l-1}}^2 - \sigma_{t_l}^2})$
5: $B_t \leftarrow U_{t_l} \Sigma_{t_l}^{(trunc)}$
6: return : $B_t \ and \ U_{t_k}$
```

The overall space complexity of Algorithm DetUpdate is $O(m \max_t \{n_t\} + ml)$, therefore, the additional space overhead for the algorithm is only $O(ml)$ ($O(m \max_t \{n_t\})$) space is needed to just read and store the input matrices). In Algorithm DetUpdate, the matrix B_t is a sketch of the matrix $N_{[t]} = [N_1,\ \ldots,\ N_t]$. Let B_{t_k} be the rank-k approximation of B_t .

In addition it is proven that :

$$\|N_{[t]_k}\|_F^2 - \|B_{t_k}\|_F^2 \leq k/(l-k)\|N_{[t]} - N_{[t]_k}\|_F^2$$

This dictates that, while being computationally more efficient, the sketch matrices generated by Algorithm DetUpdate have the same guarantees as that generated by Frequent Directions.

Randomized Matrix Sketching Update

A major improvement in the previous algorithm (DetUpdate) derieved from the observation that the low-rank SVD (Step 2) can be replaced by a randomized low-rank matrix approximation. This leads to even greater computational savings however, this comes at a cost of slightly higher error in sketching as compared to DetUpdate. In this algorithm a Randomized low-rank matrix approximation technique suggested by Halko [11] is adapted, which is based on combining a randomized pre-processing step (multiplying by a random matrix and QR decomposition) along with a simple post-processing step (eigenvalue decomposition of a small matrix). The complete sketching procedure is described in Algorithm RandUpdate.

Algorithm 4 RandUpdate (randomized streaming update of the left singular vectors at time t)

```
Input: $N_t \in \Re^{m \times n_t},\ k \leq l \ and\ E_{[t-1]} \in \Re^{m \times l}$

 1: $M_t \leftarrow [E_{[t-1]}, N_t]$
 2: $r \leftarrow 100l$
 3: $Generate\ an\ m \times r\ random\ Gaussian\ matrix\ \Omega$
 4: $Y \leftarrow M_t M_t^T \Omega$
 5: $QR \leftarrow QR(Y)$
 6: $A_t \Sigma_t A_t^T \leftarrow EIG(Q^T M_t M_t^T Q)$
 7: $U_t \leftarrow QA_t$
 8: $U_{t_l} \leftarrow [\mathbf{u}_1, \dots, \mathbf{u}_l]$
 9: $\Sigma_{t_l}^{(trunc)} \leftarrow diag(\sqrt{\sigma_{t_1}^2 - \sigma_{t_l}^2},\ \dots,\ \sqrt{\sigma_{t_{l-1}}^2 - \sigma_{t_l}^2})$
10: $E_t \leftarrow U_{t_l} \Sigma_{t_l}^{(trunc)}$
11: $U_{t_k} \leftarrow [\mathbf{u}_1, \dots, \mathbf{u}_k]$
12: return : $E_t\ and\ U_{t_k}$
```

At timestep t, Algorithm RandUpdate takes $O(lT_{mult} + (m + n_t)l^2)$ time with $m \geq l$, where T_{mult} denotes the cost of matrix-vector multiplication with the input matrix M_t. The matrix-vector multiplication is a well-studied topic with numerous known efficient sequential/parallel algorithms. Note that this running time is smaller than that of Algorithm DetUpdate. which at timestep t takes $O(\max\{mn_t l, ml^2\})$ time. The overall space complexity of Algorithm RandUpdate is $O(m \max_t \{n_t\} + mr)$, therefore, the additional space overhead for the algorithm is again only $O(mr) = O(ml)$.

3.3.3 Performance comparison

In their work Hao Huang and Shiva Kasiviswanathan [1] provide proof that their Algorithms DetUpdate and RandUpdate produces good matrix sketches at every time t.

Variables:

$\breve{U}_{t_k}$	RandUpdate rank-k singular vectors
$\hat{U}_{t_k}$	True rank-k singular vectors
κ	$\sigma_1(N_{[t]})/\sigma_k(N_{[t]})$, where $\sigma_1(N_{[t]}) \geq \cdots \geq \sigma_m(N_{[t]})$ are the singular values of $N_{[t]}$
L	$\min_{i \neq j} \lvert\lambda_i - \lambda_j\rvert > 0$, where λ_i be the ith eigenvalue of $N_{[t]}N_{[t]}^T$

At first, they compare the anomaly scores generated by using either $\hat{U}_{t_k}$ or $\breve{U}_{t_k}$ in Algorithm AnomDetect. The following theorem shows that under some reasonable assumptions and settings of parameters, the Algorithm RandUpdate can be used efficiently for singular value updating in the Algorithm AnomDetect and still obtain anomaly scores that are close to that obtained using the true singular vectors ($\hat{U}_{t_k}$).

Let $N_1, \ldots, N_t$ be a sequence of matrices with $N_{[t]} = [N_1, \ldots, N_t]$. Let $N_{[t]_k} = \hat{U}_{t_k}\hat{\Sigma}_{t_k}\hat{V}_{t_k}^T$ be the best rank-k approximation to $N_{[t]}$. Then for any unit vector $\mathbf{y} \in \Re^m$, $\breve{U}_{t_k}$, with probability at least $1 - t \cdot 6e^{-99l}$, satisfies:

$$|\min_{x \in \Re^k} \|\mathbf{y} - \hat{U}_{t_k}\mathbf{x}\| - \|\mathbf{y} - \breve{U}_{t_k}\mathbf{x}\|| \leq \frac{\sqrt{2}L}{\sqrt{L + 8\kappa^2\|N_{[t]}\|^2 \sqrt[4]{L^2 + 16\kappa^4\|N_{[t]}\|^4}}}$$

The above bound on the difference in anomaly scores is an increasing function in L.

The same bound satisfies the algorithm DetUpdate. The only difference being that the error due to randomization is zero because it computes the exact low-rank matrix at each time step. Therefore, the requirement on l is slightly weaker.

In addition, they state that the bounds on l that are presented in their work should be treated as existential results, as setting these bounds are tricky. Consequently, they recommend setting $l \approx \sqrt{m}$, which suffices to get good results for both Algorithms DetUpdate and RandUpdate. But, both can be used with any l within $k \leq l \leq m$, the above bounds on l are only to show theoretically that their performances are similar to using global singular value decomposition updates in Algorithm AnomDetect.

Chapter 4

Spark Streaming Solution

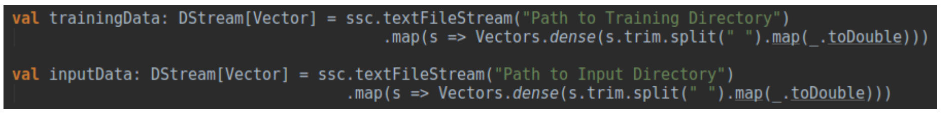
4.1 Introduction

In this chapter we present our implementation of the above algorithms in the Apache Spark Streaming framework. We used the Scala programming language featuring ScalaNLP's Breeze library for Matrix calculations, decompositions and many helpful tools for linear algebraic problems [12]. We decided to implement our solution with Spark Streaming, reason being that it allows us to receive and process the data in batches, with low latency and ease of implementation. We chose to code in Scala [13] instead of Java or Python because Spark core is programmed in Scala thus, the Scala RDDs are being processed 2 times faster than the others. We picked Breeze instead of MLlib because the later as of now (2017) is also based on Breeze, it imposes some limitations on the Matrix calculations and in addition it does not fully support some matrix decompositions that we need in this work. Finally, we use HDFS [14] as the file system for the frameworks input and output Stream mainly because for the ease of implementation (in some professional cases Amazon's S3 is a better option).

Figure 4.1: Technologies used in this work

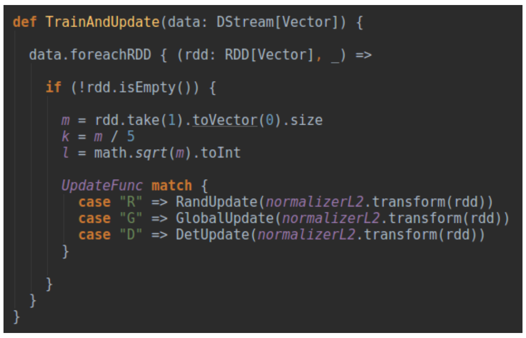
4.2 Anomaly Detection Model

It is fair to state that we took notice of how the Spark MLlib's developers create a Streaming function from their github repository. Hence, we create two Discretized Streams (DStreams), one for the training set of data and another for the actual data that we aim to monitor. In our experiments, the data are being inserted to HDFS folders and with the Spark API we monitor those folders. The Spark API detects changes to the said folders and it streams those changes to the Spark Framework creating the DStream.

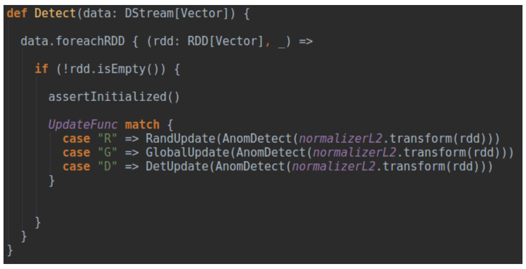


```
val trainingData: DStream[Vector] = ssc.textFileStream("Path to Training Directory")
                                    .map(s => Vectors.dense(s.trim.split(" ").map(_.toDouble)))

val inputData: DStream[Vector] = ssc.textFileStream("Path to Input Directory")
                                 .map(s => Vectors.dense(s.trim.split(" ").map(_.toDouble)))
```

At first, from the training data we initialize the Sketches, the singular vectors, the size of the feature space of the dataset (m), the rank of the approximated matrix ($k = m/5$) and the size of the sketch matrix ($l = \sqrt{m}$). Then, when we have new data in the input stream, AnomDetect is being called, which creates a score for each data point. The next step is to save the outliers (higher anomaly score than the current threshold) and the Normal points. Finally, we update (with one of the three methods) the sketch and the rank-k singular vectors based on the Normal points of the current time step.



```
def TrainAndUpdate(data: DStream[Vector]) {

  data.foreachRDD { (rdd: RDD[Vector], _) =>

    if (!rdd.isEmpty()) {

      m = rdd.take(1).toVector(0).size
      k = m / 5
      l = math.sqrt(m).toInt

      UpdateFunc match {
        case "R" => RandUpdate(normalizerL2.transform(rdd))
        case "G" => GlobalUpdate(normalizerL2.transform(rdd))
        case "D" => DetUpdate(normalizerL2.transform(rdd))
      }

    }
  }
}
```



```
def Detect(data: DStream[Vector]) {

  data.foreachRDD { (rdd: RDD[Vector], _) =>

    if (!rdd.isEmpty()) {

      assertInitialized()

      UpdateFunc match {
        case "R" => RandUpdate(AnomDetect(normalizerL2.transform(rdd)))
        case "G" => GlobalUpdate(AnomDetect(normalizerL2.transform(rdd)))
        case "D" => DetUpdate(AnomDetect(normalizerL2.transform(rdd)))
      }


    }
  }
}
```

The above Model is pictured in the next page.

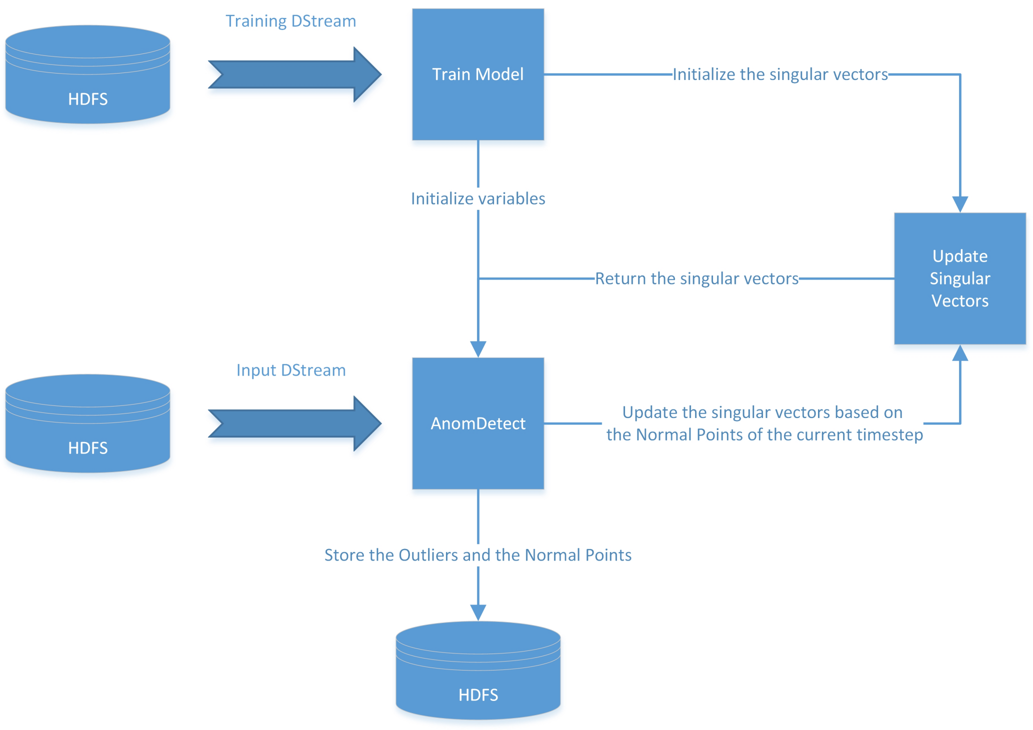
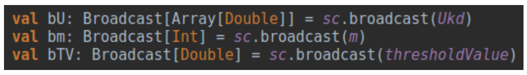

Training DStream
HDFS
Train Model
Initialize the singular vectors
Initialize variables
Update Singular Vectors
Return the singular vectors
Input DStream
HDFS
AnomDetect
Update the singular vectors based on the Normal Points of the current timestep
Store the Outliers and the Normal Points
HDFS

Figure 4.2: Spark Model

4.3 Actions and Transformations

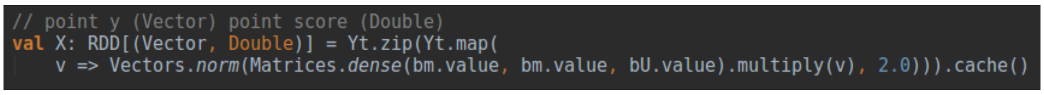
4.3.1 Spark Streaming AnomDetect

In this section we present the actions and transformations that happen on each RDD of the data Stream. The first action in our algorithm is the forEach function on the input DStream that outputs one RDD for every batch interval. Then, the next action is the isEmpty that checks if the RDD has any data in the current interval. The third step, is a transformation on the RDD which makes each column on the RDD have an L_2-norm.
After that, in the AnomDetect transformation we need to broadcast some variables to the workers so they can do the transformations that we require. Those variables are the size of the feature space, the $(\mathbb{I}_m - U_{(t-1)_k} U^T_{(t-1)_k})$ from the left singular vector update step and the threshold.



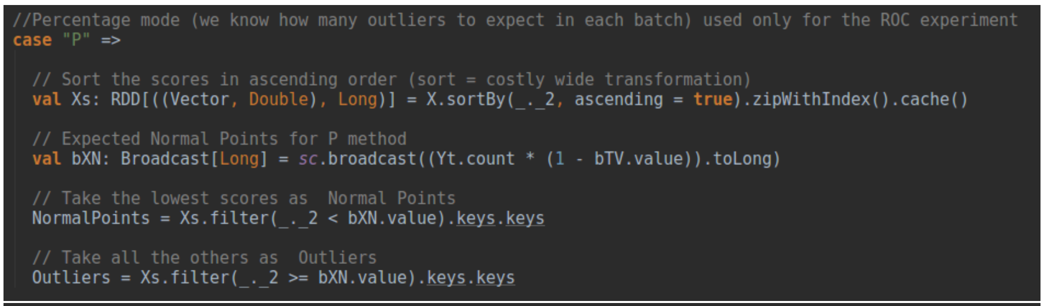
```
val bU: Broadcast[Array[Double]] = sc.broadcast(Ukd)
val bm: Broadcast[Int] = sc.broadcast(m)
val bTV: Broadcast[Double] = sc.broadcast(thresholdValue)
```

Now, that all is set the score is calculated. The first transformation is zip, creating a new column (Double) in the initial input RDD[Vector] that is the score of the respective data point. Inside that transformation we do a map in order to calculate the said score $\|(\mathbb{I}_m - U_{(t-1)_k} U^T_{(t-1)_k})\mathbf{y}_i\|$. In addition, we cache the result of that transformation in the worker nodes memory because we are going to used the results in multiple following transformations.

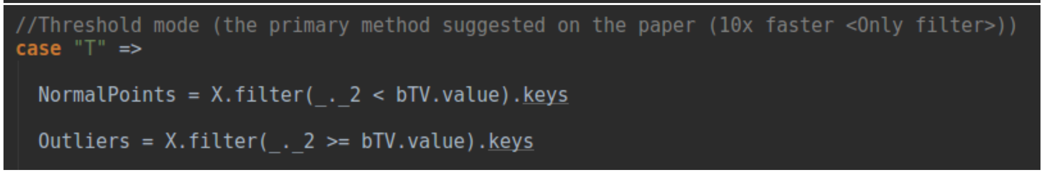


```
// point y (Vector) point score (Double)
val X: RDD[(Vector, Double)] = Yt.zip(Yt.map(
    v => Vectors.norm(Matrices.dense(bm.value, bm.value, bU.value).multiply(v), 2.0))).cache()
```

Inside the AnomDetect transformation two methods are offered for the thresholding. The first one is based on the percentage of outliers that we expect in every batch. So, we sort the RDD based on the score. Then, we broadcast the expected NormalPoint count in order to filter the top points until the expected count as NormalPoints and the rest as outliers. The second, is the method described in the paper [1] that just filters the dataset based on a threshold.

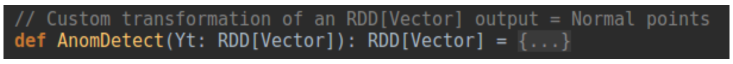


```
//Percentage mode (we know how many outliers to expect in each batch) used only for the ROC experiment
case "P" =>

  // Sort the scores in ascending order (sort = costly wide transformation)
  val Xs: RDD[((Vector, Double), Long)] = X.sortBy(_._2, ascending = true).zipWithIndex().cache()

  // Expected Normal Points for P method
  val bXN: Broadcast[Long] = sc.broadcast((Yt.count * (1 - bTV.value)).toLong)

  // Take the lowest scores as  Normal Points
  NormalPoints = Xs.filter(_._2 < bXN.value).keys.keys

  // Take all the others as  Outliers
  Outliers = Xs.filter(_._2 >= bXN.value).keys.keys
```



```
//Threshold mode (the primary method suggested on the paper (10x faster <Only filter>))
case "T" =>

  NormalPoints = X.filter(_._2 < bTV.value).keys

  Outliers = X.filter(_._2 >= bTV.value).keys
```

Finally, all those transformations are narrow transformations, which means that are executed in each worker in a distributed fashion with no need of communication. Except, the sortBy transformation that is a wide one and need communication and then shuffling between the workers. Furthermore, AmonDetect is written in a way to be perceived as a custom RDD transformation, taking an RDD[Vector] as input (current batch) and yielding an RDD[Vector] as a result (NormalPoints).

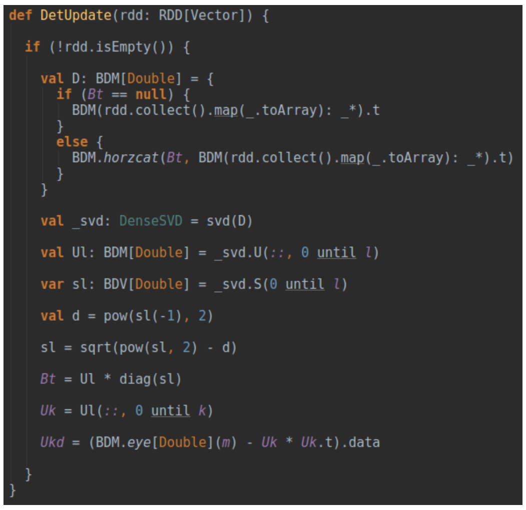


```
// Custom transformation of an RDD[Vector] output = Normal points
def AnomDetect(Yt: RDD[Vector]): RDD[Vector] = {...}
```

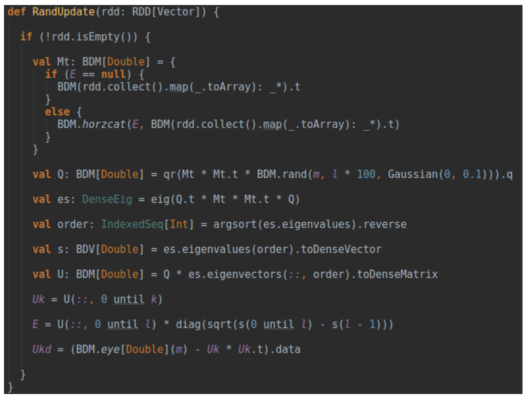
4.3.2 Spark Streaming Singular Vector Update

The first single node operation is an action collecting the RDD containing the NormalPoints into the driver node and map it as Breeze Dense Matrix. We have implemented all three Update algorithms from [1]. To begin with, the first algorithm is GlobalUpdate, which is not recommended due to the fact that it requires increasing amounts of memory overtime. So, if our streaming algorithm runs for a long amount of time the memory will eventually run out. As a result, we focus on the two sketch based algorithms that do not have the above problem (DetUpdate and RandUpdate). Both of them were a straight forward implementation from the paper, with the only change being that they return the $(\mathbb{I}_m - {U_{(t)}}_k {U^T_{(t)}}_k)$ instead of just ${U_{(t-1)}}_k$. We found out that just sending ${U_{(t-1)}}_k$ and then doing the computations to calculate

$(\mathbb{I}_m - {U_{(t)}}_k {U_{(t)}^T}_k)$ inside the AnomDetect, costs a lot more than doing them inside the Update. That is the case because we have all the data we need in the Update and, also, it is a single node operation, eliminating the complexity of our computations.

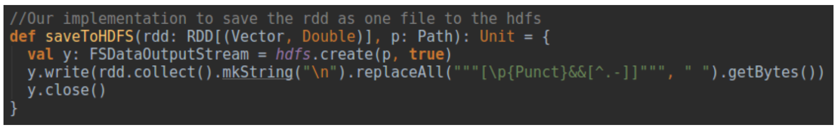


```scala
def DetUpdate(rdd: RDD[Vector]) {

  if (!rdd.isEmpty()) {

    val D: BDM[Double] = {
      if (Bt == null) {
        BDM(rdd.collect().map(_.toArray): _*).t
      }
      else {
        BDM.horzcat(Bt, BDM(rdd.collect().map(_.toArray): _*).t)
      }
    }

    val _svd: DenseSVD = svd(D)

    val Ul: BDM[Double] = _svd.U(::, 0 until l)

    var sl: BDV[Double] = _svd.S(0 until l)

    val d = pow(sl(-1), 2)

    sl = sqrt(pow(sl, 2) - d)

    Bt = Ul * diag(sl)

    Uk = Ul(::, 0 until k)

    Ukd = (BDM.eye[Double](m) - Uk * Uk.t).data

  }
}
```



```scala
def RandUpdate(rdd: RDD[Vector]) {

  if (!rdd.isEmpty()) {

    val Mt: BDM[Double] = {
      if (E == null) {
        BDM(rdd.collect().map(_.toArray): _*).t
      }
      else {
        BDM.horzcat(E, BDM(rdd.collect().map(_.toArray): _*).t)
      }
    }

    val Q: BDM[Double] = qr(Mt * Mt.t * BDM.rand(m, l * 100, Gaussian(0, 0.1))).q

    val es: DenseEig = eig(Q.t * Mt * Mt.t * Q)

    val order: IndexedSeq[Int] = argsort(es.eigenvalues).reverse

    val s: BDV[Double] = es.eigenvalues(order).toDenseVector

    val U: BDM[Double] = Q * es.eigenvectors(::, order).toDenseMatrix

    Uk = U(::, 0 until k)

    E = U(::, 0 until l) * diag(sqrt(s(0 until l) - s(l - 1)))

    Ukd = (BDM.eye[Double](m) - Uk * Uk.t).data

  }
}
```

In addition, the preferred way to store data on the HDFS is as single big files rather than the default Spark choice. Spark writes one file per partition of the input data, meaning the if we have multiple partitions per timestep our HDFS memory limit will be exceeded (Small files problem [4.4]). To counter that we have implemented an action that writes the data as one file.



```scala
//Our implementation to save the rdd as one file to the hdfs
def saveToHDFS(rdd: RDD[(Vector, Double)], p: Path): Unit = {
  val y: FSDataOutputStream = hdfs.create(p, true)
  y.write(rdd.collect().mkString("\n").replaceAll("""[\p{Punct}&&[^.-]]""", " ").getBytes())
  y.close()
}
```

in order to store the results to the HDFS it is preferred to save the results as one file to create a local file to the node and then copy it to the HDFS. Rather than creating an RDD and then using the action saveAsTextFile. Furthermore, that solves the small files problem in the HDFS because each partition of an RDD would have been stored in a different file.

In the next page we present a flowchart of the above actions and transformations.

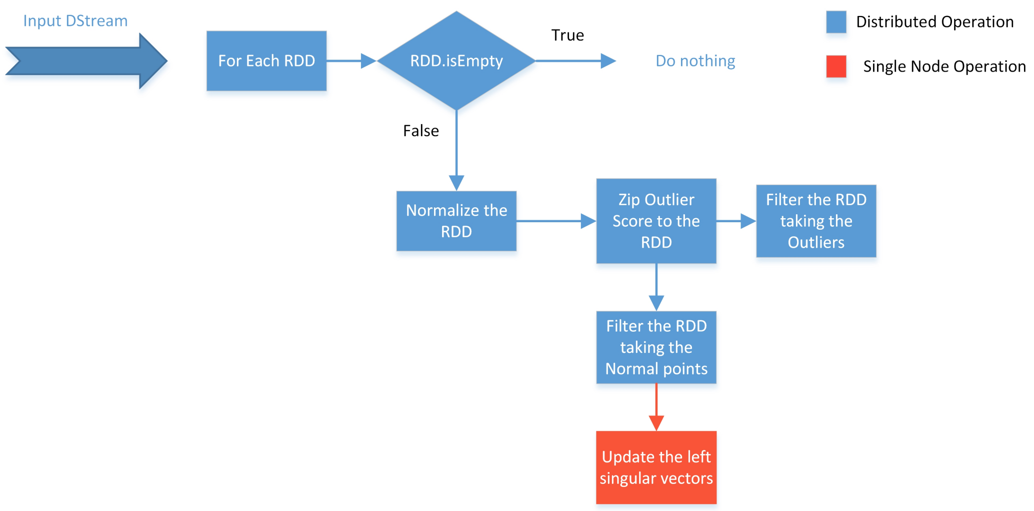

Input DStream
For Each RDD
RDD.isEmpty
True
Do nothing
False
Normalize the RDD
Zip Outlier Score to the RDD
Filter the RDD taking the Outliers
Filter the RDD taking the Normal points
Update the left singular vectors
Distributed Operation
Single Node Operation

Figure 4.3: Actions and Transformations Flowchart

4.3.3 Spark Streaming Batch interval and Concurrent Jobs

The Spark framework needs a batch interval in which it will pull the data from the streaming sources. It is recommended to pull the data in intervals, where the time of those intervals is more than the processing time of each batch. This recommendation is based on the way Spark Streaming works, each batch process is sent to a queue and if the previous one has not finished the current one waits. An other solution to that problem, if we do not want to change the batch interval, is to change a configuration parameter "spark.streaming.concurrentJobs" (default "1") that enables Spark Streaming to process more than one jobs at a time. But, that is not recommended because if used without care it can lead to some weird situations. In our implementation we set this parameter to 2 in order to prevent a big batch to interfere with the real time nature of our implementation.

4.4 Small Files Problem in HDFS

In the above sections we mentioned the so called small files problem inside the HDFS. That occurs due to the way HDFS prefers to devour a limited number of large files rather than a large number of small files. A small file is one which is significantly smaller than the HDFS block size (default 64MB). If we are storing lots of small files, the problem is that HDFS can not handle lots of files. Reason being that every file, directory and block in HDFS is represented as an object in the namenode's memory, each of which occupies 150 bytes, as a rule of thumb. So 10 million files, each using a block, would use about 3 gigabytes of memory. Scaling up much beyond this level is a problem with current hardware. Certainly a billion files is not feasible. To make matters worse, in Spark each file represents one partition in the resulting RDD, meaning that we can easily end up with an RDD with more than a million tiny partitions. In order to not destroy our processing efficiency, we will need to repartition that RDD into something more manageable, which will require lots of expensive shuffling over the network. Furthermore, HDFS is not geared up to efficiently accessing small files being primarily designed for streaming access of large files. Reading through small files normally causes lots of seeks and lots of hopping from data node to data node to retrieve each small file, all of which is an inefficient data access pattern.

That problem also occurs in other storage solutions like S3. If we store all the data gzipped in S3, it could result in lots of small gzipped files. In that case, not only does Spark have to pull those files over the network, it also has to uncompress them. Because gzipped files can be uncompressed only if you have the entire file on one core, our executors are going to spending a lot of time unzipping files.

So, we aim to keep the input and output files in relatively big sizes.

Chapter 5

Experimental Results

5.1 Introduction

We conducted experiments on some of the datasets from Hao Huang and Shiva Kasiviswanathan paper [1] in order to prove that our Spark implementation of their algorithm inside the Spark Streaming framework produces the same results considering the quality both in runtime and in accuracy. In addition, we conduct scalability experiments with the Spark Cluster in order to see if the algorithms performance increases with the more resources we provide.

Accuracy

To measure the accuracy of the algorithm we use ROC curves. In statistics, a receiver operating characteristic (ROC) curve, is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. The ROC curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. The true-positive rate is also known as sensitivity, recall or probability of detection in machine learning. The false-positive rate is also known as the fall-out or probability of false alarm. Accuracy is measured by the area under the ROC curve. An area of 1 represents a perfect test; an area of 0.50 represents a worthless test. And precision is measured by the following formula $Precision = \frac{TP}{TP + FP}$

A rough guide for classifying the accuracy of a diagnostic test, is the traditional academic point system:

- $0.90 - 1$ = excellent
- $0.80 - 0.90$ = good
- $0.70 - 0.80$ = fair
- $0.60 - 0.70$= poor
- $0.50 - 0.60$ = fail

Runtime

We measure the average runtime on batches containing 5000 datapoints from the outlier detection phase to the singular vectors update.

Scalability

The scalability is being measured in three different ways:

- Single Batch: With this experiment we see how match scaling we can achieve with increasing the input size of a single batch. For example, can we still achieve near real time results even with a big influx of data on a single time step.
- Overtime: The current experiment provides us information about the data that our framework can accumulate over time. This will provide information if our streaming implementation is stable overtime.
- Parallelism: Here we will observe the increase in performance depending on the increase of parallelism in the spark architecture.

Spark Cluster Setup

In our experiments we used the Spark on YARN cluster on Hortonworks HDP 2.2.4 of our University at Softnet Lab. Our driver node has 8 cores and 2 GB of RAM and two worker nodes with 8 cores and 2 GB of RAM each. But our parallelism tests were run on a standalone cluster due to problems in the version of the Softnet cluster (Spark 1.2.1) that did not support some parts of our code (Spark 2.2.0).

5.2 ROC Curve

The first test that we run is on the Cod-RNA and Protein-homology datasets with 5000 data point batches. We aim to see if we have the same results as the authors of [1]. In the following figure 5.2, we observe that we have the same results with an area under the ROC curve of 0.8, which indicates a fairly good accuracy in the current dataset. In addition, it yields a precision of 65 %

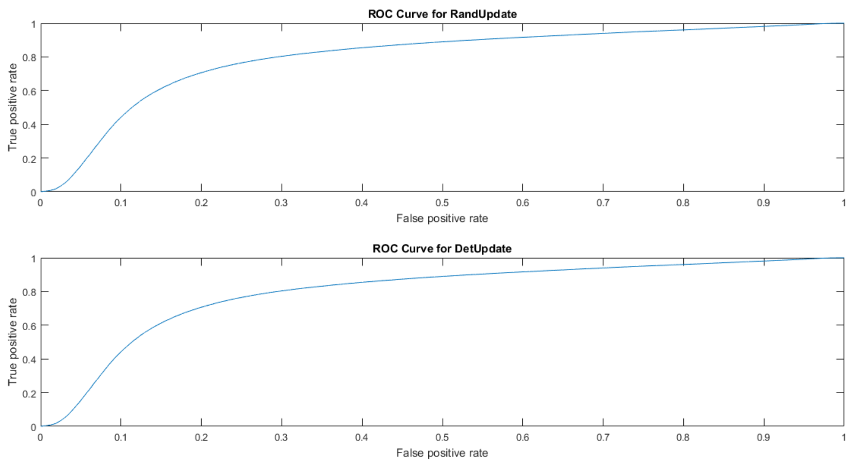


Figure 5.1: ROC curves of RandUpdate and DetUpdate for the Cod-RNA Dataset

Now we produce the ROC curve for the Protein-homology dataset. At the beginning, we observe that it has a really bad ROC curve with 0.5 area beneath it. But, it produces results with a precision of 99 %. Furthermore, they are the same as those of the authors of [1].

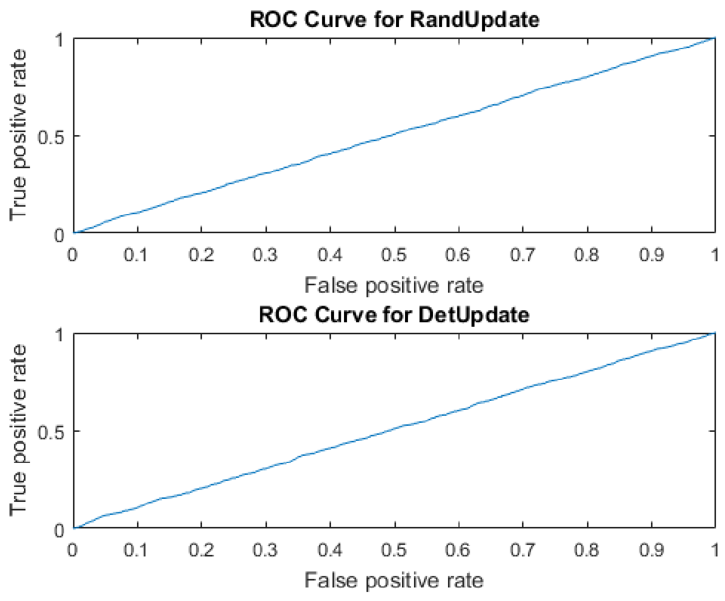


Figure 5.2: ROC curves of RandUpdate and DetUpdate for the Protein-homology Dataset

5.3 Runtime

The following experiment provides us with results concerning how fast our implementation runs. In the table below, we compare the results that we achieved with our Spark Streaming implementation with those of the authors to see if the Spark framework improves runtime. The test were conducted in the Cod-RNA and Protein-homology datasets with batches of 5000 datapoints and the runtime that we present is the average of all the batches. We measure from the start of AnomDetect till the end of the singular vector Update. Our first observation, is that Rand update is much better, especially in the Protein-homology dataset, which has a much bigger feature size and thus, the SVD algorithm is more costly for the DetUpdate algorithm. The algorithm RandUpdate manges to keep the processing in Real-time. In addition, the measurements inside the Spark Cluster are included 5.3, 5.4. In those figures we observe a peak in runtime at the beginning due to the initialization and the Spark Core starting overhead cost. Furthermore, the RandUpdate algorithm if far more stable, never exceeding the 1 second mark after training.

Algorithms/Dataset	Cod-RNA(8 features)	Protein-homology(74 features)
DetUpdate	0.821	6.357
RandUpdate	0.114	0.458

Table 5.1: Average RunTime(sec) of each algorithm in each Dataset.

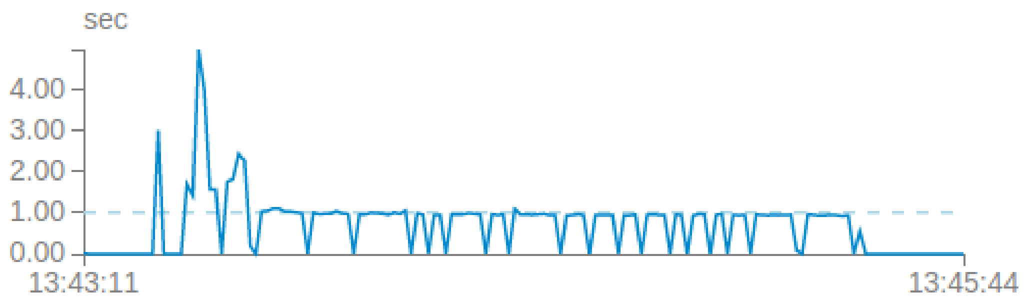


Figure 5.3: AnomDetect with DetUpdate Runtime as depicted in the spark cluster for the Cod-RNA Dataset

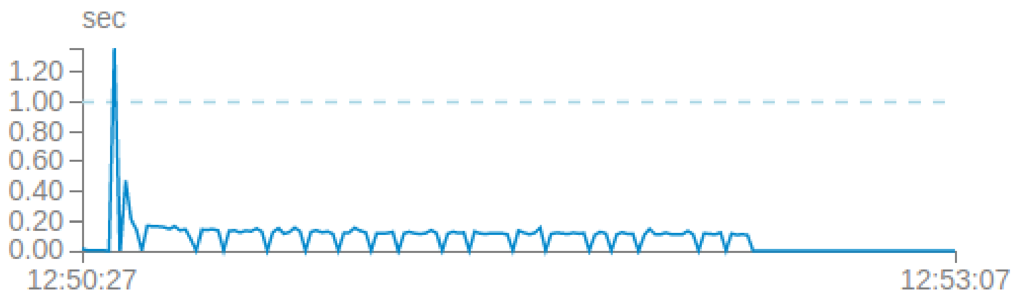


Figure 5.4: AnomDetect with RandUpdate Runtime as depicted in the spark cluster the Cod-RNA Dataset

5.4 Scalability

In this section we conduct three experiments to the Cod-RNA dataset to see how scalable our implementation is. At first, we test how it will react in bigger batch sizes. Then, we let our Spark Streaming application run for an hour to see how stable it is in a continuous workload. Finally, we increase the level of parallelism in our Spark application to see if by running some actions or transformations in parallel improves runtime.

5.4.1 Increasing Batch Size

In the figure 5.5 bellow we experiment on increasing batch sizes and try to see how our spark implementation reacts in regard of Runtime with 2 worker nodes. At first we observe that the randomized update approach yields faster results exceeding the 1 second mark after 300.000 datapoint and reaching just over 3 seconds on 1 million datapoints. The deterministic approach exceeds the 1 second mark at 100.000 data points and eventually at 9 seconds it processes 1 million datapoints.

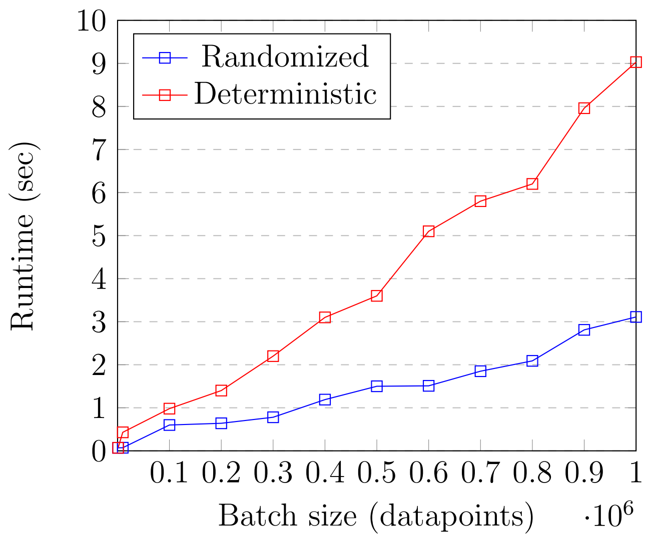


Figure 5.5: Increasing Batch Size Runtime Cod-RNA dataset 2 worker nodes

5.4.2 Stability in continuous workload

The current experiment is conducted in order to see see how our implementation works in a continuous workload and in order to measure how the output that we send to the HDFS affects performance. We observe in the figure 5.6, which depicts the last 17 minutes of an 1 hour uptime of our algorithm, that it is stable and runs under 1 second in all the batches of those 17 minutes. Furthermore, the average delay is 0.073 seconds. Those results were to be expected due to the lazy state of the RDDs and the fact that Spark dumps the RRD data of the previous batches if we do not instruct it otherwise. The figure has the last 17 minutes because the Spark cluster does not provide larger delay plots.

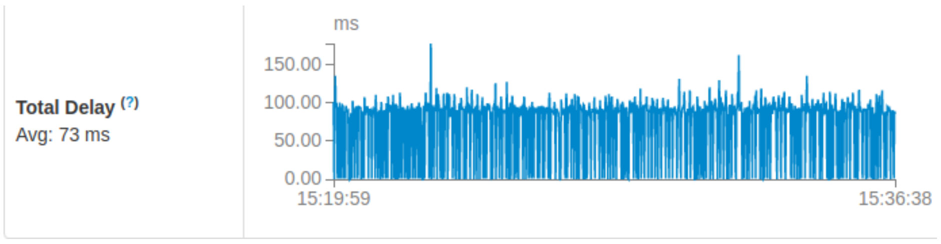


Figure 5.6: The last 17 minutes as depicted in the Spark Cluster

5.4.3 Parallelism

The last experiment is carried out in order to see if our implementation can run better in parallel, both with and without writing the data to the HDFS. The experiments were conducted for batches of 5000 over 2 minutes in order to see how much task time takes each worker in order to complete each task. We observe that the runtime improvement is acceptable with an improvement around 38% on both experiments going from one to two workers and, finally, reaching over 50% when we run it with four workers. From that point, due to the size of our batches we do not see any further improvement. Although, if we had bigger batches, we could have benefited from more workers. That was expected because only a part of our algorithm did not run on a distributed fashion.

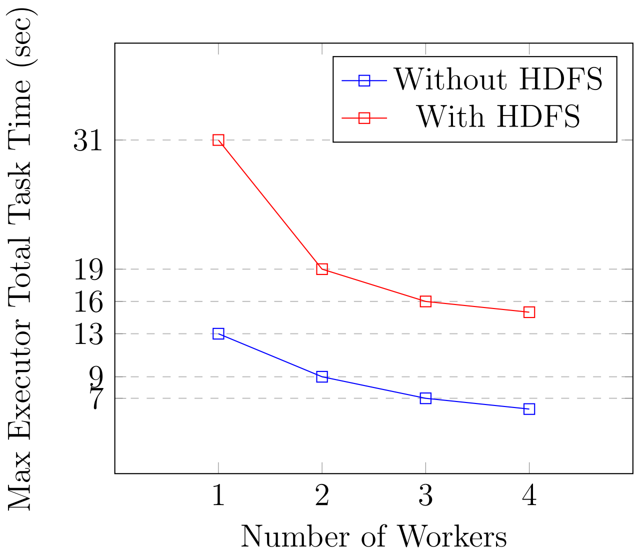


Figure 5.7: Performance increase due to increase in the Spark executors

Chapter 6

Conclusion

In this work we migrated the algorithm of Hao Huang and Shiva Kasiviswanathan [1] inside the Spark Streaming framework. They proposed a new deterministic and randomized sketching-based approaches to efficiently and effectively detect anomalies in large data streams. We proved that their idea can work in a modern streaming data analytics framework. In addition, we conducted various experiments proving that this implementation can work in near real time. Moreover, we expanded it to work in a distributed fashion on the AnomDetect part of the algorithm being able to utilize the power of the Spark cluster. Furthermore, the experimental results prove the effectiveness and efficiency of the proposed approaches.

Chapter 7

Future Work

This work could be extended and improved three ways.

At first, a performance improvement would be to calculate the outlier space and then threshold in order for the normal points to be far from it. This is an improvement because most of the time the outliers are much less than the normal points and we will have less data sent to the driver to calculate the left singular vectors.

Secondly, the last centralized step in our implementation could be parallelized for an increase in performance in big data clusters.

Finally, the Spark Streaming framework as of now is in a transforming state. The RDD abstraction is slowly getting deprecated and better ones emerge that are fully implemented in Spark Core and Spark SQL like the DataFrames and DataSets. So, when the Spark Streaming framework decides to make the change, we can migrate this work there.

Bibliography

[1] Hao Huang, Shiva Prasad Kasiviswanathan. *Streaming Anomaly Detection Using Randomized Matrix Sketching.* Proceedings of the VLDB Endowment, 2015.

[2] Hans-Peter Kriegel, Peer Krger, Arthur Zimek. *Outlier Detection Techniques.* The 2010 SIAM International Conference on Data Mining

[3] What is Streaming Data?
`https://aws.amazon.com/streaming-data/`

[4] Spark Core Project
`https://spark.apache.org/`

[5] Spark Streaming
`https://spark.apache.org/streaming/`

[6] Spark Cluster
`http://spark.apache.org/docs/latest/cluster-overview.html`

[7] E. Liberty. *Simple and Deterministic Matrix Sketching.* In ACM SIGKDD, pages 581588, 2013.

[8] M. Ghashami and J. M. Phillips. *Relative Errors for Deterministic Low-Rank Matrix Approximations.* In SODA, pages 707717, 2014.

[9] M. Ghashami, E. Liberty, J. M. Phillips, and D. P. Woodruff. *Frequent Directions : Simple and Deterministic Matrix Sketching.* CoRR, abs/1501.01711, 2015.

[10] P. Businger. *Updating a Singular Value Decomposition.* BIT, 10(3):376385, 1970.

[11] N. Halko, P. G. Martinsson, and J. A. Tropp. *Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions*. SIAM Review, 53(2), 2011.

[12] Breeze numerical processing library for Scala
`http://www.scalanlp.org/`

[13] Scala
`https://www.scala-lang.org/`

[14] HDFS
`https://hadoop.apache.org/`