AMRules for Fraud Detection with Spark Streaming

Emmanouil Fragiadoulakis

Technical University of Crete

School of

Electrical and Computer Engineering

Thesis Committee:

Associate Professor Antonios Deligiannakis, Thesis Adviser

Professor Minos Garofalakis

Associate Professor Michail G. Lagoudakis

October 2018

Abstract

In this day and age, an important part of our daily interaction with our electronic devices is on-line payments, which results in a great amount of transactions. In order to handle these transactions, to determine if they are fraudulent, we need an efficient, distributed and streamable machine learning algorithm, that can process big amount of incoming data and react to it instantly. Thus, we implemented the distributed Adaptive Model Rules on Spark Streaming, which is an extension of the Spark Core API, that enables the development of scalable, fault-tolerant streaming applications. Adaptive Model Rules is an one-pass algorithm for training its model from streaming data and is robust to outliers and irrelevant features. The experimental results concluded, that there is a noticeable speedup from Vertical Adaptive Model Rules to Hybrid Adaptive Model Rules at the cost of reducing accuracy.

Acknowledgements

Firstly, I would like to thank my parents, without whom my education would not be possible, along with my brother, who was constantly able to help me whenever I needed.

I would like to thank everyone from the department of Electrical and Computer Engineering. More specifically, I would like to thank my thesis adviser, Associate Professor Antonios Deligiannakis, who helped me and guided me throughout this work, as well as the rest of the thesis committee, Professor Minos Garofalakis and Associate Professor Michail G. Lagoudakis, who took the time to review my work.

Finally, I would like to thank my friends, who were there with me throughout my education.

To my parents and my brother.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

New technologies allow us to generate data by interacting with software on our daily activities [15]. A important part of our daily interaction with these technologies is the use of credit cards. Credit cards dominate among other payment methods, thus fraud appears as a huge issue in credit card transactions. Therefore, it is important for applications to be able to process incoming data and react to it in an online fashion using comprehensive detection mechanisms.

Since, credit card transactions can be produced anywhere, anytime, anyhow, there is major interest in preventing or detecting fraudulent transactions [5]. Fraud prevention is the process of blocking fraudulent transactions at source with some common techniques for fraud prevention to be Card Verification Method (CVM), Personal Identification Number (PIN) etc. On the other hand, fraud detection is the mechanism that classifies a transaction as fraudulent or genuine. A Fraud Detection System can be constructed, so that it can detect frauds from a massive online sample of transactions.

A predictive model can use rules, that are based on the knowledge acquired from fraud experts, but these rules need to be maintained by the fraud experts. However, a machine learning approach can detect fraud patterns and predict transactions as

possibly fraudulent, without human supervision [6]. Automatic fraud detection systems are vital, considering the fact that fraud analysts can not easily process large samples of a dataset, that update over time. In 2013, the Adaptive Model Rules (AMRules) were presented as the first rule-based learning algorithm for regression problems on streaming samples of data [1]. And in 2014, the distributed implementation of AMRules was presented in order to achieve scalability and handle big data streams.

Nowadays, machine learning algorithms are scalable, adaptable and streamable. However, there are some challenges in producing a machine learning algorithm, that detects fraudulent transactions. The main challenge is that, there are not enough samples, since the datasets contain sensitive information and the available datasets provide us with small portion of fraud transactions, hence the fraud data are highly unbalanced. Finally, there is also the problem, that the fraud distribution changes as the adversaries' strategies evolve.

1.1 Contributions

In this thesis, it is presented a distributed implementation of AMRules on Spark Streaming. The goal is to reduce the complexity of implementation as well as the processing time, since Spark Streaming is a fast, scalable and fault-tolerant analytics engine.

1.2 Outline

The Chapter 2 provides us with the theoretical backround needed to be able to understand main concepts discussed in this thesis. Chapter 3 and Chapter 4 give us the main concept behind the algorithms in this diploma thesis and the implementation of them using Spark Streaming, respectively. In Chapter 5 we would see the exper-

imental results of this work and finally the Chapter 6 concludes the work and gives recommendations for future interest.

Chapter 2

Background

Over the years, data were processed offline, but now there is no reason to wait for the complete dataset. Streaming analysis provides the ability to identify patterns and make predictions based on them simultaneously. Moreover, the amount of streaming data presented the need for a distributed system, that can efficiently implement machine learning algorithms. The theoretical background behind these concepts will be discussed in this Chapter. In Section 2.1 we will see what machine learning is and, how and where can be applied. In Section 2.2 we will see from what components the distributed system Apache Spark is made of and we will also dive into Spark Streaming for more information.

2.1 Machine Learning

Machine learning (ML) [12] gives the ability to a machine to learn from data, in order to create a model. The model adapts with the purpose of detecting patterns and making predictions based on these patterns. The most common machine learning models are described below.

- **Supervised Learning :** In this type of machine learning model, two types of data are available, labeled and unlabeled. The labeled data are used to train the machine learning model and the unlabeled data are used to test the model and make predictions.

- **Unsupervised Learning :** Contrariwise, this type of model does not have historical data to learn from. It tries to find regularities in the given data or extract the most significant features from that data.

Online Learning

There is the assumption that all the training data should be available beforehand, but not with online learning. In this case, the complete sample is not accessible, but only streams of data. In online learning there are intervals of time. In each interval, a new training example becomes available to the learning algorithm, which uses its current model to predict the label. After that, the correct label is revealed to the learner so that it can incur loss and update its model.

Credit Card Fraud Detection Use Case

A model in machine learning, that is created with the purpose of detecting fraud, it is called Fraud Detection Model. More specific with the help of ML models, we can detect credit card frauds. Credit card frauds [11] are unauthorized transactions, made or attempted by a malicious entity. The patterns of fraud [7] are dynamic, and with that arises the need for the credit card fraud detection model to be adaptive.

2.2 Apache Spark

Apache Spark [3] is a lightning-fast cluster computing engine for Big Data and Machine Learning. It is an open source framework, which allows you to write general-

purpose distributed programs and build applications on top of it. Spark [18] was developed in UC Berkeley and was first introduced in 2010 by Matei Zaharia et al..

Apache Spark provides the following features.

- **Speed :** Spark is 100 times faster in memory and 10 times faster on disk, when compared with Hadoop MapReduce.
- **Ease of use :** Spark has build-in APIs for multiple languages like Java, Scala, Python or R. So, it is easy to create applications, that run multiple parallel operations.
- **Generality :** The Apache Spark Ecosystem is designed to combine SQL, streaming, machine learning and graph analytics. It includes libraries for each of them in order to be combined in the same application.
- **Support :** Spark is able to run on Hadoop, Mesos, Kubernetes, standalone, or in the cloud. It is capable of accessing diverse data sources like HDFS, Cassanda, HBase and S3.

The Spark project [9] contains different closely integrated components. The base of these components is Spark Core, which provides support for Spark SQL, Spark Streaming, MLlib and GraphX. A description for the components, shown in Figure 2.1, follows below.

Spark Core

Spark Core [9] holds the basic functionality of Spark. Spark Core's APIs can provide support such as task scheduling, memory management, fault recovery, interaction

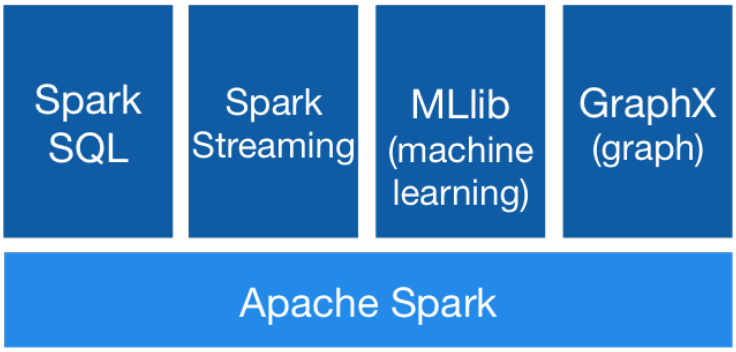


Figure 2.1: The Spark Stack [3]

with storage systems etc, for a wide range of applications. Furthermore, the Resilient Distributed Datasets [17] (RDDs) are the main concept of Spark. RDDs allow in-memory computations on large clusters in a fault-tolerant way. RDDs can be divided into partitions, they are also immutable and lazily evaluated, which means they cannot be altered and the transformations are not performed immediately, respectively.

Spark SQL

Apache Spark's package Spark SQL [3] [4] processes structured data. One of Spark SQL's main characteristics is its integration between relational and procedural processing, which is achieved by the DataFrame API. Moreover, it has the ability to connect uniformly to different data sources, including Hive, Avro, Parquet, ORC, JSON and JDBC. Spark SQL also enables to write queries using HiveQL, access existing Hive warehouses and connect to Hive UDFs.

Spark Streaming

Spark Streaming [3] allows scalable, high-throughput, fault-tolerant stream processing of data received in real time. Spark Streaming will be discussed extensively in Section 2.2.1.

MLlib (machine learning)

MLlib [3] [10] is Spark's scalable library, that contains multiple types of machine learning algorithms. MLlib is supported by several programming languages like Java, Scala, Python and R, and its high-level API manages to simplify the construction, evaluation and tuning of end-to-end machine learning pipelines. Finally, MLlib's performance excels in terms of quality and speed, because of Spark's iterative computation.

GraphX (graph processing)

GraphX [3] [16] is a distributed graph computation engine that combines graph and data parallel computation. GraphX contains fundamental operators as well as the Pregel API.

2.2.1 Spark Streaming

As it is mentioned earlier Spark Streaming enables the development of scalable, fault-tolerant streaming applications. Spark Streaming was build to match the scalability, fault recovery and throughput of Spark cluster with the help of the programming model, called Discretized Streams [20] [19](DStreams). A DStream groups together a series of RDDs and allows the user to perform various operations on them.

A Spark Streaming program is able to accept data by loading it periodically from various storage sources such as Kafka, Flume, HDFS/S3, Kinesis, or TCP sockets. Both transformations and output operations can be applied on the incoming streams.

Transformations produce a new DStream each interval and they can be stateless (i.e. *map*, *reduce*, *groupBy*) or steteful (i.e. *window*, *reduceByWindow*), and output operations (i.e. *save* or *foreachRDD*) let the program save data to an external system or run a user code snippets on each RDD. The Spark Streaming Architecture described above is shown in Figure 2.2.

Figure 2.2: The Spark Streaming Architecture [3]

Discretized Streams (DStreams)

Discretized Streams or DStreams are a high-level abstraction on Spark Streaming and correspond to incoming streams of data. A DStream is a continuous series of RDDs, which is an immutable, distributed dataset. Each transformation or output operation is applied on the RDDs through the DStreams. Each RDD in a DStream holds incoming data from a time interval. Discretized Stream is shown in the following Figure 2.3.

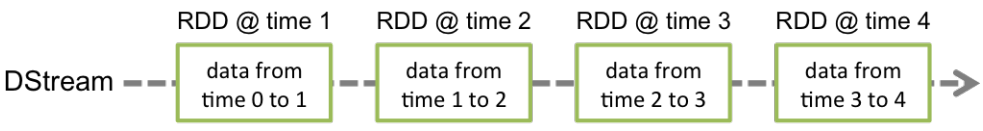


Figure 2.3: DStream per interval [3]

2.3 Samza vs Spark Streaming

For the implementation of distributed AMRules, Spark Streaming was chosen instead of Samza, which is the chosen distributed stream processing framework for the original implementation. A comparison between these two distributed frameworks follows below.

Table 2.1: Samza vs Spark Streaming [13] [2]

	Samza	**Spark Streaming**
Ordering	native streaming	micro-batching
Latency	milliseconds	seconds
Guarantee	at-least-once	exactly-once
Fault-tolerance	YARN	cluster manager, automatic
Parallelism	processing	processing, receiving
Deployment	YARN, local	Spark Standalone, Apache Mesos, Hadoop Yarn, Amazon EC2
Language	Java, Scala	Java, Scala, Python, R
Maturity	young	big community

One of the main differences these distributed frameworks have, as shown in Table 2.1, is their stream processing model. Samza's model is native streaming, while Spark Streaming's is micro-batching, meaning that Samza processes incoming records as they come and Spark Streaming processes incoming records in micro-batches per interval. This impacts their performance possibilities and leads into latency, which is milliseconds for Samza and seconds for Spark Streaming.

Samza is able to provide us with an at-least-once message delivery guarantee, but Spark Streaming can guarantee an exactly-once message delivery. Moreover, failure in Samza and Spark Streaming can happen on worker or driver nodes. Samza's way to fault-tolerance is to work with YARN in both worker and driver nodes, in contrast with Spark Streaming's way, which is using cluster manager and standalone (automatic) for worker and driver nodes, respectively.

Furthermore, they have differences in parallelism and partitioning, Samza's parallelism lies in splitting processing into independent tasks, but Spark Streaming's lies in splitting jobs in small tasks and sending them to executors. A distinction regarding their stream computations is that Spark Streaming requires deterministic transformation operations, but Samza does not.

These distributed systems also differ in their support in deployment and languages. While Samza only supports YARN and local execution, Spark streaming supports Spark standalone, Apache Mesos and Hadoop YARN, and also has script for launching in Amazon EC2. In Addition, Samza supports only Java and Scala, while Spark Streaming has APIs for Java, Scala, Python and also R. In Conclusion, it is important to mention that Spark Streaming has a bigger community behind it, since it is older than Samza.

Chapter 3

Preliminaries

Training a machine learning model from streaming data requires incremental learning, using restricted computational resources as well as adjusting to changes in time. In this Chapter we present the Adaptive Model Rules (AMRules) algorithm, which is the first one-pass algorithm for training its model from streaming data. Model rules are among the most efficient regression algorithms due to the fact that rules do automatic feature selection and are robust to outliers and irrelevant features. In Section 3.1 we present the AMRules algorithm and in Section 3.2 its distributed versions Vertical and Hybrid AMRules.

3.1 Adaptive Model Rules (AMRules)

The AMRules [1] algorithm is an incremental regression algorithm for learning model rules. The AMRules algorithm is analyzed in this Section.

Firstly, the consistence of the decision rule needs to be described. In AMRules each rule consists of a head and a body. The body of the rule is composed of various conditions called features, each of which correspond to an attribute of the instances. The head of the rule is a function that reduces the mean square error of a specific attribute. Moreover, the rule is equipped with an online change detector, so that the

mean square error can be monitored, in order to get knowledge about the process. The rule has also statistics about the previous instances. Finally, the rule can be summed up in the following expression : $head \leftarrow body$.

The training process starts by initializing a set of rules (RS) and a default rule, both of which are set as empty. Each instance, is checked if it is covered by a rule in the rule set (RS), as presented in line 6 of Algorithm 1, where rule is referenced as 'r' and instance as 'example'. If a rule covers the instance, then both the change detection tests and the statistics of the rule covered are updated, as we can see in lines 7 and 15 of Algorithm 1, respectively. If the RS is unordered, the statistics of each rule covering the instance are updated. Otherwise, if the RS is ordered, the statistics of only the first rule, that covers the instances, are updated. We are able to notice in line 17 of Algorithm 1, that the loop in line 5, breaks if the RS is ordered. If none of the rules in the RS covers the instances, the default rule's statistics are updated, which is shown in line 21 of Algorithm 1.

The rule tries to expand, when N_m updates are applied on a rule, as it is shown in lines 13, 14 of Algorithm 1. The Standard Deviation Reduction (SDR) measure is computed for each feature of the instance and the ones with the higher SDR are chosen, in order to compute the *ratio* of the second-higher over the highest SDR, where the computation of the *ratio* is presented in line 5 of Algortithm 2. A high confidence interval ϵ is computed with the help of Hoeffding bound alongside the *ratio*, as we can see in lines 4 and 6 of Algorithm 2, so the decision of the rule expansion can happen. If $ratio + \epsilon < 1$, then the rule can be expanded, based on the feature of the highest SDR value. In order not to miss a valuable feature, a rule is expanded also when the Hoeffding bound ϵ is below a given threshold. Finally, if the default rule is expanded, a new rule is being created with the default rule statistics, and we can notice that in line 10 of Algorithm 2, and the default rule is being initialized.

Algorithm 1: AMRules Algorithm

Input: S : Stream of examples
ordered-set : Boolean flag
N_{min} : Minimum number of examples
λ : Threshold
α : the magnitude of changes that are allowed

Result: RS Set of Decision Rules

```
1  begin
2      Let RS ← {}
3      Let defaultRule ← 0
4      foreach example (x, yk) ∈ S do
5          foreach Rule r ∈ RS do
6              if r covers the example then
7                  Update change detection tests
8                  Compute error = x_t − x̄_t − α
9                  Call PHTest(error,λ)
10                 if Change is detected then
11                     Remove the rule
12                 else
13                     if Number of examples in L_r > N_min then
14                         r ← ExpandRule(r)
15                         Update sufficient statistics of r
16                 if ordered-set then
17                     BREAK
18         if none of the rules in RS triggers then
19             if Number of examples in L mod N_min = 0 then
20                 RS ← RS ∪ ExpandRule(defaultRule)
21                 Update sufficient statistics of the defaultRule
```

Algorithm 2: ExpandRule: Expanding one Rule

Input: r : One Rule
τ : Constant to solve ties
δ : Confidence
Result: r' Expanded Rule

```
1  begin
2      Let X_a be the attribute with greater SDR
3      Let X_b be the attribute with second greater SDR
4      Compute ε = sqrt(R^2 ln(1/δ) / 2n) (Hoeffding bound)
5      Compute ratio = SDR(X_b) / SDR(X_a) (Ratio of the SDR values for the best two
         splits)
6      Compute UpperBound = ratio + ε
7      if UpperBound < 1 ∨ ε < τ then
8          Extend r with a new condition based on the best attribute
9          Release sufficient statistics of L_r
10         r ← r ∪ {X_a}
11     return r
```

As mentioned earlier, the RS in AMRules can be ordered and unorderd, where only the first rule that covers the instance is used to make a prediction about the target instance and all the rules covering the instance are used to make predictions, aggregating them to compute the mean, respectively. Each rule implements 3 prediction strategies: *i)* the mean of the target attribute computed from instances covered by the rule (TargetMean), *ii)* a linear combination of the independent attributes (Perceptron), *iii)* an adaptive strategy, that chooses between the first two strategies, the one with the lower MSE in the previous instances.

3.2 Distributed AMRules

In this Section, we analyzed the steps that were followed in order to parallelize the AMRules algorithm based on the paper "Distributed Adaptive Model Rules for mining Big Data Streams" [15] by Vu et. al.

3.2.1 Vertical AMRules

Based on the fact that a rule is able to expand individually and that this process is computationally expensive, the training of the rules it is delegated to multiple learner processors to achieve parallelization.

In order to filter and redirect the incoming instances to the correct learners, a model aggregator processor is essential. The model aggregator handles a rule set, the rules of which only consist of head and body, meaning the rules do not keep statistics. The body of a rule determines if the rule's covering an instance, while the head computes the prediction. Furthermore, the model aggregator manages the statistics of the default rule, which is updated with the instances not covered by any rule in the RS. Finally, if the default rule is eligible for expansion, it expands and appends a new rule to the RS, then the model aggregator sends the new rule with the corresponding instance to one of the learners based on the rule's ID.

The model aggregator sends instances and rules to the learners. Each learner manages rules that are assigned to it based on the rule ID. At the learners, the rule's statistics are updated with the forwarded instance, which ensures that the generated rules are the same as in the sequential algorithm. If a rule expansion happens, the new feature is sent to the model aggregator to update the body of the rule. Concurrently, learners are capable of detecting changes and removing existing rules, when this occurs the learners notify the model aggregator with a message containing the removed rule ID. We can observe the design of Vertical AMRules (or VAMR) in Figure 3.1.

Since for each rule in model aggregator there is a duplicate rule in a learner, the statistics of the rule might not be updated. There can be a delay between the learner and the model aggregator regarding the update of the expanded rule, and this lies on the queue length on the model aggregator. The queue length is a result of the volume and speed of the incoming data stream. Thus, there is a chance that instances might be forwarded to a recently expanded rule, which no longer covers the instance.

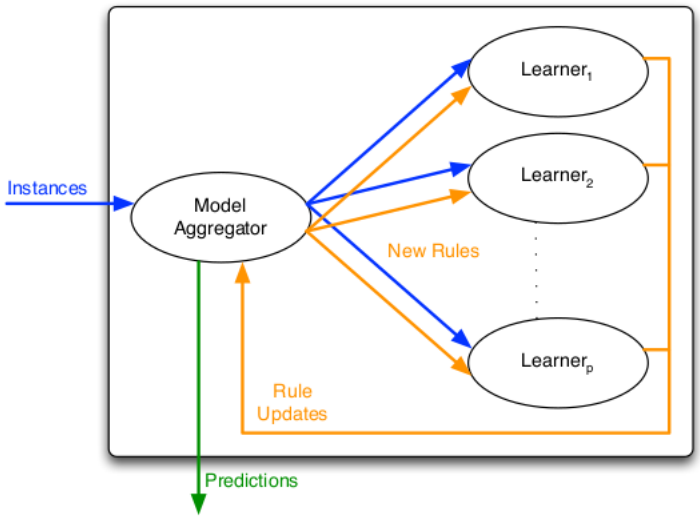


Figure 3.1: Vertical AMRules (VAMR) [15]

For the above reason, a coverage test is performed at the learner, so that the instance can be discarded, if it was incorrectly forwarded. Based on this test and that the rule expansion can only increase the selectivity of a rule, the accuracy of the algorithm stays the same for unordered rules. Although, for ordered rules these temporary inconsistencies are able to affect the statistics of the rules, considering that the instance should have been forwarded to a different rule.

3.2.2 Hybrid AMRules

The first step of creating the hybrid AMRules is to vertically parallelize the execution of the model aggregator with the help of a horizontally based parallelism. This can be accomplished by replicating the model aggregator, so that each replica contains the exact copy of the rule set (RS), but processes only a part of the incoming data stream. The basic concept is to integrate multiple model aggregators in to the design, so that each of them can process a proportional amount of instances to the number of model aggregators. In conclusion, this will affect both the prediction and the

training statistics of the default rules. In Figure 3.2 we can observe the design of the horizontally parallelized model aggregator.

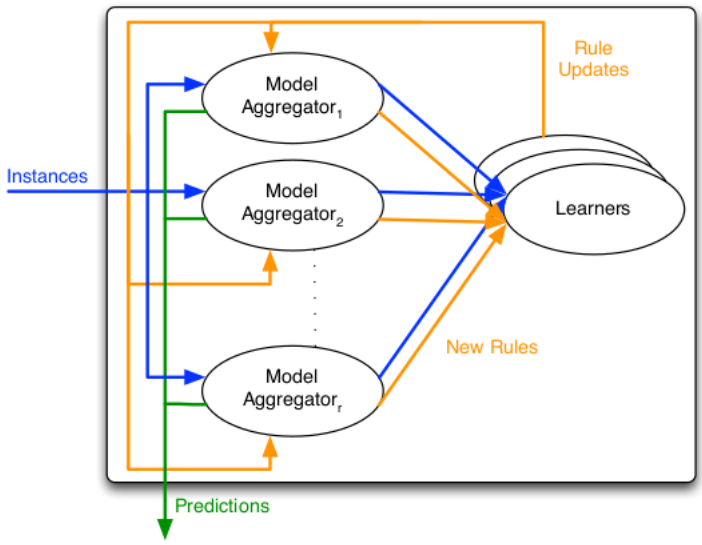


Figure 3.2: AMRules with multiple horizontally parallelized Model Aggregators [15]

Concerning the fact that each of the model aggregators can process a partition of the incoming data stream, each default rule is trained independently. Therefore, model aggregators will be creating overlapping or conflicting rules, so it is important to be able to synchronize and order the rules created by different model aggregators. Moreover, this design is less reactive compared to VAMR, since it needs more instances, so that the default rules can expand. The prediction function of each rule adapts depending on the attribute and label values of past instances, concluding that having only one part of the data stream will lead to having less information and maybe lower accuracy.

To fix the problems created by parallelizing the model aggregators, a centralized rule is created. The distribution in the creation of the rules created issues, that can be avoided by moving the default rules from the model aggregators to one specialized default rule learner processor. In result, all the model aggregators now will be synchronized.

Since the default rule is in an independent component, the instances, that are not covered by any rules in the model aggregators, are forwarded to the default rule learner. This learner updates its statistics with the input instances and when the default rule expands, it sends the new rule to the model aggregator as well as the learner, selected based on the rule ID.

The design of the Hybrid AMRules (HAMR), which is a combination of vertical and horizontal parallelism, is shown in Figure 3.3.

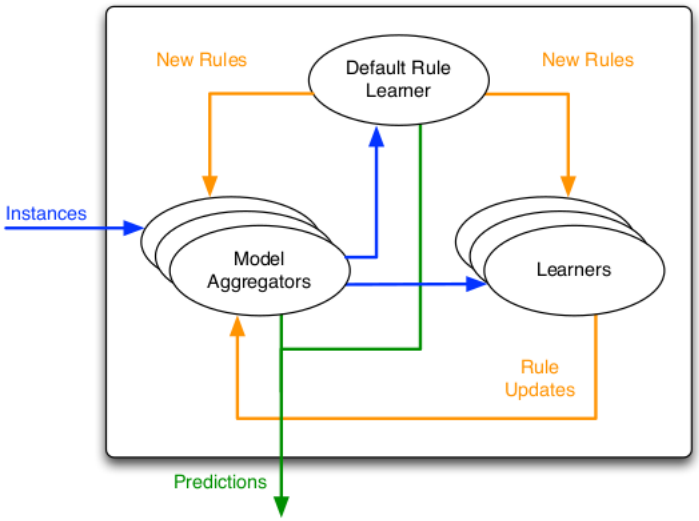


Figure 3.3: Hybrid AMRules (HAMR) with multiple Model aggregators and separate Default Rule Learner

Chapter 4

Implementation on Spark Streaming

In this thesis, we provide an alternative implementation for the distributed versions of the Adaptive Model Rules (AMRules) algorithm, Vertical and Hybrid AMRules. The original implementation was based on Samza, while this one will be based on Spark Streaming. To adjust the algorithm for Spark Streaming, we chose Scala instead of Java, since Spark is written in Scala and provides flexibility in terms of coding. In the following Sections, we show how the algorithms were implemented, so that they could run with Spark Streaming.

4.1 Distributed AMRules Model

Firstly, the basic model should be described for both Vertical and Hybrid AMRules. As shown in Figure 4.1, three main Processor Interfaces (PIs) are needed to create the model. These PIs are the Source PI, the Classifier PI and the Evaluator PI.

The Source PI is responsible for processing the files received from HDFS (Hadoop File System) and forward the data to the Classifier PI. The Source PI receives files from HDFS and transforms the data into Instances, a type in library

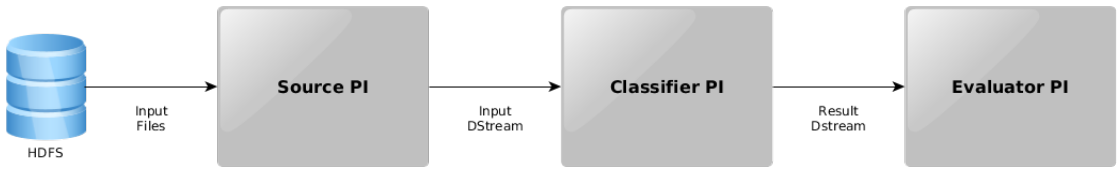


Figure 4.1: AMRules Model

org.apache.samoa.instances. In this process, the attribute values and the class value are assigned to the correct Instance fields. After a batch of data are converted into Instances, an InputDStream is being created, which we can see in lines 3-18 of Listing 4.1, to forward the Dstream to the Classifier PI. Along with the Instance, three more fields are forwarded, one of type Long, which is the instance ID, and two of type Boolean, which suggest the purpose the instance should be used, testing and training, as we are able to see in line 13 of Listing 4.1. Dstreams are being sent to the Classifier PI per certain batch interval. Finally, the Classifier PI receives from the Source PI the DStream of the following type: *DStream[(Long, Instance, Boolean, Boolean)]*. We are able to see in Table 4.1 an example of what the DStream looks like as it enters the Classifier PI with our chosen dataset. Further details about the dataset itself are given in Section 5.2. Since the AMRules algorithm is an one-pass algorithm, the Booleans representing the testing and training are both set to **true**.

The 'Source PI' process runs in lines 8 and 9 of Listing 4.3 (and in lines 9 and 10 of Listing 4.4 for HAMR). In line 8 (line 9 for HAMR listing) we see the initialization of FileReader. We initialiaze the FileReader constractor with the batch interval, the file location and the size of the batch. FileReader is a class that reads the files from HDFS from a certain location and creates the input DStream for the Classifier PI. We get the desirable DStream, which is stored as variable called *sourcepi*, by running the getExamples function in line 9 (line 10 for HAMR). In Listing 4.1 is shown the getExamples function, which takes as parameter the spark streaming context.

Listing 4.1: FileReader's function getExamples Scala Code

```
1  def getExamples(ssc: StreamingContext): DStream[(Long, Instance, Boolean,
       Boolean)] = {
2      prepareForUseImpl()
3      new InputDStream[(Long, Instance, Boolean, Boolean)](ssc) {
4          override def start(): Unit = {}
5          override def stop(): Unit = {}
6          override def compute(validTime: Time): Option[RDD[(Long, Instance,
               Boolean, Boolean)]] = {
7              val examples = Array.fill[(Long, Instance, Boolean, Boolean)]
8              (FileReader.this.batchsizeOption)({
9                  FileReader.this.instanceID += 1
10                 if (FileReader.this.instanceID ==
                       FileReader.this.instanceLimitOption.getValue())
11                     FileReader.this.instanceID = -1
12                     /* one pass algorithm -> isTraining=true && isTesting=true*/
13                 (FileReader.this.instanceID, nextInstance().getData,true,true)
14             })
15             Some(ssc.sparkContext.parallelize(examples))
16         }
17         override def slideDuration = Seconds(slideDurationOption)
18     }
19 }
```

Table 4.1: Classifier PI Input Example

Long	**Instance**	**Boolean**	**Boolean**
621	471, 1.37749, ... 0.00380, 48, 0	true	true
622	472, -1.10091, ..., -0.05723, 0.92, 0	true	true
623	472, 1.01705, ..., 0.02079, 17.57, 0	true	true
624	472, -3.04354, ..., 0.03576, 529, 0	true	true
625	472, 1.04078, ..., 0.01902, 59.88, 0	true	true
626	476, -0.86755, ..., 0.14624, 1, 0	true	true

The Classifier PI receives the DStreams and processes the RDDs inside the DStream to obtain a prediction for each Instance. The model differs between Vertical and Hybrid AMRules, and will be explained further in Sections 4.1.1 and 4.1.2. The result of the Classifier PI has the following form: *DStream[Option[((Long, Instance, Boolean, Boolean),Array[Double])]]*, the second field represents the prediction that was computed for the instance in the first field. The predictions have been defined as type Option, a type in Scala library, so that we can avoid NullPointerException by taking the only the ones, which have been defined. Finally, the result DStream is being sent to the Evaluator PI. We can see an example of the resulting DStream in Table 4.2. We notice in this example, that the instance with instance ID 624 has not been detected as fraud, and that is because the model has not had yet enough fraud transactions to be trained, so that the transaction can be detected correctly. A transaction is considered fraud if the prediction is above 0.5, an example with correctly predicted fraud transaction is displayed in the Table 4.3, where prediction of the transaction with ID 6642 is *0.771200*.

Table 4.2: Classifier PI Output Example 1

(Long, Instance, Boolean, Boolean)	**Array[Double]**
621, 471, 1.37749, ..., 0.00380, 48, 0, true, true	0.001612
622, 472, -1.10091, ..., -0.05723, 0.92, 0, true, true	0.001610
623, 472, 1.01705, ..., 0.02079, 17.57, 0, true, true	0.001607
624, 472, -3.04354, ..., 0.03576, 529, 1, true, true	0.001605
625, 472, 1.04078, ..., 0.01902, 59.88, 0, true, true	0.003205
626, 476, -0.86755, ..., 0.14624, 1, 0, true, true	0.003200

Table 4.3: Classifier PI Output Example 2

(Long, Instance, Boolean, Boolean)	**Array[Double]**
6642, 8169.0, 0.85732, ..., 0.14846, 1.0, 1, true, true	0.771200
6643, 8169.0, 1.42249, ..., 0.00325, 10.95, 0, true, true	0.024292
6644, 8170.0, 1.03006, ..., 0.02752, 118.08, 0, true, true	9.850599E-4

The Evaluator PI computes the binary classification metrics in order to define the accuracy of the algorithms. Spark provides us with the MLlib library, that can compute easily these metrics. The constructor for the metrics is initialized with an RDD, which holds two Double values, one for the predicted value and one for the real class label. Metrics like the precision, ROC Curve or area under ROC Curve, which are accessible through *BinaryClassificationMetrics*, define accuracy. In Listing 4.2 is shown the scala code of the Evaluator PI. The resulting DStream of Classifier PI is stored as a variable called *results*, as shown in Listings' 4.3 and 4.4 twelfth lines, and is processed in the Evaluator PI to acquire metrics like ROC Curve and area under ROC Curve, as shown in lines 5 and 8 of Listing 4.2, respectively.

Listing 4.2: Evaluator PI Scala Code

```
1  results.foreachRDD { rdd =>
2      val predandlabels = rdd.filter(_.isDefined)
3          .map(_.get).map{case (x,y) => (y(0), x._2.classValue())}
4      // ROC Curve
5      val roc = metrics.roc
6      roc.foreach(case (x,y) => println(s"$x\t$y"))
7      // area under ROC
8      val auROC = metrics.areaUnderROC
9      println(s"Area under ROC = $auROC")
10 }
```

4.1.1 Vertical AMRules Design

Figure 4.2 shows the design of Vertical AMrules. The two components the Model Aggregator PI and the Learner PI constitute the Classifier PI. Though, only the Learner PI is distributed, as it was described in Chapter 3, and is represented with blue.

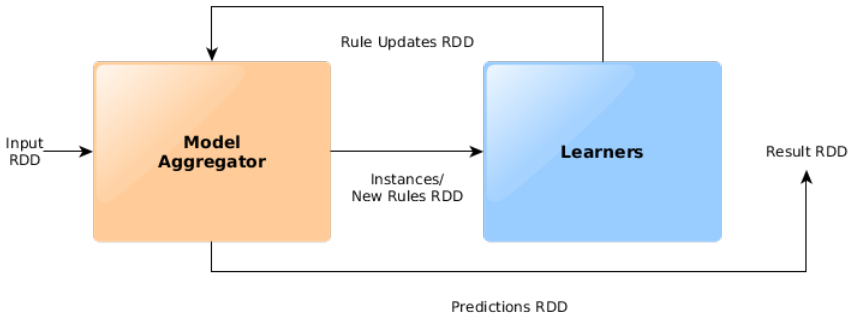


Figure 4.2: Vertical AMRules Classifier Model

The Model Aggregator PI receives the input RDD and processes its data. The operations on the Model Aggregator is non-distributed and will be explained in Section 4.2.1. The Model Aggregator forwards an RDD, which holds the predictions along with the corresponding instance, to the Classifier PI output. Moreover, the Model Aggregator sends the RDD to the Learner PI for the training process. Besides the input RDD from the Source PI, the Model Aggregator receives an RDD from the Learners, in order to update or remove rules in the Rule Set. This RDD is of the following type : *RDD[Option[(Int,ActiveRule,Boolean), (Int,RuleSplitNode,RulePassiveRegressionNode)]]*, where Int type fields represent the ruleID and Boolean type field suggests if the rule is removing. Examples of rule updates and removals are shown in Tables 4.4, 4.5. More specifically, the rules with ruleIDs 10 and 11 in Table 4.4 are being removed from Model Aggregator's rule set, since the Boolean type field is **true**, and the rules with ruleIDs 6,7 and 8 in Table 4.5 update the corresponding rules in Model Aggregator's rule set. In conclusion, the rule type *(Int,ActiveRule,Boolean)* is used to remove a certain rule from the rule set and the rule type *(Int,RuleSplitNode,RulePassiveRegressionNode)* is used to update a certain rule in the rule set.

The Model Aggregator sends rules and instances with the corresponding ruleID to the Learners, so that the classification model can be trained. The Learner PI

Table 4.4: Rule Data Example

Int	**ActiveRule**	**Boolean**
6	*ActiveRule*	false
7	*ActiveRule*	false
10	null	true
11	null	true

Table 4.5: Predicate Data Example

Int	**RuleSplitNode**	**RulePassiveRegressionNode**
6	*RuleSplitNode*	*RulePassiveRegressionNode*
7	*RuleSplitNode*	*RulePassiveRegressionNode*
8	*RuleSplitNode*	*RulePassiveRegressionNode*

receives from the Model Aggregator, a RDD of type: *RDD[(Int,ActiveRule,Boolean)]* and *RDD[(Instance,Int)]*, for adding and updating rules, respectively. For example, in Table 4.4 the rules with ruleID 6 and 7 are added to the corresponding Learners since the Boolean field is **false** and in Table 4.6 the instances are used to update the rule with ruleID 2. These RDDs are being sent separately, so that they can be partitioned by ruleID and achieve a correct parallelism. Finally, after the Learners have processed the RDDs, they send an RDD to the Model Aggregator, so that the rule set can be updated.

Table 4.6: Assignment Data Example

Instance	**Int**
2923.0, -0.14540, ..., 0.15118, 59.95, 0	2
2923.0, -1.72590, ..., 0.13050, 202.41, 0	2
2923.0, -1.07508, ..., 0.14501, 96.52, 0	2

4.1.2 Hybrid AMrules Design

The design in Figure 4.3 represents the model of Hybrid AMRules classifier. This Classifier PI consists of the Model Aggregator PI, the Learner PI, which are dis-

tributed components, and the Default Rule PI, which is a non-distributed component. Here, as well the distributed components are represented with blue.

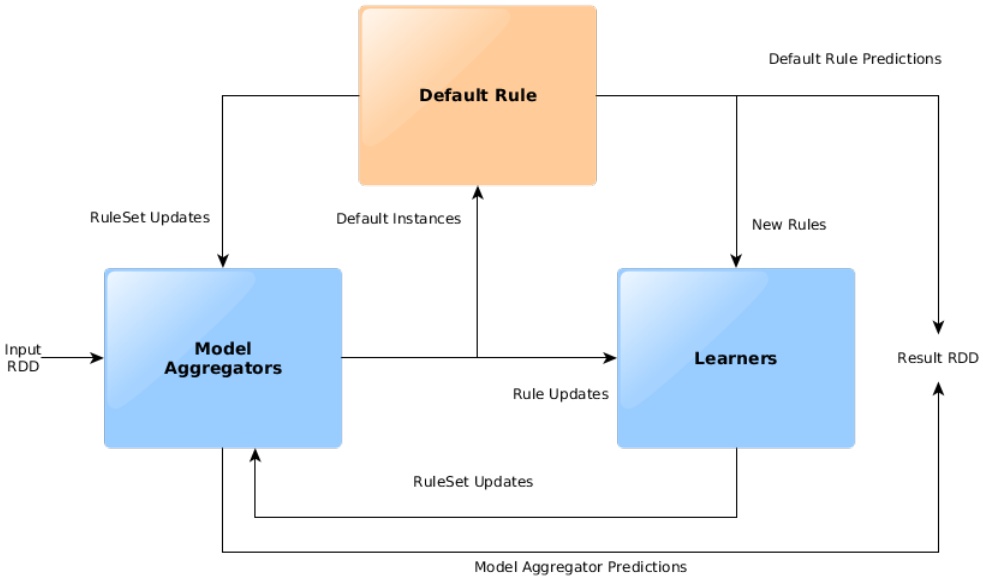


Figure 4.3: Hybrid AMRules Classifier Model

Firstly, the Model Aggregator PI receives the input RDD : *RDD[(Long,Instance, Boolean,Boolean)]*, as shown in Table 4.1, and processes the instances to acquire predictions for each of them. However, if none of the rules in the rule set satisfies an instance, it is sent to the Default Rule PI in the form of the above RDD, first to be tested by the default rule and then to train the default rule. The Model Aggregator PI also sends RDDs to the Learner PI with rule updates, each update is being sent to the correct Learner PI based on the ruleID, which is being updated, to achieve parallelism.

The Learner PIs receive RDDs from the Model Aggregator PI and the Default Rule PI to update each the corresponding rule and to add new rules. The type of RDD the Model Aggregator PI sends to the Learner PI is *RDD[(Int,Instance)]* and the one Default Rule PI sends is *RDD[(Int,(Int,ActiveRule,Boolean))]*, as displayed in Table 4.6 and in Table 4.4 (rules with ruleIDs 6 and 7), respectively. The key

for both RDDs is the ruleID, so that the RDDs can be assigned properly among the Learners.

Finally, the Model Aggregators receive RDDs from the Learners and the Default Rule, in order the rule set to be updated. From the Default Rule PI they receive the following RDD type: *RDD[Option[(Int,ActiveRule,Boolean)]]* and from the Learner PI the RDD of type *RDD[Option[(Int,ActiveRule,Boolean), (Int,RuleSplitNode,RulePassiveRegressionNode)]]*, where again the rule type *(Int, ActiveRule,Boolean)* is used to remove a certain rule from the rule set and the rule type *(Int,RuleSplitNode,RulePassiveRegressionNode)* is used to update a certain rule in the rule set.

4.2 RDD Operations

Spark RDDs support two types of operations: transformations and actions [14]. A Spark transformation is a function that produces new RDD from the existing RDDs. Some basic transformations are *map*, *filter* etc. On the other hand, actions are Spark RDD operations, that return values to the driver program after running a computation on the RDD. For example, *foreach* and *collect* are some actions. Furthermore, transformations are lazy, meaning that they get executed only when an action is called.

Each operation performed on the RDDs to implement the algorithms is explained in Sections 4.2.1 and 4.2.2. However, A generic description of each operation, which is performed either on a RDD or on a DStream, is shown in Tables 4.7 and 4.8.

4.2.1 VAMR Operations

A complete representation of each operation performed on RDDs for the VAMR algorithms is shown in Figure 4.4. Distributed operations are displayed in blue,

Table 4.7: RDD Operations

	Operations	Description
Transformations	filter(func)	Returns a new RDD, containing only elements that meet the predicate function *func*.
	map(func)	The map function iterates over every line in RDD and creates a new RDD. The map() transformation uses any function *func*, and that function is applied to every element of the RDD.
	partitionBy(prt)	Returns a copy of the RDD partitioned using the specified partitioner *prt*.
Actions	foreach()	The foreach() function is used, when there is a need to apply some operations on each element of an RDD without returning anything to the driver.
	collect()	Returns the RDD's content to driver program.

whereas non-distributed are displayed in orange. The classifier receives the DStream with the instances and for each RDD in DStream we perform operations to first test and then train the model.

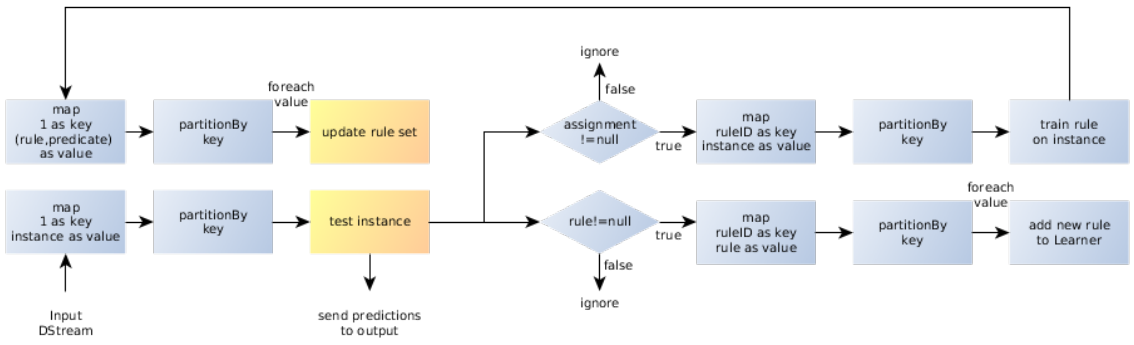


Figure 4.4: Operations VAMR

In the interest of the actions on the Model Aggregator to be non-distributed, the instances are mapped with 1 as key, thus all the instances will be forwarded to one partition after *partitionBy* operation, we can see these operations in line 14 of

Table 4.8: DStream Operations [3]

	Operations	Description
Transformations	transform(func)	Returns a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream. This can be used to do arbitrary RDD operations on the DStream.
Actions	foreachRDD(func)	This function should push the data in each RDD to an external system, such as saving the RDD to files, or writing it over the network to a database. Note that the function *func* is executed in the driver process running the streaming application, and will usually have RDD actions in it that will force the computation of the streaming RDDs.

Listing 4.3. Now, that all the instances are in one partition, they can be tested to acquire the predictions, which will be sent to the output. The result is an RDD which is stored as a variable called *model*. This RDD is cached, as it is shown in line 15 of Listing 4.3, so it can persist in memory and will not be recomputed until the next batch of data. It is important to mention, that all of these operations are transformations, hence non of these are executed until an action is performed on the resulting RDD.

The RDD called *model* is of type *RDD[(Option[(Long,Instance,Boolean), Array[Double]], (Int,ActiveRule,Boolean),(Instance,Int))]* and is used in two separate occasions for the Learner PI. Firstly, it is used to update the learners by adding a new rule, referring to the lines 17-19 of Listing 4.3, in order to do that, the predictions and assignments are removed from the RDD using Spark transformations *filter* and *map*. The resulting RDD's type is *RDD[(Int,(Int,ActiveRule,Boolean))]* and the RDD's contents are being distributed among partitions based on the key, which is

the ruleID. Then a *foreach* action is executed to add the rules to the corresponding learners. Now that an action is being performed on the resulting RDD, every transformation before that is being executed. Secondly, the RDD called *model* is used to train rules with instances, as shown in lines 20-22 of Listing 4.3, and for that reason, the predictions and rules are removed from the RDD, resulting in an RDD of type *RDD[(Int, Instance)]*. The contents of the RDD are partitioned based on the ruleID, so that the rules can be updated with the instances, they covered. The RDD is stored as a variable called *learner*.

The RDD *learner* is called only once to update the Model Aggregator and for that reason there is no need to cache it. On this RDD a filter is applied to filter out undefined updates and also is mapped using 1 as key, so that the process by the Model Aggregator will be non-distributed. We can see the above operations in line 24 of Listing 4.3. In conclusion, the *foreach* action is performed on the resulting RDD to update the Model Aggregator's rule set.

As a result on the complete DStream operation, an RDD with the predictions is being returned. The resulting DStream is of type *DStream[(Long,Instance,Boolean,Boolean), Array[Double]]* and it is used to evaluate the Vertical Adaptive Model Rules algorithm.

Listing 4.3: VAMR Scala Code

```
1 /* Configuration and initialization of model */
2 val conf: SparkConf = new SparkConf().setAppName("VAMR")
3 // Spark Streaming Context
4 val ssc = new StreamingContext(conf, Seconds(args(0).toInt))
5 val myhashpartitioner = new HashPartitioner(ssc.sparkContext.defaultParallelism)
6
7 /* SourcePI */
8 val sourcepi: FileReader = new FileReader(args(0).toInt, args(1),args(2).toInt)
9 val instances = sourcepi.getExamples(ssc)
```

```
10
11 /* Classifier PI */
12 val results = instances.transform{ rdd =>
13     /* Model Aggregator PI (Test)*/
14     val model = rdd.map((1, _)).partitionBy(myhashpartitioner)
15         .map(x => classifierpi.aggregator.processInstance(x._2)).cache()
16     /* Learner PI (Train) */
17     model.filter(_._2 != null).map { case (_, r, _) => (r._1, r) }
18         .partitionBy(myhashpartitioner)
19         .foreach(data => classifierpi.learner.addProcess(data._2))
20     val learner = model.filter(_._3 != null).map{case (_, _, a) => (a._2,a._1)}
21         .partitionBy(myhashpartitioner)
22         .map(a => classifierpi.learner.trainProcess(a.swap))
23     /* Model Aggregator PI (Rule Updates) */
24     learner.filter(_.isDefined).map((1, _)).partitionBy(myhashpartitioner)
25         .foreach(input => classifierpi.aggregator.updateProcess(input._2.get.swap))
26     model.map { case (pred, _, _) => pred }
27 }
```

It is important to note, that the decision for choosing to use 1 as key for the non-distributed operation of the algorithm and to use partitionBy to assign the data in to partitions based their key was made in the interest of time improvement. Originally, it was chosen repartition and groupBy for the respective operations. Repartition is an operation that can change the number of partitions of an RDD. This operation was chosen to repartition the RDD to one partition, so that the non-distributed process could be implemented, though it was noticed that repartitioning the RDDs it was a time consuming process and as result of that it was decided to leave the RDD partitions as is and forward the data to one partition based on their key. Furthermore, groupBy is an operation that groups the data based on a key. GroupBy was used to group data base on the given key, so that they could be processed together, but to

group and iterate afterwards through them made it time consuming. And for that reason it was chosen partitionBy, which only defines the partition of each instance based on the key and does not need to iterate through the data afterwards, so that they can be processed as a group, which is less time consuming.

4.2.2 HAMR Operations

Here as well, each operation is shown in the following Figure 4.5 and distributed from non-distributed operations are distinguishable through the color difference.

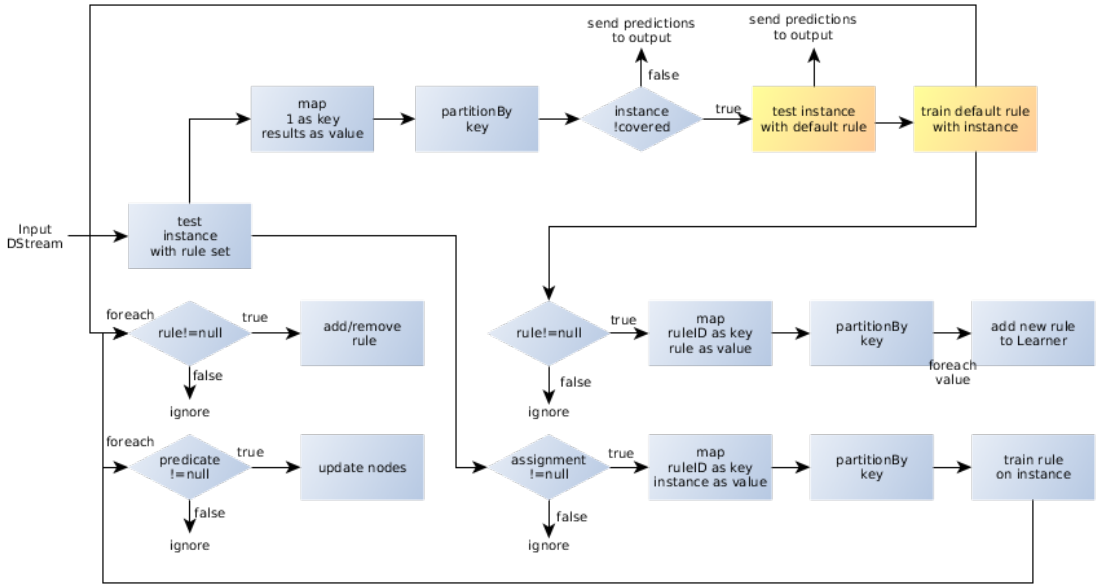


Figure 4.5: Operations HAMR

Firstly, the Model Aggregators process the input RDD and test the instances with the rule set to get a prediction. Since, in this algorithm the Model Aggregator PI is implemented in a distributed fashion, there is no need to forward any instance to a particular partition. Besides the predictions, assignments (rule updates) and instances, that were not able to be covered by any rule, are forwarded to Learner PI and DefaultRule PI, respectively.

The transformed RDD, that resulted from the Model Aggregator PI, is being transformed further, so that it can be processed by the Default Rule PI. The contents of the RDD are mapped with 1 as key to achieve non-distributed implementation for the Default Rule PI, as shown in line 14 of Listing 4.4. The instances, that they were not covered are tested with the default rule, as shown in line 23 of Listing 4.4, to get the predictions and then the default rule is trained by them, as shown in line 31 of Listing 4.4. The resulting RDD of these operations is cached and stored in a variable named *model*. Since all the above operations are transformations, none of these have been processed yet.

The RDD *model* is called in the next step, so that learners can be updated. Since, the Learners' updates conclude in a *foreach* action, as we are able to see in line 46 of Listing 4.4, all the transformations stored in *model* are performed. The *model* RDD is transformed with *filter* to exclude undefined rule updates' field and with *map* to set ruleID as key, so that *partitionBy* transformation can be used to partition the contents properly to the learners and update the correct rule. Moreover, *model* RDD's contents are used to train a rule with the instance that covered the rule, as we can see in line 49 of Listing 4.4. For that reason, it is being filtered to exclude the undefined assignments'. A *map* transformation is performed, which results in the following RDD type: RDD[(Int,(Instance,Int))], with the Int types defining ruleIDs and Instance types the corresponding covered instances, this RDD is used to train the rules.

Finally, the Model Aggregator PI's rule set needs to be updated with the updates from the Default Rule PI and Learner PI, and that is shown in lines 52-55 of Listing 4.4. To achieve that the defined updates are selected with the *filter* transformation and then *foreach* action is used to update the rule set.

Listing 4.4: HAMR Scala Code

```
/* Configuration and initialization of model */
val conf: SparkConf = new SparkConf().setAppName(s"HAMR-${2/*args(3)*/}")
// Spark Streaming Context
val ssc = new StreamingContext(conf, Seconds(args(0).toInt))
val myhashpartitioner = new HashPartitioner(ssc.sparkContext.defaultParallelism)
val myhashpartitionerLearner = new HashPartitioner(args(3).toInt)

/* SourcePI */
val sourcepi: FileReader = new FileReader(args(0).toInt, args(1), args(2).toInt)
val instances = sourcepi.getExamples(ssc)

val results = instances.transform{ rdd =>
    /* Model Aggregator PI*/
    val model = rdd.map(classifierpi.model.processInstance).map((1,_))
        .partitionBy(myhashpartitioner)
        .map{case (_,prediction) =>
            /* Default Rule PI ( Test Instance and Train Rule) */
            val defaultRulePrediction: scala.Option[((Long, Instance, Boolean,
                Boolean),Array[Double])] = {
                if (prediction._3.isDefined){
                    if (prediction._3.get._4)
                        // Test Instance
                        Some(prediction._3.get,
                         classifierpi.root.processInstance(prediction._3.get._2).get)
                    else None
                } else None
            }
            val ruleUpdatesRoot: scala.Option[(Int, ActiveRule, Boolean)] = {
                if (prediction._3.isDefined){
                    if (prediction._3.get._3)
                        // Train default rule
                        classifierpi.root.trainOnInstanceImpl(prediction._3.get._2)
```

```
32                  } else None
33              else None
34          }
35          val predResult = {
36              if (prediction._1.isDefined) prediction._1
37              else if (defaultRulePrediction.isDefined) defaultRulePrediction
38              else None
39          }
40          (predResult, ruleUpdatesRoot, prediction._2)
41      }.cache()
42
43  /* Learner PI */
44  model.filter(_._2.isDefined).filter(_._2.get != null)
45      .map{case (_,r,_) => (r.get._1,r.get)}.partitionBy(myhashpartitioner)
46      .foreach(r => classifierpi.learner.updateProcess(r._2))
47  val learner = model.filter(_._3.isDefined).filter(_._3 != null)
48      .map{case (_,_,a) => (a.get._2,a.get)}.partitionBy(myhashpartitioner)
49      .map(a => classifierpi.learner.trainRuleOnInstance(a._2._2,a._2._1))
50
51  /* Model Aggregator (Rule Updates) */
52  model.filter(_._2.isDefined).map{case (_,r,_) => (r.get)}
53      .foreach(r => classifierpi.model.updateProcess(null, r))
54  learner.filter(_.isDefined)
55      .foreach(x => classifierpi.model.updateProcess(x.get._2,x.get._1))
56
57  model.map{case (pred,_,_) => pred} // return predictions
58 }
```

Chapter 5

Experimental Results

In this Chapter, we provided results of the experiments performed for each algorithm measuring their accuracy and scalability. The accuracy of the algorithms will be shown through the BinaryClassificationMetrics and the scalability through various changes in batch size and parallelism.

5.1 Setup

For the evaluation of the algorithms, the experiments were conducted using Spark on YARN cluster, the services of which run on HDP 2.6.3.0 (Hortonworks Data Platform). The Softnet Lab of Technical University of Crete cluster holds 22 servers, where the operating System of each server is Ubuntu 14.04. Finally, we used the HDFS to store our files.

5.2 Dataset

The dataset used for the experiments acquired from Kaggle [8].The dataset has been collected and analyzed during a research collaboration of Worldline and the Machine

Learning Group of ULB (Universit Libre de Bruxelles) on big data mining and fraud detection.

The dataset contains transactions made by credit cards in September 2013 by European cardholders. It presents transactions that occurred in two days, where 492 out of 284,807 transactions are frauds. It is highly unbalanced, since the positive class (frauds) account for 0.172% of all transactions.

Table 5.1: Dataset Class Statistics

Class	**Rows**	**Percentage**
Negative (legit)	284315	99.827
Positive (fraud)	492	0.1727

The dataset contains only numerical attributes which are the result of a PCA transformation. Obviously, due to confidentiality issues, the original features and more background information about the data cannot be provided. Features V1, V2, ... V28 are the principal components obtained with PCA, the only features which have not been transformed with PCA are 'Time' and 'Amount'. Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature 'Amount' is the transaction Amount. Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.

Table 5.2: Dataset Example

Time	**V1**	**V2**	**...**	**V28**	**Amount**	**Class**
471	1.37749	-0.66256	...	0.00380	48	"0"
472	-1.10091	1.02958	...	-0.05723	0.92	"0"
472	1.01705	-0.18504	...	0.02079	17.57	"0"
472	-3.04354	-3.15730	...	0.03576	529	"1"
472	1.04078	0.10956	...	0.01902	59.88	"0"
476	-0.86755	-0.41863	...	0.14624	1	"0"

5.3 Accuracy Evaluation

BinaryClassificationMetrics of spark.mllib provides a suite of metrics for the purpose of evaluating the performance of machine learning models. The current application of Credit Card Fraud Detection falls under the classification type of machine learning. The evaluation of classification models all share similar principals. In a supervised classification problem, like this one, exists a true output and a model generated predicted output for each data point, where each data point can be assigned to one of the four categories :

- **True Positive :** label is fraud and the prediction is also fraud
- **True Negative :** label is legit and the prediction is also legit
- **False Positive :** label is legit, but the prediction is fraud
- **False Negative :** label is fraud, but the prediction is legit

In Table 5.3, the predictions for the instances are shown based on the four categories described above. It is noticeable that the True Positives are almost half total number of frauds, that is because the chosen dataset is highly unbalanced. The choice, to leave the dataset as is and not make any alterations to balance frauds and legits, has been made, in order to simulate a real life experiment with real life data, and see how the algorithms perform under these circumstances.

Table 5.3: Evaluation

Algorithms	**TP**	**TN**	**FN**	**FP**
VAMR	262	284292	230	23
HAMR	254	284297	238	18

At the TP column, we notice that a slight difference in predictions occurs. The reason behind it is, that in VAMR implementation 4.3 we only have rule set updates

to the Model Aggregator from the Learner PI, in contrast with HAMR implementation 4.4, where we have rule set updates to model aggregator from both Default Rule PI and Learner PI. This means that the updates from Default Rule PI do not happen concurrently in HAMR like in VAMR. And to be more specific, in line 53 of Listing 4.4 is shown, that the updates on the Model Aggregator PI from the default rule do not happen, while the Model Aggregator PI first processes the instance like in VAMR implementation, where if there are any updates from default rule, these happen as the Model Aggregator PI processes the Instance in line 15 of Listing 4.3.

However, pure accuracy is not generally a good metric, when a dataset is highly unbalanced. A good way to examine an algorithm's accuracy is metrics like ROC curve, because they take into account the type of error. The way to examine the algorithms and evaluate their accuracy is to get their Receiver Operating Characteristic (ROC) curves. In statistics, ROC curves illustrate the diagnostic ability of a binary classifier system as its discrimination threshold is varied. Basically, the way to compare algorithms based on their ROC curve is to calculate the area under ROC using the following type $\int_0^1 \frac{TP}{P} \mathrm{d}(\frac{FP}{N})$. The auROC for VAMR equals with 0.9317 and the auROC for HAMR is 0.9315. Due to the fact that the area under curve is about the same, we are able to conclude that the algorithms have the same level of accuracy. Moreover, since the area under curve for both algorithms is close to 1, we can confidently say, that we have a great classifier.

In order to get a clearer picture about the algorithms' accuracy, we used the same dataset for the Samza implementation and the results, that were obtained, are shown in Table 5.4 as well as the ROC curve in Figure 5.3. As it is presented in the table, the TPs and the TNs are about the same as the VAMR implementation in Spark Streaming, despite the fact that Samza streams the instances sequentially and Spark streams micro-batches of instances. On the subject of the ROC curve and the area under it, there is similarity but it is lower in Samza implementation. The area under

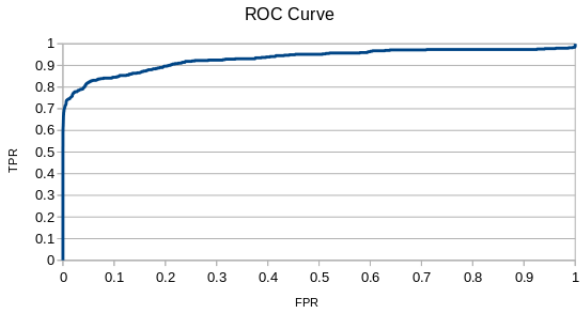


Figure 5.1: VAMR ROC curve

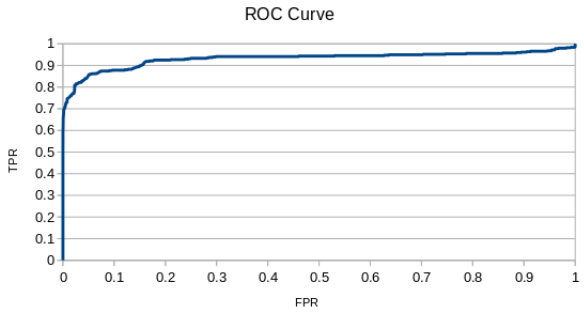


Figure 5.2: HAMR ROC curve

ROC in Samza implementation equals with 0.9060, that is because there are more FPs in comparison with Spark Streaming implementations.

Table 5.4: Samza results

	TP	**TN**	**FN**	**FP**
Samza implementation	261	284288	231	27

5.4 Performance Evaluation

To evaluate the performance regarding time on Spark Streaming implementation, experiments were performed on the Softnet cluster of the Technical University of Crete.

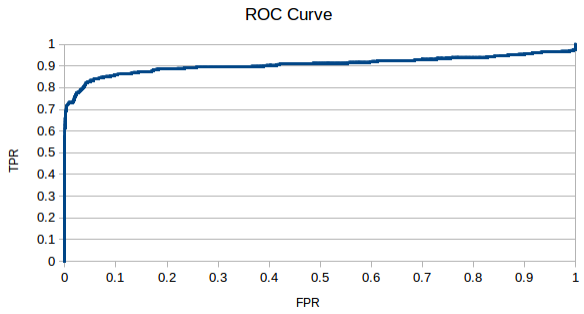
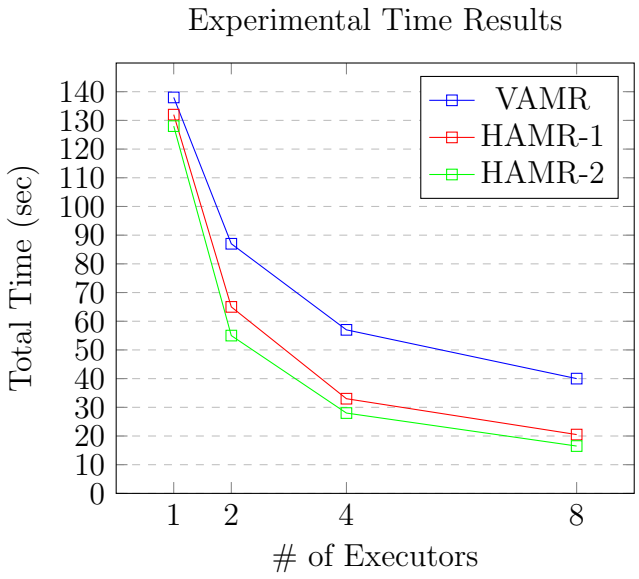


Figure 5.3: Samza ROC curve

The first experiment shows the scalability of the algorithms. For each experiment performed, the timeout was set to 250 secs, the interval was set to 1 sec and the amount of instances per interval was set to 5K. Meaning that, for each experiment the total amount of instances, that were processed, was 1.125mil. Algorithms HAMR-1 and HAMR-2 represent the Hybrid AMRules algorithm with one and two learners assigned to it, respectively. Furthermore, the parallelization of Model Aggregator for the HAMR algorithm corresponds to the assigned executors. To show the scalability of the algorithms, the number of executors was increased. Each algorithm run for 1, 2, 4 and 8 executors.

The results for the scalability experiments are shown in the plots above. As expected the run-time for each algorithm decreases, when the number of executor increases. For example, in VAMR algorithm, when the number of executors increases from 1 to 2, to 4, to 8 the total processing time increases from 138, to 87, to 57, to 40 secs. It noticeable that the time does not completely cuts down in half, the reason is that in VAMR algorithm the Model Aggregator PI is a non-distributed component, hence it is not expected the time to decrease fully in half. Moreover, there is an apparent time difference from VAMR to HAMR-1 and HAMR-2, due to the fact that in HAMR algorithm the Model Aggregator PI, which processes the input instances, is also parallelizable. As shown in plot, when there is no parallelization, meaning

Figure 5.4: Experimental Time Results

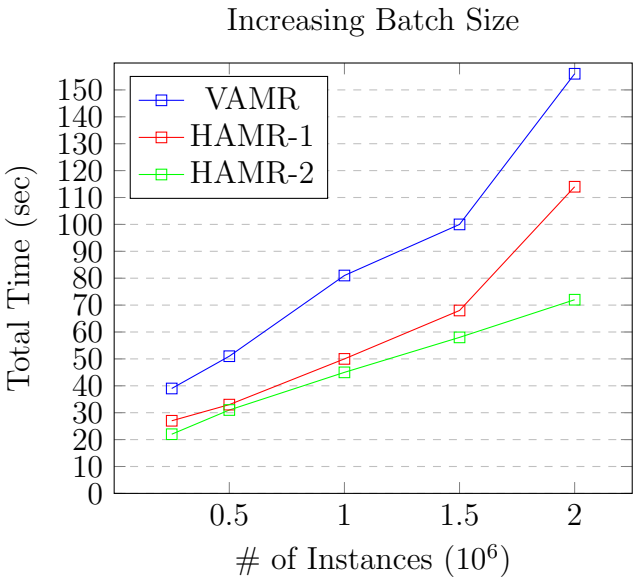


there is only one executor, the runtimes are about the same, the reason for that is that the algorithm is the same without any parallelization, so it takes the same time to process.

In the second experiment, that was performed, we increased the batch size on VAMR, HAMR-1 and HAMR-2. For all the experiments of increasing batch size, we assigned 2 executors, that is because we have already concluded that the time decreases as we increase the number of executors and we wanted to see how our Spark Streaming implementation reacts with 2 worker nodes, regarding the run-time. Moreover, we set the timeout to 250 seconds. As a result of that, for 1K, 2K, 4K, 6K and 8K instances per second, the application will process 250K, 500K, 1.mil, 1.5mil and 2mil instances in total until it finishes running. And to be more exact, the tasks process a total input size of 153.1MB, 301.2MB, 1GB, 2.4GB and 4GB, respectively.

As it is shown in the figure above, the HAMR-1 and HAMR-2 plots are below the VAMR plot. That is because VAMR parallelizes only the Learner PI, where HAMR parallelizes both Model Aggregator PI and Learner PI. Furthermore, it is

Figure 5.5: Increasing Batch Size



noticeable, that when the amount of instances per second increases the total time of processing the data increases. Finally, it is apparent that after the 6K instances per second mark (1.5mil) the algorithms VAMR and HAMR-1 slow down, in comparison with algortihm HAMR-2, where this algorithm has parallelization both in the Model Aggregator PI and the Learner PI.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Our goal for this thesis was to utilize the community of Apache Spark and adapt the distributed Adaptive Model Rules algorithm to Spark Streaming. The accessibility to data is creating new opportunities for machine learning applications like Credit Card Fraud Detection, where it is important to be useful and effective. Nowadays, credit cards are used all the time for our daily transactions, it is of course useful to develop an algorithm, that can detect fraudulent transactions. Spark's distributed environment, is designed to address the challenges of data's volume, velocity and variety. The algorithm's effectiveness is proven by the experiments performed, which show the fast processing of big amount of data as well as good accuracy. With our adaption of the algorithm from Samza to Spark Streaming we were able to further improve some of the problems at hand.

6.2 Future Work

The fact that there is a reduction of accuracy in VAMR and HAMR lies on the existence of cyclic topology of the algorithm, i.e., model aggregators send instances

to learners and learners are expected to send back updates to model aggregator. These cycles contribute to the rule update delay in model aggregators. Besides the fact that there is a slim reduction of accuracy in the evaluation, it is expected to increase as we give input more instances. Thus, it would be useful to design an acyclic topology to improve accuracy. Otherwise, the support for priority in streams could also remove this problem.

Bibliography

[1] Ezilda Almeida, Carlos Ferreira, and Joao Gama. Adaptive model rules from data streams. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 480–492. Springer, 2013.

[2] Comparisons Spark-Streaming. https://samza.apache.org/learn/documentation/0.8/comparisons/spark-streaming.html. Accessed: 2018-06-01.

[3] Apache Spark. https://spark.apache.org/. Accessed: 2018-04-02.

[4] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[5] Richard J Bolton and David J Hand. Statistical fraud detection: A review. *Statistical science*, pages 235–249, 2002.

[6] Andrea Dal Pozzolo, Olivier Caelen, Yann-Ael Le Borgne, Serge Waterschoot, and Gianluca Bontempi. Learned lessons in credit card fraud detection from a practitioner perspective. *Expert systems with applications*, 41(10):4915–4928, 2014.

[7] Tom Fawcett and Foster Provost. Adaptive fraud detection. *Data mining and knowledge discovery*, 1(3):291–316, 1997.

[8] Kaggle - Credit Card Fraud. https://www.kaggle.com/mlg-ulb/creditcardfraud. Accessed: 20178-06-06.

[9] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning spark: lightning-fast big data analysis*. "O'Reilly Media, Inc.", 2015.

[10] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[11] Tova Milo, Slava Novgorodov, and Wang-Chiew Tan. Interactive rule refinement for fraud detection. EDBT, 2018.

[12] Christian Robert. Machine learning, a probabilistic perspective, 2014.

[13] Jim Scott. Stream processing everywhere what to use? https://mapr.com/blog/stream-processing-everywhere-what-use/. Accessed: 2018-06-01.

[14] Spark RDD Operations-Transformation and Action with Example. https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/. Accessed: 2018-07-20.

[15] Anh Thu Vu, Gianmarco De Francisci Morales, João Gama, and Albert Bifet. Distributed adaptive model rules for mining big data streams. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 345–353. IEEE, 2014.

[16] Reynold S Xin, Daniel Crankshaw, Ankur Dave, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: Unifying data-parallel and graph-parallel analytics. *arXiv preprint arXiv:1402.2394*, 2014.

[17] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion 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*, pages 2–2. USENIX Association, 2012.

[18] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[19] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

[20] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. *HotCloud*, 12:10–10, 2012.