SCARFF: a Scalable Framework for Streaming Credit Card Fraud Detection with Spark [1]

Fabrizio Carcillo[a], Andrea Dal Pozzolo[a], Yann-Aël Le Borgne[a], Olivier Caelen[b], Yannis Mazzer[b], Gianluca Bontempi[a]

[a] *Machine Learning Group, Computer Science Department, Faculty of Sciences ULB, Université Libre de Bruxelles, Brussels, Belgium.*
(email: {fcarcill, yleborgn, gbonte}@ulb.ac.be, dalpozz@gmail.com)
[b] *R&D High Processing & Volume team, Worldline, Belgium.*
(email: {yannis.mazzer, olivier.caelen} @worldline.com).

Abstract

The expansion of the electronic commerce, together with an increasing confidence of customers in electronic payments, makes of fraud detection a critical factor. Detecting frauds in (nearly) real time setting demands the design and the implementation of scalable learning techniques able to ingest and analyse massive amounts of streaming data. Recent advances in analytics and the availability of open source solutions for Big Data storage and processing open new perspectives to the fraud detection field. In this paper we present a SCAlable Real-time Fraud Finder (SCARFF) which integrates Big Data tools (Kafka, Spark and Cassandra) with a machine learning approach which deals with imbalance, nonstationarity and feedback latency. Experimental results on a massive dataset of real credit card transactions show that this framework is scalable, efficient and accurate over a big stream of transactions.

Keywords: Big Data, Fraud Detection, Streaming Analytics, Machine Learning, Scalable Software, Kafka, Spark, Cassandra

1. Introduction

The increasing adoption of electronic payments is opening new perspectives to fraudsters and asks for innovative countermeasures to their criminal

activities. If on the one hand fraudsters continuously improve their techniques to emulate genuine behaviour, on the other hand it becomes affordable for the companies managing transactional services to collect data about customers and monitor their behavior.

The need of automatic systems able to detect frauds from historical data led to the design of a number of machine learning algorithms for fraud detection [1, 2, 3]. Supervised methods, typically based on binary classification, as well as unsupervised and one-class classification [4, 5] have been proposed in literature. Most of these works address some specific issues of fraud detection, notably class imbalance [6, 7, 8] (the percentage of fraudulent transactions is usually very small), concept drift [9, 10, 11, 12, 13, 14] (the distribution of fraudulent transactions might change in time) and stream processing [15, 16].

The authors of this paper studied and analysed in detail the existing literature in previous works [17, 18, 19, 20] and proposed an original solution for accurate classification of fraudulent credit card transactions in imbalanced and non-stationary settings. In particular we assessed the superiority of undersampling versus oversampling techniques in our specific problem, we proposed a sliding window approach to effectively tackle concept-drift and we addressed in [19, 20] an issue often overlooked in literature: the *verification latency* due to the fact that in real settings the transaction label is obtained only after that human investigators contacted the card holders.

Though a large number of learning techniques have been proposed, most solutions assume a conventional setting where the entire dataset is resident in memory. It follows that very few studies made the implementation of these techniques scalable and studied their performances. Also what exists is typically related to other domains than the fraud: for instance [21] and [22] studied already the issue of data imbalance in a Hadoop/MapReduce framework [2] but only for public and bioinformatics data.

In domains closer to fraud detection most of the existing works are preliminary or in progress. H. Hormoz et al. [23] made a comparison between a serial implementation and a Hadoop/MapReduce batch processing solution based on Artificial Immune Systems (AIS). The same authors made some tests on cloud services and provided accuracy measurements [24]. A web service framework for near real-time credit card fraud detection is described, together with some preliminary results, in [25]. A big data architecture based

[2] `https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html`

on Flume, Hadoop and HDFS is proposed in [26] but no validation results are provided. An example of application in a non banking environment is presented in [27] where J. Chen et al. describe the Hadoop based fraud detection infrastructure at Alibaba. Other works in progress can be found on several git servers [28, 29, 30, 31, 32, 33, 34].

In this paper we start from the conclusions of our published works [17, 19, 20] and we propose a realistic and scalable implementation of a fraud detection system. SCARFF (SCAlable Real-time Fraud Finder) is an open source platform which processes and analyses streaming data in order to return reliable alerts in a nearly real-time setting. These are the main original contributions:

1. the design, implementation and test of an entirely open-source solution integrating state-of-the-art components from the Apache ecosystem. This architecture deals seamlessly with data ingestion, streaming, feature engineering, storage and classification;
2. a scalable learning solution able to provide accurate classification in a context characterized by nonstationarity, class imbalance and verification latency. This is obtained by implementing in a scalable and distributed manner an ensemble solution able to deal with concept drift and delayed feedback;
3. the design of a distributed on-line feature engineering functionality, which constantly updates historical features relevant to better identify fraud patterns. This on-line functionality relies on a MapReduce programming model;
4. a real-world extensive assessment, in terms of scalability, computational performance and precision, carried out by testing the platform on a stream of more than 8 millions of transactions (corresponding to more than 1.9 millions of cards) provided by our industrial partner;
5. the virtualisation of the complete workflow proposed in this article as a Docker container, making the workflow fully reproducible.

The paper is organized as follows. Section 2 introduces the main characteristics of real-world Fraud-Detection Systems. Section 3 gives an overview of the big data tools from the Apache ecosystem that are integrated in our framework. Section 4 details the learning and the streaming functionalities of the platform. Finally, in section 5 we assess the scalability, computational speed and precision on a real dataset, as a function of allocated resources and incoming transaction rates.

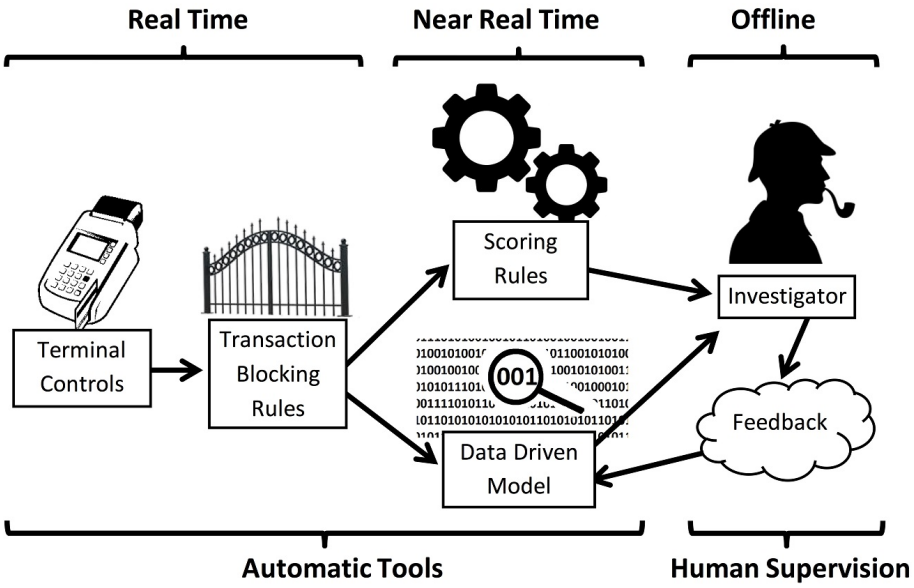

Figure 1: A diagram illustrating the layers of control in a FDS [35, 20]. The paper focuses on the data driven model part.

2. Real-world Fraud Detection Systems

Real world Fraud-Detection Systems (FDSs) for credit card transactions rely on both automatic and manual operations [35, 20] (Fig. 1). Manual operations are performed offline by human investigators, while automatic components are implemented by algorithms that work in real-time and near real-time configurations. Real-time operations take place before the payment is authorized, while near real-time operations are executed after the payment occurred.

Real-time processing consists of a set of security checks of the transaction. If these checks are not passed, the transaction is stopped, otherwise the amount is virtually transferred from one account to another. Real-time operations can be divided in Terminal controls and Transaction Blocking Rules (TBRs). Terminal controls concern terminal-card interaction (e.g.

checking if the PIN code is correct) or terminal-server interaction (e.g. checking if there is a sufficient balance on the account). TBRs are a set of *if/else* conditions, properly designed by fraud experts to block evident fraudulent attempts (e.g. *IF attempt from a shop in black list THEN deny transaction*). Those rules are seldom updated and because of their real-time nature (execution in milliseconds) they cannot rely on a feature engineering step returning complex features (e.g. cardholder profile or past cardholder behaviour).

Once the payment has been registered, near real-time operations are used to score transactions for fraud investigation. Two near real-time kinds of control are typically performed: Scoring Rules (SRs) and Data Driven Model (DDM).

SRs are expert based rules like the TBRs, but of a more complex nature since they can take advantage of the output of a feature engineering step. For instance these rules can use the cardholders profile and behaviour (e.g. *IF the cardholder is a 80 year old man who never used his card on-line AND the transaction involves a big amount AND the transaction comes from an offshore website THEN return a score of 0.9*). The SRs output is a score obtained by merging the output of multiple rules and it is used to raise, if necessary, an alert on the transaction. SRs are known to be effective for specific fraudster behaviours or recurrent fraudulent patterns.

The second near real-time component is the Data Driven Model (DDM) which is based on Machine Learning classifiers trained to predict the probability of a new transaction to be fraudulent. The scalable implementation of this module, which is the main focus of this paper, will be detailed in section 4.

The offline control layer is managed by investigators who take care of the *alerts* returned by the TBRs, SRs and the DDM. By *alert* we mean a transaction associated to high fraud risk and for which a human investigation is needed. In practice an alert is raised according to one of those criteria:

- the estimated risk of fraud associated to the transaction is over a threshold;
- the transaction belongs to the *top-N* transactions with the highest risk.

In the first case we may obtain an unpredictable number of alerts per day, while in the second case we may better organize the effort of the investigators by asking them to process alerts at a constant pace.

The number of credit cards a team of investigators can process depends on the organization guidelines, as well as the number of investigators available for such a task. Usually an organization investing a certain amount of money on investigation (i.e. employing a number of investigators), expects that at least a given number of alerts will be examined by its investigators. For this reason we have chosen to implement the second option in our pipeline.

3. The Big Data ecosystem

This paper proposes a scalable implementation of the DDM learning module which relies on standard tools from the Apache ecosystem, notably Kafka, Spark and Cassandra (Fig. 2). A major advantage of these components is that they similarly handle fault tolerance and tasks distribution.

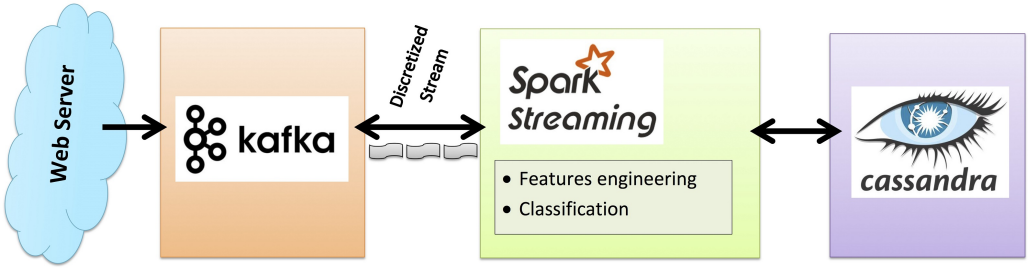

Figure 2: The Big Data pipeline. In our experimental setting, we used a bash program to emulate the web server which inputs data in the pipeline.

3.1. Transactions Collection

Apache Kafka[3] is a distributed publish-subscribe messaging queue system that is commonly used for log collection. It has a multi-producers management system able to retrieve messages from multiple sources. For testing purposes we emulate the streaming through a bash program injecting transactions in Kafka at a desired rate per second. In case of need (e.g. system outage) the transactions may be retrieved also during a time interval (set by the user) posterior to their processing. In general, data partitioning and retention make of Kafka a useful tool for fault tolerant transaction collection [36].

[3] `http://kafka.apache.org`

3.2. Data Analysis

Apache Spark[4] is an in-memory, streaming-enabled, Map-Reduce implementation which automatically distributes the computation among the assigned resources and aggregates the results on a distributed file system. The central idea of this tool is to organize data in a distributed object, the Resilient Distributed Dataset (RDD) [37]. In case of partition lost, the RDD object contains sufficient information to retrieve the data structure [37, 38]. Spark includes a built-in library for machine learning (package `MLlib` [39]), as well as one for streaming (package `Streaming`). A strong point of Spark is its capacity to enable batch and streaming analysis in the same platform.

The proposed framework relies on Spark Streaming which processes data stream in mini-batches trailing latency of the order of seconds. Though this could be considered as a disadvantage in some streaming contexts, it is harmless in our nearly real-time setting.

The Spark module of the framework is written in Scala [40], a language which combines object-oriented and functional programming. Scala runs on top of Java VM and it is fully compatible with Java libraries. Overall, Spark accomplishes three missions in our pipeline: the aggregation of historical transactions to perform feature engineering, the online classification of the transactions returning the estimated fraud risk and the cold storage of transactions in Cassandra.

3.3. Data Storage

Apache Cassandra[5] is a distributed database designed for scalability, able to support replication across multiple nodes or datacenters. It offers linear scalability, fault tolerance, low latency when querying [41] and manages consistency of requests at the node level. Data is stored on multiple nodes organized in a ring shape (i.e. there is no master and every node is as important as the others), thus avoiding a single point of failure. The creation of a Cassandra table requires the setting of some parameters (e.g. the primary key) having an impact on performances. We use a compound primary key made of a partition and a clustering key. The partition key is an identifier of the hour when the transaction has been received, making easy to retrieve old transactions and to compute statistics for a certain cardholder over a given

[4] `http://spark.apache.org`
[5] `http://cassandra.apache.org`

period. The clustering key defines the order of the records in a partition and it is composed of the card identifier and the timestamp.

4. Online learning and streaming solutions

This section details the functionalities of the proposed framework. Our pipeline implements two main functionalities: a machine learning classification engine and a streaming component. In the first subsection, 4.1, the selected machine learning techniques are described. The machine learning engine includes a weighted ensemble of two classifiers. The second subsection, 4.2, focuses on the streaming component. Here, more details will be given regarding the data preprocessing (4.2.2), the data throughput (4.2.1), the features engineering (4.2.3), the online classification(4.2.4) and the data storage (4.2.5).

4.1. The machine learning engine

This module is designed on the basis of our recent research in fraud detection [19, 20] and it aims to take into account the specificity of a Fraud Detection System where automatic tools have to interact with human investigators. The role of fraud investigators is to focus on the most suspicious transactions and to contact cardholders. This means that the automatic system receives a binary feedback (fraud or genuine) only on the small subset of transactions (few hundreds per day) which triggered an alert. For the rest of the transactions no feedback is received unless the cardholder reports a fraud. This means that non-alerted transactions can be assumed to be genuine only after some time. The learning strategy discussed in [19, 20] and implemented here, is able to integrate this verification latency by taking into consideration both transactions for which we have investigators' *feedback* and those labeled by customer with some *delay*. In particular, the classification relies on Random Forests [42, 43], which have been shown to be particularly effective in fraud detection problems [44, 45, 46].

The resulting algorithm estimating the risk of fraud is then composed of two classifiers:

- a *Feedback* Random Forest classifier $\mathcal{F}_t$ trained on the observations generated in the last f days and for which a *Feedback* was returned by investigators;

- a *Delayed* classifier $\mathcal{D}_t$ made of an ensemble of Balanced Random Trees (BRTs) [47, 43] trained on the old transactions for which we can reasonably consider the class as known. Note that this classifier is typically learned on a much larger number of samples than the *Feedback* one.

Every tree in $\mathcal{D}_t$ is day specific, i.e. it uses only transactions of a given day (Fig. 3). This allows an easier distribution of the computation and aggregation of the results.

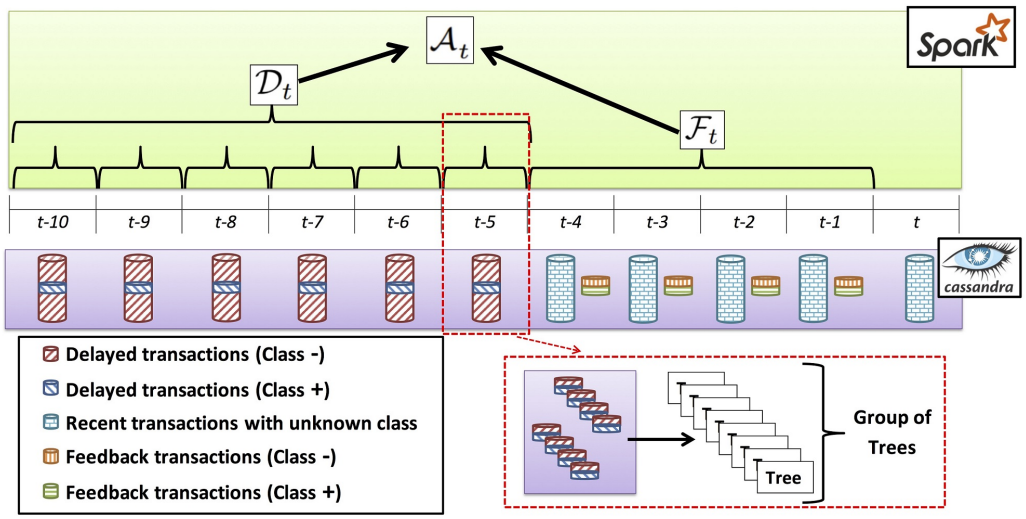

Figure 3: In this illustrative example, the Feedback model is a Random Forest trained on investigator feedback from day $t-4$ to $t-1$. The Delayed model is an ensemble of Balanced Random Trees (BRTs), each trained on the observations of every single day from day $t-10$ to $t-5$. The transaction risk score is a function of the scores of the two models. Note that the *Delayed* classifier follows a sliding window approach: as new BRTs are trained and added to the ensemble, the oldest ones are discarded. The same is true for the *Feedback* classifier, where a sliding window approach is followed when selecting the subset of transactions used for the classifier training.

Though the two classifiers are updated periodically (e.g. once per day), they are continuously used in the streaming module (subsection 4.2) to assess the risk of fraud.

In order to deal with concept drift, a sliding window approach [19] is used to update both $\mathcal{F}_t$ and $\mathcal{D}_t$ on the basis of new transactions. The classifier $\mathcal{F}_t$

is trained on the transactions (feedback) of the latest 14 days. The classifier $\mathcal{D}_t$ implements an updating strategy that keeps the BRT corresponding to a window of 13 days and discards the oldest ones.

Given an incoming transaction i at time t, coded by a feature vector x_i, the classifiers $\mathcal{F}_t$ and $\mathcal{D}_t$ produce respectively the posterior probabilities $\mathcal{P}_{\mathcal{F}_t}(+|x_i)$ and $\mathcal{P}_{\mathcal{D}_t}(+|x_i)$, where $+$ denotes a fraud and $-$ a genuine transaction. The aggregated posterior probability $\mathcal{P}_{\mathcal{A}_t}(+|x_i)$ is obtained by a weighted average of posterior probabilities from the individual classifiers:

$$\mathcal{P}_{\mathcal{A}_t}(+|x_i) = w^A \mathcal{P}_{\mathcal{F}_t}(+|x_i) + (1 - w^A)\mathcal{P}_{\mathcal{D}_t}(+|x_i) \quad (1)$$

where $w^A \in [0, 1]$ and $\mathcal{A}_t$ is the overall model which wraps $\mathcal{F}_t$ and $\mathcal{D}_t$. On the basis of the analysis conducted in [19] we set $w^A = 0.5$.

The imbalanced nature of the classification problem led us to implement our own scalable version of a Balanced Random Forest (BRF). For this purpose we integrated Scala code with Weka [48], a well established open source tool for machine learning in Java. The result is a scalable BRF where every tree is trained on a subsample of the majority class (genuine cases) and the entire minority class (fraudulent cases). The pseudo-code of the scalable learner is detailed in **Algorithm 1**.

Algorithm 1 Distributed implementation of a Balanced Random Forest

```
1: nTrees ← number of trees per partition
2: frauds ← array of frauds
3: Broadcast frauds
4: genuine ← RDD of genuine
5: for any partition (genuine) do
6:     treeArray ← initialize an array
7:     for i ← 0, nTrees − 1 do
8:         subsetGenuine ← random subsample of the partition
9:         balanced.set ← Union(frauds, subsetGenuine)
10:        balanced.tree ← build a classifier using balanced.set
11:        treeArray ← append balanced.tree to treeArray
12:    end for
13: end for any partition
14: treeArrayGlobal ← collect all treeArray from partitions
```

Note that the genuine transactions are stored in the RDD *genuine* while

all the fraudulent transactions (array $frauds$) are broadcast[6] to every executor. For any partition of the RDD *genuine* we build $nTrees$ BRTs and we collect them in *treeArray*. Finally we store all the models created in the partitions in the *treeArrayGlobal* object. *Delayed* classification task is then performed by aggregating the outcome of all the BRFs. A typical way to perform aggregation relies on weighting

$$\mathcal{P}_{\mathcal{D}_t}(+|x_i) = \sum_{n=1}^{k} w_n^D \mathcal{P}_{BRF_{t-d-n}}(+|x_i) \tag{2}$$

where w^D is a vector of k weights which sums to 1 and d is the delay (number of days) for the reception of the labels. Different strategies can be used to set the weights, e.g. proportionally to the size of the tree training set or to the number of incorrect decisions (Dynamic Weighted Majority). A better ensemble learning strategy may be used to optimize the detection task. To further investigate ensemble strategies for streaming classification, we suggest the read of these surveys [49, 50, 51].

4.2. The streaming analytics engine

This engine implements the following functionalities:

- data throughput;
- data preprocessing;
- features engineering;
- online classification;
- data storage.

4.2.1. Data throughput

Data throughput in Spark from Kafka produces a DStream object (Discretized Stream) [52], the basic abstraction provided by Spark Streaming to represent a continuous stream of data. A DStream object is a continuous

[6]In Spark, the broadcast mechanism allows to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner.

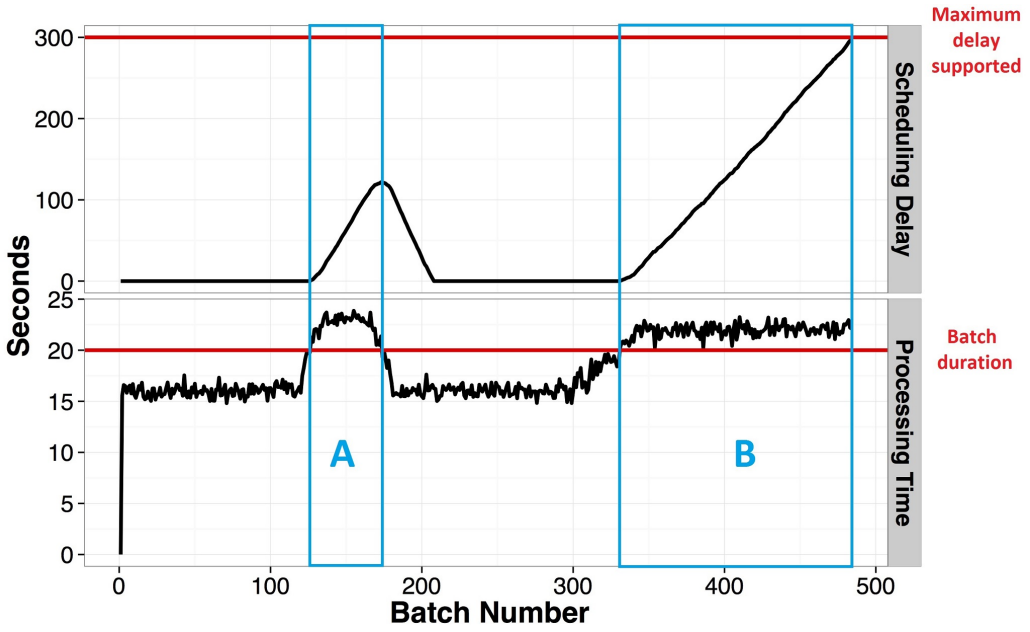

Figure 4: Behavior of the scheduling delay vs. batch duration in a synthetic example where the available resources can afford a maximum delay of 300 seconds. Configuration A refers to a situation where, only for a limited period, processing time is longer than the batch duration. Configuration B corresponds to a longer violation period of maximal batch duration. This event leads to an increase of delay up to a limit imposed by the available resources.

series of RDDs, obtained by periodically generating and appending RDDs. The frequency at which streaming data are partitioned into batches (a.k.a. batch duration) is an important parameter of a DStream object. In fact, the processing of a new batch starts as soon as a new RDD is generated and the processing of the previous batch has been completed. This entails that the processing time of an RDD should be smaller than the batch duration. If this is not the case, the batch is stacked in a queue and the execution postponed. Such a delay is sustainable for a limited period of time only (see configuration A in Fig. 4). If incoming data flow at a too high rate for a long period the application fails as soon as all the storage resources are exhausted (see configuration B in Fig. 4).

4.2.2. Data Preprocessing

This step deals with the treatment of missing values (replaced by median values) and with the coding of the categorical features (e.g. product class or merchant business type) characterized by a large number of values. The coding step consists in replacing each categorical value by a numeric value representing the a priori probability of the category to be associated to a fraudulent transaction, as presented in [53] (subsection 9.2.4). The probability is estimated from historical data and stored in a *dictionary* [54].

The effectiveness of such preprocessing is confirmed by previous research as well as by our industrial partner experience. From a more theoretical perspective it can be seen as an instance of *cascade generalization* [55] where preliminary naive classifiers are used as inputs to a more powerful classifiers. Potential risks of concept drift in this procedure could be addressed by updating the dictionary every time a new batch of labels is received. A detailed survey on data preprocessing for data stream mining can be found in [56].

4.2.3. Feature engineering

This step consists in the retrieval of historical data stored in the Cassandra database and the computation of aggregated statistics. Commonly used statistics are the maximum, minimum, count and average of relevant numerical variables (e.g. transaction amount), which derive from recent transactions of the concerned cardholder. In this step, crucial parameters are the size of the historical time window (e.g. week or month) and the number of recent transactions taken into consideration. Given the streaming nature of the problem, the modification of these parameters can noticeably impact the required resources and the affordable rate of data throughput. Alternative feature engineering techniques are discussed in literature [57, 58].

4.2.4. Online classification

This step consists first in classifying any incoming transaction by using the most recent classification model returned by the procedure described in subsection 4.1. Once the classification is performed, the system updates a dashboard containing a priority list of transactions (alerts) sorted by estimated risk. In our prototype this dashboard is simply a database table. Obviously a more user-friendly interface should be considered in a production environment.

4.2.5. Data storage

The final step consists in storing transactions and their aggregated information in a Cassandra table by means of a Spark Cassandra Connector, an open source library developed by Datastax [7]. Transaction and aggregated features are periodically retrieved to build the training set of the machine learning engine.

4.2.6. Pseudo-code

The entire streaming procedure can then be summarized by the pseudo-code in **Algorithm 2**. Given a $DStream$, for any of its RDD components (denoted $inTrx$) we perform a series of tasks:

- if the day is over [Row:12], we retrain the models $\mathcal{F}$ and $\mathcal{D}$ (section 4.1), we save on Cassandra DB the $topN$ alerts from $AlertTable$, we reset $AlertTable$ and we discard the unneeded transactions;
- for any given time interval in the array $window$, we retrieve information about previous transactions from $tableTrx$ and for any cardholder ($idTrx$) [Row:21];
- once the feature vector $featHist$ is built, the transaction may be classified according to the up-to-date classifiers $\mathcal{F}$ and $\mathcal{D}$ [Rows:27-30] and the riskiest $topN$ alerts stored in the alert table.

5. Experiments

This section assesses the proposed scalable architecture according to different criteria:

- Scalability;
- Impact of internal parametrization on computational performance;
- Classification precision.

Experiments were carried out on a cluster of ten machines, each with 24 cores and 80GB of RAM. Spark was run on top of the cluster resource manager Yarn [59]. For all experiments, each executor was allocated 1GB of RAM,

[7] `https://github.com/datastax/spark-cassandra-connector`

Algorithm 2 Streaming procedure

$DStream \leftarrow$ RDD collection
$tableTrx \leftarrow$ empty cassandra table
$tableRank \leftarrow$ empty cassandra table
$AlertTable \leftarrow$ empty array
$topN \leftarrow$ number of alerts to retain
$window \leftarrow$ array of window intervals for
features aggregation
$modelDate \leftarrow$ day of the last model update
for any RDD ($DStream$) **do**
    $inTrx \leftarrow$ current RDD
    $currentDate \leftarrow$ date of $inTrx$
    if $currentDate \neq modelDate$ **then**
        $trainFeedback \leftarrow$ train a new $\mathcal{F}$
        $trainDelayed \leftarrow$ train a new $\mathcal{D}$
        $modelDate \leftarrow currentDate$
        $tableRank \leftarrow AlertTable$
        $AlertTable \leftarrow$ empty array
    end if
    $idTrx \leftarrow getUniqueIds(inTrx)$
    $featHist \leftarrow$ empty array
    for $i \leftarrow 0$, $size(window) - 1$ **do**
        $h \leftarrow retrieveHist(idTrx, tableTrx, window(i))$
        $featHist \leftarrow$ append h to $featHist$
    end for
    $augTrx \leftarrow$ merge $inTrx$ and $featHist$
    $tableTrx \leftarrow$ insert $augTrx$
    $feedProb \leftarrow$ classify $augTrx$ using $trainFeedback$ and get the probability for class fraud
    $delProb \leftarrow$ classify $augTrx$ using $trainDelayed$ and get the probability for class fraud
    $totalProb \leftarrow$ ensemble $delProb$ and $feedProb$
    $AlertTable \leftarrow$ append $totalProb$ to $AlertTable$ and keep the $topN$ alerts with the highest risk
end for any RDD

Data subset				
Subset name	# trx	% of fraud. trx	# cards	% of fraud. cards
DS	8,356,811	0.4	1,921,457	0.2

Table 1: Dataset used for experiments

and the driver was allocated 10GB of RAM. Further discussion over memory usage will be presented in this section.

The dataset DS used for experiments is a selection of 40 consecutive days of transactions recorded from 2014, October, 18 to November, 26. This dataset includes more than 8 millions of e-commerce transactions from almost 2 millions cardholders, 18 descriptive features and the label (genuine or fraudulent). Table 1 reports the presence of frauds in terms of fraudulent transactions and fraudulent cards. Note that the feature engineering step is performed on a one week time window leading to the creation of 17 additional features.

The *Feedback* classifier $\mathcal{F}_t$ is trained over all the transactions from the 100 cards alerted per day and for a period of 14 days. Given that there are on average four transactions per card per day, $\mathcal{F}_t$ is trained with about 5,600 transactions.

The *Delayed* classifier $\mathcal{D}_t$ is trained on the set of transactions (about 2.7 million) occurring during 13 days (from day $t-8$ to $t-20$). The total number of transactions can be roughly estimated as follows

$$2.4\frac{trx}{sec} \times 86,400\frac{sec}{day} \times 13days = 2,695,680trx$$

Note however that the effective size of the final training set is smaller and dictated by the undersampling step which returns a more balanced dataset.

Note also that in the experiments we use a weighting strategy for aggregation (Equation 2) where weights are proportional to the number of training samples per tree.

5.1. Scalability

This section aims to assess the scalability of our pipeline by running three times the entire detection procedure on the dataset DS with an increasing number of Spark executors (25, 35 and 45). We set the batch duration [8] to

[8] This is an attribute of the object `StreamingContext` in Spark

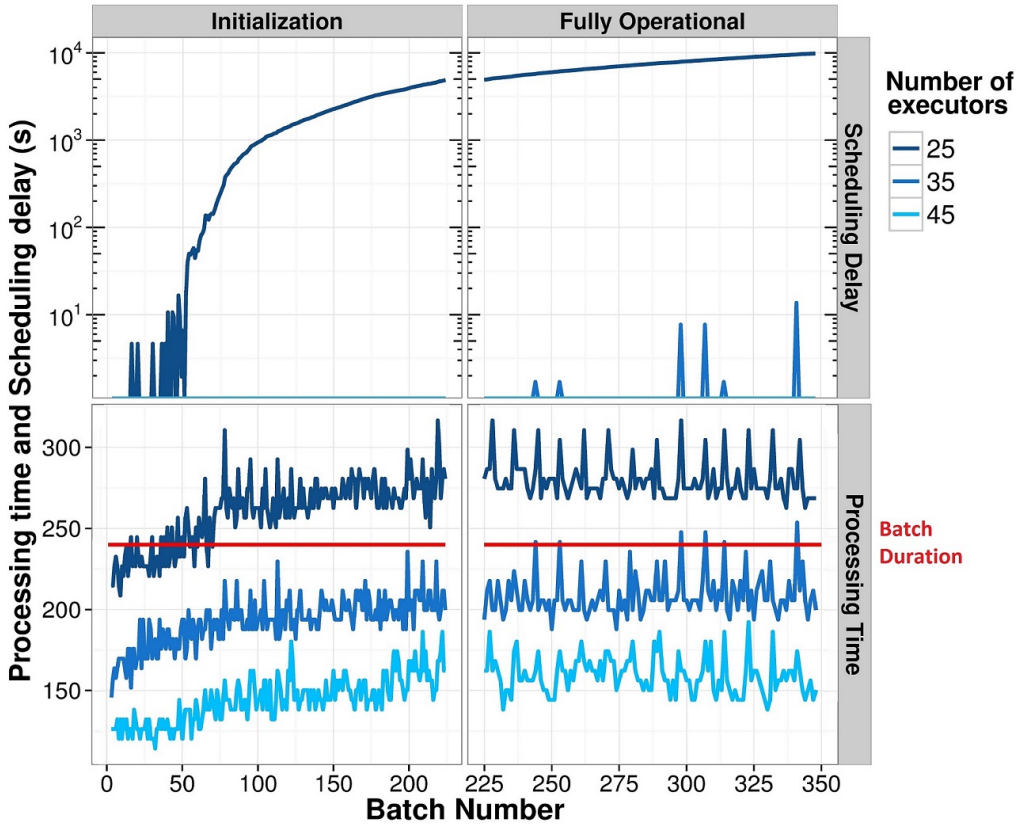

Figure 5: Processing time (lower part) and scheduling delay (upper part) behavior as a function of the number of executors. The left facet focuses on the *Initialization* phase, while the right facet reports the behaviour during the *Fully Operational* phase. On the x-axes we have an in time ordered sequence of batches analysed by the application, while on the y-axis we have the number of seconds needed to process a given batch.

240 seconds (section 4.2.1) and the data incoming rate at 100 transactions per second (trx/sec). Note that the real throughput rate associated to *DS* is 2.4 trx/sec.

Fig. 5 is divided in two vertical facets, displaying the execution behavior during an *Initialization* phase and a *Fully Operational* phase, respectively. During the *Initialization* phase, the system is in a bootstrap state, the Cassandra database is not completely filled and classification is not yet started. By *Fully Operational* phase we mean that all the functionalities (preprocess-

ing, feature engineering and classification) are fully working. Fig. 5 shows that during the *Initialization* phase, the processing time increases because of (i) the growing number of stored transactions and (ii) the increasing classification time due to the growing complexity of the random forest used for classification.

The *Fully Operational* phase begins as soon as the number of days set for features engineering is elapsed and we reach the desired number of models in the ensemble. From this moment on, the learning system starts discarding the oldest transactions and the oldest models from the ensemble, thus keeping constant the memory occupation.

Considering the *Fully Operational* phase, a first observation is that the processing time for the *25 executors* run is longer than the limit set by the batch duration. In this case a delay will be accumulating with a potential risk of application failure (see also Fig. 4).

This is not the case for 35 and 45 executors, respectively, since the processing time is typically shorter than the batch duration. Nevertheless in the *35 executors* case, we still observe some peaks passing over the batch duration threshold. Those peaks refer to batches where the retraining of the model takes place in addition to the feature engineering and the online classification operations. The fact that some distributions pass over the 240 seconds threshold has not necessarily a negative impact on the resulting performances. This is due to the fact that in real production setting, these situations occur only once a day with minor impact on final schedules.

For this reason we have rearranged the *Fully Operational* data in Fig. 6 in order to make explicit the processing time due to the streaming and to the learning step, respectively. It appears again evident that the configuration with 10, 20 or 30 executors is not sufficient to absorb a streaming rate of 100 trx/sec.

Let us remark also that in Fig. 6 the average processing time is decreasing with the number of executors suggesting that the application is scalable. However the decreasing rate of improvement suggests that the improvement could be negligible from a certain number of executors on. This is typically due to the fact that the map-reduce process may become too expensive for a large number of executors since the benefit of dividing the computation among executors is counterbalanced by the cost of shuffling too many data among those executors.

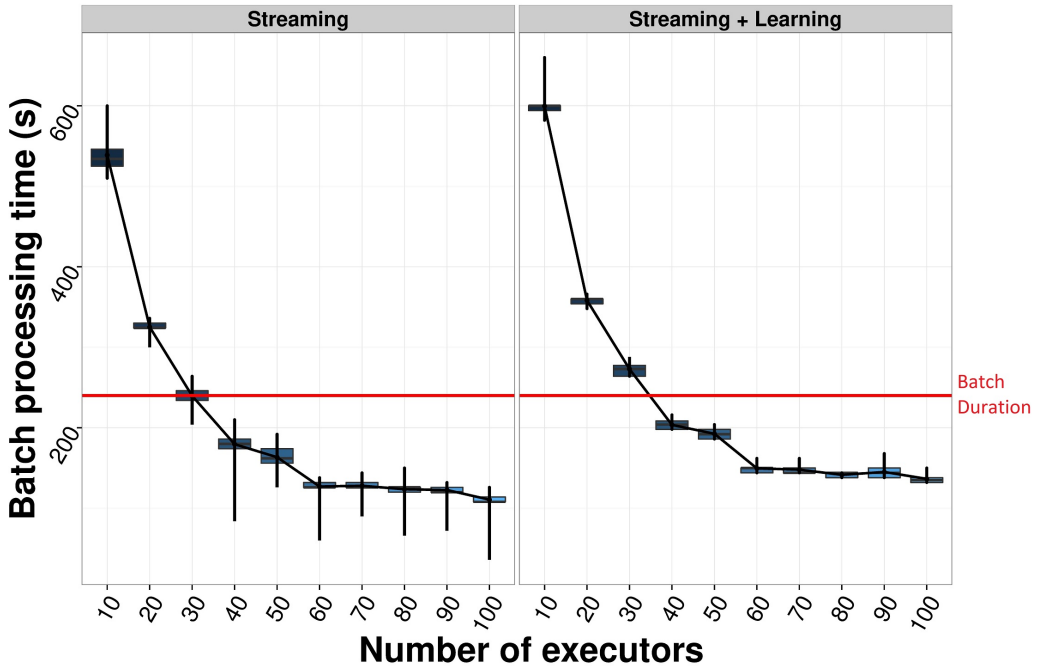

Figure 6: Processing time distribution in the *Fully Operational* phase, for different number of executors. Left: execution time due to *Streaming* only. Right: execution time due to *Streaming* and *Learning*.

5.2. Impact of internal parametrization on computational performance

In the previous section, we remarked that it is possible to define the minimal number of executors able to manage a given incoming rate (e.g. 100 trx/sec). However, given the potential saturation of a distributed approach, it is interesting to study how to deal with high incoming rates without necessarily increase the size of the cluster. A possible solution comes from an appropriate tuning of internal parameters like the batch duration time. However an increase of the batch duration time implies two drawbacks:

- a deterioration of the precision of the features engineering step;
- a delay in raising alerts.

The precision of features engineering is reduced since during its calculation, only the transactions stored in advance may be used; therefore if we have two

transactions from the same card in a given batch, information about the first transaction will not be included to engineer features related to the second one.

The second drawback is a minor one since, as discussed previously, in a fraud detection scenario where human investigators have to contact clients to obtain their feedback, a delay of few minutes makes little difference.

In Fig. 7 we report the results obtained with the dataset DS, 100 executors and by raising the incoming rate to 240 trx/sec. The aim is to test the robustness of the infrastructure for long periods at high throughput rates.

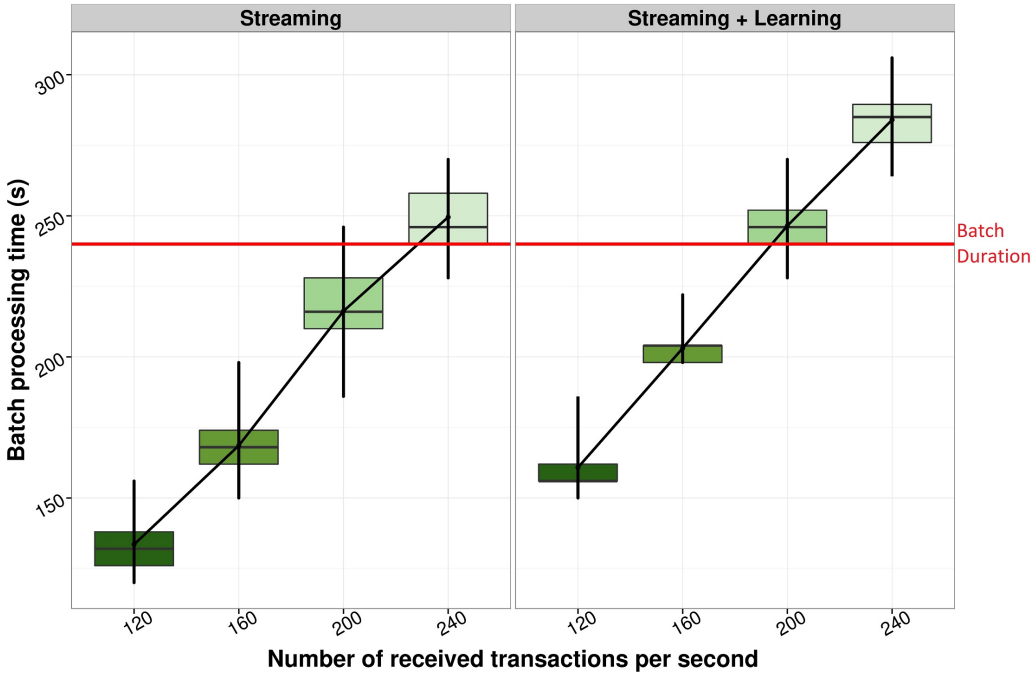

Figure 7: Processing time distribution in the *Fully Operational* phase for different throughput rates, fixed batch duration of 240 seconds and using 100 executors. The left side shows the distribution of *Streaming* times while the *Streaming + Learning* times are shown on the right side.

It is worth to notice that 200 trx/sec is not necessarily an upper limit and that still higher rates could be obtained either by increasing the batch duration or the number of executors.

In terms of RAM-Hours (the average RAM usage in gigabyte per hour required by a given process [60]), our solution exhibits two distinct behaviours. In the *Initialization* phase, the memory use grows linearly in time until the *Fully Operational* phase. In the *Fully Operational* phase, memory use is constant (0.05 RAM-Hour in case of a 200 trx/sec stream and 100 executors).

Another interesting information concerns how the processing time is distributed among the different tasks of the Streaming functionality. From Fig. 8, it appears that the heaviest task is the *Feature Engineering* (which includes the aggregation described in 4.2), followed by the reading time from Cassandra.

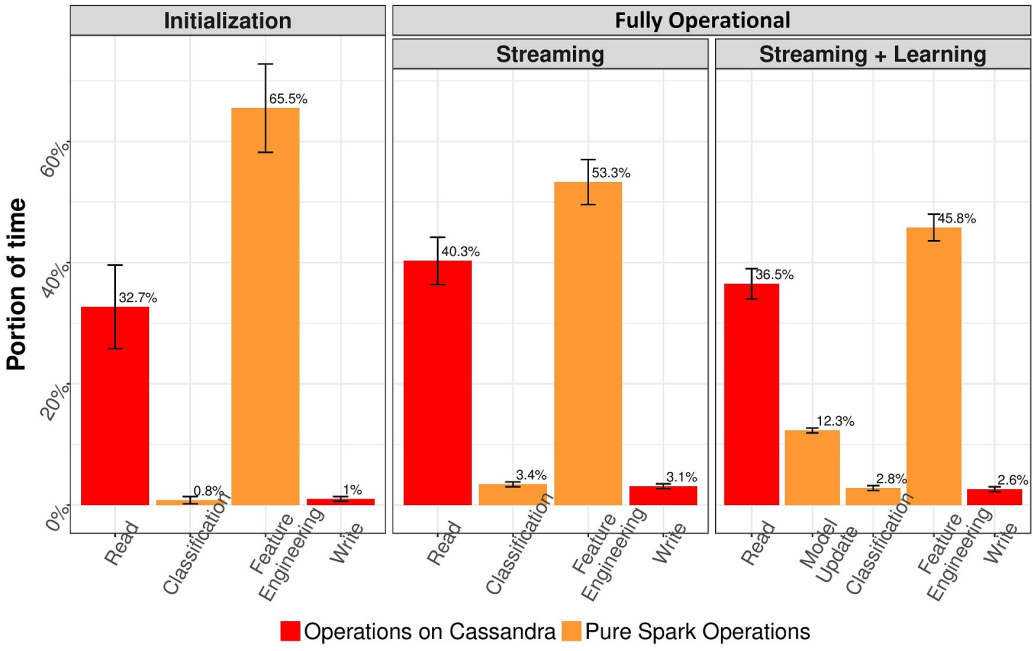

Figure 8: Distribution of processing time among *Reads* and *Writes* on Cassandra, *Feature engineering*, *Model Update* and *Classification* tasks.

The distribution of *Read* operations and *Write* operations is strongly skewed in our plot. That is happening because we are writing few lines and reading many lines. Indeed for the aggregation purposes, we need to retrieve old transactions information and this concerns far more lines than the one pushed in Cassandra. As expected, the *Read* tasks as well the *Classification* tasks, consume more resources in the *Fully Operational* phase than in the

Initialization one. Note that the time for *Feature Engineering* is similar during the two phases in absolute values: the decrease visible in the figure is only in percentage terms (*Read* and *Write* times increase). The *Fully Operational* phase includes two sub-phases (*Streaming* and *Streaming + Learning*). An additional component, the *Model Update*, characterizes the latter sub-phase and impact for the 12.3% of the total processing time.

5.3. Classification Precision

Fraud Detection Systems are designed to have accurate detection performance. A good measure for the precision, proposed in [20] and previously used in rare item detection [61], is the Card Precision (CP), which is the proportion of detected fraudulent cards among the alerted ones.

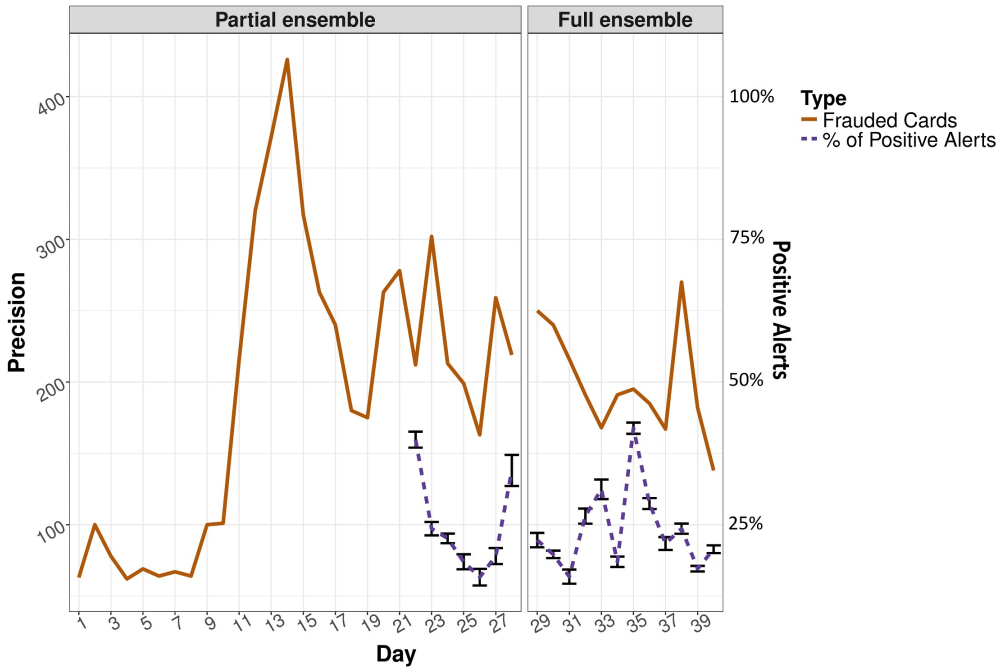

Figure 9: The line chart shows the distribution of frauded cards (solid/yellow line, left y-axis) and the percentage of detected frauds in the examined period (dashed/violet line, right y-axis). The confidence intervals were obtained by repeating the experiment ten times and changing different random seeds.

It is therefore important to assess the CP returned by a scalable implementation of the fraud detection procedure. The precision obtained in our

experiments with the dataset DS is essentially in line with the results published in [20]. The minor discordance is due to the fact that we are using only a subset of the features used in [20].

Overall we obtained an average precision $CP_k = 0.24$ where k is set to 100 since this is the average number of cards that can be daily checked by the investigators working for our industrial partner. This means that on average 24 alerts out of 100 are correct.

Fig. 9 reports the total number of fraudulent cards (solid/yellow line) and the percentage of detected fraudulent cards (dashed/violet line) for the analysed period. As expected, we see an improvement of the precision when we move from the *Partial ensemble* to the *Full ensemble* phase, essentially due to the growing size of the classifier ensemble and the improvement of the Feedback model (initially trained on inaccurate alerts). Fig. 10 and Table 2 illustrate well the effectiveness of the averaging strategy: most of the time and on average the single classifiers perform worse than the ensemble. Those results have been obtained over ten runs of SCARFF, streaming the same time series, but changing the randomization seed. We have also computed two paired t-tests between the CP_k obtained using the Ensemble Classifier and those obtained by its two components. The CP_k of the Ensemble Classifier results to be statistically bigger than the Delayed and Feedback Classifier (p-values smaller than 10e-3).

Fully Operational Stage	
Classifier	CP_k
Delayed Classifier	16.1%
Feedback Classifier	22.4%
Ensemble Classifier	24.1%

Table 2: Precision CP_k for multiple classifiers during the fully operational stage.

Since the CP_k assessment refers to $k = 100$ alerts only, we also report in Fig. 11 the Area Under the receiver operating characteristic Curve (AUC),which is a more general measure of accuracy used in fraud detection scenarios [17, 46, 62]. The two lines represent respectively the AUC considering all the transactions and the most likely fraudulent transactions for each credit card.

Finally, a last measure of accuracy derived from the experiment relates to the capacity of our implemented model to detect a fraudulent card before what actually happened in the recorded dataset. It happens in fact that some

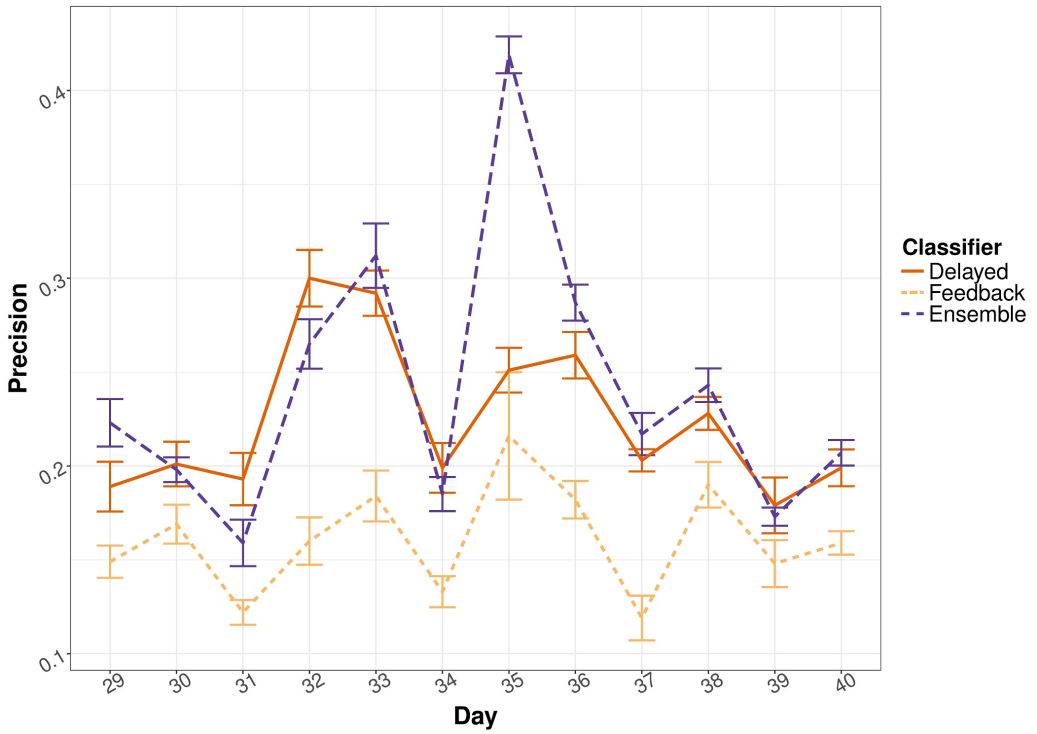

Figure 10: Percentage of detected frauds from the Delayed model, the Feedback model and the Ensemble of the two. Note that the aggregate model precision (dashed/violet line) is usually higher. The confidence intervals were obtained by repeating the experiment ten times and changing different random seeds.

fraudulent cards were stopped only after several fraudulent transactions. On the basis of our simulation, it appears that the implemented classifier is able to detect earlier fraudulent cards 3.6% of the time.

6. Conclusions and future work

The paper presented SCARFF, an original scalable platform to automatically detect frauds in a near real-time horizon. The most original contribution of this framework is the design and the implementation of an open source big data solution for real-world Fraud Detection and its test on a massive real-world data set. We wish to emphasize that the workflow proposed in our article, while not disclosing the data, has been made fully open source

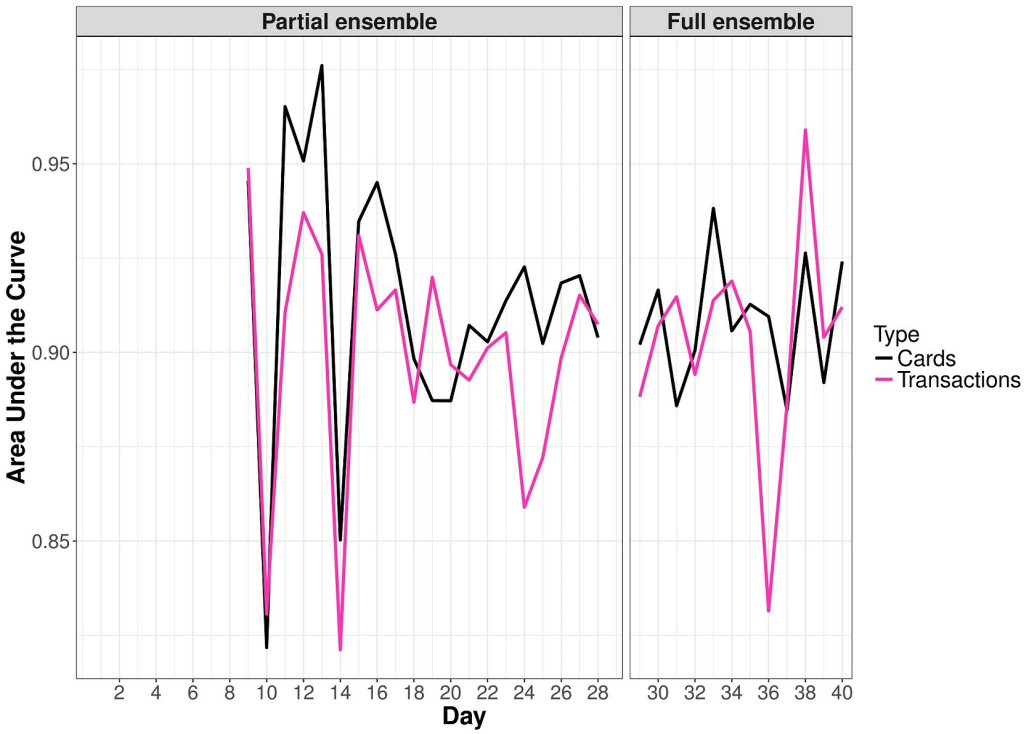

Figure 11: AUC of fraud detection at the level of card (darker color) and transaction (lighter color).

and reproducible by means of a **Docker**[9] container and an artificial dataset. To the best of our knowledge this is the most complete and detailed open source and big data solution for credit card fraud detection in the literature.

In terms of software development the paper shows that Kafka, Spark and Cassandra, may provide easy scalability and fault tolerance, to receive, aggregate and classify transactions at high rate. In the experimental session we have extensively tested the system in terms of scalability, sensitivity to parameters and classification accuracy.

We have shown that the system behaves robustly up to an incoming rate of 200 transactions per second, which is a remarkable result once compared with the 2.4 transactions per seconds rate currently managed by our industrial

[9] `https://hub.docker.com/r/fabriziocarcillo/scarff/`

partner. Moreover, a rate of 200 transactions per seconds should not be considered as a hard upper limit since an appropriate setting of the number of executors and the batch duration could allow still higher rates.

In terms of precision we have confirmed previous results obtained in a conventional architecture with data resident in main memory.

Nevertheless, as a concluding remark, it is important to add some words of caution about the maturity of big data solutions for large scale deployments. Big data solutions are supported by a growing open-source community which leads to a very fast evolution and, at the same time, to a high rate of new releases. If on the one hand this ensures rapid debugging, on the other hand it may induce instability in the existing running solutions. The problem is evident when one is trying to combine several functionalities of different tools in the same platform. For instance we encountered several problems in querying a Cassandra table from Spark: we had a very hard time in doing ad hoc queries to Cassandra which had as consequence that we often decided to get the whole table from Cassandra and filter it on Spark (a suboptimal solution).

Overall, we consider that the most important message is that the adoption of a big data solution introduces a number of parameters having an impact on the resulting computational and classification performances. In order to obtain an efficient solution to a specific detection problem, several trade-offs have to be made explicit and managed both at the software and hardware levels. In our experience the most important trade-off concerned the number of transactions processed per second, the complexity of the feature engineering step, the batch duration and the number of available executors.

Future work will focus on porting the existing solution to the industrial partner, testing the efficiency in a Cloud environment, assessing the robustness to the adoption of alternative service providers (e.g. other databases than Cassandra) and generalizing the framework to other streaming settings (e.g. analytics of multivariate sensor streams). From a more theoretical point of view, we would like to investigate innovative approaches especially in the area of semi-supervised and active learning.

Acknowledgement

The authors FC, YLB and GB acknowledge the funding of the Brufence project (Scalable machine learning for automating defense system) supported

by INNOVIRIS (Brussels Institute for the encouragement of scientific research and innovation). ADP acknowledges the funding of the Doctiris (Adaptive real-time machine learning for credit card fraud detection) project supported by INNOVIRIS (Brussels Institute for the encouragement of scientific research and innovation).

References

[1] S. Ghosh, D. L. Reilly, Credit card fraud detection with a neural-network, in: Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences., Vol. 3, IEEE, 1994, pp. 621–630.

[2] D. Sánchez, M. Vila, L. Cerda, J. Serrano, Association rules applied to credit card fraud detection, Expert Systems with Applications 36 (2) (2009) 3630–3640.

[3] Y. Sahin, S. Bulkan, E. Duman, A cost-sensitive decision tree approach for fraud detection, Expert Systems with Applications 40 (15) (2013) 5916–5923.

[4] B. Krawczyk, M. Woźniak, Incremental weighted one-class classifier for mining stationary data streams, Journal of Computational Science 9 (2015) 19–25.

[5] B. Krawczyk, M. Woźniak, One-class classifiers with incremental learning and forgetting for data streams with concept drift, Soft Computing 19 (12) (2015) 3387–3400.

[6] B. Krawczyk, Learning from imbalanced data: open challenges and future directions, Progress in Artificial Intelligence 5 (4) (2016) 221–232.

[7] A. Dal Pozzolo, O. Caelen, G. Bontempi, When is undersampling effective in unbalanced classification tasks?, in: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, 2015, pp. 200–215.

[8] A. Dal Pozzolo, O. Caelen, R. A. Johnson, G. Bontempi, Calibrating probability with undersampling for unbalanced classification, in: Symposium Series on Computational Intelligence, IEEE, 2015, pp. 159–166.

[9] J. a. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, A. Bouchachia, A survey on concept drift adaptation, ACM Comput. Surv. 46 (4) (2014) 44:1–44:37.

[10] C. Alippi, G. Boracchi, M. Roveri, Just-in-time classifiers for recurrent concepts, IEEE Transactions on Neural Networks and Learning Systems 24 (4) (2013) 620–634.

[11] E. R. Faria, I. J. Gonçalves, A. C. de Carvalho, J. Gama, Novelty detection in data streams, Artificial Intelligence Review 45 (2) (2016) 235–269.

[12] Z. S. Abdallah, M. M. Gaber, B. Srinivasan, S. Krishnaswamy, Anynovel: detection of novel concepts in evolving data streams, Evolving Systems 7 (2) (2016) 73–93.

[13] H. Yang, S. Fong, Countering the concept-drift problem in big data using iovfdt, in: 2013 IEEE International Congress on Big Data, IEEE, 2013, pp. 126–132.

[14] H. Yang, S. Fong, Countering the concept-drift problems in big data by an incrementally optimized stream mining model, Journal of Systems and Software 102 (2015) 158–166.

[15] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, S. Whittle, Millwheel: Fault-tolerant stream processing at internet scale (2013) 734–746.

[16] Q. Lin, B. C. Ooi, Z. Wang, C. Yu, Scalable distributed stream join processing, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ACM, 2015, pp. 811–825.

[17] A. Dal Pozzolo, O. Caelen, Y.-A. Le Borgne, S. Waterschoot, G. Bontempi, Learned lessons in credit card fraud detection from a practitioner perspective, Expert Systems with Applications 41 (10) (2014) 4915–4928.

[18] A. Dal Pozzolo, R. A. Johnson, O. Caelen, S. Waterschoot, N. V. Chawla, G. Bontempi, Using hddt to avoid instances propagation in unbalanced and evolving data streams, in: 2014 International Joint Conference on Neural Networks (IJCNN), IEEE, 2014, pp. 588–594.

[19] A. Dal Pozzolo, G. Boracchi, O. Caelen, C. Alippi, G. Bontempi, Credit card fraud detection and concept-drift adaptation with delayed supervised information, in: International Joint Conference on Neural Networks (IJCNN), IEEE, 2015, pp. 1–8.

[20] A. Dal Pozzolo, G. Boracchi, O. Caelen, C. Alippi, G. Bontempi, Credit card fraud detection: a realistic modeling and a novel learning strategy, IEEE Transactions on Neural Networks and Learning Systems (Accepted) (2017).

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

[22] I. Triguero, S. del Río, V. López, J. Bacardit, J. M. Benítez, F. Herrera, Rosefw-rf: the winner algorithm for the ecbdl14 big data competition: an extremely imbalanced big data bioinformatics problem, Knowledge-Based Systems 87 (2015) 69–79.

[23] H. Hormozi, M. K. Akbari, E. Hormozi, M. S. Javan, Credit cards fraud detection by negative selection algorithm on hadoop (to reduce the training time), in: Information and Knowledge Technology (IKT), 2013, pp. 40–43.

[24] E. Hormozi, M. K. Akbari, H. Hormozi, M. S. Javan, Accuracy evaluation of a credit card fraud detection system on hadoop mapreduce, in: Information and Knowledge Technology (IKT), 2013, pp. 35–39.

[25] A. Tselykh, D. Petukhov, Web service for detecting credit card fraud in near real-time, in: Proceedings of the 8th International Conference on Security of Information and Networks, ACM, 2015, pp. 114–117.

[26] S. Phulari, U. Shantling Lamture, S. Vilas Madage, K. Tirupati Bhandari, Pattern analysis and fraud detection using hadoop framework, International Journal of Engineering Science and Innovative Technology (IJESIT) Volume 5, Issue 1 (2016) 92–100.

[27] J. Chen, Y. Tao, H. Wang, T. Chen, Big data based fraud risk management at alibaba, The Journal of Finance and Data Science 1 (1) (2015) 1–10.

[28] https://github.com/mapr-demos/frdo, [Online; accessed 7-July-2016].

[29] https://github.com/vakshorton/CreditCardTransactionMonitor, [Online; accessed 7-July-2016].

[30] https://github.com/bhomass/marseille, [Online; accessed 7-July-2016].

[31] https://github.com/Azure/azure-content/blob/master/articles/stream-analytics/stream-analytics-real-time-fraud-detection.md, [Online; accessed 7-July-2016].

[32] https://github.com/hadooparchitecturebook/fraud-detection-tutorial, [Online; accessed 7-July-2016].

[33] https://github.com/pranab/beymani, [Online; accessed 7-July-2016].

[34] https://github.com/namebrandon/Sparkov, [Online; accessed 7-July-2016].

[35] K. Veeramachaneni, I. Arnaldo, V. Korrapati, C. Bassias, K. Li, Ai2: Training a big data machine to defend, in: IEEE 2nd International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS), IEEE, 2016, pp. 49–54.

[36] J. Kreps, N. Narkhede, J. Rao, Kafka: A distributed messaging system for log processing, in: Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece, 2011, p. 1.

[37] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12, 2012, pp. 2–2.

[38] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: Cluster computing with working sets, in: Proceedings of the 2Nd

USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, 2010, pp. 10–10.

[39] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al., Mllib: Machine learning in apache spark, arXiv preprint arXiv:1505.06807.

[40] M. Odersky, al., An overview of the scala programming language, Tech. Rep. IC/2004/64, EPFL Lausanne, Switzerland (2004).

[41] A. Lakshman, P. Malik, Cassandra: a decentralized structured storage system, ACM SIGOPS Operating Systems Review 44 (2) (2010) 35–40.

[42] L. Breiman, Random forests, Machine learning 45 (1) (2001) 5–32.

[43] L. Rokach, Decision forest: Twenty years of research, Information Fusion 27 (2016) 111–125.

[44] S. Bhattacharyya, S. Jha, K. Tharakunnel, J. C. Westland, Data mining for credit card fraud: A comparative study, Decision Support Systems 50 (3) (2011) 602–613.

[45] A. C. Bahnsen, D. Aouada, B. Ottersten, Example-dependent cost-sensitive decision trees, Expert Systems with Applications 42 (19) (2015) 6609–6619.

[46] V. Van Vlasselaer, C. Bravo, O. Caelen, T. Eliassi-Rad, L. Akoglu, M. Snoeck, B. Baesens, Apate: A novel approach for automated credit card transaction fraud detection using network-based extensions, Decision Support Systems 75 (2015) 38–48.

[47] C. Chen, A. Liaw, L. Breiman, Using random forest to learn imbalanced data, University of California, Berkeley 110.

[48] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I. H. Witten, The weka data mining software: an update, ACM SIGKDD Explorations Newsletter 11 (1) (2009) 10–18.

[49] M. Woźniak, M. Graña, E. Corchado, A survey of multiple classifier systems as hybrid systems, Information Fusion 16 (2014) 3–17.

[50] B. Krawczyk, L. L. Minku, J. Gama, J. Stefanowski, M. Woźniak, Ensemble learning for data stream analysis: a survey, Information Fusion 37 (2017) 132–156.

[51] H. M. Gomes, J. P. Barddal, F. Enembreck, A. Bifet, A survey on ensemble learning for data stream classification, ACM Computing Surveys (CSUR) 50 (2) (2017) 23.

[52] M. Zaharia, T. Das, H. Li, S. Shenker, I. Stoica, Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters, in: Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing, HotCloud'12, 2012, pp. 10–10.

[53] J. Friedman, T. Hastie, R. Tibshirani, The elements of statistical learning, Vol. 1, Springer series in statistics Springer, Berlin, 2001.

[54] A. Dal Pozzolo, Adaptive Machine Learning for Credit Card Fraud Detection, Ph.D. thesis, Université libre de Bruxelles (2015).

[55] J. Gama, P. Brazdil, Cascade generalization, Machine Learning 41 (3) (2000) 315–343.

[56] S. Ramrez-Gallego, B. Krawczyk, S. Garca, M. Woniak, F. Herrera, A survey on data preprocessing for data stream mining, Neurocomputing 239 (C) (2017) 39–57.

[57] A. C. Bahnsen, D. Aouada, A. Stojanovic, B. Ottersten, Feature engineering strategies for credit card fraud detection, Expert Systems With Applications 51 (2016) 134–142.

[58] C. Whitrow, D. J. Hand, P. Juszczak, D. Weston, N. M. Adams, Transaction aggregation as a strategy for credit card fraud detection, Data Mining and Knowledge Discovery 18 (1) (2009) 30–55.

[59] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, E. Baldeschwieler, Apache hadoop yarn: Yet another resource negotiator, in: Proceedings of the 4th Annual Symposium on Cloud Computing, SOCC '13, 2013, pp. 5:1–5:16.

[60] A. Bifet, G. Holmes, B. Pfahringer, E. Frank, Fast perceptron decision tree learning from evolving data streams, Advances in knowledge discovery and data mining (2010) 299–310.

[61] G. Fan, M. Zhu, Detection of rare items with target, Statistics and Its Interface 4 (2011) 11–17.

[62] S. Viaene, R. A. Derrig, G. Dedene, A case study of applying boosting naive bayes to claim fraud diagnosis, IEEE Transactions on Knowledge and Data Engineering 16 (5) (2004) 612–620.