Adaptive Random Forest for Detecting Fraudulent Credit Card Transactions

Abstract - This is a project implementing a real-time fraud-detection system (FDS) for Credit Card Fraud Transactions using Adaptive Random Forest in Apache Spark platform. The increasing use of credit card in online transactions reflects the rise of a new, fast and easy way of interchange in modern world. Based on extensive data collected for online card payments in the United Kingdom, is expected that by 2026, the number of card payments per day will grow to 60 million.[1] This new form of transaction involves the risk of fraudulent attacks and fishing attempts, therefore provokes the urgent need of developing fast and online FDS. This project suggests a new system for detecting and monitoring online transactions using the latest Apache Spark processing engine, Structured Streaming, and implementing an Adaptive Ensemble Classification Method, Random Forest. Our goal is to design a learner, for extremely large datasets which do not fit in main memory, adapting to concept drift and evolving data. We show the effectiveness of our method on both synthetic and real world datasets and we manage to reach a 92% accuracy on average.

Keywords Random Forest · Concept drift · Apache Spark · Weighted Voting ·

1. Introduction

1.1 Machine Learning in the Big Data World

From early 2000, machine learning and statistics are becoming common practice for solving classification problems. Recently many statistical methods have been adapted to Big Data such that subsampling and divide and conquer approaches have become common practice.[2] Thus, the notion of mining a fixed-sized database is giving way to the notion of mining an open-ended data stream as it arrives.[3] This fact depicts the existing problem statement for large and massive data. The logical assumption with which we construct machine learning models nowadays, is that the data can be too large to fit to computer memory and therefore the state of our model relies on a fraction of data. Some applications considering unbound data, are synopses of data streams and effective prediction models, which are an open research area. For that reason, many big streaming platforms concentrated their work to develop real time machine learning processing tools. A robust tool is "MLib" from Apache Spark, which is a part of the Hadoop Ecosystem within the Big Data analytic framework, including a range of Machine Learning algorithms such Naïve Bayes, KNN, K-means and Random Forest Decision Trees to name but a few. Despite the widely acceptance of MLib libraries, there is need for more custom algorithms with specific features.

1.2 Random Forest

Random Forests (RF) were introduced by Breiman [4] in the need of an effective tool for prediction.[1] Random Forests consists of a collection of weak learners such that each one produces a predicted class. Firstly, RF is responsible to collect all the individual predictions and apply majority voting, and secondly produce the final prediction. One of the many worth mentioned points in his paper is the claim: "Injecting the right kind of randomness, makes RF more accurate".[2] This randomness is introduced by two different forms. The first one lies around the idea of drawing random samples with replacement from the training set (bagging) and the second one lies around the idea of randomly selecting a subset of features for each decision tree at the process of node splitting. Breiman, also, addresses three very important key variables: *overfitting*, *variance* and *bias*. In Section 2 of his paper proves that due to the Strong Law of Large Numbers

[1] Breiman's first approach simulates a prediction model in a non-streaming setting.
[2] A step forward of this idea is the implementation of Extremely randomized trees [5]

RF always converge so that overfitting is not a problem. He shows, in the experiment section, that adaptive bagging and bootstrap algorithms reduce bias as well variance. A notion of reduced bias can be explained by the fact that RF relies on the power of the "crowd"-learners. Finally, by applying bootstrap sampling, we end up that approximately 64% of data points are used by each weak learner and the other 36% is the out of the bootstrap sample or out-of-bag (OOB). As we will discuss in Section 4, in streaming context, bagging is performed by an alternative manner achieving equal results to Breiman's bagging and OOB can be reduced significantly. In 2001, a state-of-the-art algorithm of Hoeffding Tree was introduced by Domingos and Hulter for mining high-speed data streams [3] responding to challenges set by the production rate of data and the aforementioned incapability of computer resources to contain massive data.

1.3 Concept Drift

Our project was inspired by an updated version of [3] for time-changing data streams.[6] It is also inspired by the paper «Adaptive random forests (ARF) for evolving data stream classification» of Heitor Gomes.[7] The last two papers use the basic Hoeffding concept for training the classification model but differ on the way they adapt to concept drift. Heitor Gomes claims that they avoid bounding ARF to a specific drift detecting algorithm to facilitate future adaptations and they propose a new abstract warning and drift concept for triggering the concept-drift algorithm of theirs training model. Both papers, so do we, avoid deleting already trained trees once a drift is detected but create background trees ready to replace the existing ones.

This paper is organized as follows. After this introduction, we briefly state some minimum information for the Apache Spark Streaming Platform and the implemented dataset as well as the preprocessing carried-out of data fed to the Streaming Job, Section 2. In Section 3, we present the Project Pipeline starting from some necessary theoretical insights setting some basic concepts of understanding.

2. Methods – Materials – Formulation

2.1 Apache Spark Streaming

Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine.[8] In short, Structured Streaming provides fast, scalable, fault-tolerant, end-to-end exactly-once stream processing without the user having to reason about streaming. Apache Spark, promises that the queries are processed using *micro-batching* with latency from 100 milliseconds to 1 millisecond (when tuning some parameters). Essentially, Structured Streaming is a new, higher-level API for streaming structured data. Structured Streaming unlike Apache Streaming uses DataFrames which are, in practice, a Dataset of Row objects. The key point that differs Structured Streaming from Apache Streaming is that the case of Structured Streaming, Spark knows at compile time what type of information it will receive, and has not have to wait until run time to actual figure out the structure of data.

2.2 Data Set

The material of this project comes from the Kaggle [9], an online community of data scientists that allows users to find and publish data sets. As we have mentioned before credit card fraud detection is a motivation subject for machine learning and data streams.[10][11] The dataset, [12] consists of 284.807 online transactions where only 0.172% of all transactions is classified as fraud (positive class) which means that dataset is highly unbalanced.

In the Section 4, we will present some ways for solving such a problem by introducing oversampling (by replicating the minority class).

The data contains only numerical input variables which are the result of a PCA transformation. The data consists from 28 features V1, V2, … V28 and three extra columns: 'Time', 'Amount', 'Class'. Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature 'Amount' is the transaction Amount, this feature can be used for example-dependent cost-sensitive learning. Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.

2.3 Data Preprocessing – Data Source

A complete FDS must consist of three phases, the training, testing and predicting phase. A small simplification, that we have to make, is that we consider immediate classified tuples in our model and not delayed tuples (samples with delayed labels). The reason for that is because often the tuples used for the training of the model belong to some past records where they have already been distinguished by the investigating team of each bank and there is one to one relation between features and the known class. This assumption helps us to build a precise model from past records, able to predict incoming-present records. Furthermore, a worth mentioning point is that the dataset, and the credit card fraud detection in particular, is the reason for this project and not the purpose. We do not want to create a model defiantly for the needs of solving this particular problem but we want to build a model which can be applied to a variety of such similar problems. So, some small assumptions are acceptable up to the point that we do not cut corners from building a general model. Similar behavior to training samples, have the testing samples. Testing, in general, serves the purpose of monitoring the system. Often, these testing samples come from a tiny fraction of daily transactions amount and they might have the same number of samples from the two classes. The fact that we know that true label of the tuple and we compare it with the predicted label from our model, helps us to have a better understanding about the performance of our model. In Section 5, we present all the metrics we used in order to monitor our model. Last but not least, we have the predicting samples, where we don’t know the true label but based on our model, we predict its label. In this case, we can have a real time answer whether or not this sample is fraudulent or not. Each one of the aforementioned phases is produced by a separate source, therefore we have three sources. This concept will be presented in detail in Section 4.1.

In order to distinguish from which source each tuple is coming from, we have added an extra column for identification. As mentioned before, the original tuple contains the values of each attribute and the corresponding class, (in case of training and testing) where as in case of predicting, each tuple contains only the set of values. So, an example of an instance looks like that: {v1, v2, v3, …, c, id, incnr}, where c= Class, id is either -5 in case of testing or 5 in case of training and incnr is the incremental number of the tuples since the first produced tuple. In case of a predicting tuple id is equal to -10.

3. Machine Learning Model - Hoeffding tree

3.1 Hoeffding Tree Concept

Domingos and Hulten propose a new system for infinite set of data without the need to store any of the incoming instances. Their classification problem is described as follows: N examples in the form of *(x, y)*, where *x* is the set of numeric attributes and *y* is the discrete class, are fed to a model. The goal is, from those examples, to produce a model *y=f(x)* that will predict the correct *y* of new incoming examples *x* with high accuracy. In our approach each Hoeffding Tree is a binary tree which consists of a set of nodes where each node split the data into two subsets. The reason for that is thoroughly explained in Section 3.5. There are two types of nodes, the intermediate nodes and the leaves. The first ones, are guiding each incoming tuple to the next level and the leaves are predicting the class based on the most dominant class within the node. Each node, despite what type it is, has a set of parameters that are necessary for constructing a tree. Those parameters are the followings:

- The splitting attribute and the splitting value. Based on these two values each incoming tuple is guided to the next level, during the traverse from the root to the leaves.
- The label counts, therefore the label of the node.
- The number of tuples it has seen, as well as the maximum number of tuples it has to see in order to start the splitting process. In Section 3.3, there are mentioned all the criteria for node splitting.
- Information Gain which corresponds to the best attributes' performance for its best value.
- A set of attributes, that are randomly selected.
- Its child nodes, left and right node and its parent node.

3.1.1 Building a Hoeffding Tree

Algorithm: Create Hoeffding Tree

```
Input:  Max:                is the number of how many features we have to select from
        m_features:         is the number of the size of the random subset of Max
        max_examples_seen:  is the number of examples between checks for split
        delta:              one minus the desired probability of choosing the correct feature at any
                            given node
        tie_threshold:      is the number between splitting values of selected feature for split
Output: root of Hoeffding Tree
1. For each attribute on m_features do:
2.     Create a HashMap for samples
3.     Create a HashMap for label counts and Initialize them with zero
4. instances_seen ← 0
5. correctly classified ← 0
6. weight ←1
7. InitializeRoot (m_features, max_examples_seen, delta, tie_threshold)
8. Reservoir Sampling (m_features, Max)
```

<u>Pseudo Code Analysis</u>

Line 2: We know a priori the number of attributes that we have, so we create as many lists as the number of attributes.

Line 3: We know that we have a binary problem (0: Non Fraud, 1: Fraud) so we need only 2 <0, ?> and <1, ?> in order to keep track the label counts.
Line 6: At first all Hoeffding Trees have the same weight.
Line 7: *InitializeRoot* is a function for initialize variables such as: information gain, root label, splitting attribute and value, left and right child and parent node and sets m_features, max_examples_seen, delta, tie_threshold to the user's input.
Line 8: In order to select m random features from total features we use Reservoir Sampling.

3.1.2 Train a Hoeffding Tree

Their solution describes a model which has to process a fraction of incoming tuples in order to find the best attribute at an internal node of the tree. In order to know the sufficient number of tuples with which we can conclude to the best splitting attribute, we need to set the statistical Hoeffding Bound. Hoeffding Bound guarantees that given n observations of a random variable r whose range is R (in our case log2) the computed mean $\bar{r}$ of this variable differs at most by ε with probability *1-delta* from the true mean *r*.

The ε is defined as such:

$$\varepsilon = \sqrt{\frac{R^2 \ln\left(\frac{1}{delta}\right)}{2n}}$$

Algorithm: Train Hoeffding Tree

```
Input:   node:          is the root of the Hoeffding Tree
         input_sample:  is an array of values of the corresponding attribute
Output: root of Hoeffding Tree
1.  filtered_input ←Filter the input_sample respectively to m_features.
2.  TargetNode ← Traverse (node, filtered_input)
3.  If NeedForSplit (TargetNode) then
4.      AttemptSplit (TargetNode);
5.      InsertNewSample (TargetNode, filtered_input);
6.  else
7.      InsertNewSample (TargetNode, filtered_input);
8.  end
```

<u>Pseudo Code Analysis</u>

Line 1: We have to only select the features that the HT was built based on the Algorithm of Creating HT.
Line 2: We traverse the input sample through the HT and we find the node where it should be added.
Line 3: *NeedForSplit* is a function that checks where or not we should split the current node. The criteria are two:
– If the number of examples seen is greater or equal to the maximum number of examples it should see for considering a split
– If the node is homogeneous which means if the node is pure and all samples are of the same class.
Line 4: For a given node, it attempts to split the node. Firstly, finds the best attributes to split the node and secondly, finds if the best attributes satisfy the condition (based on epsilon and tie_threshold). If these two conditions are satisfied it performs the splitting procedure.
Line 5: Add the input sample to the target node. This process includes:
– Adding each value of each attribute to the correspond HashMap.

– Add the label of the input sample to the correspond HashMap.
– Update the node's label counter.
– Update the counter of samples seen.

Line7: If there is no need for split, we just add the input sample to the target node.

3.1.3 Test and Predict using Hoeffding Tree

A Hoeffding Tree has to keep track with its progress and have to broadcast its information to the State. In order to do that it needs to contain the number of correctly classified tuples, the weight and the number of seen instances. For defining at each Hoeffding Tree (h) the weight (w_h) we have to take the ratio between the correctly classified tuples (c_h) and the total tuples seen (n_h), $w_h = \frac{c_h}{n_h}, where\ c_h < n_h$

Algorithm: Test and Predict Hoeffding Tree

```
Input:   node:              is the root of the Hoeffding Tree
         input_sample:      is an array of values of the corresponding attribute
         purpose_id:        is the id for identifying the different tuples
Output:  predicted_value:   is the prediction from the Hoeffding Tree
1.  filtered_input ←Filter the input attributes respectively to m_features
2.  If    purpose_id is a testing OR prediction sample then
3.            predicted_value ← TestHT (node, filtered_input)
4.  else if purpose_id is a training sample then
5.            increase by one the instances seen counter
6.            predicted_value ← TestHT (node, filtered_input)
7.            If predicted_value is equal to the true label of the training sample then
8.                   increase by one the correctly classified counter
9.            end
10.           UpdateWeight (correctly_classified, instances_seen)
11. end
12. return predicted_value
```

<u>Pseudo Code Analysis</u>

purpose_id: As we have discussed previously, we need to use an id variable for distinction between predicted, testing and training tuples.
– purpose_id= -5 correspond to testing examples
– purpose_id = -10 correspond to predicted examples
– purpose_id= 5 correspond to training examples

Line 3: *TestHT* is a function which traverses through the HT and returns the label of the target node.
Line 10: *UpdateWeight* follows the concept we described previously.

3.2 Random Forest Concept

Let us assume that given is a set of training data $\boldsymbol{X_t} = \{(\boldsymbol{x_m}, \boldsymbol{y_m}), \boldsymbol{m} = \boldsymbol{1}, \ldots \boldsymbol{M}\}$ where $\boldsymbol{x_m}$ is an input observation and $\boldsymbol{y_m}$ is a predictor output. A weak learner can be created using the training set $\boldsymbol{X_t}$. A weak learner is a predictor $f(\boldsymbol{x}, X_t)$ having a low bias and a high variance [13]. By randomly sampling from the set $\boldsymbol{X_t}$, a collection of weak learners $f(\boldsymbol{x}, X_t, \boldsymbol{\theta_k})$ can be created, with $f(\boldsymbol{x}, X_t, \boldsymbol{\theta_k})$ being the *kth* weak learner and $\boldsymbol{\theta_k}$ is the random vectors electing data points for the *kth* weak learner.

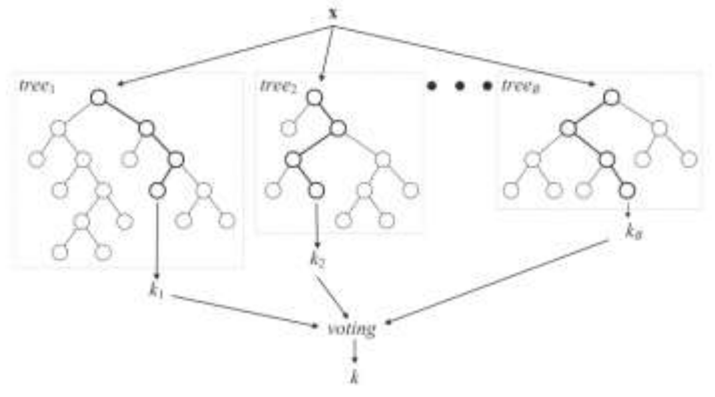


Fig. 1. A general architecture of a random forest.

3.1.3 Randomly Selection of k-features

Ensemble methods work best when the predictors are as independent from one another as possible. One way to get diverse classifiers is to train them using very different algorithms. This increases the chance that they will make very different types of errors, improving the ensemble's accuracy. Second way to get diverse classifiers is to use the same training algorithm for every predictor and train them on different random subsets of the training set ,thus we managed this using leveraging sampling which explained below. Another way to get in even more predictor diversity is to use randomly selection a subset of the input features. The benefits of using randomly selection a subset of the input features is: (1) useful when dealing with high-dimensional inputs, (2) introduces extra randomness when growing trees so the Random forest results in greater tree diversity, which trades a higher bias for a lower variance, generally yielding an overall better model, (3) reduce the computational cost of finding the best-split feature at each node on every tree of Random Forest.

3.1.2 Leveraging Sampling

In data stream learning it is infeasible to perform multiple passes over input data as you cannot keep the entire stream. Thus, an adaptation of Random Forest to streaming data depends on an appropriate online bootstrap aggregating process. To explain our adaptation to address this requirement we need to discuss how bagging works in non-streaming, and how it is simulated in a streaming setting. In non-streaming bagging [13], each of the n base models is trained in a bootstrap sample of size Z created by drawing random samples with replacement from the training set. Each bootstrapped sample contains an original training instance K times, where P(K = k) follows a binomial distribution. For large values of Z this binomial distribution adheres to a Poisson ($\lambda = 1$) distribution. Based on that, authors in Oza (2005) [14] proposed the online bagging algorithm, which approximates the original random sampling with replacement by weighting instances[3] according to a Poisson($\lambda = 1$) distribution. We use Poisson ($\lambda = 6$), as in leveraging bagging (Bifet et al. 2010) [15], instead of Poisson ($\lambda = 1$). This "leverages" resampling, and has the practical effect of increasing the probability of assigning higher weights to instances while training the base models, thus we managed to increase the diversity of the weights and modify the input space of the classifiers inside the Random Forest. However, the optimal value of λ may be different for each dataset.

3.1.4 Out-Of-Bag Error

With bagging, some instances may be sampled several times for any given predictor, while others may not be sampled at all. When used a Poisson($\lambda = 1$) distribution, this means that only about 63% of the training instances are sampled on average for each predictor. The remaining 37% of the training instances that are not sampled are called out-of-bag (oob) instances. Note that they are not the same 37% for all predictors. Thus, we decided to use leveraging sampling succeeding to reduce the training instances that are not sampled and to increase the diversity of the weights to instances as explained before.

3.3 Splitting Evaluation Function - Information Gain

The Split Evaluation Function used is Entropy and Information Gain, since the success criterion of a split is decreasing the impurity for a set of training examples that arrive to a certain leaf. In the case of entropy, we care about choosing the attribute that minimizes its value whereas for Information Gain the best attribute is the one with the higher value. Entropy is used to calculate the homogeneity of a node. If the node is entirely homogenous the entropy would be 0. If the sample is equal to 1 that means that there are equal parts of tuples inside the node. Furthermore, Information Gain can be defined as the information acquired about a particular feature when we observe the rest features without it. Below, we provide the definition for entropy and Information Gain.

Entropy H for a given leaf l with variable $C = \{c_1, c_2\}$ where both elements correspond to the two classes of our binary problem. $H(C)$ has to be equal or greater than 0. ($H(C) > 0$) and describes the probability over the two discrete distributions. $p = \{P(C = c_1), P(C = c_2)\}$.

$$H(l) = -\sum_{i=1}^{|p|} p_i \log_2 p_i = -p_1 \log_2 p_1 - p_2 \log_2 p_2$$

On the other hand, we provide the definition of Information gain at a leaf s for a certain attribute X_α.

$$G(l, X_a) = H(l) \ - \sum_{i=1}^{|p|} \frac{|l_v|}{|l|} H(l_v)$$

[3] In this context, weighting an instance with a value w for a given base model is analogous to training the base model w times with that instance

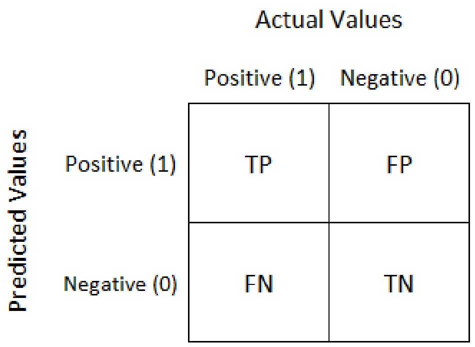
3.4 Continuous Features

Unlike the Domingos paper,[3] we deal with continuous attributes without knowing the arithmetic boundaries and distribution of each feature. This fact indicates that we only have the following case: $A < \theta$ & $A > \theta$, where A is the value of a given feature and the θ the existing splitting point. That results that the constructed tree has transformed from a tree with multiple children to a binary tree. Finding the optimal splitting point for a given feature is performed by firstly finding the quartiles of the given feature's value space and then testing the three returned values as for which one maximizes the Information Gain.

3. 5. Classification Performance Metrics

3.5.1 Confusion Matrix Voting

Several indices were employed to monitor our classification method. Apart from the accuracy score, which is calculated as the ratio of correctly classified samples by the total number of samples, showing the overall accuracy of the method, other metrics of classification performance were also used for the evaluation of the algorithm. In order to calculate these metrics, the number of True Positive (TP), False Positive (FP), False Negative (FN) and True Negative (TN) samples were computed.



Predicted Values	Actual Values: Positive (1)	Actual Values: Negative (0)
Positive (1)	TP	FP
Negative (0)	FN	TN

– *Recall* (or *Sensitivity*-true positive rate) of a tuple is its ability to determine the fraud cases correctly. This is also obvious through its calculation: $TPR = \frac{TP}{TP+FN}$

– Also, the misclassification of a transaction as non-fraud is quite serious and is desirable to be as low as possible. This is estimated through the false negative rate or 1 – TPR.

– *Specificity* is the True Negative rate, $TNR = \frac{TN}{TN+FP}$ and is the ability of a test to determine the non-fraud cases correctly.

–The exact opposite of Specificity, misclassifying a non-fraud transaction as fraud is not as severe, and it is measured by the inverse of *Specificity* (1-TNR).

–*Precision* (positive predictive value) is calculated as: $PPV = \frac{TP}{TP+FP}$. Precision can be interpreted as the ability of the classifier not to label as positive a sample that is negative.

– *F1 score* is defined as the combination of Sensitivity and Precision as it is calculated using these two metrics: $F1 = \frac{PPVTPR}{PPV+TPR} = \frac{2TP}{2TP+FP+FN}$.

4. Project Pipelining

4.1.1 General Architecture

In this section we will walk you through in the typical lifespan of a training, testing and prediction tuple which is fed to our FDS. Starting from the training phase of our model.

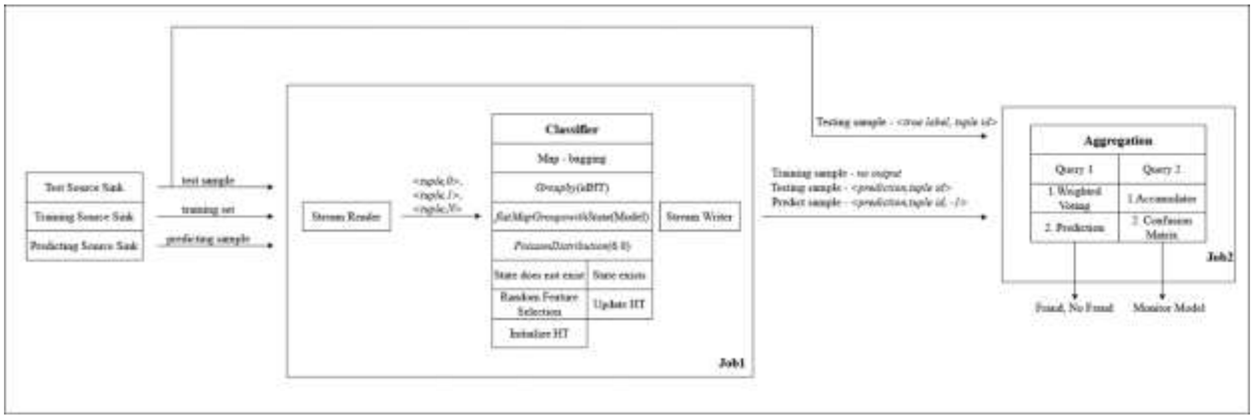


Figure1. Project architecture

4.1.2 Classifier – Random Forest

The role of Job1 is to processing the incoming stream and maintaining the Random Forest.

Algorithm: Job1- Classifier

```
Input:    rawData:           is the tuples from source
Output:   predicted_tuple:   is the prediction from the Random Forest
1.  structuredData ←FlatMap (rawData)
2.  results ← structuredData.groupByKey(structuredData.idHT). flatMapGroupsWithSate [HT, Outstate]
3.          if state does exist then
4.                  Get from state the Hoeffding Tree
5.                  for each tuple in the iterator do
6.                          if keyTuple belongs to testing or prediction then
7.                                  Do Test and Predict Hoeffding Tree Algorithm
8.                          end
9.          else if state does not exist then
10.         end
```

<u>Pseudo Code Analysis</u>

Line 1: The FlatMap function splits the rawData tuple into the data, purposeId and keyTuple part. The data part is the part that starts from the first column until the next-to-last column. The purposeId is the next-to-last column and the keyTuple is the last column. So, StructuredData follows the structure of the InputData class.

– case class InputData (data: String, purposeId: Int, keyTuple: Int, idHT: Int)

The reason we use the *FlatMap* function instead of the *Map* function because we take advantage of the property of *FlatMap* which is that it can return more than one tuple given one input tuple. Our purpose, besides structuring the rawData, is to populate each tuple according to the number of Hoeffding Trees. (the reason for that presented in Section 3.1.2). So, the structuredData is a number of copies of a structured version of rawData.

Line 2: In this line there are several maters to discuss; firstly, in order to build a Random Forest, we have to populate a number of Hoeffding Trees. This is made possible by grouping our data based on the Hoeffding Tree id. Apache Spark Structured Streaming guarantees that if a pleiad of samples is grouped by a column, each group is processed individually by a distributed manner from the processing engine. Secondly, in order to have a machine learning model, like ours, we have to save the current state, that's why we use *flatMapGroupsWithState*. In fact, as far as we know, this transformation is the only way to perform such a functionality. The state consists of the Hoeffding model and the outstate. The structure of outstate is the following:

– case class OutputState (listKeyTuple: List [Int], listRes: List [Int], listLabel: List [Int], listOfPurposeId: List [Int], weightTree:Double, idHT: Int)

4.1.3 Aggregation - Weighted Voting

The role of Job2 is to calculate the confusion matrix, the performance metrics of Random Forest and finally to aggregate the results-votes of trees so that calculates the label of predicting tuples. Below follows the section of Algorithm of Job2 and the Pseudo Code Analysis.

Algorithm: Job2

```
Input:  rawData:          is the tuples from Job1
Output: predicted_tuple:  is the prediction from the Random Forest
1. structuredData ←FlatMap (rawData)
2. groupedResult ← structuredData.groupBy(keytuple, purposeId). WeightedVotingAggregation
3. predictedTuples ← groupedResult.Filter (prediction_tuples)
4. testingTuples ← groupedResult.Filter (testing_tuples). ConfusionMetrixAccumulator
```

<u>Pseudo Code Analysis</u>

Input: The rawData, which are the incoming tuples from Job1, follow the same structure with the output structure of Job1.

Line 1: The Flatmap function splits the rawData tuple into the class Result given that we are working on structured stream. Below follows the structure of the Result case class.

– case class Result (keyTuple: Int, res: Int, label: Int, purposeId: Int, idT: Int, weightTree: Double)

Line 2: In this line, is become the groupby operation base on keyTuple,purposeId . The purpose of groupby operation is twofold. Firstly, for each predicting tuple aggregates the votes from multiple instances of it, which corresponding to the vote of each tree of Random Forest and using the weight of each tree calculates

the weighted sum of predicting tuple, so in the end calculates the label. Secondly, for testing tuples just keep the testing instances so that can then be calculated the confusion matrix for Random Forest.
Line 3: In this line, the groupby results are filtered so that only the predicted tuples are selected and emitted the label of each predicted tuple.
Line 4: In this line, the groupby results are filtered so that only the testing tuples are selected and then are calculated the performance metrics of Random Forest. The calculation is implemented using the abstract class of Accumulator.

5. Experiments -Results

In order to evaluate the methods described herein, we have to conduct an extensive testing. Firstly, our goal is to prove that the machine learning model is able to classify a given record to the respective class and secondly, to prove that our system is scalable and distributed and can run in a cluster environment. Therefore, our testing is split based on the testing datasets, to the first phase where we test the classification ability of our system and to the second phase where we examine the performance of our system according to distribution and scalability. The need of this split in the experimental process is due to the fact that there are no public datasets bigger than 150.000 records where data are not unbalanced. As we mentioned in Section 2.3 *(Data Preprocessing – Data Source)* we used a widely known dataset for Credit Card Fraud detection which is a suitable dataset for testing with the drawback of non-uniformity between classes. We also used GMSC[4] dataset, which is used for determining whether or not a loan should be granted, containing 150.000 borrowers with 10 attributed each. Both datasets do not represent the volume we needed in order to scientifically contend that our system is running with "Big Data". On the other hand, there are smaller datasets which are result of under sampling of the major class and they are more reliable for testing the classification ability of our system. The dataset used for this purpose is called Swiss banknote counterfeit detection[5] and the goal is to identify genuine and counterfeit banknotes, even if half of the data is counterfeit. Contains 200 entries with 100 in each class and 6 attributes.

5.1 First Phase – Swiss banknote counterfeit detection

The metrics presented below are results of testing after the following hyper parameter tuning (In section 3.1.1 (Building a Hoeffding Tree) the variables are extensively explained):

variable	number of Trees	m_features	Max	max_examples_seen	delta	tie_threshold
number	100	6	6	5	10^{-7}	5%

Results:

Metric	Accuracy	Sensitivity	Specificity	Precision	F1_score
Result	77%	87%	64%	72%	79%

[4] Give Me Some Credit dataset https://www.kaggle.com/c/GiveMeSomeCredit
[5] Swiss banknote counterfeit detection https://www.kaggle.com/chrizzles/swiss-banknote-conterfeit-detection

5.2 Second Phase – GMSC

The metrics presented below are results of testing after the following hyper parameter tuning (In section 3.1.1 (Building a Hoeffding Tree) the variables are extensively explained):

variable	number of Trees	m_features	Max	max_examples_seen	delta	tie_threshold
number	100	10	10	1000	10^{-7}	5%

Results:

Metric	Accuracy	Sensitivity	Specificity	Precision	F1_score
Result	90%	49%	92%	65%	59%

5.3 Second Phase – Credit Card Fraud Detection

The metrics presented below are results of testing after the following hyper parameter tuning (In section 3.1.1 (Building a Hoeffding Tree) the variables are extensively explained):

variable	number of Trees	m_features	Max	max_examples_seen	delta	tie_threshold
number	100	28	28	1000	10^{-7}	5%

Results:

Metric	Accuracy	Sensitivity	Specificity	Precision	F1_score
Result	93%	51%	91%	58%	54%

6. Conclusion

The presented methodology demonstrates an approach of classifying data streams using Random Forest in a distributed manner. Our system uses the state-of-the-art processing engine of Apache Spark and establishes itself among similar systems. Future research will continue to improve the overall classification accuracy and speed of our system.

References:

[1] Published by James Cherowbrier: https://www.statista.com/statistics/719708/card-payments-per-day-forecast-united-kingdom/

[2] Robin Genuer, Jean-Michel Poggi, Christine Tuleau-Malot. Random Forests for Big Data: https://arxiv.org/pdf/1511.08327.pdf

[3] Pedro Domingos, Geoff Hulten: Mining high-speed data streams: https://dl.acm.org/doi/10.1145/347090.347107

[4] Leo Breiman: Random Forests: https://link.springer.com/article/10.1023/A:1010933404324

[5] Pierre Geurts, Damien Ernst & Louis Wehenkel: Extremely randomized trees: https://link.springer.com/article/10.1007/s10994-006-6226-1

[6] Geoff Hulten, Pedro Domingos, Laurie Spencer: Mining time-changing data streams: https://dl.acm.org/doi/10.1145/502512.502529

[7] Heitor M. Gomes, Albert Bifet, Jesse Read: Adaptive random forests for evolving data stream classification: https://link.springer.com/article/10.1007/s10994-017-5642-8

[8] Structured Streaming Programming Guide: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#overview

[9] Kaggle: https://en.wikipedia.org/wiki/Kaggle

[10] Amiripalli Shanmuk Srinivas: Application of Big Data Analytics and Pattern Recognition Aggregated With Random Forest for Detecting Fraudulent Credit Card Transactions (CCFD-BPRRF): https://www.researchgate.net/publication/332369280_Application_of_Big_Data_Analytics_and_Pattern_Recognition_Aggregated_With_Random_Forest_for_Detecting_Fraudulent_Credit_Card_Transactions_CCFD-BPRRF

[11] Andrea Dal Pozzolo, Giacomo Boracchi, Olivier Caelen: Credit card fraud detection and concept-drift adaptation with delayed supervised information https://ieeexplore.ieee.org/document/7280527

[12] Credit Card Fraud Detection, Anonymized credit card transactions labeled as fraudulent or genuine: https://www.kaggle.com/mlg-ulb/creditcardfraud

[13] Breiman, L. (1996). Bagging predictors. Machine Learning, 24(2), 123–140.

[14] Oza, N. C. (2005). Online bagging and boosting. IEEE International Conference on Systems, Man and Cybernetics, 3, 2340–2345

[15] Bifet, A., Holmes, G., & Pfahringer, B. (2010). Leveraging bagging for evolving data streams. In PKDD (pp. 135–150).