Performance Evaluation of a Cost-Sensitive Differential Evolution Classifier Using Spark - Imbalanced Binary Classification

Jamil Al-Sawwa and Simone A. Ludwig

Department of Computer Science, North Dakota State University, Fargo, ND, USA

Abstract

Nowadays, the amount of data that has been collected or generated in many sectors has been growing exponentially because of the rapid development of technologies such as the Internet of Things (IoT). Additionally, the nature of this data is imbalanced. The need for extracting valuable information for decision support from this data poses a challenge to the scientific community to find a solution to cope with large imbalanced data. In previous work, our cost-sensitive differential evolution classification algorithm showed efficient performance for handling highly imbalanced data sets. However, our algorithm shows inefficient performance when applied to big data sets, thus lacking to scale with data size increases. In this paper, we design and implement a parallel version of our cost-sensitive differential evolution classifier using the Apache Spark framework (SCDE). The aim is to handle large and binary imbalanced data. The main idea of the algorithm is to find the optimal centroid for each target label using differential evolution by minimizing the total misclassification cost and then assign unlabeled data points to the closest centroid. Our experiments include a real data set that is based on intrusion detection in order to evaluate our algorithm's scalability and performance. The experimental results show that SCDE efficiently handles imbalanced binary data and scales very well with data size increases. Moreover, the speedup and scaleup results that are obtained by SCDE are close to linear.

Keywords: Differential Evolution, Classification, Cost-Sensitive, Apache Spark, Imbalanced Data Set, Big Data Analytics, Intrusion Detection

1. Introduction

Data classification is a supervised learning task which aims to analyze historical data by discovering hidden relationships between features and class labels in order to classify future data. The classification task starts by splitting a data set into two data sets, training and testing data set. One machine learning (ML) algorithm is usually applied to the training data set to discover the hidden

relationships in order to create a prediction model. After that, the prediction model is applied to instances of the testing data set to predict the outcomes [1].

In various real-life applications such as Biomedical, Intrusion Detection System, and Credit Card Fraud, the data is typically imbalanced where the number of instances that belong to one class label (minority class) is significantly lower compared to another class label (majority class). Furthermore, in some circumstances the misclassification cost of the minority class is much larger than the misclassification cost of another class [2]. For example, in an intrusion detection system, undetected attacks are much more serious and costly than detecting normal behavior as an attack. During the last decades, the researchers proposed various approaches to cope with imbalanced data which can be mainly classified into two groups [3],[4]:

- Data-level approach: In this approach, the data firstly goes through preprocessing to balance the distribution of class labels using sampling methods such as Over-Sampling and Under-Sampling before applying an ML algorithm.

- Algorithm-level approach: The tradition ML algorithm is modified to tackle imbalanced data.

Evolutionary Algorithms (EAs) are stochastic optimization algorithms that imitate the biological evolution theory. EAs are based on the population which consists of individuals that represent a candidate solution. The idea of EAs is to find the best solution by applying the principle of "survival of the fittest" to generate a new population. During the last decades, researchers proposed various algorithms based on this theory such as Differential Evolution (DE). DE is one of the most popular EAs, which was proposed by Storn and Price in 1997 [5]. DE is mainly inspired by the Genetic Algorithm (GA) with the difference of generating new offspring. DE showed itself as a simple and powerful algorithm for solving real-world optimization problems in continuous spaces [6]. Furthermore, several variants of the original DE algorithm have been proposed to improve the performance of DE [7],[8].

With the rapid development of technology, the collected or generated data has been growing exponentially. Furthermore, the characteristics of this data; variety, velocity, and complexity, are the significant challenges that researchers face. Recently, many big data frameworks have been proposed to handle big data and its characteristics such as Apache Spark. Apache Spark is an in-memory computing big data framework that runs on a cluster of nodes, which was initiated and implemented by a research team from the University of California-Berkeley in 2009 [9]. Apache Spark addresses the major drawback of the Hadoop Map-Reduce framework [10] which is the overhead due to disk I/O operations that are needed to write an intermediate result in a shared file system during running a Map and Reduce job by distributing the data across a cluster of nodes and keeping it in memory as long as necessary. Moreover, Apache Spark overcomes the shortcoming of Hadoop MapReduce for handling iterative and interactive jobs [11]. Furthermore, Apache Spark is more efficient for han-

dling the iterative and interactive job and runs 100 times faster than Hadoop Map-Reduce for various applications [11].

The Resilient Distributed Dataset (RDD) is an immutable collection of data sets (objects) distributed across a high-performance cluster of nodes in a fault-tolerant manner. RDD is created to load an external data set from a shared file system such as HDFS or to parallelize a group of objects in the driver program. The driver program is responsible for creating a directed acyclic graph (DAG) which is a logical execution plan for RDD. The Spark engine uses the DAG to recover any failure RDDs while running the application. Besides, the driver program negotiates with the cluster manager to allocate the resources and run the executors on the worker nodes. After the cluster manager registers the executors to the driver program, the driver program sends the tasks to the executors for parallel processing [11],[12],[13].

Apache Spark provides two types of operations that can be performed on the RDD, which are a transformation and an action. The transformation operation is used to create a new RDD from an existing RDD using pre-defined functions such as Map and MapToPair. On the contrary, the action operation is used to compute a result by running a computation job on the RDD such as reduce and count, and the result can be returned to the driver program or written on a shared file system. The transformation operations on the RDD are executed lazily, which means that the transformations on the RDD are only executed when a Spark application triggeres an action operation on the RDD. Moreover, Apache Spark supports two types of a shared variable instead of a shared global memory, which are Broadcast and Accumulator. The broadcast variable is a read-only variable that is created by the driver program and is distributed to the executors to be used during the running task. The Accumulator variable is a write-only variable, which aggregates values from the executors [11],[12],[13].

In [14], we proposed a cost-sensitive differential evolution classification algorithm to tackle imbalanced binary data sets. We introduced a new objective function by minimizing the misclassification cost instead of the misclassification error to address the inefficient performance of the current variants of the differential evolution classification algorithm [15, 16] when applied to imbalanced binary data sets. The proposed algorithm was tested using several cancer data sets (imbalanced binary data sets) and the experimental results showed that the proposed algorithm's performance outperformed the performance of the current variants of the differential evolution classification algorithm in terms of Area Under Curve (AUC) and G-mean. Moreover, the performance of our algorithm is competitive compared to the performance of five cost-sensitive classification algorithms concerning AUC and G-mean. However, all differential evolution classification algorithms shows inefficient performance when applied to big data sets and thus lacks to scale with data size increases. In this paper, we propose a parallel cost-sensitive differential evolution classification algorithm based on Apache Spark, which is further referred to as SCDE. The idea of SCDE is to find the optimal centroid of each class label in a training data set by minimizing the total misclassification cost. The aim of the algorithm is to handle binary imbalanced data sets and take advantage of Apache Spark to work on massive

data sets to achieve high performance and scalability. To the best of our knowledge, this is the first work that implements a nature-inspired based classification algorithm using Spark to handle large binary imbalanced data.

The remainder of this paper is organized as follows: Section 2 provides an overview of existing works in parallel data mining algorithms using big data frameworks. Section 3 illustrates our proposed approach - SCDE. Section 4 describes the data set and preprocessing. In Section 5, we present the scalability and performance analysis of SCDE. Finally, Section 6 presents our conclusions and future work.

2. Related Work

Recently, designing and implementing scalable solutions for traditional ML algorithms or data mining algorithms that are based on optimization methods have received recent attention from the scientific community. In particular, the inefficiency of these algorithms for handling and analyzing big data in order to extract valuable information is investigated. The researchers have proposed a scalable solution to accommodate the rapid growth of data and run these algorithms in a suitable time using a high-performance cluster of nodes [17]. In this section, we will start presenting the existing parallel solutions of these algorithms based on a big data framework. Additionally, the parallel implementation of the optimization methods are also described.

In [18], the authors proposed a parallel version of DE using the MapReduce (MR) paradigm in order to improve the running time of the optimization method when solving a large-scale problem. In this work, the fitness evaluation is carried out through the Map and Reduce phases. Nevertheless, the experimental results showed that the exhaustive I/O disk operations during the shuffling and sorting phases are reducing the parallelization performance. Two variant parallelization approaches of DE based on Apache Spark, a master-slave and an island-based, have been proposed in [19] to overcome the drawback of the previous work. The master-slave and island-based approaches were experimented on the AWS cloud using synthetic and real biology-inspired benchmarks. The results revealed that both approaches show good scalability with increasing numbers of nodes. Other works related to the parallel implementation of the optimization method can be found in [20],[21],[22],[23].

The authors in [24] proposed a parallel implementation of the DE clustering algorithm using the MapReduce framework. DE clustering is carried out through three levels. First, a MapReduce job is launched to carry out the mutation and crossover operations for generating new offspring. Then, the current population and offsprings are evaluated using the fitness function by running the MapReduce job. After that, the selection operation is performed to improve the current population. The proposed approach was tested using 18 real gene expression data sets, and the experimental results achieved by the proposed approach are found to be more reliable compared to K-means and PSO MapReduce.

A Spark-based clustering algorithm using particle swarm optimization (S-PSO) was proposed in [25]. In this work, a new strategy was introduced to improve the quality of the clustering by running the k-means algorithm in the final stage. S-PSO runs through repeated three steps which are fitness evaluation, personal and global best updating, and position and velocity updating. The fitness evaluation, and position and velocity updating steps are run in a parallel manner using the map and reduce functions. After S-PSO is completed, the global best position is taken as an initial centroid for the k-means algorithm that runs in parallel. Using real and synthetic data sets, S-PSO achieved good performance and scalability results. Other works for implementing PSO clustering using Apache Spark can be found in [26], [27].

In [28], a new ABC clustering algorithm based on the MapReduce model (MR-ABC) was proposed. MR-ABC aims to find the optimal centroids by minimizing the sum of the Euclidian distances. MR-ABC carries out fitness level evaluation by launching a MapReduce job. During the MapReduce job, the Map function creates a new key-value pair whereby the value is the minimum computed distance between the data point and the centroids. Then, emitting the key-value pairs to the Reduce function to calculate the average distance by aggregating the values with the same key. MR-ABC was experimented using real and synthetic data sets, and the analysis of the results showed the effectiveness and robustness of the MR-ABC for solving the clustering task.

Another work found in [29], a Spark-based artificial bee colony for clustering was proposed. The idea of clustering is the same as MR-ABC, but the authors used the Apache Spark framework to run the algorithm. The algorithm evaluates the fitness of bees (individuals) in a parallel manner by distributing the RDDs to the worker nodes along with the individuals. The algorithm was tested using the KDDCUP99 data set to evaluate its effectiveness. The experimental result analysis showed that the algorithm obtained good accuracy and achieved almost linear speedup.

A novel clustering algorithm using enhanced grey wolf optimizer that is run using the MapReduce framework (MR-EGWO) and was introduced in [30]. MR-EGWO starts by dividing the data set into partitions that are distributed among the Hadoop nodes. Then, the MapReduce job is launched to create a set of key/value pairs using the Map function. After that, the set of key/value pairs is used by the Reduce function to collect values based on a key. MR-EGWO was tested using real and synthetic data sets. The results showed that MR-EGWO outperformed four MapReduce-based clustering algorithms according to the F-measure and also obtained significant speedup results.

3. Cost-Sensitive Differential Evolution Classifier Based On Spark

In [14], our cost-sensitive differential evolution classification algorithm showed itself as efficient and robust to handle highly imbalanced binary data sets. However one of the drawbacks is that it shows inefficient performance when applied to big data sets. In this work, we proposed a scalable design for the cost-sensitive

differential evolution classifier based on Apache Spark (SCDE) in order to handle big data sets. The SCDE algorithm is based on differential evolution to find the optimal centroid of each class label in a training data set. Then, each instance in a testing data set is assigned to the closest centroid based on the Euclidean distance. SCDE starts with a predefined number of individuals NP that form the initial population $Pop_{initial}$. Each individual $\vec{x}_i$ is encoded as follows:

$$\vec{x}_i = \{\vec{v}_{c_1}, \vec{v}_{c_2}, \ldots, \vec{v}_{c_n}\}, i = \{1, 2, \ldots, NP\} \tag{1}$$

where $\vec{v}_{c_n}$ is a d-dimensional centroid vector of class label c_n.

Furthermore, each individual (vector) has an identification number (ID). Each centroid vector of individual $\vec{x}_i$ in the initial population $Pop_{initial}$ is randomly initialized in the d-dimensional problem space. After initialization, the individuals are evaluated using an objective function F to measure their fitness as follows:

$$F(\vec{x}_i) = \sum_{j=1}^{N} F\psi_{cost}(\vec{I}_j) \tag{2}$$

$$F\psi_{cost}(\vec{I}_j) = \begin{cases} Cost^{+} & if \quad \hat{y} \neq y \quad and \quad y = + \\ Cost^{-} & if \quad \hat{y} \neq y \quad and \quad y = - \\ Otherwise & 0 \end{cases} \tag{3}$$

Here, $Cost^{+}$ is the misclassification cost of the positive class label, $Cost^{-}$ is the misclassification cost of the negative class label, N is the total number of instances in the data set, $\hat{y}$ is the predicted class label of instance $\vec{I}_j$, and y is the actual class label of $\vec{I}_j$. The fitness level of the individual vector $\vec{x}_i$ is evaluated in two steps. In the first step, all instances in a training data set are assigned to the closest centroid according to the Euclidean distance, Equation 4. After that, the second step is carried out by summing over the misclassification cost of all instances that are misclassified.

$$d\left(\vec{a}, \vec{b}\right) = \sqrt{\sum_{k=1}^{n} (a_k - b_k)^2} \tag{4}$$

Then, $Pop_{initial}$ goes through a repeated evolution process (mutation, crossover, and selection) to improve $Pop_{initial}$.

The operations for generating a new population in the following generation $G + 1$, besides the fitness evaluation, need to be adapted in order to work on large data sets. For example, suppose the data set contains 100 million examples with two class labels and the population size is 20, thus, the cost-sensitive DE classifier needs to compute $100{,}000{,}000 \times 2 \times 20 = 4{,}000{,}000{,}000$ distance values during one iteration. The fitness evaluation of the individuals requires more computational time and consumes much more memory space compared to the operations (mutation, crossover, and selection) for generating a new population in generation $G+1$. In SCDE, the fitness evaluation of the individuals is carried

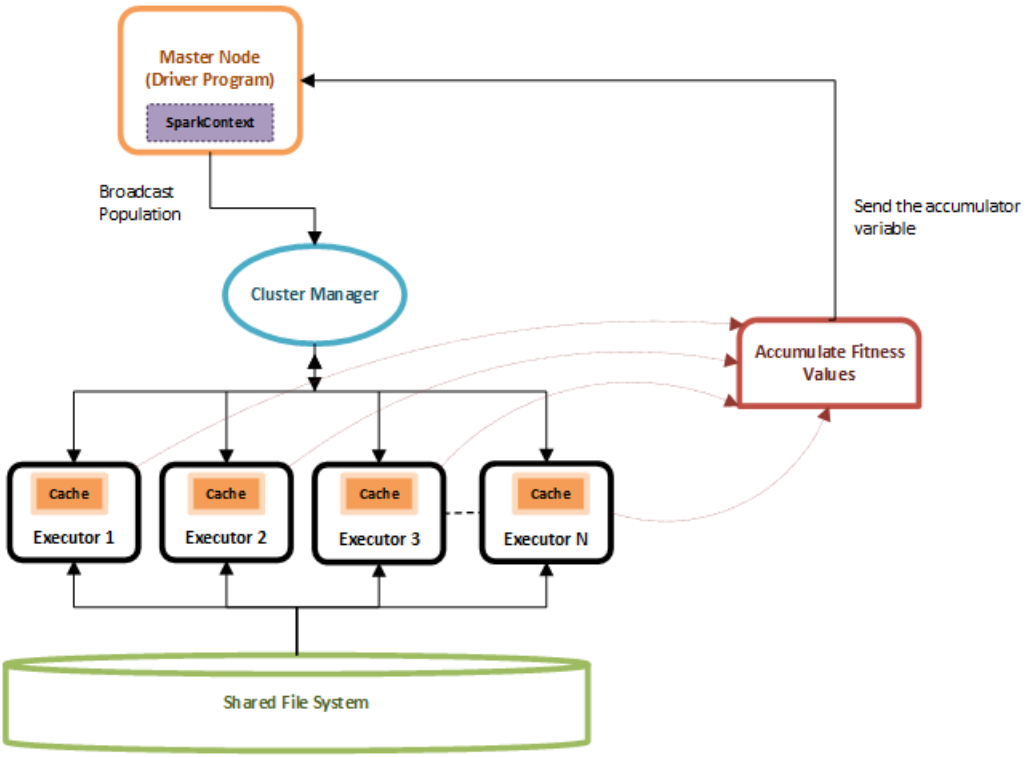


Figure 1: SCDE Architecture

out across a cluster of worker nodes while the operations for generating a new population for $G+1$ is performed in the master node.

Figure 1 shows the architecture of SCDE. In Figure 1, the driver program starts by creating a directed acyclic graph (DAG) for the RDD operations and sends the tasks to the executors. In addition, the driver program sends the intial population as a broadcast variable to the executors through a cluster manager for the fitness evaluation. After that, the executors starts reading the part of the RDD to work on. In this stage, each executor starts extracting one instance vector I at a time from the RDD. Then, for each individual, I is assigned to the closest centroid based on the Euclidean distance. If the class label of the assigned centroid does not match with the actual class label cl of I, then vector's ID and the cost of misclassification cl are added to the accumulator variable $Accumulator_{FV}$ as shown in Algorithm 1. It should be noted here that the data instances are read only once by the executors and are cached in the executors' memory for the next iterations.

Algorithm 1 Fitness Function Evaluation - Executor

```
for each data instance x in RDD do
    for each individual (vector) in Broadcast Variable do
        Assign x to closest centroid
        if Assigned_Class(x) != Actual_Class(x) then
            Add MisclassificationCost and Individual's ID to accumulator Accumulator_FV
        end if
    end for
end for
```

After the executors finished their work, the accumulator variable $AccumulatorFV$ is sent to the driver program (master node). In the driver program, the generation of the new population for the next generation $G+1$ starts with updating the fitness of all the individuals. Then, Pop sequentially goes through the mutation, crossover, and selection operations to generate a new population. Here, after the crossover operation is carried out, the trail vectors are sent to the executors for fitness evaluation. Then, the selection operation is performed by applying the principle of "the survival of the fittest" to choose between the current individual and its trial vector based on the fitness value. The portion of the pseudo code of the driver program is shown in Algorithm2.

In our work, the scaling factor F is a random value chosen using Equation 5 [31]. In addition, the crossover rate CR value is linearly decreasing with increasing numbers of generations (Equation 6) [31]. The aim is to explore the entire space at the beginning by replacing most of the target vector elements with the mutant vector elements. But at later generations, the CR value will linearly decrease, thus, more elements will inherit from the target vector. This leads to explore the interior space comprehensively [31]. It should be noted here

Algorithm 2 The portion of pseudo code - Driver program

- Initializing all the individuals in the initial population $Pop_{initial}$ randomly.

- Sending the $Pop_{initial}$ to the exectors for fitness evaluation.

- Updating the fitness value for the individuals in $Pop_{initial}$

while Maximum number of generation is not reached **do**
for Each individual $\vec{x}_i$ in the current population Pop **do**
- A mutant vector $\vec{m}_i$ is generated using $DE/best/1/bin$ schema as follows:

$\vec{m}_i = \vec{x}_{best,G} + F.(\vec{x}_{r1} - \vec{x}_{r2})$ [1]
- A trail vector $\vec{t}_i$ (candidate) is generated by combining $\vec{x}_i$ and $\vec{m}_i$ as follows:

$$\vec{t}_{j,i} = \begin{cases} \vec{m}_{j,i} & if \quad (rand(0,1.0) < CR \quad or \quad rand_j == j)^2 \\ \vec{x}_{j,i} & otherwise \end{cases}$$

end for
Sending the trail vectors to the executors for fitness evaluation.
The selection operation is carried out to generate a new population.
end while

that each element in the trail vector should be within the range [0,1].

$$F = 0.5 * (1 + rand(0,1)) \tag{5}$$

$$CR = CR_{max} - \left((CR_{max} - CR_{min}) \frac{G}{G_{max}} \right) \tag{6}$$

4. Data set and Environment

In our work, we used a new and reliable intrusion data set, which is CICIDS2017, to evaluate the performance and scalability of SCDE. CICIDS2017 was generated by the Canadian Institute for Cybersecurity, which contains the most recent types of attacks captured during five days starting from July 3, 2017. The CICIDS2017 data set consists of eight CSV files, which are the network traffic analysis results that were collected using the CICFlowMeter tool

[1] Where *best* is an index of the best vector, and $r1$ and $r2$ are random integer values within the range $\{1, 2, \ldots, NP\}$, which should be dissimilar and different than the index values of the best vector and the target vector. F is the scaling factor.

[2] CR is the crossover rate.

Table 1: Properties of Data Sets

Data set	# Features	Label	# Instances	Percentage	Total
Botnet	4	BENIGN (Normal)	188,955	98.98%	190,911
		Botnet Attack *	1,956	01.02%	
Brute Force	3	BENIGN (Normal)	431,813	96.90%	445,645
		Brute Force Attack *	13,832	03.10%	
Web Attack	4	BENIGN (Normal)	168,051	98.72%	170,231
		Web Attack *	2,180	01.28%	
Port Scan	5	BENIGN (Normal)	127,292	44.49%	286,086
		Port Scan Attack	158,804	55.51%	

* A minority class label.

[32]. Each CSV file contains one type of attack along with normal traffic (Benign). Besides, each record has 78 features describing the behavior of 25 users. For our experiments, we focused on four types of attacks which are Brute force, Botnet, Port Scan, and Web Attack that were taken from CICIDS2017. Table 1 shows the data sets including the number of features, the total number of instances in each data set as well as the number of instances and the distribution percentage of each class label. From the table we can easily see that three out of four data sets are highly imbalanced. For example, Botnet attack represents only 01.02% of instances in the Botnet data set.

It should be noted here that the instances that have a missing value were removed. Furthermore, we performed supervised feature selection on the data sets using WEKA [33] to reduce the number of features as shown in Table 1, and then applied the Min-Max normalization technique to normalize all data sets.

The Brute Force and Web Attack data sets are multi-labels data sets. Therefore, we decided to transform both data sets into binary data sets by mapping various attack types to one attack. In the Brute Force data set, "SSH-Patator" and "FTP-Patator" labels are mapped to one label named "Brute Force". For the Web Attack data set, "Web Attack-Sql", "Web Attack-XSS", and "Web Attack-Brute Force" labels are mapped to the "Web Attack" label.

The experiments were run on the SDSC Dell Cluster with Intel Haswell Processors (COMET) operated by the San Diego Supercomputer Center at UC San Diego[3]. The SDSC-Comet cluster consists of 1,948 nodes that have 24 Intel Xeon cores (2.5 GHz speed) and 128GB of DRAM. We implemented and run the SCDE algorithm using Java Runtime 1.8, Spark version 2.1, and the standalone cluster manager.

5. Experiments and Results

In this section, we will start by describing the measurements that were used to evaluate the performance and scalability of SCDE and also the parameter setting of SCDE . Then, the experiments and the results are explained. We will focus in particular on the scalability for SCDE including speedup and scaleup.

5.1. Evaluation Measures

In this paper, we used various measures to assess the robustness and scalability of the SCDE algorithm. To assess the performance and effectiveness of SCDE, we used the Detection Rate, False Alarm Rate, and Geometric Mean (G-mean) measures. The following are the descriptions of these measures:

- True positive (TP): The number of attack instances that are detected correctly.
- True negatives (TN): The number of normal behavior instances that are classified correctly.
- False positives (FP): The number of normal behavior instances that are classified as attacks.
- False negatives (FN): The number of attack instances that are undetected.
- Detection Rate: The ratio of detecting the attacks is calculated as follows:

$$\frac{TP}{TP + FN} \tag{7}$$

- False Alarm Rate: The ratio of the wrong prediction of normal behavior is calculated as follows:

$$\frac{FP}{TN + FP} \tag{8}$$

[3]https://portal.xsede.org/sdsc-comet

– G-mean: This metric is one of the essential measures to evaluate the performance of a classifier on a highly imbalanced data set [34], which is calculated as follows:

$$G - mean = \sqrt{TPR(sensitivity) \times TNR(specificity)} \tag{9}$$

For the scalability evaluation, we used the following measures:

– Speedup [35]: Is a metric that measures the parallelization capability of the application, which is the ratio of the computation time on a single node T_1 to the computation time on P nodes T_p. The speedup is calculated as follows:

$$Speedup = \frac{T_1}{T_p} \tag{10}$$

Here, the number of nodes P is increased in a certain ratio while the data set size is fixed.

– Scaleup [35]: Is a metric that measures how the cluster of nodes are utilized efficiently by the parallel algorithm, which is calculated as follows:

$$Scaleup = \frac{T_{sp}}{T_{Rsp}} \tag{11}$$

Here, the data set size S and the P nodes are increased by the same ratio R.

For the experiments, the parameters of SCDE were taken from [31], except the maximum number of generations, which are:

- Maximum number of generations = 200
- Population size $NP = 100$
- Crossover range [$\mathrm{CR}_{\min}$=0.5, $\mathrm{CR}_{\max}$ = 1.0]
- Scaling factor F: a random value in range [0.5, 1.0]

5.2. Performance Analysis

For the performance experiment, we performed 25 independent runs for SCDE on the data sets in Table 1 to evaluate the performance and effectiveness of SCDE. In addition, we compared the performance of SCDE with the performance of three chosen cost-sensitive classification algorithms. The chosen algorithms are Logistic Regression [36], Naïve Bayes [37], and RBF Network [38]. We ran the cost-sensitive classification algorithms using the Waikato Environment for Knowledge Analysis (WEKA) tool version 3 [39],[40].

For this experiment, the misclassification cost of the majority and minority class labels for the Botnet, Brute Force, Web Attack data sets are empirically

Table 2: Misclassification cost of class labels for each data set

Data set	**Class labels**	
Botnet	Botnet Attack *	Normal
	97.0	1.0
Brute Force	Brute Force Attack *	Normal
	31.0	1.0
Web Attack	Web Attack *	Normal
	76.0	1.0
Port Scan	Port Scan Attack	Normal *
	1.0	1.2

* Minority class label.

Table 3: G-mean Results in Percent

Machine Learning Algorithms	**Bot Attack**	**Brute Force**	**Web Attack**	**Port Scan**	**Average**
Logistic Regression	87.12	99.84	90.29	99.21	**94.12**
Naïve Bayes	77.94	99.64	34.76	97.75	**77.52**
RBF Network	88.45	59.37	90.57	98.28	**84.17**
SCDE-F_{cost}	91.80 [±0.06]	99.56 [±0.00]	90.52 [±0.71]	99.14 [±0.04]	**95.26**

Table 4: Detection Rate Results in Percent

Machine Learning Algorithms	**Bot Attack**	**Brute Force**	**Web Attack**	**Port Scan**	**Average**
Logistic Regression	100.00	99.99	84.22	99.53	**95.93**
Naïve Bayes	100.00	99.96	96.60	99.31	**98.97**
RBF Network	100.00	99.89	84.22	99.29	**95.85**
SCDE-F_{cost}	99.81 [±0.11]	100.00 [±0.00]	88.14 [±2.50]	99.22 [±0.01]	**96.79**

Table 5: False Alarm Rate Results in Percent

Machine Learning Algorithms	**Bot Attack**	**Brute Force**	**Web Attack**	**Port Scan**	**Average**
Logistic Regression	24.10	0.31	3.20	1.12	**7.18**
Naïve Bayes	39.25	0.69	87.49	3.79	**32.81**
RBF Network	21.77	64.72	2.60	2.72	**22.95**
SCDE-F_{cost}	15.57 [±0.02]	0.89 [±0.00]	6.99 [±1.55]	0.94 [±0.07]	**6.10**

Table 6: Duplicate Data Sets

Data set	**Duplication Rate**	**Size**
20T	20 times	8,912,900
40T	40 times	17,825,800
60T	60 times	26,738,700

determined. Table 2 shows the best misclassification cost that was obtained for each data set.

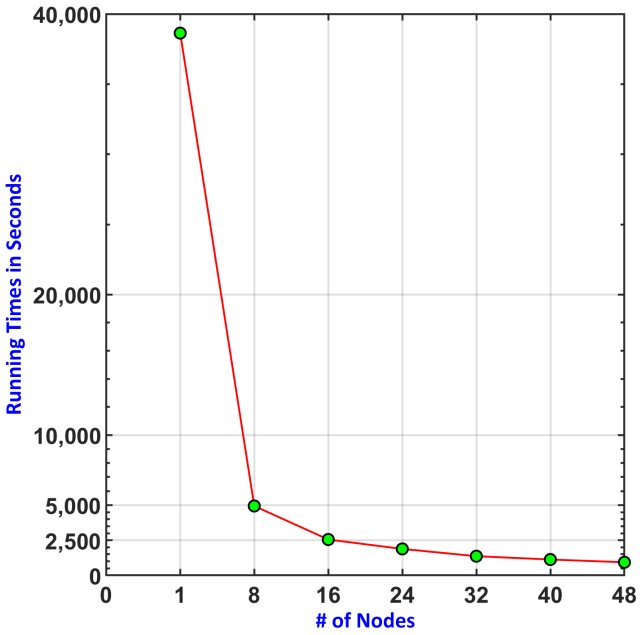
Tables 3, 4, and 5 show the results achieved by SCDE and three cost-sensitive classification algorithms for all data sets in terms of G-mean, detection rate, and false alarm rate, respectively. Furthermore, the G-mean, detection rate, and false alarm rate results were averaged over the given data sets, as shown in the last column in Tables 3, 4, and 5, respectively. Moreover, the standard deviation of 25 independent runs for SCDE are given within the brackets.

From the results, SCDE obtained the best average G-mean and false alarm rate results compared to the Logistic Regression, Naïve Bayes, and the RBF Network algorithm, where the results were 95.26% and 6.10%, respectively. Furthermore, SCDE achieved a very good average detection rate, where the result was 96.79%. Overall, SCDE is competitive compared to three cost-sensitive ML algorithms as well as it can detect attacks efficiently with a low false alarm rate.

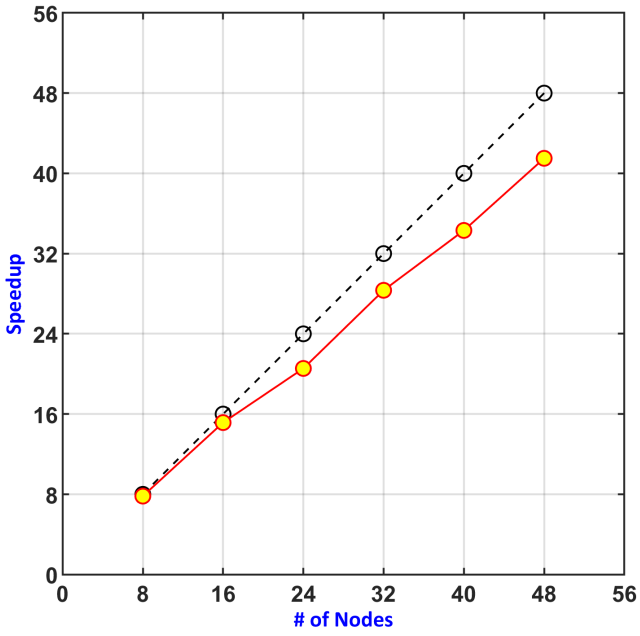
5.3. Scalability Analysis

Since the data sets in Table 1 are too small to measure the speedup of SCDE on a large cluster of nodes, we decided to duplicate the largest data set in Table 1, which is the Brute Force data set by replicating the original data set 20, 40, and 60 times. Table 6 shows the sizes of the Brute Force data sets after duplication.

In the speedup measurement experiment, we ran SCDE on all duplicate data sets for up to 48 nodes on the COMET cluster. For each run, the number of nodes is increased by multiples of eight while the data size is kept fixed. We

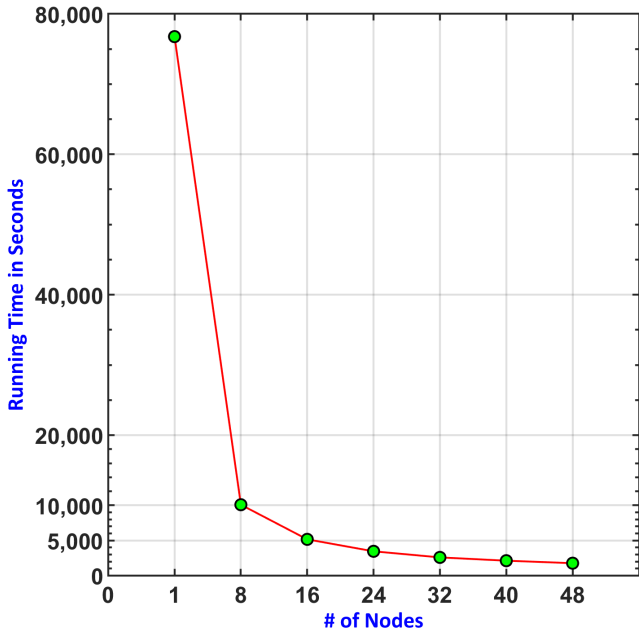


(a) Running Time

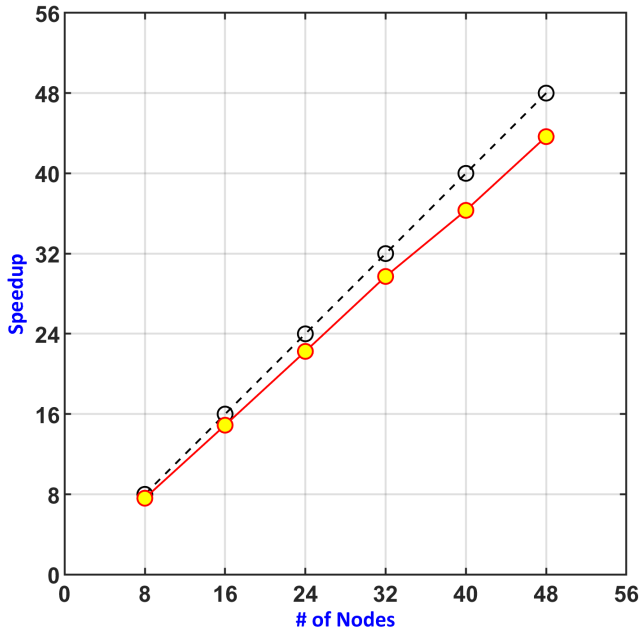


(b) Speedup

Figure 2: Running Time and Speedup using the 20T data set

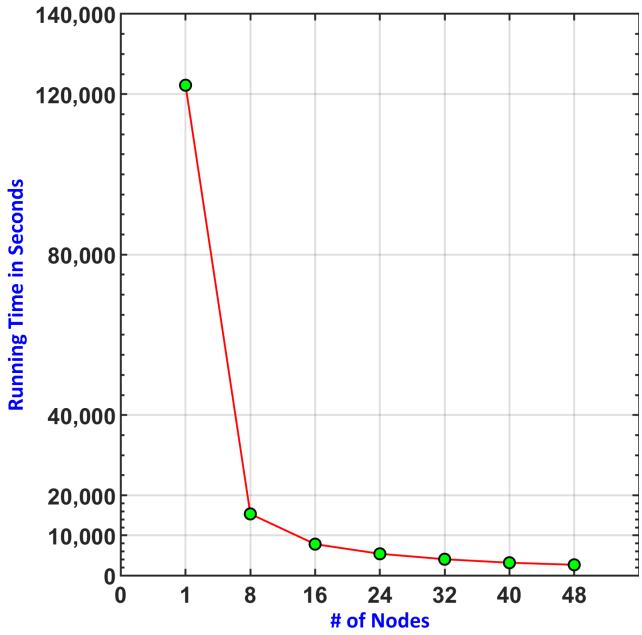


(a) Running Time

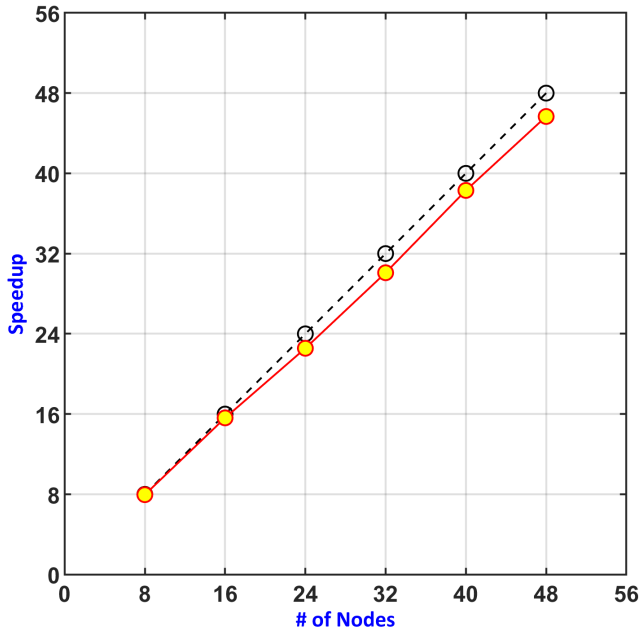


(b) Speedup

Figure 3: Running Time and Speedup using the 40T data set



(a) Running Time



(b) Speedup

Figure 4: Running Time and Speedup using the 60T data set

reported the running time in seconds and the speedup of SCDE for all duplicate data sets as shown in Figures 2 to 4. The black dashed line in Figures 2b, 4b, 4b represents the linear speedup.

From Figures 2a, 3a, and 4a, we can easily observe that the running time of SCDE for all data sets is decreasing approximately exponentially. For example, the running time of SCDE for the 20T, 40T, and 60T data sets using 48 nodes is reduced to 931s, 1,758s, and 2,676s, respectively, compared to the running time of SCDE using one node which are 38,633s, 76,761s, 122,219s for the 20T, 40T, and 60T data sets, respectively.

In the speedup evaluation, we can see from Figure 2b that the speedup results for 8 and 16 nodes using the 20T data set are approximately the linear speedup. The speedup values then drift away a little from the linear speedup starting from 24 nodes, where the speedup results for 24, 32, 40, and 48 nodes using the 20T data set are 20.55, 28.34, 34.31, and 41.50, respectively.

In Figure 3b , the speedup results using the 40T data set are almost linear for 8, 16, 24, and 32 nodes where the speedup results are 7.61, 14.89, 22.25, 29.72, respectively. For 40 and 48 nodes, the speedup results are 36.31, and 43.66, which are very close to the linear speedup.

For the 60T data set, the speedup results are almost identical to the linear speedup where the speedup results for 8, 16, 24, 32, 40, and 48 nodes are 7.96, 15.62, 22.56, 30.09, 38.30, and 45.67, respectively as shown in Figure 4b.

Overall, we can conclude from the previous results that the running time of SCDE is significantly improved with increasing numbers of nodes. Furthermore, the speedup achieved by SCDE is close to the linear speedup.

For the Scaleup evaluation experiment, we used the Brute Force data set to run SCDE on the COMET cluster to measure the capability of SCDE on utilizing the cluster of nodes efficiently. For this experiment, we doubled the data set size and number of nodes for each run starting from 445,645 records for data size and 2 nodes. Figure 5 shows the scaleup results achieved by SCDE. From this figure, SCDE shows good scaleup results that have approximately a constant ratio ranging between 0.96 and 1.0. Moreover, the results are close to 1.0 which is the optimal value.

6. Conclusion and Future Works

In this paper, we designed and implemented a scalable cost-sensitive differential evolution (SCDE) algorithm using Apache Spark to solve the classification task for imbalanced and massive data. SCDE is based on DE to find the optimal centroid vector for each class label by minimizing the total misclassification cost. SCDE assigns each instance in a testing data set to the closest centroid.

The experiments used real intrusion detection data sets to evaluate SCDE's performance and scalability. The experimental results revealed that SCDE efficiently detects an attack with a low false alarm rate. Furthermore, the scalability analysis showed that the improvement factor of SCDE's running times are very close to the linear values for most data sets, and SCDE scales efficiently with data size increases.

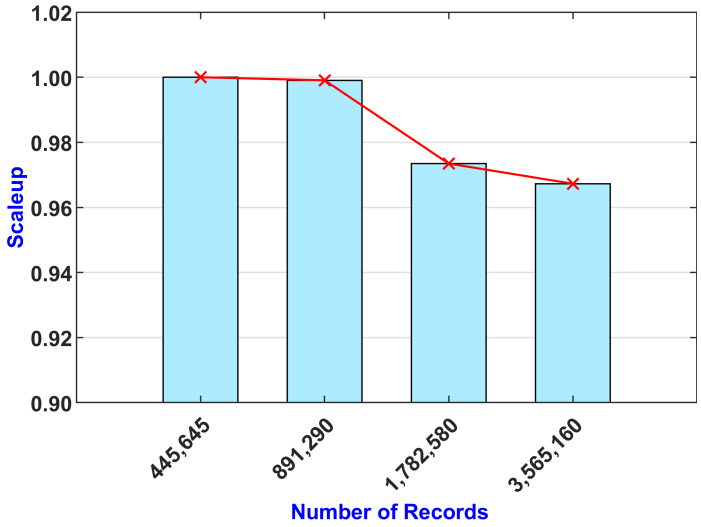


Figure 5: SCDE Scaleup

Overall, we can conclude that SCDE is robust and efficient for handling binary imbalanced data and achieved good speedup and scaleup results that are close to linear.

Our future work aims to conduct experiments on terabyte-size data sets using hundreds nodes. Moreover, finding a solution to cope with imbalanced multi-label data set is another future topic of investigation.

Acknowledgment

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562.

[1] S. Sayad, Real Time Data Mining, Self-Help Publishers, 2011.

[2] C. Qiu, L. Jiang, G. Kong, A differential evolution-based method for class-imbalanced cost-sensitive learning, in: Neural Networks (IJCNN), 2015 International Joint Conference on, IEEE, 2015, pp. 1–8.

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

[4] S. Fotouhi, S. Asadi, M. W. Kattan, A comprehensive data level analysis for cancer diagnosis on imbalanced data, Journal of biomedical informatics.

[5] R. Storn, K. Price, Differential evolution-a simple and efficient heuristic for global optimization over continuous spaces, Journal of global optimization 11 (4) (1997) 341–359.

[6] P. Rocca, G. Oliveri, A. Massa, Differential evolution as applied to electromagnetics, IEEE Antennas and Propagation Magazine 53 (1) (2011) 38–49.

[7] S. Das, A. Konar, U. K. Chakraborty, Two improved differential evolution schemes for faster global search, in: Proceedings of the 7th annual conference on Genetic and evolutionary computation, ACM, 2005, pp. 991–998.

[8] D. Dawar, S. A. Ludwig, Effect of strategy adaptation on differential evolution in presence and absence of parameter adaptation: An investigation, Journal of Artificial Intelligence and Soft Computing Research 8 (3) (2018) 211–235.

[9] 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, USENIX Association, 2012, pp. 2–2.

[10] Apache hadoop- mapreduce, `https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html` , Accessed on Feb-10-2019.

[11] S. Kumar, Apache Spark for Java Developers, Packt Publishing Limited, 2017.

[12] H. Karau, A. Konwinski, P. Wendell, M. Zaharia, Learning Spark: Lightning-Fast Big Data Analysis, OReilly, 2015.

[13] Spark 2.1.0 documentation, `https://spark.apache.org/docs/2.1.0/` , Accessed on Feb-10-2019.

[14] J. Al-Sawwa, S. A. Ludwig, A cost-sensitive centroid-based differential evolution classification algorithm applied to cancer data sets, accepted at 2019 IEEE Symposium Series on Computational Intelligence (SSCI).

[15] I. De Falco, A. Della Cioppa, E. Tarantino, Automatic classification of handsegmented image parts with differential evolution, in: Workshops on Applications of Evolutionary Computation, Springer, 2006, pp. 403–414.

[16] P. Luukka, J. Lampinen, A classification method based on principal component analysis and differential evolution algorithm applied for prediction diagnosis from clinical emr heart data sets, in: Computational Intelligence in Optimization, Springer, 2010, pp. 263–283.

[17] S. Khan, K. A. Shakil, M. Alam, Big data computing using cloud-based technologies, challenges and future perspectives, arXiv preprint arXiv:1712.05233.

[18] C. Zhou, Fast parallelization of differential evolution algorithm using mapreduce, in: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, GECCO '10, ACM, New York, NY, USA, 2010, pp. 1113–1114. doi:10.1145/1830483.1830689.
URL http://doi.acm.org.ezproxy.lib.ndsu.nodak.edu/10.1145/1830483.1830689

[19] D. Teijeiro, X. C. Pardo, P. González, J. R. Banga, R. Doallo, Implementing parallel differential evolution on spark, in: European Conference on the Applications of Evolutionary Computation, Springer, 2016, pp. 75–90.

[20] C. Deng, X. Tan, X. Dong, Y. Tan, A parallel version of differential evolution based on resilient distributed datasets model, in: Bio-Inspired Computing-Theories and Applications, Springer, 2015, pp. 84–93.

[21] A. W. McNabb, C. K. Monson, K. D. Seppi, Parallel pso using mapreduce, in: 2007 IEEE Congress on Evolutionary Computation, IEEE, 2007, pp. 7–14.

[22] G. Miryala, S. A. Ludwig, Comparing spark with mapreduce: Glowworm swarm optimization applied to multimodal functions, International Journal of Swarm Intelligence Research (IJSIR) 9 (3) (2018) 1–22.

[23] L. Cui, Parallel pso in spark, Master's thesis, University of Stavanger, Norway (2014).

[24] M. Daoudi, S. Hamena, Z. Benmounah, M. Batouche, Parallel diffrential evolution clustering algorithm based on mapreduce, in: 2014 6th International Conference of Soft Computing and Pattern Recognition (SoCPaR), IEEE, 2014, pp. 337–341.

[25] M. Moslah, M. A. B. HajKacem, N. Essoussi, Spark-based design of clustering using particle swarm optimization, in: Clustering Methods for Big Data Analytics, Springer, 2019, pp. 91–113.

[26] M. Sherar, F. Zulkernine, Particle swarm optimization for large-scale clustering on apache spark, in: 2017 IEEE Symposium Series on Computational Intelligence (SSCI), IEEE, 2017, pp. 1–8.

[27] K. Govindarajan, D. Boulanger, V. S. Kumar, et al., Parallel particle swarm optimization (ppso) clustering for learning analytics, in: 2015 IEEE International Conference on Big Data (Big Data), IEEE, 2015, pp. 1461–1465.

[28] A. Banharnsakun, A mapreduce-based artificial bee colony for large-scale data clustering, Pattern Recognition Letters 93 (2017) 78–84.

[29] Y. Wang, Q. Qian, A spark-based artificial bee colony algorithm for large-scale data clustering, in: 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), IEEE, 2018, pp. 1213–1218.

[30] A. K. Tripathi, K. Sharma, M. Bala, A novel clustering method using enhanced grey wolf optimizer and mapreduce, Big data research 14 (2018) 93–100.

[31] A. Abraham, S. Das, A. Konar, Document clustering using differential evolution., in: IEEE Congress on Evolutionary Computation, 2006, pp. 1784–1791.

[32] Canadian institute for cybersecurity, `https://www.unb.ca/cic/datasets/ids-2017.html` , Accessed on Feb-10-2019.

[33] 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.

[34] J. Akosa, Predictive accuracy: A misleading performance measure for highly imbalanced data, in: Proceedings of the SAS Global Forum, 2017.

[35] A. Grama, A. Gupta, G. Karypis, V. Kumar, Introduction to Parallel Computing, Second Edition, Addison-Wesley, 2003.

[36] N. Landwehr, M. Hall, E. Frank, Logistic model trees 95 (1-2) (2005) 161–205.

[37] G. H. John, P. Langley, Estimating continuous distributions in bayesian classifiers, in: Eleventh Conference on Uncertainty in Artificial Intelligence, Morgan Kaufmann, 1995, pp. 338–345.

[38] D. Broomhead, D. Lowe, Multivariable functional interpolation and adaptive networks, Complex Systems 2 (3).

[39] 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.

[40] I. H. Witten, E. Frank, M. A. Hall, C. J. Pal, Data Mining: Practical machine learning tools and techniques, Morgan Kaufmann, 2016.