Preprints of the Federated Conference on
Computer Science and Information Systems pp. 181–192

Parallel computation of information gain using Hadoop and MapReduce

Eftim Zdravevski*, Petre Lameski†, Andrea Kulakov‡, Sonja Filiposka§, Dimitar Trajanov¶, Boro Jakimovski‖
Faculty of Computer Science and Engineering
Ss.Cyril and Methodius University, Skopje, Macedonia
Email: *eftim.zdravevski@finki.ukim.mk, †petre.lameski@finki.ukim.mk, ‡andrea.kulakov@finki.ukim.mk,
§sonja.filiposka@finki.ukim.mk, ¶dimitar.trajanov@finki.ukim.mk, ‖boro.jakimovski@finki.ukim.mk

***Abstract*—Nowadays, companies collect data at an increasingly high rate to the extent that traditional implementation of algorithms cannot cope with it in reasonable time. On the other hand, analysis of the available data is a key to the business success. In a Big Data setting tasks like feature selection, finding discretization thresholds of continuous data, building decision threes, etc are especially difficult. In this paper we discuss how a parallel implementation of the algorithm for computing the information gain can address these issues. Our approach is based on writing Pig Latin scripts that are compiled into MapReduce jobs which then can be executed on Hadoop clusters. In order to implement the algorithm first we define a framework for developing arbitrary algorithms and then we apply it for the task at hand. With intent to analyze the impact of the parallelization, we have processed the FedCSIS AAIA'14 dataset with the proposed implementation of the information gain. During the experiments we evaluate the speedup of the parallelization compared to a one-node cluster. We also analyze how to optimally determine the number of map and reduce tasks for a given cluster. To demonstrate the portability of the implementation we present results using an on-premises and Amazon AWS clusters. Finally, we illustrate the scalability of the implementation by evaluating it on a replicated version of the same dataset which is 80 times larger than the original.**

***Keywords*—Hadoop, MapReduce, information gain, parallelization, feature ranking**

I. Introduction

THE volume of data that needs to be processed has increased significantly in recent years. Most of the organizations in the world base their decisions on the data they collect and they need large volumes of data to be processed in as little time as possible. Over the years many ideas have been developed for solving the Big Data challenge. Increasing the processing power is the logical way to go but this has proven to be effective up to a certain point. After that the hardware scaling is not yet effective enough. The idea of distributing the computation has become popular in recent years since the publications of Google's approaches for MapReduce [1] in 2004 and the concept of Big Table [2] in 2006. Other companies have followed similar paths introducing open-source solutions. One such system is Apache Hadoop that contains a set of algorithms for distributed processing, storage of large datasets on computer clusters, scheduling etc. It is a framework that is employed by industry leaders like Yahoo, Facebook, Ebay, Adobe, etc [3].

Machine learning algorithms such as decision trees [4], neural networks [5], Naive Bayes [6, 7] and many others can automatically analyze data and make conclusions, predictions or even find patterns that otherwise cannot be detected. The main drawback of these algorithms is the degrading performance in presence of redundant and irrelevant features. Other algorithms such as Support Vector Machines are able to cope with this problem to some extent, however, this ability increases the computational time so much that the algorithm doesn't give result in reasonable time. This has already been confirmed in the literature [8, 9, 10]. One way to resolve this is to perform feature selection [11, 9, 12], defined as the task of selection of feature subsets that describe the hypothesis at least as well as the original set. In [13] the most widely used methods for feature selection are introduced.

The rest of this paper is organized as follows. First, in section II we review the most recent approaches to parallelization of various algorithms. Afterwards in section III we describe the definition and applications of information gain. Next, in section IV we present we describe the services in the Hadoop ecosystem and then we present a framework for parallelization of algorithms. Thereupon, in section V we apply it for parallel and distributed computation of information gain based on MapReduce. Next, in section VI we present the experimental setup and the obtained results. Finally, in section VII we discuss the contribution of our work and our plans for further research.

II. Related work

This section describes some of the most recent work on parallelizing different algorithms with MapReduce. The general approaches and limitations of different data mining algorithms when applied to massive datasets are described in [14]. Here some common data mining problems are explained from a Big Data perspective, but a MapReduce implementation is given only for some common problems like matrix manipulation and joins between tables.

A good overview of the parallel programming paradigms and frameworks in the Big Data era is presented in [15]. Here the authors describe the MapReduce paradigm, but more importantly introduce the frameworks that are built on top of it like: Pig Latin for processing data flows, Hive for non-real time querying of partitioned tables, and Spark and Twister for iterative parallel algorithms.

The authors in [16] address the problem of efficient feature evaluation for logistic regression on very large data sets. Here they present a new forward feature selection heuristic

©2015

that ranks features by their estimated effect on the resulting model's performance. They test the method on already available datasets from UCI, but also generate artificial datasets for which they know the logistic regression coefficients. They use that to evaluate the selected features.

By using the MapReduce paradigm in [17] a data intensive parallel feature selection method is proposed. In each map node, a method is used to calculate the mutual information and combinatory contribution degree is used to determine the number of selected features.

In [18] an implementation based on the MapReduce programming model of Naive Bayes is proposed. During the map phase all counts needed for calculating the conditional probabilities are emitted, and during the reduce phase they are aggregated.

A parallel implementation of the SVM algorithm for scalable spam filtering using MapReduce is proposed in [19]. By distributing, processing and optimizing the subsets of the training data across multiple participating nodes, the distributed SVM reduces the training time significantly. Merging of the results is actually a union of the individually computed support vectors. The cost of the parallelization is that because not all training data is available on all nodes, the performance can degrade. However, if the data is properly distributed on the nodes in regards to stratification per class, this problem can be mitigated.

A method for reducing the dataset to a small but representative subset is proposed in [20]. The idea is to use the representative subset for faster machine learning because the dataset size will be significantly reduces. The speedup is being calculated against a cluster with one node. However, if the dataset is too large, or the computation takes a lot of time the authors suggest to use more than one for estimating the speedup. By doing this one can calculate the speed of the current configuration versus the cluster with some smaller number of nodes.

In [21] an approach based on MapReduce for distributed column subset selection is proposed. In this approach each node has access to a random subset of features. This approach has a limitation that the datataset has to be manually splitted and the MapReduce jobs need to be written on lower level. The reason for this is that HBase segments the data horizontally by rows, so either the dataset needs to be transposed or to manually start different jobs and not to rely on a higher level language like Pig Latin or Hive.

Authors in [22] propose a wrapper approach for parallel feature selection. Here features are added to the selected set if after their addition, the performance of the classifier does not degrade. Then in a second phase from the subset obtained in the previous step, features are removed if their discarding does not degrade the classifier performance.

Apache Mahout [23] is an environment for quickly creating scalable performant machine learning applications based on MapReduce. Even though there are plenty of algorithms available in it, at the time of this writing, only two algorithms related to feature selection and dimensionality reduction in Mahout are available: Singular Value Decomposition (SVD) and Stochastic SVD.

III. Information gain and its applications

Information gain is a synonym for KullbackLeibler divergence and has variety of applications. Very often it is used for ranking individual features as described in [24, 25]. The research discussed in [26] shows how information gain can be used for feature selection in text categorization problems. Authors in [27] propose using the information gain for discretization of continuous valued features into discrete intervals. In like manner, in [28] information gain is analyzed as an unsupervised method for discretization of continuous features. Likewise, in [29] it is applied for improving decision tree performance by prior discretization of continuous-valued attributes. In fact these papers have inspired many other researchers to propose various other applications based on the information gain and entropy. In [30] the information gain in conjunction with methods based on particle filters is used for exploration, mapping, and localization. Another application of information entropy for extending the rough set based notion of a reduct is proposed in [31]. There it is applied for calculation of minimal subsets of features keeping information about decision labels at a reasonable level.

In the remaining of this section when describing the information gain we use the notation we have also used in [32]. In order to calculate the information gain, first the entropy $H(X)$ of the dataset should be calculated. Let X denote a set of training examples, and each of them x_i is in the form $(x_i^1, x_i^2, ..., x_i^k, y_i)$. Let each column (i.e. feature) be a discrete random variable that takes on values from set $V^j, j = 1..k$. Let the set of possible labels (i.e. classes) is L, such as $y_i \in L$. Then the entropy of the dataset X can be calculated with equation (1), where $p(l)$ is the probability of instance x_i to be labeled as l (i.e $y_i = l$) and is defined with equation (2).

$$H(X) = -\sum_{l \in L} p(l) \log p(l) \tag{1}$$

$$p(l) = \frac{|\{x_i \in X | y_i = y\}|}{|X|} \tag{2}$$

The information gain of the j-th feature of the dataset X can be calculated with equation (3), where first part in the sum is the probability of the instance x_i to have value v of the j-th feature. The second part in the sum in equation (3) denotes the entropy of the subset of instances of X that have the value v of the j-th feature.

$$IG(X,j) = H(X) - \sum_{v \in V^j} \frac{\left|\left\{x_i \in X | x_i^j = v\right\}\right|}{|X|} H(\left\{x_i \in X | x_i^j = v\right\}) \tag{3}$$

As shown by equations (1),(2) and (3), calculation of information gain of all features boils down to counting the number of instances per feature, value and class. After we compute these counts, we can calculate the probabilities and consequently calculate the information gain. In section V we propose parallel implementation for calculating the information gain of each feature j in the dataset X.

IV. Framework for parallelization

Parallelization of algorithms introduces a handful of potential software bugs of usually related to race conditions, communication and synchronization between the different subtasks. Owing to that, writing parallel computer programs is more challenging than writing sequential ones. In Hadoop most of those challenges are already addressed by various mechanisms and services, so when it is used as a platform for implementation of algorithms, the programmer does not have to put much effort for solving those kinds of issues. Before explaining the details we want to point out that the proposed framework uses the principle of data-parallelism. Having that in mind, the same principles could be used in a regular SQL environment. Nevertheless, while many of the limitations and benefits of SQL vs NoSQL are much argued in the research community, the scalability properties of NoSQL databases are undisputed.

Given that understanding of the Hadoop ecosystem is essential for understanding our parallelization framework, first in the next subsection IV-A we review its services. Then in the following subsections we describe the several phases of the proposed framework. We have applied similar logic in [32], but the solution was not a generic one and was custom for the task at hand. On Fig. 1 is shown a general overview of the data flow during these phases. For data partitioning we propose using HBase tables which are pre-splitted for optimal data distribution, where as for parallel processing and writing MapReduce jobs we suggest using Pig Latin with appropriate user-defined functions.

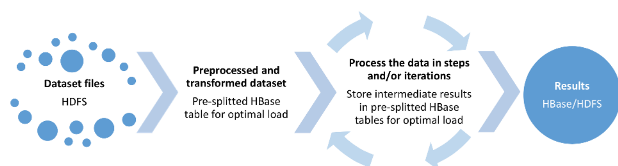

Fig. 1. Data flow phases for processing HBase tables with Pig Latin

A. Hadoop

The MapReduce [1, 33] paradigm is essential to the distributed computation and storage that Hadoop achieves. It consists of two phases: map and reduce. The first phase, map, splits the data into subsets. The reduce phase, aggregates the result from the output that the map phase produces. Procedures that can be performed in the map phase are: filtering, sorting, projecting and reading the data. The map phase returns an intermediate result consisted of keys and values. The reduce procedures use this data and perform aggregation. Hadoop delegates the data from the map phase to the reduce procedures. The MapReduce simplicity makes it very efficient for large-scale implementations on thousands of nodes.

Hadoop with its different services provides all the logistics and monitoring for the processes like scheduling, distribution, communication and data transfer, and also provides redundancy and fault tolerance. Many services or subsystems exist in Hadoop, but the most notable are: YARN (MapReduce2), HDFS and HBase [34][35].

YARN (Yet Another Resource Negotiator) [36] takes care of job scheduling, monitoring and resource management. Two separate daemons are responsible for these tasks: a global ResourceManager and per-application ApplicationMaster. The ResourceManager deploys resources among all the applications and the per-application ApplicationMaster negotiates for resources with the ResourceManager and works with the NodeManager to execute tasks and perform monitoring. YARN does the resource allocation and the distribution of MapReduce jobs to the apropriate nodes.

Hadoop Distributed File System (HDFS) [37] provides scalable, fault-tolerant, distributed storage system that works closely with MapReduce. It was designed to span large clusters of commodity servers. An HDFS cluster is consisted of a NameNode and DataNodes. The NameNode is responsible for the cluster metadata and DataNodes are responsible for data storage. The data is usually split into large blocks (typically 128 megabytes), independently replicated across multiple DataNodes.

HBase is an open source, non-relational, distributed database modeled after Google's BigTable. It runs on top of HDFS (Hadoop Distributed Filesystem), providing BigTable-like capabilities for Hadoop [38, 39, 40]. HBase is a NoSQL (Not Only SQL) database in which the tables are designed by analyzing usage patterns. This allows simplicity of design, horizontal scaling, and finer availability control. The data structures in NoSQL databases, such as HBase, allow faster executions of some operations than the execution of similar operations in relational databases. This mostly depends on the problem that must be solved. Tables in HBase can be used as the input and output for MapReduce jobs run in Hadoop. According to Eric Brewers CAP theorem, HBase is a CP type system (i.e. Consistent and Partition tolerant) [41].

The MapReduce programming model is very popular due to its simplicity. The extreme simplicity of MapReduce leads to much low-level coding that needs to be done for some operations that are much simpler when using relational databases. This increases development time, introduces bugs and may obstruct optimizations [42]. A group at Yahoo motivated by these repeatable tasks on daily basis, has developed a scripting language called Pig Latin. Pig is a high-level dataflow system that is a compromise between SQL and MapReduce. Pig offers constructs for data manipulation similar to SQL, which can be integrated in an explicit dataflow. Pig programs are compiled into sequences of MapReduce jobs, and executed in the Hadoop MapReduce environment [43].

B. Loading data into HDFS

This is the first and most simple phase. This phase should be performed once or multiple times, depending on how the dataset is structured. The most common formats for datasets are:

- CSV (comma separated values). This format is usually used to store dense datasets.
- ARFF (Attribute-Relation File Format). Also used to store dense datasets.
- EAV (Entity Attribute Value). Used to store sparse matrices that have a lot of zeros and some non-zero

elements.

If the dataset is only one file then it will be copied from the Linux File System to HDFS using a simple command. This means that for such cases during this step we cannot have parallelism. However if the dataset is dispersed into multiple files, then all of them can be copied simultaneously to HDFS. Be that as it may, this step usually is very fast compared to the following steps for machine learning, so its parallelization may not be necessary at all.

C. Facilitating data parallelism with HBase

After the previous step IV-B is finished the dataset files reside on HDFS. As it is extensively described in [37], each file in HDFS is replicated across several nodes for reliability. A typical file in HDFS is gigabytes to terabytes in size, splitted in blocks of 128 MB by default. If the files are too small than that could degrade the performance of the system and limit the level of parallelism. Map tasks usually process a block of input at a time. If the file is very small and there are a lot of them, then each map task processes very little input, and there are a lot more map tasks, each of which imposes extra bookkeeping overhead. Ideally the dataset will be one large file dispersed on multiple blocks on HDFS so when loaded, transformed and stored into HBase, greater parallelism can be achieved. Be that as it may, datasets are not always so large that HDFS can distribute them on all nodes and get optimal parallelization. One way to mitigate this is by splitting the dataset in multiple smaller files and store them in one folder, so later the Pig script can read from all files in the folder instead of a specific file. Nevertheless, this step again is usually very fast especially compared to the steps that comprise the actual algorithm, so we do not recommend to spend too much time on optimizing the file sizes for better parallelism.

Even though we can achieve parallelism while processing files stored on HDFS, the control of degree of parallelism is difficult, more involved and at very low-level. For instance, we if we have a large file then HDFS will automatically partition it and distribute it on different nodes. Be that as it may, we do not have control of how HDFS will do this, on how many partitions it will store it, where are they going to be distributed, etc. On the opposite side, if we have a small dataset file, it will not be partitioned at all. To have a better control on this one would need to manually split the file in the desired number of chunks and then let HDFS distribute them. Moreover, this process has to be repeated again and again if we have continuous stream of data.

On the other hand, HBase offers many other services built on top of HDFS, among which is a much better control of the degree of parallelism. This is due to the fact that the data in HBase is stored in a structured manner, while having various mechanisms that simplify random reads and writes from rows and columns. Namely, HBase tables are divided into potentially many regions, while one or more regions are serviced by a region server. The tables can be horizontally and vertically segmented while they are physically stored in HBase. Because many machine learning applications access the data by rows, in this paper we will continue to discuss only horizontal segmentation. As HBase was designed with very large tables in mind, a common use case is the following. A table at creation has only one region, which is serviced by one region server (a physical node in the Hadoop cluster). When this table is loaded with data it gets bigger and at some point it will become too big, so HBase will split its region into two regions. Then the new region will be assigned to the same region server or can be moved to another region server. The default splitting threshold is 10 GB. There are numerous reasons why HBase was designed that way, and we will not go into details about that. From parallelization perspective, this can pose a challenge, because for the automatic splits there are no guarantees that every region will contain equal amount of data, when are the splits going to occur exactly, are the regions going to be served by different region servers (nodes) etc. Further more, if one is using Hadoop for research purposes only then the dataset may not be that large, thus never overcoming the threshold for splitting. To overcome this challenge we can pre-split the tables on creation. This in turn means that the table can be configured at creation time to be stored on as many-regions as needed. Usually the number of regions is a multiple of the number of HBase region servers. The logic for having more region servers than acutal nodes is because the nodes are multi-core machines, so different threads on the same node can service different regions.

Before loading the dataset in HBase, we need to define the table structure and create it. Column names and data types are provided when storing data in each row, so at creation time we need to only specify a table name and a column family. There are some advanced configuration features that can be specified, but they are not topic of this discussion. Be that as it may, there is one very important decision that we need to make before loading data in the table. Because HBase tables, unlike SQL tables, cannot have secondary indexes, the primary key (row key) needs to be designed according to the usage patterns of the table. There are many considerations when designing the row key and they are very important for production use of HBase tables. However, for scientific use and for parallelizing machine learning algorithms, we need a simple design that allows uniform data distribution across nodes. In most scientific datasets the data instances (i.e. rows) do not have ids for their instances, or if they do they are not used for the actual machine learning. Nevertheless, in order to store a row in a HBase table, it needs a row key. For flat flies like CSV or ARFF the row key can be the line number of the instance. However, sequential row keys are very bad choice for HBase tables because the inserts will always be on the last region, therefore having no parallelism during the load, a problem called Region Server hotspotting. There are multiple ways of overcoming this problem, and one of them is a technique called salting [38]. With this technique each sequential id is salted with a prefix. The prefix is usually the modulo number of the original sequential id and the number of regions. Even though, this is very important topic, the step of loading the datasets in HBase is not the focus point of this paper.

Once the dataset files are loaded into HDFS we need to transform them if needed and store them in HBase. If we have totally M rows in the dataset, and R regions, then we would like to distribute the data uniformly so each region gets M/R rows. This in turn means that we need to specify $R-1$ split points when creating the table. If we use sequential ids for the row key (like the line number in the file), than these split

points would be: $M/R, 2M/R, 3M/R, ..., (R-1)M/R$. If we use a more sophisticated row key design, then the split points should reflect that design. For instance, if we take the modulo number of the id and the number of regions, then each region would get almost the same number of rows. This design of the row key allows fast random reads and writes, and additionally it facilitates addition of new data to the table at a later time without needing to redesign the table for equally dispersed load across regions. The following example shows how a table can be pre-splitted on creation. The row key design is described with the function in listing 1. It returns a tuple in which the first element is the padded modulo number and the second part is the padded sequential id. The numbers are padded with zeros so that they are lexicographically sorted.

```
1 (pad(seq_id % num_regions), pad(seq_id))
```

Listing 1. Row key design

Once the HBase table that should contain the dataset is created with appropriate split points for even data distribution across the cluster, we can start loading the data. One can write pure MapReduce jobs in Java or Python. If we choose that path, then we need to write a separate map and reduce function for each task. However, by using the scripting language Pig Latin [42, 43] we can write scripts from a higher-level perspective. These Pig scripts generate MapReduce tasks in the background so the programming effort is simplified and the development time is greatly reduced. The downside of using Pig is that when Pig scripts are compiled into MapReduce jobs, there is some overhead. Additionally one may write a more optimal implementation of map and reduce functions manually than the ones generated by the Pig compiler. Nevertheless, these are corner cases and for longer running MapReduce tasks the overhead is insignificant in the range of up to couple of minutes in the worst case. When loading the data usually only a map phase is required. It reads the data from the HDFS files and stores it in HBase tables. In most cases when loading the data there is no grouping of keys, so a reduce phase is not needed. During this step we can add various methods for data preprocessing like discretization, transformation and other methods that rely only on the value in one row of the dataset.

D. Processing HBase tables

After the dataset is loaded in a HBase table we can continue implementing machine learning algorithms. In general, this phase can be comprised of several substeps or iterations of data processing, depending on the nature of the algorithm that is being implemented. For each of the intermediate steps that we need to store some data we need a HBase table that will also be pre-split at creation time, similar to what we described in the previous subsection IV-C. What happens in the background when a particular HBase table is being processed, is very peculiar. Pig will determine the number of regions of the table and it will start that number of map tasks. Then each map task will process the data of a particular table region on the node where the data resides, therefore leveraging data locality. Note that, in order to benefit from the principle of data locality, each node in the cluster should run HDFS, HBase and YARN (MapReduce) services. The number of reduce tasks is by default one, but this can be also manually specified and

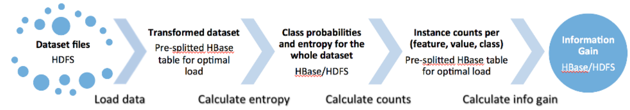

Fig. 2. Data flow during the parallel computation of information gain

does not depend of the table structures. If we specify more than one reduce tasks, then this will solicit a merge phase which will combine the intermediate results from each reduce task. Usually, for smaller datasets specifying more than one reduce task does not improve performance, on the contrary, it can degrade performance. However, being able to specify the number of reduce tasks provides flexibility that can improve the performance for larger datasets or some specific problems.

During this phase, depending on the type of algorithm that is being parallelized, many tables can be used by multiple MapReduce jobs. In the following section V we illustrate this by when we parallelize the computation of information gain.

V. Parallel computation of information gain

In this section we illustrate how we can use the framework proposed in section IV for computing the information gain of a dataset. Fig. 2 shows the data flow during the steps of the framework. The first steps of loading the data to HDFS and subsequently into HBase tables corresponds to what we explained in subsections IV-B and IV-C. Then, after the dataset is loaded in a HBase table, the processing takes place in three phases explained in the following subsections and they correspond to the specific properties of the current algorithm. This is an illustration how the framework step described in subsection IV-D can contain multiple phases consisted of many different MapReduce tasks that read and store data from several HBase tables.

A. Calculating entropy of a dataset

As explained in section III, the definition of information gain requires calculation of the entropy of the whole dataset. In order to calculate it, first we need to count the number of instances per class, and afterwards to sum the class probabilities. Notably this is a very simple step and does not require parallelization because its complexity is $O(N)$, where N is the number of instances in the dataset. Moreover, if we are interested in only sorting the features without having the actual information gain for each of them, then we can eliminate the entropy from equation (3). Be that as it may, for other applications we need the actual information gain. If we decide to parallelize this step, despite of its simplicity, then we need two MapReduce jobs. The Pig Latin script shown in listing 2 performs this. Each parameter starting with *\$* can be passed to Pig script when it is started. In line 2 we specify that we only want to read the cell where the class is stored, denoted as $\$label$. The first MapReduce job that calculates the counts per class and the class probabilities corresponds to the code from line 2 to line 16. Then from line 17 till the end of the script the second MapReduce job calculates the entropy of the dataset. Notwithstanding, the peculiar thing that is demonstrated here is how easy MapReduce jobs can be combined into a flow. In like manner, one can combine many MapReduce jobs in one flow without any need of manual synchronization between them.

```
register '$udf_path' as paddingUDFs;
pfdata_tmp = LOAD '$table_dataset' USING
    org.apache.pig.backend.hadoop.
hbase.HBaseStorage('r:$label',
       '-loadKey=true'
) AS (rowkey:tuple(prefix_padded:chararray,
    id_padded:chararray),
  class:int);
pfdata_class_group = GROUP pfdata_tmp BY
    class;
pfdata_class = FOREACH pfdata_class_group
    GENERATE
  flatten(group) as class,
  COUNT(pfdata_tmp.class) as count;
pfdata_class_prob = FOREACH pfdata_class
    GENERATE
  class,
  count,
  (count/$num_instances) as prob:double,
  ((-count/$num_instances)*
  UDFs.log2(count/$num_instances)) as
      entropy:double;
total_entropy_group = GROUP pfdata_class_prob
    ALL;
total_entropy = FOREACH total_entropy_group
    GENERATE
  SUM(pfdata_class_prob.entropy) as
      entropy:double;
STORE total_entropy INTO
    '$hdfs_export_entropy' USING
    PigStorage('\t');
```

Listing 2. Pig script for calculating entropy of a dataset

B. Counting instances per feature index, feature value and class

After the entropy is calculated, the definition of information gain, as presented with (3), requires counts of instances per feature index, feature value and class. This step is the most computationally expensive step in the algorithm. The source code of this step is shown in listing 3 and it is based on the pseudo code we have reported in [32]. Parameters that are passed to the script are the table names, number of features, index of the class value, number of padding digits, etc. First we need to load the dataset from a HBase table (lines 3 through 6). A row of the dataset is represented by the row key, and the dictionary r in which keys are the column names and values are the actual values. This representation allows us to store only the non-zero values of a dataset. Then we need to expand each row of the dataset (denoted as dictionary r) to tuples: *(feature index, feature value, class, 1)*. This is performed in lines 7 through 8 with the user-defined function *decode_sparse_row*. If the dataset has M rows (instances) and N columns (features), then from each row we will generate N tuples because now also the zero-valued cells are also included. To summarize, when the whole dataset is processed $M \times N$ tuples will be generated. These tuples are afterwards grouped by the key *(feature index, feature value, class)* in lines 9 through 12, and finally the count is stored in another table (lines 13 through 15). All of the code in listing 3 is compiled in one MapReduce job. The number of generated map tasks will be equal to the number of regions of the input table (denoted by *$table_dataset* in the script), and the number of reduce tasks is set by the parameter *$parallel*.

```
register '$udf_path' using jython as UDFs;
set default_parallel $parallel;
pfdata_tmp = LOAD '$table_dataset' USING
    org.apache.pig.backend.hadoop.hbase.
  HBaseStorage('r:*', '-loadKey=true') AS
     (rowkey:tuple(prefix_padded:chararray,
        id_padded:chararray),
     r:map[]);
pfdata_short = FOREACH pfdata_tmp GENERATE
  FLATTEN(UDFs.decode_sparse_row(r,
      $num_features,
  $num_features_digits, '$feature_data_type',
      '$label'));
feature_value_class_counts_group = GROUP
    pfdata_short BY (feature_index,
    feature_value, class);
feature_value_class_counts = FOREACH
    feature_value_class_counts_group GENERATE
  group as rowkey,
  SUM(pfdata_short.instance_count) as
      instance_count;
STORE feature_value_class_counts INTO
    '$table_feature_index_tmp' USING
     org.apache.pig.backend.hadoop.hbase.
     HBaseStorage('r:instance_count');
```

Listing 3. Counting number of instances per feature index, feature value and class with Pig Latin

C. Calculating the information gain

Having the counts calculated in the previous step V-B, this step only calculates the probabilities and entropies in (3) and stores this result in HBase or HDFS. Nevertheless, it is usually the second longest running step from this list. The code for this is shown in listing 4. First in the lines 3 through 8 it reads the tuples *(feature index, feature value, class, instance_count)* which were calculated in the previous step, and now are stored in the table *$table_feature_index_tmp*. This table was properly pre-splitted on creation so the Pig script will be compiled in one MapReduce job with multiple map tasks. In particular, if the number of features is N and the desired number of regions of the table is R, then we specify $R-1$ split points, and in that way each region will contain the tuples for N/R features. We acknowledge that this might not be ideal distribution because some features might have significantly more distict values then others, but nevertheless, it provides decent parallelism. Given that this step is not as computationally intensive as the previous, we did not consider it necessary to further optimize this table. Then when the MapReduce job is compiled, it will have R map tasks and each of them will work with the data of the appropriate table region.

```
register '$udf_path' using jython as UDFs;
set default_parallel $parallel;
feature_value_class_counts_tmp = LOAD
    '$table_feature_index_tmp' USING
    org.apache.pig.backend.hadoop.hbase.
HBaseStorage('r:instanceCount',
      '-loadKey=true'
) AS (
  id:tuple(feature_index:chararray,
      feature_value:int, class:int),
  instanceCount:double);
```

```
10 feature_value_class_counts = FOREACH
       feature_value_class_counts_tmp GENERATE
11 flatten(id) as (feature_index, feature_value,
       class),
12 instanceCount;
13 feature_index_group = GROUP
       feature_value_class_counts BY
14      (feature_index);
15 feature_index_info_gain = FOREACH
       feature_index_group GENERATE
16    flatten(group) as feature_index_padded,
17    flatten(UDFs.
18    calc_feature_info_gain(($entropy), group,
         feature_value_class_counts,
         ($num_instances))) as info_gain:double;
19 STORE feature_index_info_gain INTO
       '$table_feature_index_info_gain' USING
       org.apache.pig.backend.hadoop.
20 hbase.HBaseStorage('r:ig');
```

Listing 4. Calculating information gain with Pig Latin

The most peculiar part in the script in listing 4 is at line 13. Here all tuples are grouped by feature index. When the Pig Script is translated into a MapReduce job, the during the map phase the feature index is emitted as a key, and during the reduce phase all tuples that are for the same key (in this case the feature index) are grouped together on the same node. The Python UDF *calc_feature_info_gain* utilizes this because for each feature it has the count of instances of all its values per class. Having that it is easy to compute the information gain by (3). Finally, the results can be stored in a HDFS file or in a HBase table. In this script we store them in the HBase table $\$table_feature_index_info_gain$, performed in the last line.

VI. Experiments

With intention to monitor various aspects of the parallel implementation, a relatively large dataset was essential. Furthermore, we did not want to focus on significant preprocessing like discretization or transformation of values so we can easily compare our results with other research. The FedCSIS AAIA'14 data mining competition dataset [44] has exactly those properties. It is a sparse matrix with 50000 instances and 11852 numeric features, most of which are have the value 0 or 1. There are about 0.9% non-zero values in it. It represents a multi-label problem that has 3 binary labels, that can be merged with the powerset technique as used in [45] into one one-label multi-class problem that has 8 (2^3) possible classes.

We have tested the same dataset on three completely different Hadoop clusters. Each of them was running the same version of Apache Hadoop 2.3.0 (integrated in Cloudera CDH 5.3.0). This is an extension to what we did in [32], where we analyzed the speedup only on one on-premises cluster. Additionally in this paper we analyze the effect of the number of reduce tasks, while in [32] we have used only one reduce task. Finally, the most important difference is that we have evaluated the scalability of the approach by replicating the dataset 80 times. We have performed this by replicating the dataset horizontally so from each instance there are 80 exact copies. This in turn results in a dataset that has 4 million instances and almost 12 thousand features. It should be noted that the computational complexity of the algorithm depends only on the dataset size and not on its sparsity or feature types. Keeping in mind that our goal is to evaluate the execution time and speedup based on the cluster size, the expansion of the dataset serves this purpose.

The first cluster (denoted by *Amazon32* in the remaining of the paper) was deployed on Amazon AWS. It contained a total of 32 nodes, each of them a m1.xlarge instance with 15GB RAM and 8 compute units (4 cores with 2 compute units each). From the 32 nodes, 8 were hosting HBase Region Servers and HDFS Data Nodes, 3 were specifically dedicated to HDFS Data Nodes and 19 were running only YARN. We acknowledge that this configuration may not be optimal for the current task, but we were given access to this cluster without the ability to modify its configuration. Therefore we have decided to run tests using up to 8 nodes at a time, because when using more it would be difficult to estimate the speedup.

The second cluster (denoted by *FCSE24* in the remaining of the paper) was deployed on-premises at the Faculty of Computer Science and Engineering (FCSE) at the Ss.Cyril and Methodius University, Skopje, Macedonia. It had a total of 24 nodes, each of them an Intel Xeon Processor E5640 with 12M Cache, 2.66 GHz, 24 GB RAM, 4 cores and 8 threads. From them 21 were configured to run the following services: HBase Region Servers, HDFS DataNodes and YARN MapReduce NodeManagers. The remaining nodes were used for other Hadoop and Cloudera management services.

The third cluster (denoted by *FCSE65* in the remaining of the paper) was also deployed on-premises and it was an extended version of the second, containing a total of 65 nodes, of which 54 were running the following services: HBase Region Servers, HDFS DataNodes and YARN MapReduce NodeManagers. A variant of this cluster with 59 instead of 54 active nodes was also used for the experiments presented in [32].

During our tests none of these clusters was executing other tasks. On all of them we ran tests with different table structures in order to simulate clusters with smaller sizes. By pre-splitting the HBase tables to a specific number of regions we were able to force Pig Latin to start the desired number of map tasks for each job. For all these configurations we are computing the speedup of the parallelization against a cluster with one node. We are simulating the one-node cluster by configuring the tables to have only one region, thus all MapReduce jobs that read from those tables have only one map task. We have tested using different number of reduce task by setting a configuration property in the Pig scripts. The remaining of this section is divided in two, VI-A containing summary information for all steps that are fast and did not benefit significantly from the parallelization, and VI-B containing detailed information about the step described in V-B, which was the most computationally expensive. Table I shows the information gain of the top 50 features which can be used for verification of the correctness of our implementation. In the following subsections we describe the results from our experiments.

A. Computationally cheap steps

The dataset was stored in two files: one containing the data in EAV (entity attribute value) format, and one containing the labels. The EAV format greatly reduces the file sizes to 72 MB compared to 1.1 GB when stored in full format as CSVs.

TABLE I. TOP 50 FEATURES ORDERED BY INFORMATION GAIN

Rank	Feature	InfoGain	Rank	Feature	InfoGain
1	11701	0.07422	26	7407	0.0256033
2	143	0.07000	27	11825	0.0249701
3	11832	0.06009	28	4505	0.0249698
4	1509	0.05154	29	11100	0.0249225
5	5909	0.04936	30	10331	0.0247915
6	8635	0.04539	31	7529	0.0247519
7	2182	0.04012	32	2274	0.0247061
8	865	0.03817	33	10261	0.0246147
9	6523	0.03817	34	7592	0.0245778
10	5827	0.03795	35	4319	0.0245677
11	5188	0.03467	36	1349	0.0245448
12	5513	0.03296	37	7405	0.0245288
13	6162	0.03294	38	11463	0.0245111
14	5967	0.03271	39	11000	0.0244753
15	2835	0.03223	40	6779	0.0240003
16	139	0.0318404	41	10428	0.0236240
17	9306	0.0318030	42	460	0.0235250
18	1772	0.0296594	43	7291	0.0233440
19	3257	0.0283169	44	8853	0.0232071
20	9848	0.0283169	45	2883	0.0232064
21	675	0.0282140	46	5925	0.0231852
22	73	0.0273487	47	8114	0.0225087
23	7275	0.0266788	48	5330	0.0223354
24	7419	0.0266100	49	1156	0.0219374
25	1244	0.0262854	50	2701	0.0218273

The effect is that copying them to HDFS is very fast (about a second). The step described in subsection IV-C was actually two MapReduce jobs. The first is for loading the labels which took 58 to 70 seconds, and the second for loading the data which took 130 to 145 seconds on the on-premises and 175 to 195 seconds on the Amazon cluster. Calculating the entropy of the dataset, described in section V-A, took 118 to 152 seconds on both clusters. The step described in subsection V-B is analyzed in more detail in the following subsection VI-B. After it completed and stored the results in a pre-splitted table, calculating the information gain of each feature, described in subsection V-C, took 69 to 97 seconds on both clusters. The final step, the export of the list of information gain of all features, took 46 to 70 seconds. All of the MapReduce tasks had an overhead of up to 60 seconds for compilation of the Pig script, generating JAR files, distributing them on the cluster and negotiating resources.

When preparing the 80 times replicated dataset we stored it in a slightly different format so we can later process the data and the labels at the same time. Namely, each line of the enlarged file contains pairs of the column indexes and values of all non-zero features. This representation takes 3 GB, whereas if we stored it in pure EAV format we would need about 5.5 GB (80 $\times$ 72) owing to the redundancy of line numbers. This does not have effect of any of the other steps except of how is it stored in HDFS. This file when copied on HDFS was automatically fragmented on 24 nodes (not counting the nodes for replication). On Fig. 3 is shown the data load time depending on the cluster configuration. It should be noted that even though there are 54 active nodes in the cluster in some cases we have intentionally created tables with more table regions (108, 162 and 216) aiming to leverage the multiple cores on each node.

Important to realize is that the 24 HDFS nodes on which the file is dispersed is an upper bond to the maximum number of map tasks when processing the file from HDFS and storing it in HBase tables. As a result, even though some tables have more than 24 regions during this phase it does not have an effect of the parallelism. Nevertheless, in the next steps when the data source is an HBase table, its number of regions dictates the number of generated map tasks. Another important thing to notice is that when using less nodes than 24 for the HBase tables, the number of map tasks is still 24 because this is dictated by the data source (HDFS file) and not by the destination (HBase table). From Fig. 3 it is evident that the load time is not reduced when more than 24 nodes HBase table regions are used. Also we see that when using less then 24 table regions the bottle neck is during the writes to the HBase tables. Finally, we want to emphasize the HBase table with only one region (the right-most case on Fig. 3). Even tough it was configured to have only one region by not specifying any split points for it at creation time, during the load it got larger than some configurable threshold, so HBase automatically splitted in two regions. Nevertheless, those two regions are on the same node.

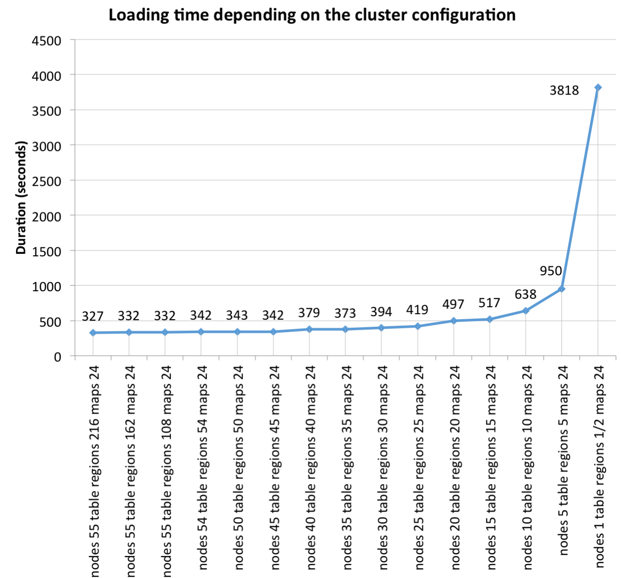

Fig. 3. Data load time for the 80 times replicated AAIA'14 dataset depending on cluster configuration

B. *Computationally expensive step - Calculating counts*

The step described in subsection V-B was the most complicated and the speedup for it varied significantly depending on the cluster size and configuration. The remaining of this subsection describes details of the impact of the parallelization of this step and all listed speedups and durations are only for it.

First, we conducted experiments using the original AAIA'14 dataset on the FSCSE65 cluster. These results were published in [32], so here we are only reviewing them. These experiments were using only one reduce task, the default in Pig Latin. Also here we used more map tasks than actual nodes because each node is a multi-core machine. The results confirmed that indeed using more map tasks is beneficial, which is intuitively logical. Nevertheless, when we further increase the number of map tasks, the performance gradually degrades. The explanation for this is that as the number of map tasks gets larger, the operating system on the nodes needs to spend more time on task switching, swapping, while

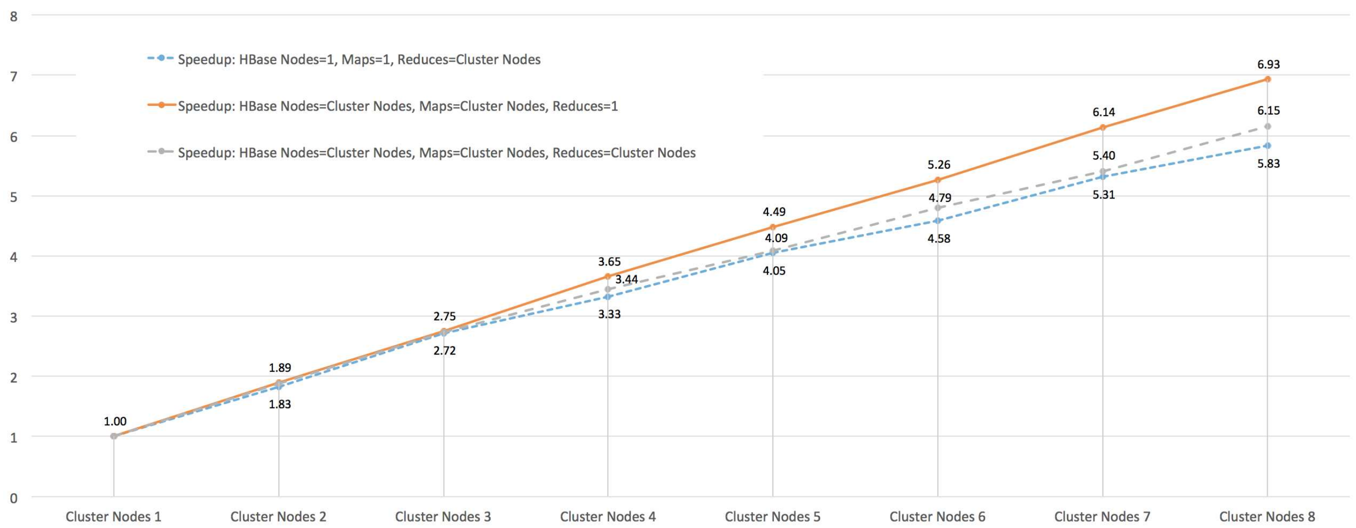

Fig. 4. Speedup depending on the number of active nodes, map and reduce tasks on the Amazon32 cluster

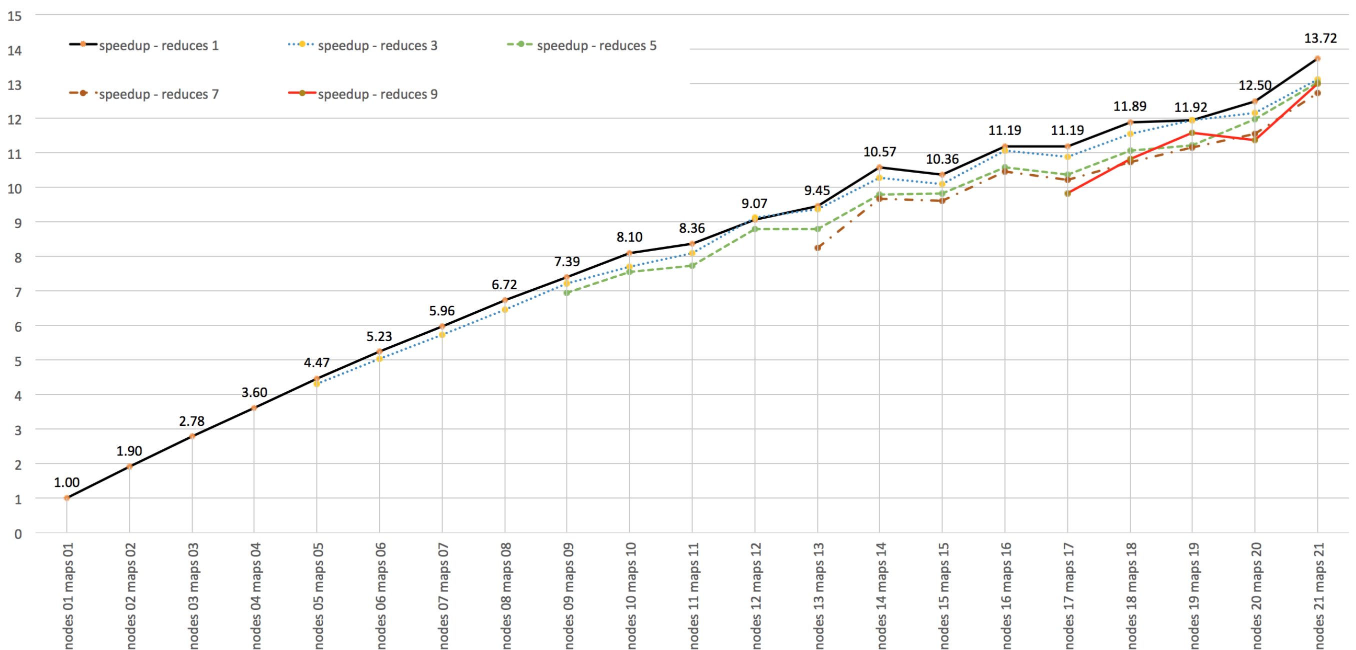

Fig. 5. Speedup depending on the number of active nodes, map and reduce tasks on the FCSE24 cluster

also needing to run many Hadoop and other services in the background. The total duration of this step on the one-node cluster was 3656 seconds, while the quickest solution obtained when using 59 nodes and 177 map tasks took 129 seconds on this cluster and the corresponding speedup was 28.34.

Then we continued our experiments on the Amazon32 cluster, trying to determine the impact of the number of nodes, maps and reduces. We have tried three options when trying to utilize the nodes of the cluster: use as much as possible nodes to run map tasks and have only one reduce task; use as much as possible nodes to run both map and reduce tasks; and use only one node for one map task and use all available nodes for reduce tasks. The speedup compared to the one-node cluster depending on the available nodes using these three options are shown on Fig. 4. It indicates that for this dataset it is best to have only one reduce phase, but use as many nodes as possible for the map tasks. This, in fact, makes sense because the work is performed during the map phase and during the reduce phase these results are only grouped together. Having more than the default of one reduce task actually increases the duration because the partial results in each reduce task need to be merged together. The total duration of this step on the one-node cluster was 4732 seconds, while the quickest solution with speedup of 6.83 took 693 seconds.

Aiming to confirm these findings we continued testing on the FCSE24 cluster, using the same approach. Additionally

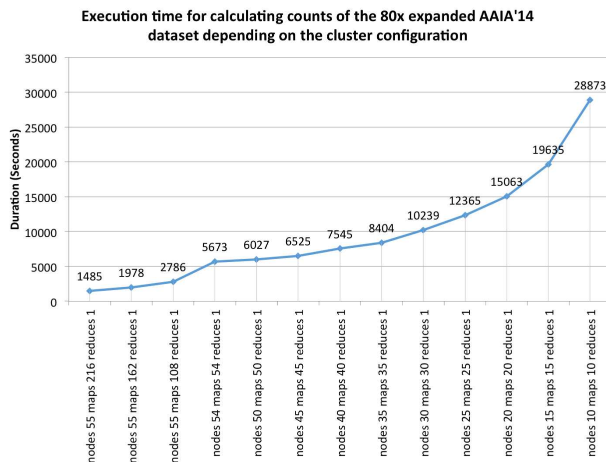

Fig. 6. Execution time for calculating counts of the 80x replicated AAIA'14 dataset depending on the cluster configuration

we have tried using 1,3,5,7 or 9 reduce tasks, depending on the number of available nodes. Our intent was to confirm that using only one reduce task (the default value in Pig Latin) will be more appropriate for a dataset of this size. The charts shown on Fig. 5 indeed confirm this assumption. The greatest speedup was always achieved when using only one reduce task, regardless of the number of available nodes. The total duration of this step on the one-node cluster was 3637 seconds, while the quickest solution with speedup of 13.72 took 265 seconds.

Finally, we have analyzed the execution time on the FCSE65 cluster using the 80 times replicated dataset. We have started experimenting using 54 nodes and gradually reducing the number of nodes by 5. When using 54 nodes we have also tried used 2, 3 and 4 times more table regions than actual nodes. Fig. 6 shows the execution times depending on the various configurations. In all cases the number of reduces was 1. Owing to the fact that this dataset is quite large, executing this step on smaller clusters took a significant amount of time. Additionally because HBase splitted the table on the one node cluster to two regions, using that execution time for calculating speedup would have been inconsistent with the previous setups. Therefore, for this experiment on Fig. 6 we are reporting the execution time and not the speedup. By performing this experiment we have confirmed that the proposed parallel implementation is scalable to large datasets for which the processing with a sequential implementation would be quite difficult if not impossible.

VII. Conclusion and future work

In this paper we have reviewed the applications of the metric information gain for ranking individual features, discretization of continuous valued features, improving decision tree performance, localization, rough sets, etc. In a Big Data setting those tasks become a significant challenge, and therefore the need for its parallelization. In this paper we have proposed a parallel implementation of it. In oder to facilitate this, we have proposed a generic framework for data parallelization and then all steps from the algorithm for computation of information gain were parallelized using it. The benefits from using the scripting language Pig Latin were evident by the code listings which allowed fast development of MapReduce jobs. We have also demonstrated how can we manually set the degree of parallelism by pre-splitting the HBase tables so they have optimal number of regions and even data distribution across regions. The experiments confirmed that for this type of algorithm it is best to use only one reduce task. We have also validated that the multi-core nodes are providing increased performance when they execute more map tasks simultaneously. By deploying the implementation on Amazon AWS and on-premises clusters we have demonstrated the portability of the approach. The correctness of the implementation was verified by comparing the ranked features with the results we obtained from WEKA. Not to neglect were also the findings related to the scalability of the approach to an even larger dataset with millions of instances and dozens of thousands of features.

In our future work we plan to utilize the proposed implementation for other task In that manner, we also need to propose valid data transformation and normalization techniques, so we can generalize the approach and make it available for datasets that contain non-discretized continuous or nominal features. Additionally, we aim to apply the current parallelization for building decision trees. Finally, we plan to parallelize other more advanced feature selection algorithms using a similar framework.

Acknowledgment

This work was partially financed by the Faculty of Computer Science and Engineering at the Ss.Cyril and Methodius University, Skopje, Macedonia.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 15–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267308.1267323

[3] "Hadoop wiki: List of institutions that are using hadoop for educational or production uses, howpublished = https://wiki.apache.org/hadoop/poweredby, note = Accessed: 2015-01-29."

[4] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN 1-55860-238-0

[5] T. M. Mitchell, *Machine Learning*, 1st ed. McGraw-Hill Science/Engineering/Math, 3 1997. ISBN 9780070428072. [Online]. Available: http://amazon.com/o/ASIN/0070428077/

[6] D. Mladenic and M. Grobelnik, "Feature selection for unbalanced class distribution and naive bayes," in

Proceedings of the Sixteenth International Conference on Machine Learning, ser. ICML '99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. ISBN 1-55860-612-2 pp. 258–267. [Online]. Available: http://dl.acm.org/citation.cfm?id=645528.657649

[7] R. O. Duda, *Pattern classification*, 2nd ed. New York: Wiley, 2001. ISBN 0471056693

[8] H. Almuallim and T. G. Dietterich, "Learning with many irrelevant features," in *Proceedings of the Ninth National Conference on Artificial Intelligence - Volume 2*, ser. AAAI'91. AAAI Press, 1991. ISBN 0-262-51059-6 pp. 547–552. [Online]. Available: http://dl.acm.org/citation.cfm?id=1865756.1865761

[9] A. L. Blum and P. Langley, "Selection of relevant features and examples in machine learning," *Artificial Intelligence*, vol. 97, no. 1–2, pp. 245 – 271, 1997. doi: http://dx.doi.org/10.1016/S0004-3702(97)00063-5 Relevance. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370297000635

[10] P. Langley, *Elements of machine learning*. San Francisco, Calif: Morgan Kaufmann, 1996. ISBN 1558603018

[11] G. H. John, R. Kohavi, and K. Pfleger, "Irrelevant features and the subset selection problem," in *Machine Learning: Proceedings of the Eleventh International Conference*. Morgan Kaufmann, 1994, pp. 121–129.

[12] B. Raman and T. R. Ioerger, "Instance based filter for feature selection," *Journal of Machine Learning Research*, vol. 1, no. 3, pp. 1–23, 2002.

[13] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *J. Mach. Learn. Res.*, vol. 3, pp. 1157–1182, Mar. 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=944919.944968

[14] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of massive datasets / Jure Leskovec, Anand Rajaraman, Jeffrey David Ullman, Standford University*, 2nd ed. Cambridge: Cambridge University Press, 2014. ISBN 9781107077232 1107077230 1316147312 9781316147313

[15] C. Dobre and F. Xhafa, "Parallel programming paradigms and frameworks in big data era," *International Journal of Parallel Programming*, vol. 42, no. 5, pp. 710–738, 2014. doi: 10.1007/s10766-013-0272-7. [Online]. Available: http://dx.doi.org/10.1007/s10766-013-0272-7

[16] S. Singh, J. Kubica, S. Larsen, and D. Sorokina, "Parallel large scale feature selection for logistic regression." in *SDM*. SIAM, 2009, pp. 1172–1183.

[17] Z. Sun and Z. Li, "Data intensive parallel feature selection method study," in *Neural Networks (IJCNN), 2014 International Joint Conference on*, July 2014. doi: 10.1109/IJCNN.2014.6889409 pp. 2256–2262.

[18] L. Zhou, H. Wang, and W. Wang, "Parallel implementation of classification algorithms based on cloud computing environment," *TELKOMNIKA Indonesian Journal of Electrical Engineering*, vol. 10, no. 5, pp. 1087–1092, 2012.

[19] G. Caruana, M. Li, and M. Qi, "A mapreduce based parallel svm for large scale spam filtering," in *Fuzzy Systems and Knowledge Discovery (FSKD), 2011 Eighth International Conference on*, vol. 4, July 2011. doi: 10.1109/FSKD.2011.6020074 pp. 2659–2662.

[20] I. Triguero, D. Peralta, J. Bacardit, S. García, and F. Herrera, "Mrpr: A mapreduce solution for prototype reduction in big data classification," *Neurocomputing*, vol. 150, pp. 331–345, 2015.

[21] A. K. Farahat, A. Elgohary, A. Ghodsi, and M. S. Kamel, "Distributed column subset selection on mapreduce," in *Data Mining (ICDM), 2013 IEEE 13th International Conference on*. IEEE, 2013, pp. 171–180.

[22] A. Guillén, A. Sorjamaa, Y. Miche, A. Lendasse, and I. Rojas, "Efficient parallel feature selection for steganography problems," in *Bio-Inspired Systems: Computational and Ambient Intelligence*, ser. Lecture Notes in Computer Science, J. Cabestany, F. Sandoval, A. Prieto, and J. Corchado, Eds. Springer Berlin Heidelberg, 2009, vol. 5517, pp. 1224–1231. ISBN 978-3-642-02477-1. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02478-8_153

[23] S. Owen, R. Anil, T. Dunning, and E. Friedman, *Mahout in Action*. Greenwich, CT, USA: Manning Publications Co., 2011. ISBN 1935182684, 9781935182689

[24] T. M. Cover and J. A. Thomas, *Elements of information theory*, 2nd ed. Hoboken, NJ: Wiley, 2006. ISBN 9780471241959 0471241954 9780471241959

[25] C. Shang, M. Li, S. Feng, Q. Jiang, and J. Fan, "Feature selection via maximizing global information gain for text classification," *Knowledge-Based Systems*, vol. 54, no. 0, pp. 298 – 309, 2013. doi: http://dx.doi.org/10.1016/j.knosys.2013.09.019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950705113003067

[26] C. Lee and G. G. Lee, "Information gain and divergence-based feature selection for machine learning-based text categorization," *Inf. Process. Manage.*, vol. 42, no. 1, pp. 155–165, Jan. 2006. doi: 10.1016/j.ipm.2004.08.006. [Online]. Available: http://dx.doi.org/10.1016/j.ipm.2004.08.006

[27] U. M. Fayyad and K. B. Irani, "Multi-interval discretization of continuous-valued attributes for classification learning," in *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993*, 1993, pp. 1022–1029.

[28] J. Dougherty, R. Kohavi, M. Sahami *et al.*, "Supervised and unsupervised discretization of continuous features," in *Machine learning: proceedings of the twelfth international conference*, vol. 12, 1995, pp. 194–202.

[29] U. M. Fayyad and K. B. Irani, "On the handling of continuous-valued attributes in decision tree generation," *Machine Learning*, vol. 8, pp. 87–102, 1992. doi: 10.1007/BF00994007. [Online]. Available: http://dx.doi.org/10.1007/BF00994007

[30] C. Stachniss, G. Grisetti, and W. Burgard, "Information gain-based exploration using rao-blackwellized particle filters," in *Robotics: Science and Systems*, vol. 2, 2005, pp. 65–72.

[31] D. Ślezak, "Approximate entropy reducts," *Fundam. Inf.*, vol. 53, no. 3-4, pp. 365–390, Aug. 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=2371245.2371255

[32] E. Zdravevski, P. Lameski, A. Kulakov, B. Jakimovski, S. Filiposka, and D. Trajanov, "Feature ranking based on information gain for large classification problems with mapreduce," in *Proceedings of the 9th IEEE International Conference on Big Data Science and Engineering*. IEEE Computer Society Conference Publishing, August 2015,

in print August 2015.
[33] D. Miner, *MapReduce design patterns.* Sebastopol, CA: O'Reilly, 2013. ISBN 9781449327170
[34] A. Holmes, *Hadoop in practice.* Shelter Island, NY: Manning, 2012. ISBN 9781617290237 1617290238
[35] T. White, *Hadoop: the definitive guide,* 3rd ed. Beijing: O'Reilly, 2012. ISBN 9781449311520
[36] "Apache hadoop nextgen mapreduce (yarn)," http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, accessed: 2015-01-29.
[37] "Hdfs architecture guide," http://hadoop.apache.org/docs/r1.2.1/hdfs design.html, accessed: 2015-01-29.
[38] L. George, *HBase the definitive guide.* Sebastopol, CA: O'Reilly, 2011. ISBN 97814493157711449315771. [Online]. Available: http://public.eblib.com/choice/publicfullrecord.aspx?p=769368
[39] Y. Jiang, *HBase administration cookbook master HBase configuration and administration for optimum database performance.* Birmingham: Packt Publishing, 2012. ISBN 9781849517157 18495171501849 517142 9781849517140. [Online]. Available: http://site.ebrary. com/id/10598980
[40] N. Dimiduk and A. Khurana, *HBase in action.* Shelter Island, NY: Manning, 2013. ISBN 16172905219781617290527
[41] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing,* ser. PODC '00. New York, NY, USA: ACM, 2000. doi: 10.1145/343477.343502. ISBN 1-58113-183-6 pp. 7–. [Online]. Available: http://doi.acm.org/10.1145/343477.343502
[42] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a high-level dataflow system on top of map-reduce: The pig experience," *Proc. VLDB Endow.,* vol. 2, no. 2, pp. 1414–1425, Aug. 2009. doi: 10.14778/1687553.1687568. [Online]. Available: http://dx.doi.org/10.14778/1687553.1687568
[43] A. Gates, *Programming Pig.* Sebastopol: O'Reilly Media, 2011. ISBN 9781449317690 1449317693 9781449317683 1449317685. [Online]. Available: http://public.eblib.com/choice/publicfullrecord.aspx?p=801461
[44] A. Janusz, A. Krasuski, S. Stawicki, M. Rosiak, D. Slezak, and H. S. Nguyen, "Key risk factors for polish state fire service: A data mining competition at knowledge pit," in *Computer Science and Information Systems (FedCSIS),* 2014 Federated Conference on, Sept 2014. doi: 10.15439/2014F507 pp. 345–354.
[45] E. Zdravevski, P. Lameski, A. Kulakov, and D. Gjorgjevikj, "Feature selection and allocation to diverse subsets for multi-label learning problems with large datasets," in *Computer Science and Information Systems (FedCSIS),* 2014 Federated Conference on, Sept 2014. doi: 10.15439/2014F500 pp. 387–394.
[46] A. H. Team, "Apache HBase reference guide," http://hbase.apache.org/book.html, accessed: 2015-03-29.