An Information Theory-Based Feature Selection Framework for Big Data under Apache Spark

Sergio Ramírez-Gallego*, Héctor Mouriño-Talín†, David Martínez-Rego†,
Verónica Bolón-Canedo†, José Manuel Benítez*, Amparo Alonso-Betanzos† and Francisco Herrera*‡
*Department of Computer Science and Artificial Intelligence, CITIC-UGR, University of Granada, 18071 Granada, Spain. Emails: sramirez@decsai.ugr.es, j.m.benitez@decsai.ugr.es, herrera@decsai.ugr.es
†Department of Computer Science, University of A Coruña, 15071 A Coruña, Spain.
Emails: h.mtalin@udc.es, dmartinez@udc.es, veronica.bolon@udc.es, ciamparo@udc.es
‡Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah, Saudi Arabia.

Abstract

With the advent of extremely high-dimensional datasets, dimensionality reduction techniques are becoming mandatory. Of the many techniques available, feature selection is of growing interest for its ability to identify both relevant features and frequently repeated instances in huge datasets. We aim to demonstrate that standard feature selection methods can be parallelized in Big Data platforms like Apache Spark so as to boost both performance and accuracy. We propose a distributed implementation of a generic feature selection framework that includes a broad group of well-known information theory-based methods. Experimental results for a broad set of real-world datasets show that our distributed framework is capable of rapidly dealing with ultra-high-dimensional datasets as well as those with a huge number of samples, outperforming the sequential version in all the cases studied.

Index Terms

High-dimensional, Filtering methods, Feature selection, Apache Spark, Big Data.

I. Introduction

In the last few decades, the dimensionality of datasets used for machine learning (ML) or data mining tasks has increased significantly, representing an unprecedented challenge for researchers, given that existing algorithms do not always respond in a suitably timely way to extremely high dimensionality. In fact, analyzing the datasets posted in the popular libSVM Database [1], it can be observed that maximum data dimensionality was about 62 000 in the 1990s, but increased to some 16 million in the first decade of this century and has further increased to some 29 million in the current decade. In this new scenario, existing learning methods need to be adapted so as to be able to deal with millions of features.

With the advent of very high-dimensional datasets, identifying relevant features has become paramount. Dimensionality reduction techniques can be applied to reduce the dimensionality of the original data and even to improve learning performance [2], [3], [4]. These dimensionality reduction techniques usually come in two flavors: feature selection (FS) and feature extraction. Each approach has its own merits. Feature extraction techniques combine original features to yield a new set of features, whereas FS techniques remove irrelevant and redundant features. Since FS maintains the original features, it is especially useful for applications where the original features are important for model interpretation and knowledge extraction [5], [6]. For this reason, this model was the focus of our research.

Existing FS methods are not expected to scale well when dealing with Big Data due to the fact that efficiency may significantly deteriorate or that the FS approach may even become inapplicable [7]. Scalable distributed programming protocols and frameworks have emerged in the last decade to manage the problem of Big Data. The first programming model developed was MapReduce [8] along with its open-source implementation Apache Hadoop [9], [10]. More recently, Apache Spark [11], [12] has been presented as a new distributed framework with a fast, general large-scale data processing engine that is popular with ML researchers due to its suitability for iterative procedures.

Likewise, several libraries for approaching ML tasks in Big Data environments have emerged in recent years. The first such library was Mahout [13], followed up by MLlib [14], built on the Spark system [12]. Thanks to Spark's capacity for in-memory computation that speeds up iterative processes, algorithms developed for this kind of platform have become pervasive in the industry. Although several gold-standard algorithms for ML tasks have been redesigned to incorporate a distributed implementation for Big Data technologies, this is not yet the case for FS algorithms. To date only a simple chi-squared approach[1] and a Random Forest improvement [15] have been proposed to deal with this problem.

This research aims to fill the existing gap by demonstrating that standard FS methods can be designed for Big Data platforms that can be usefully applied to big datasets while boosting both performance and accuracy. We propose a new distributed design for a FS generic framework based on information theory [16] that is implemented using the Apache Spark paradigm. To make this adaptation feasible, a wide variety of techniques from the distributed environment have been used, including information

[1]http://spark.apache.org/docs/latest/mllib-feature-extraction.html#chisqselector

caching, data partitioning and replication of relevant variables. Note that adapting this framework to Spark implies a major challenge as it requires deep restructuring of classic algorithms.

To test the effectiveness of our framework, we applied it to a complete set of real-world datasets (up to $O(10^7)$ features and instances). The results point to competitive performance (in terms of generalization and efficiency) when dealing with datasets that are huge in terms of both number of features and instances. As an illustrative example, we were able to select 100 features in a dataset with 29×10^6 features and 19×10^6 instances in under 60 minutes (using a 432-core cluster).

The remainder of this paper is organized as follows: Section 2 provides some background information on FS, Big Data, the MapReduce programming model and other frameworks. Section 3 describes the distributed framework we propose for FS in Big Data. Section 4 presents and discusses the experiments carried out. Finally, Section 5 concludes the paper.

II. Background

Below we briefly introduce FS, discuss the advent of Big Data and its implications and outline the particularities of MapReduce and the models derived from this framework.

A. *Feature selection*

FS is a dimensionality reduction technique that tries to remove irrelevant and redundant features from original data. Its goal is to obtain a subset of features that properly describes a given problem with minimum performance degradation in order to obtain simpler and more accurate schemes [2].

We can formally define FS as follows: let $\boldsymbol{e_i}$ be an instance $\boldsymbol{e_i} = (e_{i1}, \dots, e_{in}, e_{iy})$, where e_{ir} corresponds to the r-th feature value of the i-th sample and e_{iy} corresponds to the value of the output class Y. Let us assume a training set D with m examples, whose instances $\boldsymbol{e_i}$ are formed by a set X of n characteristics or features, and a test set D_t. $S_\theta \subseteq X$ is this defined as a subset of selected features yielded by an FS algorithm.

FS methods can be broadly categorized as [17]:

1) **Wrapper methods**, which use an evaluation function dependent on a learning algorithm [18]. They are aimed at optimizing a predictor as part of the learning process.
2) **Filtering methods**, which use other selection techniques as separability measures or statistical dependences. They only consider the general characteristics of the dataset, as they are independent of any predictor [19].
3) **Embedded methods**, which use a search procedure that is implicit in the classifier/regressor [20].

Filtering methods usually result in better generalization due to learning independence. Nevertheless, since they usually select larger feature subsets, they sometimes require the application of a threshold. Regarding complexity, filters are normally less costly than wrappers. When the number of features is large (especially the case of Big Data), filtering methods need to be used as they are much faster than either of the other approaches.

B. *Big Data: a two-sided coin*

The Internet continues generating quintillions of bytes of data; in 2012, for example, 2.5 exabytes of data were created daily. A solution for the problem of handling large collections of data is therefore becoming increasingly urgent [21]. Exceptional technologies are now required to efficiently collect, maintain, transmit and process large datasets within tolerable time intervals. Extracting relevant information from these collections of data is now one of the most important and complex challenges facing data analytics research, especially since many knowledge extraction algorithms have been rendered obsolete in the face of such vast amounts of data.

Big Data, a term coined to describe the exponential growth and availability of data nowadays, has becomes a problem for classical data analytics. Gartner [22] referred to Big Data in terms of volume, velocity and variety, that is, the 3Vs, to which a further 2Vs were subsequently added, namely, veracity and value. Information thus defined requires a radically new approach to large-scale processing. An under-explored but no less important topic is Big Dimensionality in Big Data [23]. This phenomenon, also known as the “Curse of Big Dimensionality”, reflects the explosion of features and the combinatorial impact of new large incoming datasets with thousands or even millions of features.

Data scientists have generally focused on only one aspect of Big Data, namely, the huge number of instances, while paying less attention to the features aspect. Big Dimensionality, however, calls for new FS strategies and methods to deal with the feature explosion problem, reflected in many of the best known dataset repositories in computational intelligence (like UCI or libSVM [24], [1]), where the dimensionality of most of the newly added datasets is huge [23] (e.g. almost 30 million for the KDD2010 dataset).

An additional problem is the myriad of feature types and their combinations that are becoming standard in many real-world scenarios. For instance, multimedia content on the Internet, which represents about 60% of total traffic [25], is transmitted in thousands of different formats (audio, video, images, etc.). Another example is natural language processing (NLP), where multiple feature types such as words, n-gram templates, etc., are simultaneously used to produce comprehensible and reliable models [26].

Not all features in a problem, however, contribute equally to prediction models and results. FS is thus required, now more than ever, to ensure fast and cost-effective learning and prediction. Isolating high-value features from a raw set of potentially irrelevant, redundant and noisy features, while complying with measurement and storage requirements, is a key task in Big Data research. Although some solutions exist that apply FS techniques to large-scale data [27], most of these solutions are applied locally. For high-dimensional data, however, new distributed solutions are necessary.

C. Distributed programming models and frameworks: Apache Hadoop and Apache Spark

The MapReduce framework [8] emerged in 2003 as a revolutionary tool for handling Big Data. It was designed by Google specifically to generate and process large-scale datasets by automatically processing data in a highly distributed way using large clusters of computers. The framework is in charge of partitioning and managing data, recovering failures, scheduling jobs and communicating, leaving the programmers with a transparent and scalable tool to easily execute tasks on distributed systems.[2]

MapReduce is based on processing in two phases: mapping (Map) and reducing (Reduce). The Map function transforms key-value pairs into sets of intermediate pairs. The Reduce function merges these intermediate pairs with a matching key, with the master node splitting the data into chunks and distributing them across nodes for independent processing (in a divide-and-conquer fashion). Each node next executes the Map function on a given input subset and notifies completion to the master node. The master node then distributes the matching pairs across the nodes according to a key partitioning scheme that combines pairs using the Reduce function to form the final output.

Apache Hadoop [9], [10] is an open-source implementation of MapReduce for reliable, scalable, distributed computing. Despite being a popular open-source implementation of MapReduce, however, Hadoop is not suitable for certain applications, including online or iterative computing, high inter-process communication paradigms or in-memory computing [29].

In recent years, Apache Spark has been included in the Hadoop ecosystem [11], [12] as a powerful framework that performs faster distributed computing on Big Data. It does this by using in-memory primitives that allow it to perform 100 times faster than Hadoop for certain applications. The fact that this platform allows user programs to load data into memory and to make repeated queries means that it is particularly well suited for online and iterative processing (especially for ML algorithms). Spark is also versatile in allowing several distributed programming models like Pregel and MapReduce to be implemented.

Spark is based on a distributed data structure called resilient distributed datasets (RDDs). RDDs are an immutable, partitioned set of records that can be generated by either stored data or other RDDs. There are two types of operations for RDDs: *transformation*, which creates a new RDD from another RDD, and *action*, which returns a record/value to the main program after a set of operations is applied. Some examples of transformations are *map* (transform all records) and *filter* (select records), whereas *count* (compute the number of records) and *save* (write output) are examples of actions. The main difference between the two operations are when and how they are applied to data. Transformations, lazily applied, are annotated until an action triggers execution of pending operations in a log called *lineage*, which removes the need for checkpoints while allowing lost partitions to be easily recomputed from the log.

Users can also control other distributed features like persistence and data partitioning. For instance, users can *cache* a dataset to memory for reuse in further iterations. RDDs can also be persisted on disk if we consider that re-execution may be more costly than spilling to disk. Finally, placement of key-based data can be optimized by choosing between different Spark schemes (*range* or *hash*) or even using our own partitioner.

Spark primitives extend MapReduce concepts in such a way as to offer much more complex operations that ease code parallelization. For a full description of Spark operations, see [12]. Here we outline the most relevant operations for our algorithm:

- $mapPartitions$: Like Map, this operation independently runs a function on each partition, for each of which an iterator of tuples is fetched and another of the same type is generated.
- $groupByKey$: This operation (using a shuffle operation) groups tuples with the same key in a single vector of values.
- $sortByKey$: This operation is a distributed version of merge-sort.
- $broadcast$: This operation, normally used for large permanent variables (such as big hash tables), keeps a read-only copy of a given variable on each node rather than shipping a copy to each task.

Note that $groupByKey$ and $sortByKey$ imply a complete redistribution of data across the network. This operation, known as shuffling, is complex and costly and so should be avoided if at all possible.

Finally, created as a Spark subproject was a scalable ML library called MLlib [14], [30], composed of common learning algorithms and statistical utilities. Its main functionalities include classification, regression, clustering, collaborative filtering, optimization and dimensionality reduction (mostly feature extraction).

III. FEATURE SELECTION FILTERING FOR BIG DATA

An information theory-based framework that includes many common FS filtering algorithms has been proposed by Brown *et al.* [16] that proves that algorithms like minimum redundancy-maximum relevance (mRMR) and other algorithms are special

[2]For a exhaustive review of MapReduce and similar programming frameworks, see [28].

cases of conditional mutual information (CMI) when certain specific independence assumptions are made about both the class and the features (further details below). Here we demonstrate that these criteria are not only a sound theoretical formulation, but also fit well with modern Big Data platforms and allow us to distribute several FS methods and their complexity across a cluster of machines.

We now describe how we redesigned this framework for a distributed paradigm. Our framework contains a generic implementation of several information theory-based FS methods – including mRMR, conditional mutual information maximization (CMIM) and joint mutual information (JMI) – that, furthermore, have been designed to integrate in the Spark MLlib library. The framework can also be extended with other criteria provided by the user as long as it complies with the guidelines proposed by Brown *et al.* [16].

Below we first briefly present this framework and explain its adaptation to the Big Data environment. We then describe in detail how the selection process and the underlying information theory operations were implemented in a distributed manner using Spark primitives.

A. Information theory-based filter methods

Information measures tell us how much information has been acquired by the receiver when sent a message [31]. In predictive learning, we associate the message with the output feature in classification.

A commonly used uncertainty function is mutual information (MI) [32], which measures the amount of information one random variable contains about another, in other words, it expresses the reduction in the uncertainty of one random variable due to knowledge of the other variable:

$$\begin{aligned} I(A;B) &= H(A) - H(A|B) \\ &= \sum_{a \in A} \sum_{b \in B} p(a,b) \log \frac{p(a,b)}{p(a)p(b)}. \end{aligned} \tag{1}$$

where A and B are two random variables with marginal probability mass functions $p(a)$ and $p(b)$, respectively, $p(a,b)$ is the joint mass function and H is the entropy.

MI can likewise be conditioned to a third random variable. Thus, CMI is denoted as:

$$\begin{aligned} I(A;B|C) &= H(A|C) - H(A|B,C) \\ &= \sum_{c \in C} p(c) \sum_{a \in A} \sum_{b \in B} p(a,b,c) \log \frac{p(a,b,c)}{p(a,c)p(b,c)}. \end{aligned} \tag{2}$$

where C is a third random variable with marginal probability mass function $p(c)$ and where $p(a,c)$, $p(b,c)$ and $p(a,b,c)$ are the joint mass functions.

Filtering methods are based on a quantitative criterion or index, also known as the relevance index or scoring. This index measures the usefulness of each feature for a specific classification problem. Through the **relevance** of a feature for the class (self-interaction), we can rank features and select the most relevant ones. However, features can also be ranked using a more complex criterion such as whether it is more **redundant** than another feature (multi-interaction). For instance, redundant features (variables that carry similar information) can be discarded using the MI criterion [33]:

$$J_{mifs}(X_i) = I(X_i;Y) - \beta \sum_{X_j \in S} I(X_i;X_j),$$

where $S \subseteq S_\theta$ is the current set of selected features and β is a weight factor. Considered is the MI between each candidate $X_i \notin S$ and the class. Also introduced is a penalty proportional to the redundancy, calculated as the MI between the current set of selected features and each candidate feature.

A wide range of methods have been described in the literature that are built on these information theory measures. To homogenize use of all the criteria, Brown *et al.* [16] proposed a generic expression that allows multiple information theory criteria to be ensembled in a single FS framework, based on a greedy optimization process that assesses features using a simple scoring criterion. Through certain independence assumptions, many criteria can be transformed as linear combinations of the Shannon entropy terms MI and CMI [32]. In some cases, more complex criteria are expressed as non-linear combinations of these terms (e.g. max or min). For a detailed description of the transformation processes, see [16]. The generic formula proposed by Brown *et al.* [16] is as follows:

$$J = I(X_i;Y) - \beta \sum_{X_j \in S} I(X_j;X_i) + \gamma \sum_{X_j \in S} I(X_j;X_i|Y), \tag{3}$$

where γ represents a weight factor for the conditional redundancy component.

The formula can be divided into three components, representing the relevance of a feature X_i, the redundancy between two features X_i and X_j and the conditional redundancy between two features X_i, X_j and the class Y. Through the aforementioned assumptions, many criteria were re-written by the authors [16] to fit the generic formulation in such a way that all the methods could be implemented using a slight variation on this formula. For a comprehensive list of adapted FS methods, see [16].

B. *Feature selection filtering framework for Big Data*

We now describe the proposed FS filtering framework for Big Data using distributed operations, outlining the main changes made to adapt the classical approach to the new Big Data environment. We also analyze the implications arising from the distributed implementation of Equation 3 and the complexity arising from parallelization of the core operations of this expression, namely, MI and CMI.

Beyond implementation in Spark, we re-designed Brown's framework [16] by making improvements to the classical approach while maintaining certain features:

- **Columnar transformation:** The access pattern presented by most FS methods is feature-wise, in contrast to many other ML algorithms, which are instance-wise (they operate on rows). Although this may be considered a minor issue, it can significantly degrade performance since the natural way to compute relevance and redundancy in FS methods is normally via columns. This issue is especially important for distributed frameworks like Spark, where the data partitioning scheme has a significant impact on performance.
- **Broadcasting**: Once all features values have been grouped and partitioned into different partitions, data shift has to be minimal to avoid superfluous network and CPU usage. If the MI process is performed locally in each partition, the overall algorithm will run efficiently (almost linearly). Data shift is minimized by replicating the output feature and the last selected feature in each iteration.
- **Pre-computed data caching:** The first term in the generic criterion of Equation 3 is relevance, which basically implies calculating MI for all the input features and the output (relevance). This operation is performed once at the start of our algorithm, then cached to be re-used in subsequent evaluations of Equation 3. Likewise, subsequent marginal and joint proportions derived from these operations are also kept so as to omit some computations. This also helps isolate redundancy computation by feature as it replicates permanent information in all nodes.
- **Greedy approach:** Brown *et al.* [16] proposed a greedy search process in which only one feature is selected in each iteration. This approach transforms the quadratic complexity of typical FS algorithms into a more manageable complexity determined by the number of features to select.

1) Main FS algorithm:

Algorithm 1 is the main FS algorithm, in charge of deciding which feature to select in a sequential manner. Roughly speaking, it calculates the initial relevance for all the features, and then iterates to select the best features according to Equation 3 and the underlying MI and CMI values.

Algorithm 1 Main FS algorithm

```
Input: $D$ Dataset, an RDD of samples.
Input: $ns$ Number of features to select.
Input: $npart$ Number of partitions to set.
Input: $cindex$ Index of the output feature.
Output: $S_\theta$ Index list of selected features
  $D_c \leftarrow columnarTransformation(D, ns, npart)$
  $ni \leftarrow D.nrows; nf \leftarrow D.ncols$
  $REL \leftarrow computeRelevances(D_c, cindex, ni)$
  $CRIT \leftarrow initCriteria(REL)$
  $p_{best} \leftarrow CRIT.max$
  $sfeat \leftarrow Set(p_{best})$
  while $|S| < |S_\theta|$ do
    $RED \leftarrow computeRedundancies(D_c, p_{best}.index)$
    $CRIT \leftarrow updateCriteria(CRIT, RED)$
    $p_{best} \leftarrow CRIT.max$
    $sfeat \leftarrow addTo(p_{best}, sfeat)$
  end while
  $return(sfeat)$
```

The first step consists of transforming the data into columnar format. Once the data matrix is transformed, the algorithm obtains the relevance for each feature in X, initializing the criterion value (partial result according to Equation 3) and creating an initial ranking of the features. Relevance values are saved as part of the previous expression and are re-used in subsequent steps to update the criteria. The most relevant feature, p_{best}, is then selected and added to the set $sfeat$, initially empty. The iterative phase begins by calculating MI and CMI for p_{best}, each candidate X_i and Y. The resulting values update the accumulated redundancies (simple and conditional) of the criteria. In each iteration, the most relevant candidate features are selected as the new p_{best} and are added to $sfeat$. The loop ends once ns features (where $ns = |S_\theta|$) have been selected or when no more features remain to be selected.

2) Distributed operations: columnar transformation and MI computation: MI and CMI are undoubtedly the costliest operations to estimate in information theory-based FS. With huge datasets, these operations become impossible to calculate sequentially as the number of combinations grows. Below we describe how these calculations are parallelized as a set of distributed operations (explained in Section II-C). For all the algorithms described below, RDD variables are written in uppercase to distinguish them from ordinary variables.

Columnar transformation

As mentioned previously, the column-wise format is much more manageable for FS filtering methods than the row-wise format. Algorithm 2 explains this transformation, carried out as the first step in our algorithm. The idea behind this transformation is to transpose the local data matrix provided by each partition. This operation maintains the partitioning scheme without incurring in a high shuffling overhead. Additionally, once data are transformed, they can be cached and re-used in the subsequent loop. The result of this operation is a new matrix, with one row per feature, that generates a tuple, where k represents the feature index, $part.index$ is the index of the partition (henceforth the block index) and $matrix(k)$ is the local matrix for this feature block.

Algorithm 2 Function that transforms row-wise data into a column-wise format *(columnarTransformation)*

```
Input: $D$ Dataset, an RDD of samples.
Input: $nf$ Number of features.
Input: $npart$ Number of partitions to set.
Output: Column-wise data (RDD of feature vectors).
 1: $D_c \leftarrow$
 2: map partitions $part \in D$
 3:     $matrix \leftarrow new\ Matrix(nf)(part.length)$
 4:     for $j = 0\ until\ part.length$ do
 5:         for $i = 0\ until\ nf$ do
 6:             $matrix(i)(j) \leftarrow part(j)(i)$
 7:         end for
 8:     end for
 9:     for $k = 0\ until\ nf$ do
10:         $EMIT < k, (part.index, matrix(k)) >$
11:     end for
12: end map
13: $return(D_c.sortByKey(npart))$
```

In order to benefit from data locality, the algorithm allocates all instances of the same feature to a specific set of partitions (if possible, to just one partition). To do that, the new instances are sorted by key, with the number of partitions limited to $npart$. In subsequent phases, the partitions are mapped with the aim of generating a number of histograms per feature that count the number of occurrences by combination.

Choosing the right number of partitions is important for the next steps. If $npart$ is equal to or less than the number of features, then the number of total histograms per feature will be two at most; otherwise, the total number of histograms can be high, since the same feature can be distributed across many partitions (more than two). We recommend setting this parameter to, at most, $2\times$ the number of features.

Figure 1 depicts this process using a small example with eight instances and four features. This figure shows how the algorithm generates a block for each feature in each partition and how all blocks are sorted by feature in order to place them in the same partitions.

Computing relevance

Algorithm 3 describes how to compute relevance (MI) for all the input features and Y (as expressed in Equation 1). The design as an initialization method means that all variables in this function can be used in the subsequent algorithms. For example, the number of different values for each feature is first computed and saved as $counter$ (to limit the size of histograms).

The main idea behind relevance and redundancy functions is that calculations for each feature are performed independently. This is done by distributing only the single variables (p_{best} and Y) across the cluster and leveraging for the data locality property. The first step consists of collecting all blocks of Y from the data and putting them in a single vector for broadcasting ($bycol$). Histograms for all the candidate features with respect to Y are then calculated in $getHistograms$ (explained below). This function is common to both the relevance and redundancy phases. Three-dimensional histograms are computed between all the non-selected features and two secondary variables for redundancy, and between all the non-selected features and one variable for relevance. [3] Joint and marginal proportions are generated from the resulting histograms using matrix operations, that is, by aggregating proportions by row ($marginal$) and by computing joint for joint proportions ($joint$). Using this information we can now obtain the MI value for each candidate feature.

[3]For relevance, the *null* value is used to represent the lack of the second variable.

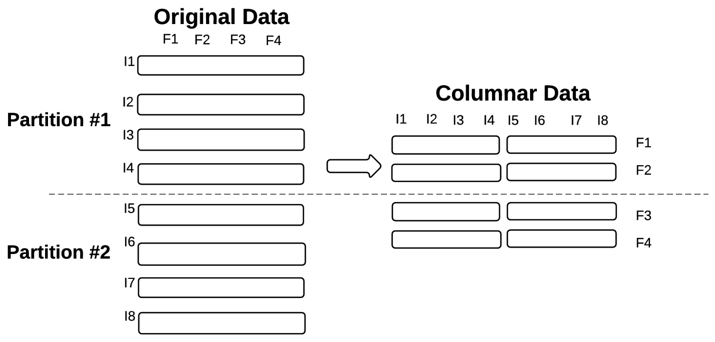

Fig. 1: Columnar transformation scheme. F and I indicate features and instances, respectively. Each rectangle on the left represents a single register in the original dataset and each rectangle on the right represents a transposed feature block in the new columnar format.

Algorithm 3 Compute mutual information between the set of features X and Y. *(computeRelevances)*

```
Input: $D_c$ RDD of tuples (index, (block, vector)).
Input: $yind$ Index of $Y$.
Input: $ni$ Number of instances.
Output: MI values for all input features.
1: $ycol \leftarrow D_c.lookup(yind)$
2: $bycol \leftarrow broadcast(ycol)$
3: $counter \leftarrow broadcast(getMaxByFeature(D_c))$
4: $H \leftarrow getHistograms(D_c, yind, bycol, null, null)$
5: $joint \leftarrow getProportions(H, ni)$
6: $marginal \leftarrow getProportions(aggregateByRow(joint), ni)$
7: $return(computeMutualInfo(H, yind, null))$
```

Computing redundancy

Simple and conditional redundancy are computed between p_{best}, each candidate feature X_i and Y. Conditional redundancy introduces a third conditional variable (Y), following the formula $I(X_j; X_i|Y)$ (Equation 2).

The operation is repeated until we obtain the number of features specified as a parameter. Algorithm 4, an extension of relevance computation (Algorithm 3), depicts this process. Blocks for p_{best} are obtained from the RDD and broadcast to all the nodes. The $getHistograms$ function is called up with two variables in order to obtain the histograms for all the candidate features with respect to p_{best} and Y. Note that the vector for Y is already available from the redundancy computation phase. Finally, both types of redundancy are computed using the function that computes MI and CMI ($computeMutualInfo$).

Algorithm 4 Compute CMI and MI between p_{best}, the set of candidate features, and Y. *(computeRedundancies)*

```
Input: $D_c$ RDD of tuples (index, (block, vector)).
Input: $jind$ Index of $p_{best}$.
Output: CMI values for all input features.
1: $jcol \leftarrow D_c.lookup(jind)$
2: $bjcol \leftarrow broadcast(jcol)$
3: $H \leftarrow getHistograms(D_c, jind, bjcol, yind, bycol)$
4: $return(computeMutualInfo(H, jind, yind))$
```

Histograms creation

Algorithm 5 computes 3-dimensional histograms for the set of candidate features with respect to p_{best} and Y and subsequently computes MI and CMI. When no conditional variable is provided, this yields histograms whose third dimension is equal to one.

The first two lines return the dimensions for the p_{best} and Y variables using $counter$ (Algorithm 1). A mapping operation on each partition is launched on the dataset that iterates over the blocks derived from the columnar transformation. Each tuple is formed of a key (the index of a candidate feature) and a value with an index block and the corresponding feature array ($(k, (block, v))$). For each instance, a matrix is initialized to zero and then incremented by one, according to the value for each combination of p_{best}, X_i, and Y. Single features (p_{best} and Y) are broadcast in the form of matrices, whose first and second axes indicate, respectively, the block index and the partial index of the value in this block. This updating operation is repeated to the end of the feature vector, after which a new tuple is emitted with the feature index as the key and the

Algorithm 5 Function that computes 3-dimensional histograms between p_{best}, the set of candidate features, and Y for CMI; or between the set of candidate features, and Y for MI. *(getHistograms)*

```
Input: $D_c$ RDD of tuples (index, (block, vector)).
Input: $jind$ Index of $Y$ or $p_{best}$.
Input: $yind$ Index of feature $Y$ (can be empty).
Input: $jcol$ Values for $Y$ or $p_{best}$, a broadcast matrix.
Input: $ycol$ Values for $Y$, a broadcast matrix (can be empty).
Output: Column-wise dataset (RDD of feature vectors).
1: $jsize \leftarrow counter(jind)$
2: $ysize \leftarrow counter(yind)$
3: $H \leftarrow$
4: map partitions $part \in partitions$
5:   for $(k, (block, v)) \leftarrow part$ do
6:     $isize \leftarrow counter(k)$
7:     $m \leftarrow newMatrix(ysize)(isize)(jsize)$
8:     for $e = 0$ $until$ $v.size$ do
9:       $j \leftarrow jcol(block)(e);\ y \leftarrow ycol(block)(e);\ i \leftarrow v(e)$
10:      $m(y)(i)(j) += 1$
11:    end for
12:    $EMIT < k, m >$
13:  end for
14: end map
15: $return(H.reduceByKey(sum))$
```

resulting matrix as the value ($< k, m >$). The mapping operation continues with ensuing blocks until the partition is finished. The final histograms are then aggregated by summation. This aggregation process will remain simple as long as the number of histograms per partition is small – which is normally the case, as a single partition usually contains all the blocks for the same feature. This is why it is important, as explained above, to reduce the number of histograms by adjusting the number of partitions.

Figure 2 depicts the histogram creation process using a simple example, showing how the algorithm generates one histogram for each partition and feature. In this case, the number of partitions corresponds to the number of features, so only one histogram is generated per feature.

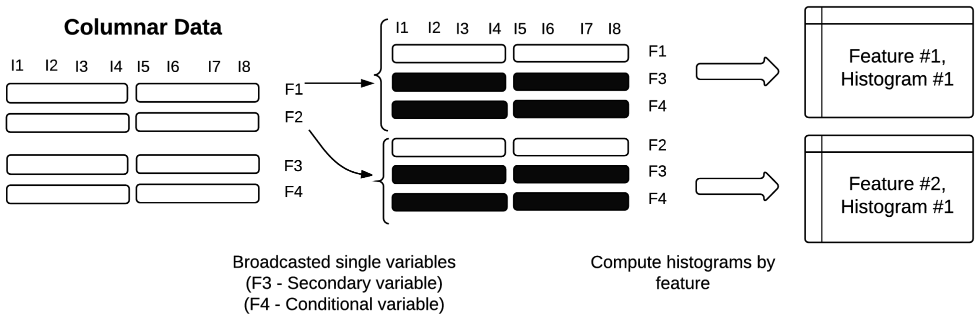
Columnar Data
I1 I2 I3 I4 I5 I6 I7 I8
F1
F2
F3
F4
Broadcasted single variables
(F3 - Secondary variable)
(F4 - Conditional variable)
Compute histograms by feature
Feature #1, Histogram #1
Feature #2, Histogram #1

Fig. 2: Histogram creation scheme. F and I indicate features and instances, respectively. The white rectangles each represent a single feature block in the columnar format and the black rectangles represent marginal and joint proportions for single variables, broadcast across the cluster.

MI and CMI computations

Algorithm 6 details the process that unifies computation of MI and CMI. The algorithm takes as input the indices of single variables (Y or p_{best} for MI, and both for CMI) and all previously computed histograms. Before launch, the algorithm broadcasts the marginal and joint matrices that correspond to these variables and sends this information to the nodes, as this information is already computed and cannot be computed independently from the previous histograms.

A mapping phase is then launched for each histogram tuple, consisting of a given feature index as the key and a 3-dimensional matrix as the value. The algorithm generates the MI and CMI values for all combinations of histograms and single variables (see Equations 1 and 2). First, however, it is necessary to compute the marginal proportions for the set of candidate features as well as the joint proportions between each X_i and Y and each X_i and p_{best} (using the matrix operations described in Algorithm 3). Once all joint and marginal proportions are calculated, a loop starts over all combinations to compute the

Algorithm 6 Calculate MI and CMI for the set of histograms with respect to Y or p_{best} for MI, and both variables for CMI. *(computeMutualInfo)*

```
Input: $H$ Histograms, an RDD of tuples (index, matrix).
Input: $bind$ Index of feature $p_{best}$.
Input: $cind$ Index of feature $Y$ (can be empty).
Output: MI and CMI values.
 1: $bprob \leftarrow broadcast(marginal(bind))$
 2: $cprob \leftarrow broadcast(marginal(cind))$
 3: $bcprob \leftarrow broadcast(joint(bind))$
 4: $MINFO \leftarrow$
 5: map $(k, m) \in H$
 6:   $aprob \leftarrow computeMarginal(m)$
 7:   $abprob \leftarrow computeJoint(m, bind)$
 8:   $acprob \leftarrow computeJoint(m, cind)$
 9:   for $c = 0$ until $getSize(m)$ do
10:     for $b = 0$ until $getnRows(m(c))$ do
11:       for $a = 0$ until $getnCols(m(c))$ do
12:         $pc \leftarrow cprob(c)$
13:         $pabc \leftarrow (m(c)(a)(b)/ninstances)/pc$
14:         $pac \leftarrow acprob(c)(a);\ pbc \leftarrow bcprob(c)(b)$
15:         $cmi += conditionalMutualInfo(pabc, pac, pbc, pc)$
16:         if $c == 0$ then
17:           $pa \leftarrow xprob(a)$
18:           $pab \leftarrow abprob(a)(b);\ pb \leftarrow yprob(b)$
19:           $mi += mutualInfo(pa, pab, pb)$
20:         end if
21:       end for
22:     end for
23:   end for
24:   $EMIT < k, (mi, cmi) >$
25: end map
26: $return(MINFO)$
```

proportion $pabc$ (which comes directly from the histogram) and the final result for each combination. These results are then aggregated to obtain the overall MI and CMI values for each feature.

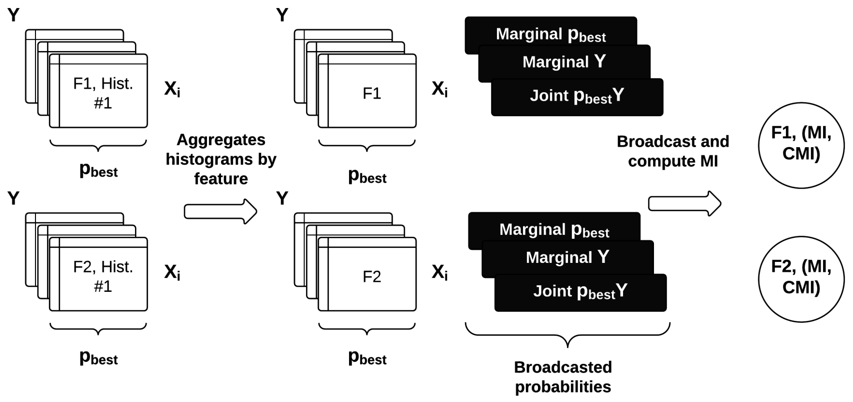

Fig. 3: MI computations. F indicates features and the matrices depict the histograms for each feature. The black rectangles represent broadcast joint and marginal probabilities.

Figure 3 depicts the MI process that is launched once all histograms have been computed. Each partition generates a histogram for each feature it contains. Histograms for the same feature are aggregated to obtain a single final histogram. Only marginal and joint proportions that cannot be computed independently from each histogram are broadcast. Using the histograms and broadcast variables, MI and CMI are independently computed per feature.

3) High-dimensional and sparse versions: The above algorithms work well with tall-and-skinny data, formed of a small number of features and a large number of instances. However, complexity grows horizontally for sparse datasets characterized by a large number of features and a variable number of instances with an undefined number of non-zero indexed elements. The algorithms affected by this issue are columnar transformation (Algorithm 2) and histogram creation (Algorithm 5). The remaining code remains unchanged except that the structure of data is reduced to a single vector (in the form $(index, vector)$). Since only one histogram is generated per feature, the block index is excluded from this structure.

Algorithm 7 describes a new model of processing that transposes data directly, generating single vectors for each feature. Sparsity is maintained in sparse vectors while the index of the original instance is used as the key. The algorithm generates a tuple for each value, in such a way that the key is formed of the feature index and the value is formed of the instance index and the value itself. Tuples are all grouped by key to create a single sparse vector (formed of sorted key-value tuples).

Algorithm 7 Function that transforms row-wise data into column-wise data (sparse version) *(sparseColumnar)*

```
Input: $D$ Dataset, an RDD of sparse samples.
Input: $npart$ Number of partitions to set.
Output: Column-wise dataset (feature vectors).
1: $D_c \leftarrow$
2: map $reg \in D$
3:     $ireg \leftarrow reg.index$
4:     for $i = 0$ $until$ $reg.length$ do
5:         $EMIT < reg(i).key, (ireg, reg(i).value) >$
6:     end for
7: end map
8: $D_c \leftarrow D_c.groupByKey(npart).mapValues(vectorize)$
```

As for histogram creation, the partition mapping operation is now replaced by a mapping operation applied to each previously generated feature vector. Algorithm 8, which describes this process, is similar to Algorithm 5, with the caveat that only one histogram is yielded for each feature; hence, no reducing operation is necessary.

Algorithm 8 Function that computes 3-dimensional histograms for the set of features X_i with respect to features p_{best} and Y (sparse version). *(sparseHistograms)*

```
Input: $D_c$ Dataset, an RDD of tuples (Int, (Int, Vector)).
Input: $jind$ Index of $Y$ or $p_{best}$.
Input: $yind$ Index of feature $Y$ (can be empty).
Input: $jcol$ Values for $Y$ or $p_{best}$, a broadcast matrix.
Input: $ycol$ Values for $Y$, a broadcast matrix (can be empty).
Output: Column-wise dataset (RDD of feature vectors).
1: $jsize \leftarrow counter(jind);\ ysize \leftarrow counter(yind)$
2: $jyhist \leftarrow frequencyMap(jind, yind)$
3: $zhist \leftarrow frequencyMap(yind)$
4: $H \leftarrow$
5: map $(k, v) \in D_c$
6:     $isize \leftarrow counter(k)$
7:     $m \leftarrow newMatrix(ysize)(isize)(jsize)$
8:     for $e = 0$ $until$ $v.size$ do
9:         $j \leftarrow jcol(e);\ y \leftarrow ycol(e)$
10:        $i \leftarrow v(e)$
11:        if $j <> 0$ then
12:            $jyhist(j)(y) = jyhist(j)(y) - 1$
13:        end if
14:        $m(y)(i)(j) += 1$
15:    end for
16:    for $((j, y), q) \leftarrow jyhist$ do
17:        $m(y)(0)(j) += q$
18:    end for
19:    for $(y, q) \leftarrow yhist$ do
20:        $m(y)(0)(0) += yhist(y) - sum(mat(y))$
21:    end for
22:    $EMIT < k, m >$
23: end map
24: $return(H)$
```

Unlike what happens in the dense version, to avoid visiting all possible sparse feature combinations, the matrix generation process is adapted, using, as much as possible, accumulators to compute combinations formed by zeros. As accumulators, the algorithm calculates the class histogram for the conditional variable and the joint class histogram for the parametric variables $jind$ and $yind$. A loop is first started for those combinations in which the first variable (i) is not equal to zero (the procedure is the same as in the dense version). If the second variable (j) is equal to zero, then the frequency counter (in the joint histogram) for the combination is reduced by one. If j is not equal to zero, then the algorithm completes the matrix with the frequencies in the joint histogram. Finally, when both variables (j and y) are equal to zero, the matrix is updated with the remaining occurrences for all classes.[4] The outcome is the feature index and the aforementioned matrix.

4) Algorithm complexity: As mentioned earlier, the FS algorithm performs a greedy search which stops once the condition defined as input is reached. Beyond this point, the sequential algorithm is influenced by the set of distributed algorithms/operations as presented above. The distributed primitives used in these algorithms need to be analyzed to check the complexity of the full proposal. Note that the first operation (columnar transformation) is quite time-consuming as it makes great use of the network and memory when shuffling the data (wide dependency). However, once data have a known partition, they can be re-used in subsequent phases (leveraging the data locality property). This transformation is performed once at the start or can even be omitted if data are already in columnar format. The list of distributed operations in this algorithm are described as follows:

- Algorithm 2: This algorithm starts with a *mapPartitions* operation that transposes the local matrix contained in the partition and emits a tuple for each feature (in a linear distributed order). The total number of tuples is n multiplied by the number of original partitions. These tuples are then shuffled across the cluster and each subset is locally sorted (in a log-linear distributed order).
- Algorithm 3-4: The first operation for both algorithms is to retrieve a single column (feature) using the *lookup* primitive (in a linear distributed order). Since the data are already partitioned, the operation efficiently only looks at the right partition. This variable is then broadcast to all the nodes, sending a single feature (m values) across the network. The following operations (histograms and MI computations) are described below.
- Algorithm 5: This algorithm is a simple *mapReduce* operation whereby the previously generated tuples are transformed to local histograms and then reduced to the final histograms by feature. This mapping operation consists of a linear function ($O(m)$) that fetches the data contained in each local matrix.
- Algorithm 6: This operation starts by broadcasting three single values (proportion values). For each feature, three linear operations are launched to compute extra probabilities. The MI values are then computed by fetching the entire 3D-histogram (in a cubic linear order). Note that the complexity of all these operations is bounded by the cardinality of the features included in the histogram.

IV. Experimental Framework and Analysis

This section describes experiments that evaluate the proposed FS framework applied to a set of real-world problems that are huge in terms of both features and instances.

A. Datasets and methods

We used five classification datasets to measure the quality and usefulness of our FS framework implementation for Spark. We classified these datasets into two groups: dense (large number of samples) and sparse (high-dimensional datasets). For the sparse datasets, the high-dimension version described in Section III-B3 was used.

The first dataset ECBDL14, used as a reference dataset at the international GECCO-2014 conference, consists of 631 features (including both numerical and categorical attributes) and 32 million instances. In this binary classification problem the class distribution is imbalanced, with 98% of negative instances. To this imbalanced problem, we applied the MapReduce version of the Random OverSampling (ROS) algorithm [34] (henceforth we will use ECBDL14 to refer to the ROS version). Another dataset used was dna, consisting of 50 000 000 instances and 201 discrete features and created in 2008 for the Pascal Large Scale Learning Challenge.[5] Only the training set was used for our experiments; since the test set does not contain the class labels, the training set was used to generate both subsets (using an 80/20 hold-out data split). The ROS technique was also applied to this dataset (henceforth dna) since this problem is also imbalanced between classes. The remaining datasets (epsilon, url and kddb) come from the LibSVM dataset repository [1]. These datasets and their descriptions can be found in the project's website.[6] Table I provides summary details of these datasets.

TABLE I: Summary description of five datasets, indicating the number of examples for training and test sets (#Train Ex., #Test Ex.), the total number of attributes (#Atts.), the number classes (#Cl) and the sparsity condition (Sparse).

Data Set	**#Train Ex.**	**#Test Ex.**	**#Atts.**	**#Cl.**	**Sparse**
epsilon	400 000	100 000	2000	2	No
dna	79 739 293	10 000 000	200	2	No
ECBDL14	65 003 913	2 897 917	630	2	No
url	1 916 904	479 226	3 231 961	2	Yes
kddb	19 264 097	748 401	29 890 095	2	Yes

[4] The class vector is always dense

[5] http://largescale.ml.tu-berlin.de/summary/

[6] http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/

As an FS benchmark method, we used the mRMR algorithm [35] since it is one of the most cited selectors in the literature. Note that the FS algorithm chosen to test performance did not affect the time results yielded by the framework since all criteria were computed in the same way.

For the comparison study, we used Support Vector Machines (SVM) [36], and Naive Bayes [37] as the classifiers. For our experiments, we used the distributed versions of these algorithms implemented in the MLlib library [14]. The parameters of the classifiers, as recommended in the authors' specifications [14], are shown in Table II. For all executions, the datasets were cached in memory as SVM and our method used iterative processes. The level of parallelism (number of partitions) was set to 864, twice the total number of cores available in the cluster.[7]

TABLE II: Parameters of the used classifiers

Method	**Parameters**
Naive Bayes	lambda = 1.0
SVM	stepSize = 1.0, batchFraction = 1.0, regularization = 1.0, iterations = 100
mRMR	level of parallelism = 864

A Spark package associated with this research can be found in the third-party Spark Repository: *http://spark-packages.org/package/sramirez/spark-infotheoretic-feature-selection*. This software was designed for integration in the MLlib Library and has an associated JIRA issue to discuss its integration in this library: *https://issues.apache.org/jira/browse/SPARK-6531*.

We used two common evaluation metrics to assess the quality of the resulting FS schemes: Area under the Receiver Operating Characteristic Curve (AUROC, henceforth AUC) to evaluate classifier accuracy, and training modeling time to evaluate FS performance.

B. Cluster configuration

For all the experiments we used a cluster composed of eighteen computing nodes and one master node. The computing nodes had the following characteristics: 2 processors x Intel Xeon CPU E5-2620, 6 cores per processor, 2.00 GHz, 15 MB cache, QDR InfiniBand Network (40 Gbps), 2 TB HDD, 64 GB RAM. We used the following configuration for the software: Hadoop 2.5.0-cdh5.3.1 from Cloudera's open-source Apache Hadoop distribution,[8] HDFS replication factor 2, HDFS default block size 128 MB, Apache Spark and MLlib 1.2.0, 432 cores (24 cores/node), 864 RAM GB (48 GB/node).

Both the HDFS and Spark master processes (the HDFS NameNode and the Spark Master) were hosted in the main node. The NameNode controlled the HDFS and coordinated the slave machines by means of their respective DataNode daemons. The Spark Master controlled all the executors in each worker node. Spark used the HDFS file system to load and save data in the same way as the Hadoop framework.

C. Results analysis

For the evaluation of the time employed by our implementation to rank the most relevant features, Table III presents the results obtained by our algorithm using different ranking thresholds (number of selected features).

TABLE III: Selection time by dataset and threshold (in seconds)

# Features	**kddb**	**url**	**dna**	**ECBDL14**	**epsilon**
10	283.61	94.06	97.83	332.90	111.42
25	774.43	186.22	148.78	596.31	173.39
50	1365.82	333.70	411.84	1084.58	292.07
100	2789.55	660.48	828.35	2420.94	542.05

As can be observed, our algorithm yielded competitive results in all cases, irrespective of the number of iterations (represented by the threshold value). Regarding the datasets with the highest data volumes, namely, kddb (ultra-high-dimensionality) and ECBDL14 (with a huge number of samples), our method was able to rank 100 features in under 60 minutes.

We conducted a comparison study between our distributed version and the sequential version developed by Brown's lab.[9]. Samples from dna were generated with different ratios of instances in order to study the scalability of our approach in comparison with the sequential version.[10] The level of parallelism was set to 200 in the distributed executions as so to facilitate comparison between the distributed version with one core per feature and the sequential version with just a single core.

Table IV and Figure 4 show the time results for our distributed versions compared to the sequential version. In the latter, the last two values (highlighted in italics) were estimated using linear interpolation since they could not be computed due to memory problems. Table IV indicates that our distributed version outperformed the sequential approach in all cases. What was especially remarkable was that the largest dataset achieved the maximum speedup rate (29.83).

TABLE IV: Selection times (sequential vs. distributed approaches) in seconds.

# Examples	Sequential	Distributed	Speedup
1 000 000	450.00	425.56	1.06
5 000 000	2 839.72	508.41	5.59
40 000 000	*23 749.77*	828.35	28.67
80 000 000	*47 646.97*	1 597.26	**29.83**

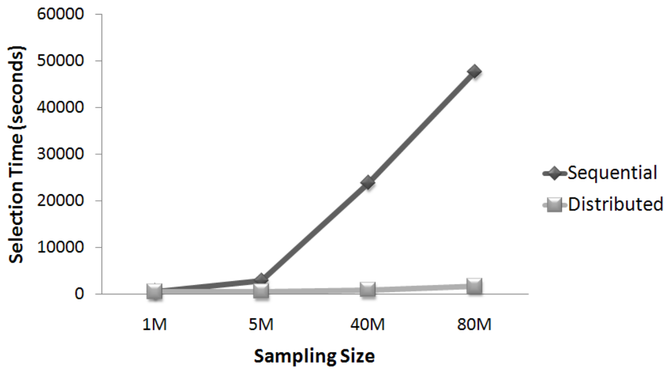

Fig. 4: Selection times (sequential vs. distributed approaches)

Finally, as an additional study of scalability we varied the number of cores, using ECBDL14 (the largest dense dataset) as a reference and using the same parameters as in the previous study. Figure 5, which depicts how our method performed depending on the number of cores (10 to 100), reveals logarithmic behavior as the number of cores increased. Note that for just 10 cores not all memory was available.

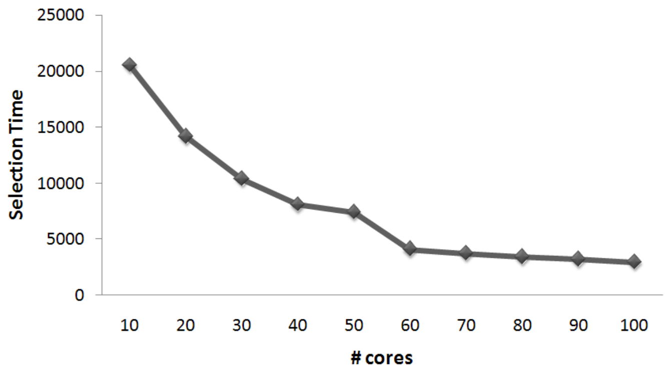

Fig. 5: Selection time by number of cores in the distributed approach (in seconds).

D. *Classification results analysis*

We also analyzed the usefulness of our FS solution when applied to large-scale classification. Figures 6 and 7 show the accuracy results for SVM and Naive Bayes using different FS schemes. All the datasets described in Table I were used in this study except kddb because those classifiers were not designed to work with such great dimensionality.

Figure 7 points to an important improvement when FS was used for url and epsilon. In contrast, the fact that the impact for dna and ECBDL14 was negligible can be explained by their high imbalance ratio or small number of features. Figure 6 presents similar results, except that the improvement for the url dataset was much smaller.

[7]The Spark creators recommend using 2-4 partitions per core: http://spark.apache.org/docs/latest/programming-guide.html#parallelized-collections
[8]http://www.cloudera.com/content/cloudera/en/documentation/cdh5/v5-0-0/CDH5-homepage.html
[9]FEAST toolbox (python version): http://www.cs.man.ac.uk/~gbrown/software/
[10]The sequential version was executed in one node of our cluster with the aforementioned characteristics.

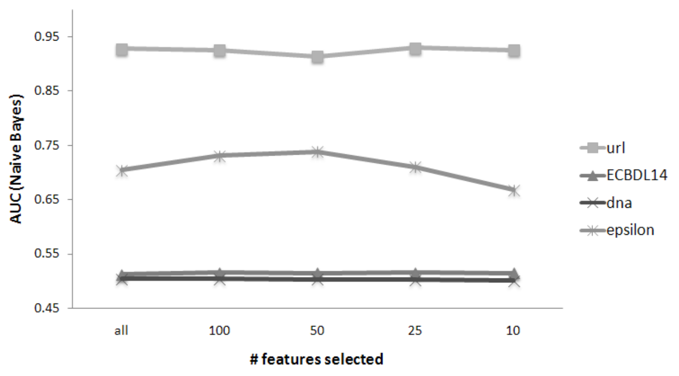

Fig. 6: AUC results for NaiveBayes using different thresholds

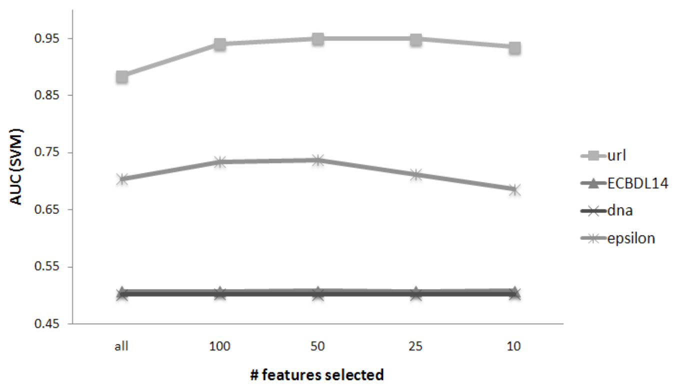

Fig. 7: AUC results for SVM using different thresholds

Beyond AUC, the time employed to create a classification model is quite important in many large-scale problems. Figures 8 and 9 show classification times for the training phase for different datasets and thresholds. The results demonstrate that the simplicity and performance of the generated models was improved after applying FS, most especially for SVM, which requires more time for modeling than Naive Bayes.

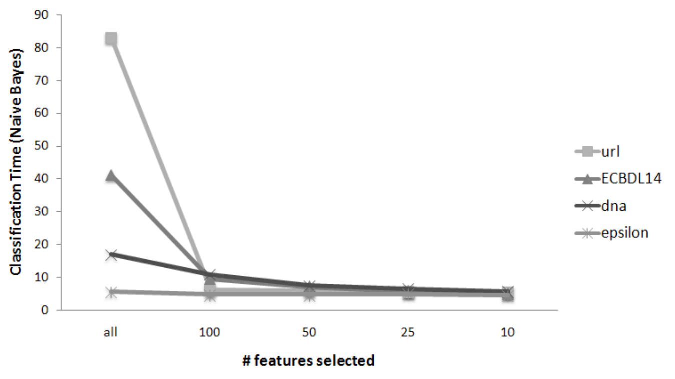

Fig. 8: NaiveBayes classification times for the training phase using different thresholds (in seconds)

The performance results demonstrate that our solution is capable of selecting features in a competitive time interval when applied to datasets that are huge- in both number of instances and features. The results also demonstrate the superiority of our distributed approach to FS over the sequential approach.

Furthermore, using our selection schemes, the classifiers yield better results in most cases, and at least similar results in the other cases. Note that, in all the studied cases, our model was much simpler and faster despite using a small percentage of the original set of features.

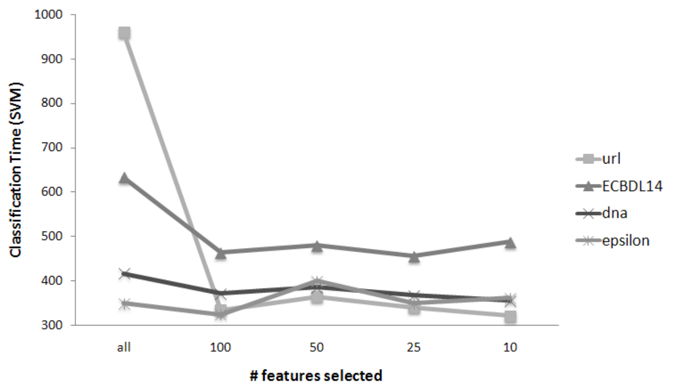

Fig. 9: SVM classification times for the training phase using different thresholds (in seconds)

V. Conclusions

In discussing the problem of processing Big Data, especially from the perspective of dimensionality, we have highlighted the impact of correctly identifying relevant features in datasets and the corresponding difficulties caused by the combinatorial effects of incoming data growing in terms of both instances and features. Despite the growing interest in dimensionality reduction for Big Data, few FS methods have been developed to date that can adequately deal with high-dimensionality problems.

Adopting an information theory approach, we adapted a generic FS framework for Big Data from a proposal by Brown *et al.* [16]. The framework contains implementations of many state-of-the-art FS algorithms, including mRMR and JMI. The adaptation entailed a radical redesign of Brown *et al.*'s framework so as to adapt it to a distributed paradigm. This research has also resulted in contributing an FS module to the emerging Spark and MLlib platforms, which, to date, included no complex FS algorithm.

Our experimental results demonstrate the usefulness of our FS solution applied to a broad set of large real-world problems. Our solution performed well with two dimensions of Big Data (samples and features) and yielded competitive performance results in dealing with both ultra-high-dimensionality datasets and datasets with a huge number of samples. Our distributed approach consistently outperformed the sequential version, enabling the resolution of problems that could not be usefully resolved using the classical approach.

Future research will focus on the following:

- Designing new information theory-based approaches for high-speed data streams and also extending these approaches to the handling of drifts in concepts.
- Analyzing the impact of approximative selection on high-dimensional data via faster solutions that do not incur a high penalty on accuracy.
- Designing a new fully automatic FS system that selects the most relevant subset of features from a full set of features, thereby eliminating the need to define the number of features to select in each execution.

Acknowledgments

This work is supported by Spanish National Research Projects TIN2014-57251-P, TIN-2015-65069-C2-1-R and TIN2016-81113-R, the BBVA Foundation Project 75/2016 BigDaPTOOLS and the Andalusian Research Plan P11-TIC-7765 and P12-TIC-2958, and also by the Xunta de Galicia through Research Project GRC2014/035 (all projects partially funded by FEDER funds of the European Union). S. Ramírez-Gallego holds a FPU scholarship (FPU13/00047) from the Spanish Ministry of Education and Science. D. Martínez-Rego and V. Bolón-Canedo acknowledge support of the Xunta de Galicia under Postdoctoral Grant POS-A/2013/196 and ED481B 2014/164-0.

References

[1] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, datasets available at http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/.

[2] I. Guyon, S. Gunn, M. Nikravesh, and L. A. Zadeh, *Feature Extraction: Foundations and Applications (Studies in Fuzziness and Soft Computing)*. Springer-Verlag New York, Inc., 2006.

[3] Z. A. Zhao and H. Liu, *Spectral feature selection for data mining*. Chapman & Hall/CRC, 2011.

[4] V. Bolón-Canedo, N. Sánchez-Maroño, and A. Alonso-Betanzos, *Feature Selection for High-Dimensional Data*, ser. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer, 2015.

[5] Q. Wu, Z. Wang, F. Deng, Z. Chi, and D. Feng, "Realistic human action recognition with multimodal feature selection and fusion," *Systems, Man, and Cybernetics: Systems, IEEE Transactions on*, vol. 43, no. 4, pp. 875–885, 2013.

[6] V. Bolón-Canedo, N. Sánchez-Maroño, A. Alonso-Betanzos, J. Benítez, and F. Herrera, "A review of microarray datasets and applied feature selection methods," *Information Sciences*, vol. 282, pp. 111 – 135, 2014.

[7] V. Bolón-Canedo, N. Sánchez-Marono, and A. Alonso-Betanzos, "Recent advances and emerging challenges of feature selection in the context of big data," *Knowledge-Based Systems*, vol. 86, pp. 33 – 45, 2015.
[8] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI 2004*, 2004, pp. 137–150.
[9] T. White, *Hadoop, The Definitive Guide*. O'Reilly Media, Inc., 2012.
[10] Apache Hadoop Project, "Apache Hadoop," 2017, [Online; accessed February 2017]. [Online]. Available: http://hadoop.apache.org/
[11] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark: Lightning-Fast Big Data Analytics*. O'Reilly Media, Incorporated, 2015.
[12] A. Spark, "Apache Spark: Lightning-fast cluster computing," 2017, [Online; accessed February 2017]. [Online]. Available: https://spark.apache.org/
[13] Apache Mahout Project, "Apache Mahout," 2017, [Online; accessed February 2017]. [Online]. Available: http://mahout.apache.org/
[14] A. Spark, "Machine Learning Library (MLlib) for Spark," 2017, [Online; accessed February 2017]. [Online]. Available: http://spark.apache.org/docs/latest/mllib-guide.html
[15] K. Sun, W. Miao, X. Zhang, and R. Rao, "An improvement to feature selection of random forests on spark," in *IEEE 17th International Conference on Computational Science and Engineering (CSE)*, 2014, pp. 774–779.
[16] G. Brown, A. Pocock, M.-J. Zhao, and M. Luján, "Conditional likelihood maximisation: A unifying framework for information theoretic feature selection," *Journal of Machine Learning Research*, vol. 13, pp. 27–66, Jan. 2012.
[17] 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.
[18] R. Kohavi and G. H. John, "Wrappers for feature subset selection," *Artificial Intelligence*, vol. 97, no. 1-2, pp. 273–324, 1997.
[19] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of Machine Learning Research*, vol. 3, pp. 1157–1182, 2003.
[20] Y. Saeys, I. n. Inza, and P. Larrañaga, "A review of feature selection techniques in bioinformatics," *Bioinformatics*, vol. 23, no. 19, pp. 2507–2517, 2007.
[21] V. Mayer-Schnberger and K. Cukier, *Big Data: A Revolution That Will Transform How We Live, Work and Think. Viktor Mayer-Schnberger*. John Murray Publishers, 2013.
[22] D. Laney, "3d data management: Controlling data volume, velocity and variety," 2001, [Online; accessed February 2017]. [Online]. Available: http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf
[23] Y. Zhai, Y. Ong, and I. W. Tsang, "The emerging "big dimensionality"," *IEEE Computational Intelligence Magazine*, vol. 9, no. 3, pp. 14–26, 2014.
[24] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml
[25] W. Roush, "TR10: Peering into Video's Future," *MIT Technology Review March 12*, 2013. [Online]. Available: http://www.technologyreview.com/Infotech/18284/
[26] Q. Mao and I. W.-H. Tsang, "Efficient multitemplate learning for structured prediction." *IEEE Transactions on Neural Networks and Learning Systems*, vol. 24, no. 2, pp. 248–261, 2013.
[27] T. Wang, W. Zhang, C. Ye, J. Wei, H. Zhong, and T. Huang, "FD4C: Automatic fault diagnosis framework for web applications in cloud computing," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 46, no. 1, pp. 61–75, 2016.
[28] A. Fernández, S. del Río, V. López, A. Bawakid, M. J. del Jesús, J. M. Benítez, and F. Herrera, "Big data with cloud computing: an insight on the computing environment, mapreduce, and programming frameworks," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 4, no. 5, pp. 380–409, 2014.
[29] J. Lin, "Mapreduce is good enough? if all you have is a hammer, throw away everything that's not a nail!" *CoRR*, vol. abs/1209.2191, 2012.
[30] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "Mllib: Machine learning in apache spark," *Journal of Machine Learning Research*, vol. 17, no. 34, pp. 1–7, 2016.
[31] M. Michael and W.-C. Lin, "Experimental study of information measure and inter-intra class distance ratios on feature selection and orderings," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. SMC-3, no. 2, pp. 172–181, 1973.
[32] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 1991.
[33] R. Battiti, "Using mutual information for selecting features in supervised neural net learning," *Neural Networks, IEEE Transactions on*, vol. 5, no. 4, pp. 537–550, 1994.
[34] S. del Río, V. López, J. M. Benítez, and F. Herrera, "On the use of mapreduce for imbalanced big data using random forest." *Information Sciences*, vol. 285, no. 285, pp. 112–137, 2014.
[35] H. Peng, F. Long, and C. Ding, "Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 27, no. 8, pp. 1226–1238, 2005.
[36] M. A. Hearst, S. T. Dumais, E. Osman, J. Platt, and B. Scholkopf, "Support vector machines," *Intelligent Systems and their Applications, IEEE*, vol. 13, no. 4, pp. 18–28, 1998.
[37] R. O. Duda and P. E. Hart, *Pattern classification and scene analysis*. Wiley New York, 1973, vol. 3.