Feature Selection in High-Dimensional Dataset Using MapReduce

Chapter *in* Communications in Computer and Information Science · February 2018

DOI: 10.1007/978-3-319-76892-2_8

3 authors:

Claudio Reggiani
Université Libre de Bruxelles
6 PUBLICATIONS **63** CITATIONS
SEE PROFILE

Yann-Aël Le Borgne
Université Libre de Bruxelles
82 PUBLICATIONS **2,664** CITATIONS
SEE PROFILE

Gianluca Bontempi
Université Libre de Bruxelles
390 PUBLICATIONS **19,750** CITATIONS
SEE PROFILE

Feature selection in high-dimensional dataset using MapReduce

Claudio Reggiani, Yann-Aël Le Borgne, and Gianluca Bontempi, *Senior Member, IEEE*

***Abstract*—Feature selection is a critical step for most data mining applications. This paper describes a distributed implementation of the minimum Redundancy Maximum Relevance algorithm, using the MapReduce programming model. The proposed approach handles both *tall/narrow* and *wide/short* datasets. We further provide an open source implementation based on Hadoop/Spark, and illustrate its scalability on datasets involving millions of observations or features.**

***Index Terms*—Feature Selection, MapReduce, Apache Spark, mRMR, Scalability.**

I. Introduction

The exponential growth of data generation, measurements and collection in scientific and engineering disciplines leads to the availability of huge and high-dimensional datasets, in domains as varied as text mining, social network, astronomy or bioinformatics to name a few. The only viable path to the analysis of such datasets is to rely on data-intensive distributed computing frameworks [1].

MapReduce has in the last decade established itself as a reference programming model for distributed computing. The model is articulated around two main classes of functions, *mappers* and *reducers*, which greatly decrease the complexity of a distributed program while allowing to express a wide range of computing tasks. MapReduce was popularised by Google research in 2008 [2], and may be executed on parallel computing platforms ranging from specialised hardware units such as parallel field programmable gate arrays (FPGAs) and graphics processing units, to large clusters of commodity machine using for example the Hadoop or Spark frameworks [2], [3], [4].

In particular, the expressiveness of the MapReduce programming model has led to the design of advanced distributed data processing libraries for machine learning and data mining, such as Hadoop Mahout and Spark MLLib. Many of the standard supervised and unsupervised learning techniques (linear and logistic regression, naive Bayes, SVM, random forest, PCA) are now available from these libraries [5], [6], [7].

Little attention has however yet been given to feature selection algorithms (FSA), which form an essential component of machine learning and data mining workflows. Besides reducing a dataset size, FSA also generally allow to improve the performance of classification and regression models by selecting the most relevant features and reducing the noise in a dataset [8].

C. Reggiani, Y. Le Borgne, G. Bontempi are with the Machine Learning Group, Faculty of Science, Université Libre de Bruxelles, Boulevard du Triomphe, CP 212, 1050 Brussels, Belgium e-mail: creggian,yleborgn,gbonte@ulb.ac.be

Three main classes of FSA can be distinguished: *filter*, *wrapper* and *embedded* methods [8], [9]. Filter methods are model-agnostic, and rank features according to some metric of information conservation such as mutual information or variance. Wrapper methods use the model as a black-box to select the most relevant features. Finally, embedded methods build the feature selection process in the training algorithm of the specific model being used.

Early research on distributing FSA mostly concerned wrapper methods, in which parallel processing was used to simultaneously assess different subsets of variables [10], [11], [12], [13], [14]. These approaches effectively speed up the search for relevant subsets of variables, but require the dataset to be sent to each computing unit, and therefore do not scale as the dataset size increases.

MapReduce based approaches, which address this scalability issue by splitting datasets in chunks, have more recently been proposed [15], [16], [17], [18], [19], [20]. In [15], an embedded approach is proposed for logistic regression. Scalability in the dataset size is obtained by relying on an approximation of the logistic regression model performance on subsets of the training set. In [16], a wrapper method based on an evolutionary algorithm is used to drive the feature search. The first approaches based on filter methods were proposed in [17], [18], using variance preservation and mutual information as feature selection metrics, respectively. Finally, two other implementations of filter-based methods have lately been proposed, addressing the column subset selection problem (CSSP) [19], and the distribution of data by features in [20].

In this paper we tackle the implementation of *minimal Redundancy Maximal Relevance* (*mRMR*) [21], a forward feature selection algorithm belonging to filter methods. mRMR was shown to be particularly effective in the context of network inference problems, where relevant features have to be selected out of thousands of other noisy features [1], [22].

The main contributions of the paper are the following: *i*) design of minimum Redundancy Maximum Relevance algorithm using MapReduce paradigm; *ii*) customization of the feature score function during the feature selection; *iii*) open-source implementation for Apache Spark available on a public repository; *iv*) scalability results of the algorithm.

The paper is structured as follows. Section II provides an overview of the MapReduce paradigm, and Section III describes the two main layouts along which data can be stored. Section IV presents our distributed implementation of mRMR, and details how it can be used with customised scoring functions. Section V finally provides a thorough experimental evaluation, where we assess the scalability of the proposed implementation by varying the number of rows and columns of the datasets, the number of selected features in the feature selection step and the number of nodes in the cluster.

II. MapReduce paradigm

MapReduce [2] is a programming paradigm designed to analyse large volumes of data in a parallel fashion. Its goal is to process data in a scalable way, and to seamlessly adapt to the available computational resources.

A MapReduce job transforms lists of input data elements into lists of output data elements. This process happens twice in a program, once for the *Map* step and once for the *Reduce* step. Those two steps are executed sequentially, and the Reduce step begins once the Map step is completed.

In the Map step, the data elements are provided as a list of key-value objects. Each element of that list is loaded, one at a time, into a function called *mapper*. The mapper transforms the input, and outputs any number of intermediate key-value objects. The original data is not modified, and the mapper output is a list of new objects.

In the Reduce step, intermediate objects that share the same key are grouped together by a *shuffling* process, and form the input to a function called *reducer*. The reducer is invoked as many times as there are keys, and its value is an iterator over the related grouped intermediate values.

Mappers and reducers run on some or all of the nodes in the cluster in a isolated environment, i.e. each function is not aware of the other ones and their task is equivalent in every node. Each mapper loads the set of files local to that machine and processes it. This design choice allows the framework to scale without any constraints about the number of nodes in the cluster. An overview of the MapReduce paradigm is reported in Figure 1.

Algorithms written in MapReduce scale with the cluster size, and Execution Time (ET) can be decreased by increasing the number of nodes. The design of the algorithm and the data layout are important factors impacting ET.

This is the case of the *equi-join* relational operator in MapReduce, where two of the most commonly used join strategies are *Repartition Join* and *Broadcast Join*. The former performs the join operation on the reducers, therefore requiring all data to be moved across the network. The latter broadcasts the smaller table across all nodes, and the operation is run as map-only job. It therefore avoids the transmission of the larger table across the network [23].

In ET terms, jobs perform better in MapReduce when transformations are executed locally during the Map step, and when the amount of information transferred during the shuffling step is minimised. In particular, MapReduce is very well-suited for associative and commutative operators, such as *sum* and *multiplication*. These can indeed be partially

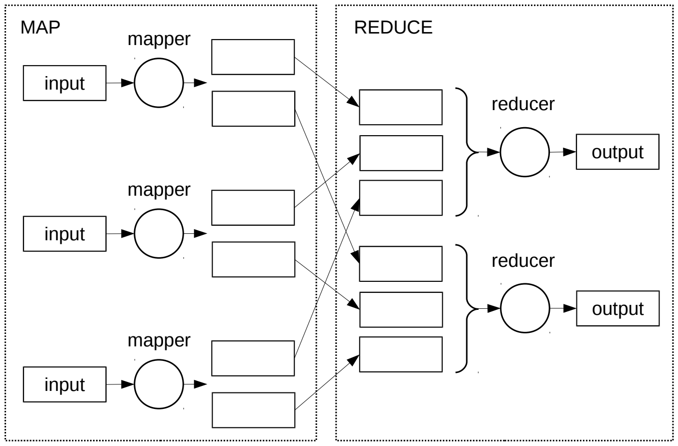

Fig. 1. MapReduce overview of the data flow. The dataset stored in the distributed storage system is split in chunks across nodes (rectangular *input* boxes). Each chunk is fed as input to the mapper functions, which may output intermediate objects. These objects are shuffled and grouped by keys across the network. Finally, the reducers generate the groups of intermediate objects and output the results. All objects (input, intermediate, output) are identified by a key-value pair.

processed using an intermediate *Combine* step, which can be applied between the Map and Reduce stages.

The combiner is an optional functionality in MapReduce. It locally aggregates mapper output objects before they are sent over the network. It operates by taking as input the intermediate key-value objects produced by the mappers, and output key-value pairs for the Reduce step. This optional process allows to reduce data transfer over the network, therefore reducing the global ET of the job. An illustration of the use and advantages of the combiner is given in Figure 2.

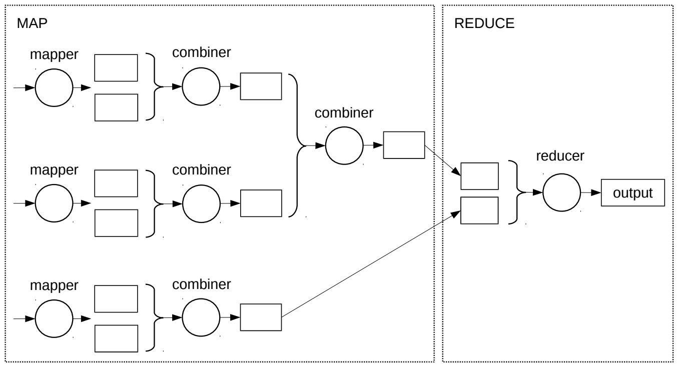

Fig. 2. MapReduce overview of the data flow with the additional combiner function. Intermediate objects produced by mappers are locally aggregated by a commutative and associative function implemented in the combiner logic. In this example two, instead of six, intermediate objects are sent over the network to the reducer function. Combiners provide an efficient way to reduce the amount of shuffled data, and to reduce the overall execution time of the job.

III. Data Layout

In learning problems, training data from a phenomenon is usually encoded in tables, using rows as observations, and columns as features. Let M be the number of observations, and

$x_{1,1}$	$x_{1,2}$	$\dots$	$\dots$	$x_{1,N}$
$x_{2,1}$	$x_{2,2}$	$\dots$	$\dots$	$x_{2,N}$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$x_{M,1}$	$x_{M,2}$	$\dots$	$\dots$	$x_{M,N}$

TABLE I
TRADITIONAL ENCODING: OBSERVATIONS ($x_{i,\cdot}$) ARE STORED ALONG ROWS, AND FEATURES ($x_{\cdot,j}$) ARE STORED ALONG COLUMNS.

$x_{1,1}$	$x_{2,1}$	$\dots$	$\dots$	$x_{M,1}$
$x_{1,2}$	$x_{2,2}$	$\dots$	$\dots$	$x_{M,2}$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$x_{1,N}$	$x_{2,N}$	$\dots$	$\dots$	$x_{M,N}$

TABLE II
ALTERNATIVE ENCODING: OBSERVATION ($x_{i,\cdot}$) ARE STORED ALONG COLUMNS, AND FEATURES ($x_{\cdot,j}$) ARE STORED ALONG ROWS.

N be the number of features. Training data can be represented as a collection of feature vectors, $\mathbf{X}$,

$$\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_M)$$

where

$$\mathbf{x}_j = (x_{j,1}, x_{j,2}, \dots, x_{j,N}) \qquad \forall j \in (1, \dots, M)\,.$$

We will refer to this type of structure as the *traditional* encoding, see Table I.

It is however worth distinguishing two types of tables: *tall and narrow* (T/N) tables, where $M \gg N$, and *short and wide* (S/W) tables, where $M \ll N$.

The distinction is important since MapReduce divides input data in chunks of rows, that are subsequently processed by the mappers. MapReduce is therefore well suited to ingest T/N table, but much less for S/W tables, since data cannot be efficiently split along rows. S/W tables are for example encountered in domains such as text mining or bioinformatics, where the number of features can be on the order of tens or hundreds of thousands, while observations may only be on the order of hundreds or thousands.

In such cases, it can be beneficial to transform S/W into T/N tables, by storing observations as columns and features as rows. We refer to this type of structure as *alternative* encoding, see Table II.

IV. ITERATIVE FEATURE SELECTION FRAMEWORK

The main drawback of the alternative encoding is the impossibility of using most (if not all) of the already developed packages and libraries available for MapReduce (e.g. MLlib of Apache Spark). We therefore present here a framework for iterative feature selection for *alternative* encoded data, and provide the implementation of mRMR for both encoding schemas.

A. minimal Redundancy Maximal Relevance

In this section, we first briefly review how mRMR works. We then detail our approach for distributing mRMR using MapReduce paradigm. Let us define the dataset as the table $\mathbf{X}$ with M rows, N columns and discrete values. We define $\mathbf{x}_k$ as the $k-th$ column vector of the dataset and $\mathbf{c}$ as the class vector. Furthermore, let us define L as the number of features to select and i_c^l and i_s^l as the sets at step l ($0 \leqslant l < L$) of candidate and selected features indices, respectively. At $l = 0$, we have $i_c^0 = \{0, 1, ..., N-1\}$ and $i_s^0 = \varnothing$. The pseudo-code of the algorithm is reported in Listing 1.

Listing 1. minimum Redundancy Maximum Relevance Pseudo-code. $I(\cdot)$ is the function that, given two vectors, returns their mutual information. $\mathbf{x}_k$ is the $k-th$ column vector of the dataset and $\mathbf{c}$ is the class vector. L is the number of features to select, i_c^l and i_s^l as the sets at step l ($0 \leqslant l < L$) of candidate and selected features indexes.

```
$i_c^0 = \{0, 1, ..., N-1\}$
$i_s^0 = \varnothing$
for $l = 0 \to L-1$
  for $k \in i_c^l$
    compute $I_{\mathbf{x}_k,\mathbf{c}} \leftarrow I(\mathbf{x}_k, \mathbf{c})$
    for $j \in i_s^l$
      compute $I_{\mathbf{x}_k,\mathbf{x}_j} \leftarrow I(\mathbf{x}_k, \mathbf{x}_j)$
    $g_k \leftarrow \begin{cases} I_{\mathbf{x}_k,\mathbf{c}} - \sum_{j \in i_s^l} I_{\mathbf{x}_k,\mathbf{x}_j} & l = 1 \\ I_{\mathbf{x}_k,\mathbf{c}} - \frac{1}{|i_s^l|-1} \sum_{j \in i_s^l} I_{\mathbf{x}_k,\mathbf{x}_j} & l > 1 \end{cases}$
  $k^* \leftarrow$ argmax($g_k$)
  $i_c^{l+1} \leftarrow i_c^l \backslash k^*$
  $i_s^{l+1} \leftarrow i_s^l \cup k^*$
output $i_s^L$
```

mRMR is an iterative greedy algorithm: at each step the candidate feature is selected based on a combination of its mutual information with the class and the selected features:

$$\mathrm{argmax}_{k \in i_c^l}\, g_k(\cdot)$$

$$g_k(\cdot) = \begin{cases} I(\mathbf{x}_k; \mathbf{c}) & l = 0 \\ I(\mathbf{x}_k; \mathbf{c}) - \sum_{j \in i_s^l} I(\mathbf{x}_k; \mathbf{x}_j) & l = 1 \\ I(\mathbf{x}_k; \mathbf{c}) - \frac{1}{|i_s^l|-1} \sum_{j \in i_s^l} I(\mathbf{x}_k; \mathbf{x}_j) & l > 1 \end{cases} \tag{1}$$

The *feature score* $g(\cdot)$ is assessed in Lines 5-8 in Listing 1.

We redesigned the algorithm using MapReduce paradigm on Apache Spark, distributing the feature evaluation into the cluster.

B. mRMR in MapReduce with traditional encoding

Let us define the dataset as a Resilient Distributed Dataset (RDD) [24] of M tuples $(\mathbf{x}, c)$, where $\mathbf{x}$ is the input (observation) vector and c is the target class value.

Considering the dataset with only discrete values, we represent with d_c the set of categorical values of the class, and with d_v the (union) set of unique categorical values of all features. If the dataset has binary values, then $d_c = d_v = \{0, 1\}$. In case of having features with different sets of categorical values, then d_v is the union of unique categorical values of all features.

The input vector is partitioned in candidate and selected features, labeled respectively as $\mathbf{x}_c$ and $\mathbf{x}_s$ ($\mathbf{x}_c \cup \mathbf{x}_s = \mathbf{x}$, $|\mathbf{x}| = N$). Variables L, i_c^l and i_s^l are defined as in the previous section and i_{class} is the class column index. Listings 2, 3 and 4 report the MapReduce job, the mapper and reducer functions, respectively, while an illustrative overview of the data flow is reported in Fig. 3.

#entry	class	features			
	c	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{x}_4$
1	0	2	0	0	-2
2	0	0	-2	2	0
3	0	0	2	0	-2
4	1	-2	0	0	0
...	...	...	...	...	...

TABLE III
EXAMPLE OF DATASET ENCODED WITH TRADITIONAL LAYOUT. THE DATASET HAS ONE BINARY CLASS COLUMN AND FOUR CATEGORICAL FEATURES, WITH THREE POSSIBLE VALUES: -2,0,2.

		d_v		
		-2	**0**	**2**
d_c	**0**	0	0	1
	1	0	0	0

TABLE IV
CONTINGENCY TABLE EMITTED BY THE MAPPER FUNCTION AS A RESULT OF PROCESSING THE PAIR $(\mathbf{x}_1, c)$ OF THE FIRST ENTRY IN TABLE III. IN THIS EXAMPLE THE CLASS VECTOR CAN ONLY HAVE TWO POSSIBLE VALUES: *0* AND *1*; ANY FEATURE CAN ONLY HAVE THREE POSSIBLE VALUES: *-2*, *0* AND *2*. THE INPUT PAIR $(e_k, e_{i_{class}})$ IS $(2, 0)$, THEREFORE THE ELEMENT CORRESPONDING TO ROW *0* AND COLUMN *2* IS SET TO 1, ALL THE OTHERS ARE SET TO 0.

		d_v		
		-2	**0**	**2**
d_c	**0**	0	2	1
	1	1	0	0

TABLE V
AGGREGATED CONTINGENCY TABLE EMITTED BY THE COMBINER FUNCTION AS A RESULT OF PROCESSING THE PAIR $(\mathbf{x}_1, c)$ OF THE FIRST FOUR ENTRIES IN TABLE III. THE CLASS VECTOR CAN ONLY HAVE TWO POSSIBLE VALUES: *0* AND *1*; ANY FEATURE CAN ONLY HAVE THREE POSSIBLE VALUES: *-2*, *0* AND *2*. CONSIDERING FOUR VALUE PAIRS COMING FROM THE CANDIDATE FEATURE VECTOR AND THE CLASS VECTOR: $(2, 0)$, $(0, 0)$, $(0, 0)$ AND $(-2, 1)$, EACH PAIR IS FIRST TRANSFORMED IN A CONTINGENCY TABLE AND THE AGGREGATED BY A ELEMENT-WISE SUM OF THE TABLES.

Listing 2. mRMR MapReduce job with traditional data encoding. L is the number of features to select, i_c^l and i_s^l are the sets at step l $(0 \leqslant l < L)$ of candidate and selected features indices. i_{class} is the class column index. d_c is the set of categorical values of the class, and d_v is the (union) set of unique categorical values of all features.

```
$i_c^0 = \{0, 1, ..., N-1\}$
$i_s^0 = \varnothing$
for $l = 0 \to L-1$
  broadcast $i_{class}$, $i_c^l$, $i_s^l$, $d_v$, $d_c$
  scores <- mapreduce(RDD, Map, Reduce)
  $k^*$ ← argmax(scores)
  $i_c^{l+1} \leftarrow i_c^l \backslash k^*$
  $i_s^{l+1} \leftarrow i_s^l \cup k^*$
output $i_s^L$
```

Listing 3. mRMR MapReduce mapper function with traditional data encoding. i_c^l and i_s^l as the sets at step l $(0 \leqslant l < L)$ of candidate and selected features indexes. i_{class} is the class column index. d_c is the set of categorical values of the class, and d_v is the (union) set of unique categorical values of all features. e is a single observation fed as input to the mapper, k and j represent column indices and *contTable* is the function that creates a contingency table.

```
# broadcasted vars: $i_{class}$, $i_c^l$, $i_s^l$, $d_v$, $d_c$
mapper(·, $e$)
  for $k \in i_c^l$
    output ($k$, contTable($e_k$, $e_{i_{class}}$, $d_v$, $d_c$))
    for $j \in i_s^l$
      output ($k$, contTable($e_k$, $e_j$, $d_v$, $d_c$))
```

Listing 4. mRMR MapReduce reducer function with traditional data encoding. k is a column index and t is a collection of contingency tables. The *mrmr_score* function process all the contingency tables associated with the column with index k and return the feature score.

```
reducer($k$, $t$)
  output($k$, mrmr_score($t$))
```

For every $(e_k, e_{i_{class}})$ pair, the mapper task outputs a contingency table, *contTable*, with rows defined as the categorical values in d_c and columns defined as the categorical values in d_v. The element corresponding to row $e_{i_{class}}$ and column e_k is set to 1, while all the others are set to 0. Using the dataset in Table III, an example of emitted contingency table is reported in Table IV. In case of (e_k, e_j) pair, the contingency table has both rows and columns defined by categorical values in d_v.

At the cost of managing discrete values only, the introduction of the contingency table minimizes the amount of data exchanged across the cluster during shuffling. Such data structure benefits of the commutative and associative properties (required by the combiner function). While the single mapper outputs one or more contingency tables for each candidate feature, those tables emitted by mappers executed on a given node can be locally reduced via the Combine step. Assuming that the first four entries in Table III are processed by four mappers in the same machine, Table V is the result of the combiner after the aggregation of four contingency tables of the $\mathbf{x}_1$ feature produced by the mappers. In this example, the combiner performs an element-wise sum of the contingency tables given as input.

At scale, considering M entries distributed across C nodes, N candidate features and S selected features, this design exchanges $N{*}(S{+}1){*}C$ instead of $N{*}(S{+}1){*}M$ contingency tables during shuffling, where typically $M \gg C$.

C. mRMR in MapReduce with alternative encoding

Data stored in alternative encoding has one column per observation and one row per feature. In this case, let us define the dataset as a RDD of M tuples $(\mathbf{x}, lab)$, where $\mathbf{x}$ is the feature vector and *lab* is the feature name, assumed to be unique among all the other feature's names. Feature and class values could be discrete and continuous as well. With respect to the design of mRMR in MapReduce with traditional encoding, a set of vectors are broadcasted across the cluster: $\mathbf{v}_{class}$ is the class vector, v_s is the collection of selected feature vectors and v_{ns} is the collection of selected feature names. Variable L is defined as in the previous section and *getEntry* function is a MapReduce task that retrieves the feature vector from the RDD, given a feature name. Listings 5 and 6 report the MapReduce job and the mapper function, respectively.

Listing 5. mRMR MapReduce job with alternative data encoding. *RDD* represents the distributed dataset and L is the number of features to select. $\mathbf{v}_{class}$ is the class vector, v_s is the collection of selected feature vectors and v_{ns} is the collection of selected feature names. The *getEntry* function retrieves the feature vector from the RDD, given a feature name.

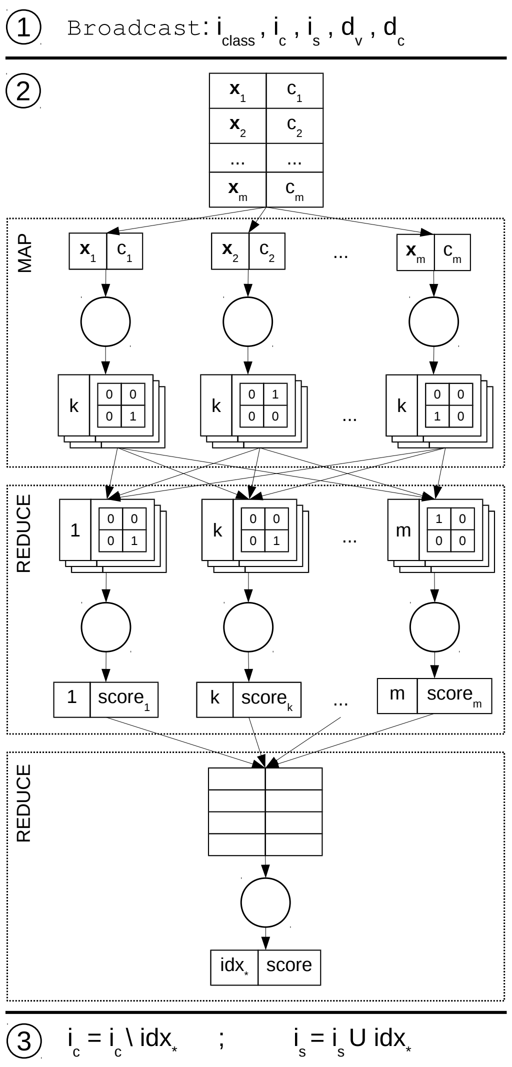
Broadcast: i_{class}, i_c, i_s, d_v, d_c
MAP
REDUCE
REDUCE
idx* score
$i_c = i_c \setminus idx_*$; $i_s = i_s \cup idx_*$

Fig. 3. Illustrative representation of a single iteration of a MapReduce job with discrete values using the traditional encoding. Steps 1-3 represent one iteration of the main loop. There are as many iterations as the number of features to select. Each iteration consists of three main steps: 1. Broadcast the class, candidate and selected column indices (i_{class}, i_c and i_s), along with the categorical values of the class and the features (d_c, d_v); 2. Calculate the feature score per each candidate feauture and find the column index (idx_*) with best score (our implementation also takes advantage of the combiner which is not shown in the figure); 3. Update the broadcasted variables and reiterate.

```
$v^0_{ns} = \varnothing$
$v^0_s = \varnothing$
for $l = 0 \to L-1$
  broadcast $\mathbf{v}_{class}$, $v^l_{ns}$, $v^l_s$
  mrmr_scores <- mapreduce(RDD, Map)
  $vn^* \leftarrow$ argmax(mrmr_scores)
  $v^*$ <- getEntry(RDD, $vn^*$)
  $v^{l+1}_{ns} \leftarrow v^l_{ns} \cup vn^*$
  $v^{l+1}_s \leftarrow v^l_s \cup v^*$
output $v^L_{ns}$
```

Listing 6. mRMR MapReduce mapper function with alternative data encoding. $\mathbf{v}_{class}$ is the class vector, v_s is the collection of selected feature vectors. The tuple $(\mathbf{x}, lab)$ is composed by the feature vector, $\mathbf{x}$, and the feature name, *lab*. The *mrmr_score* function processes the vectors and returns the feature score.

```
# broadcasted variables: $\mathbf{v}_{class}$, $v^l_s$
mapper(·, ($\mathbf{x}$, lab))
  score <- mrmr_score($\mathbf{x}$, $\mathbf{v}_{class}$, $v^l_s$)
  output (lab, score)
```

While in traditional encoding we used the contingency table as intermediate data structure, the design of mRMR in MapReduce with alternative encoding broadcasts at each iteration all required data for calculation to mappers. This design provides two main advantages: it deals with both discrete and continuous features as well, and the MapReduce job is composed by the Map step only. At the small cost of broadcasting some variables, all operations are executed locally. An illustrative overview of the data flow is reported in Fig. 4.

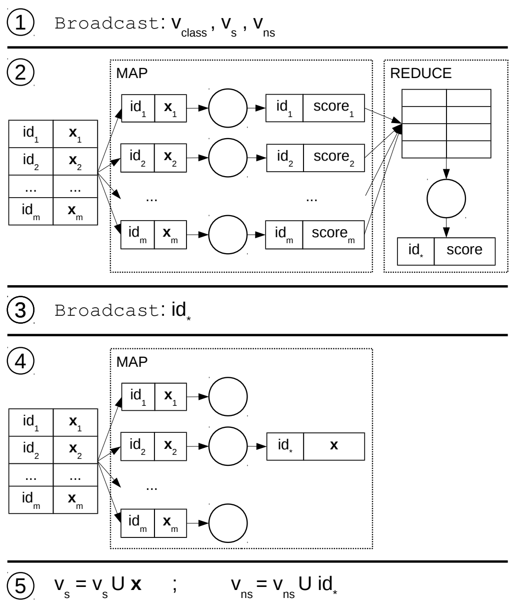
Broadcast: v_{class}, v_s, v_{ns}
MAP
REDUCE
id* score
Broadcast: id*
MAP
id* x
$v_s = v_s \cup \mathbf{x}$; $v_{ns} = v_{ns} \cup id_*$

Fig. 4. Illustrative representation of a single iteration of a MapReduce job with alternative encoding. Steps 1-5 represent one iteration of the loop; there are as many iterations as the number of features to select. Each iteration consists of five main steps: 1. broadcast the class, the selected feature vectors and the selected feature names (v_{class}, v_s, v_{ns}); 2. for every candidate feature (i.e. every row in the dataset) calculates the feature score value using the function $g(\cdot)$ in Formula 1 or a custom one; 3. broadcast the feature label (id_* in the figure) with best score; 4. retrieve the feature vector, $\mathbf{x}$, using a second MapReduce job; 5. update the broadcasted variables and reiterate.

D. Custom score with alternative enconding

By introducing mRMR in MapReduce with alternative encoding, we propose a solution to store and analyse high-dimensional datasets in distributed environments. While mRMR is a well-known feature selection method, some problems might require a different algorithm or feature score function to perform the selection of features. For this reason, we provide an interface to customize the feature score function in the alternative encoding, Listing 7. By means of the *getResult* function, the interface provides at each iteration of the algorithm three variables: the candidate feature vector *variableArray*, the class vector *classArray* and the collection of selected feature vectors *selectedVariablesArray* (respectively $\mathbf{x}$, $\mathbf{v}_{class}$ and v^l_s in Listing 6). The function has to return a scalar value representing the feature score for the candidate feature at such iteration.

Listing 7. Scala interface for custom feature score.

```
function getResult:
  arguments:
    variableArray: Array[Double]
    classArray: Array[Double]
    selectedVariablesArray:
          Array[Array[Double]]
  return: Double
```

The mRMR design in MapReduce with alternative encoding described in this paper is implemented using the *getResult* function. Its mathematical representation (as described in the original paper [21]) is reported in Formula 1, where $g(\cdot)$ defines the feature score function.

To illustrate the flexibility of the framework for feature selection with alternative encoding, we provide an example of custom feature score function implementation: an approximation of the mutual information, $f(\cdot)$, by means of the Pearson correlation coefficient (Formula 2 and Listing 8).

$$\text{argmax}_{k \in i_c^l} \, g_k(\cdot)$$

$$g_k(\cdot) = \begin{cases} f(\mathbf{x}_k;\mathbf{c}) & l = 0 \\ f(\mathbf{x}_k;\mathbf{c}) - \sum_{j \in i_s^l} f(\mathbf{x}_k;\mathbf{x}_j) & l = 1 \\ f(\mathbf{x}_k;\mathbf{c}) - \frac{1}{|i_s^l|-1} \sum_{j \in i_s^l} f(\mathbf{x}_k;\mathbf{x}_j) & l > 1 \end{cases} \tag{2}$$

Listing 8. Pseudo-code implementation of the *getResult* interface for customizing the feature score, based on Pearson correlation coefficient instead of mutual information. *pcc* calculates the Pearson correlation given two input vectors. The *size* function returns the number of elements in the collection.

```
cor2mi(v)
  -0.5 * log( 1.0-(v*v) )

getResult(variableArray, classArray,
     selectedVariablesArray)
  val v <- variableArray
  val clv <- classArray
  val sva <- selectedVariablesArray

  sc <- cor2mi(pcc(v, clv))

  sfs <- 0
  for i=0 → size(sva)
    sv <- sva[i]
    sfs <- sfs + cor2mi(pcc(v, sv))

  coeff <- 1.0
  if size(sva) > 1
    coeff <- 1.0 / size(sva)

  return sc - (coeff * sfs)
```

The interface should be implemented as a third-party library, packaged as a *jar* file, and provided as input during the spark submission job. The implementation of the custom score which approximates the mutual information by means of the Pearson correlation coefficient is provided as an example in the public repository (https://github.com/creggian/mrmr).

V. Results

We studied the scalability of the implementation of mRMR in MapReduce in both encodings in a cluster with the following specifications: hadoop cluster of 10 nodes, where each node has Dual Xeon e5 2.4Ghz processor, 24 cores, 128GB RAM and 8TB hard disk; all nodes are connected with a 1Gb ethernet connection. Using Apache Spark v1.5.0, we submit jobs with 4GB of RAM for both driver and executors.

For the evaluation of mRMR implementations we used binary artificial datasets. We followed the principles of CorrAL dataset [25], in which four features determine the class value with the following formula: $c = (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$, one is irrelevant and the last one is partially correlated with the class. In all our datasets, the class value (c) depends on the value of 8 features (Formula 3); the remainings are irrelevant.

$$c = ((x_1 \wedge x_2) \vee (x_3 \wedge x_4)) \wedge ((x_5 \wedge x_6) \vee (x_7 \wedge x_8)) \tag{3}$$

We performed four kinds of tests: by increasing the number of rows, the number of columns, the number of selected features, the number of nodes. We used two kinds of dependent variables: the relative execution time per executor and the computational gain. The former is the ratio between ET divided by ET of `1x`, the latter is the ratio between ET of 1-node and ET. We ran the tests three times to show the variability of the results; in the figures, values from indentical experimental settings are reported using a vertical bar and lines connect the average values.

A. Scalability across the number of rows

We tested the scalability across the number of rows by means of four datasets: they have 1000 columns and an increasing number of rows: 1M, 4M, 7M and 10M (M = millions). We configured the cluster and the algorithm to select 10 features in a distributed environment of 10 nodes (Fig. 5).

B. Scalability across the number of columns

We evaluated the scalability across the number of columns using four datasets: they have 1M rows and an increasing number of columns: 100, 400, 700 and 1000. We configured the cluster and the algorithm to select 10 features in a distributed environment of 10 nodes (Fig. 6).

C. Scalability across the number of selected features

We investigated the scalability across the number of selected features using a dataset with 1M rows and 50k (k = thousands) columns. We configure the cluster to distribute the work over 10 nodes, and the algorithm to select an increasing number of features: 1, 2, 4, 6, 10 (Fig. 7).

D. Scalability across the number of nodes

We tested the scalability across the number of nodes using a dataset with 1M rows and 100 columns. We configure the algorithm to select 10 features, and the cluster to distribute the work over 1, 2, 5 and 10 nodes (Fig. 8).

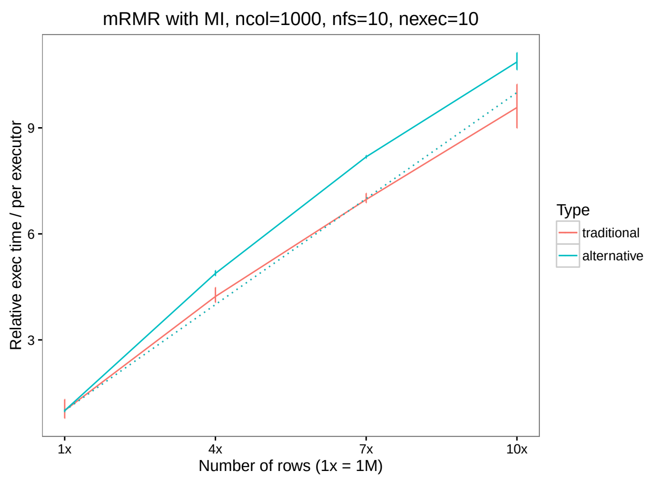

Fig. 5. mRMR scalability across the number of rows.

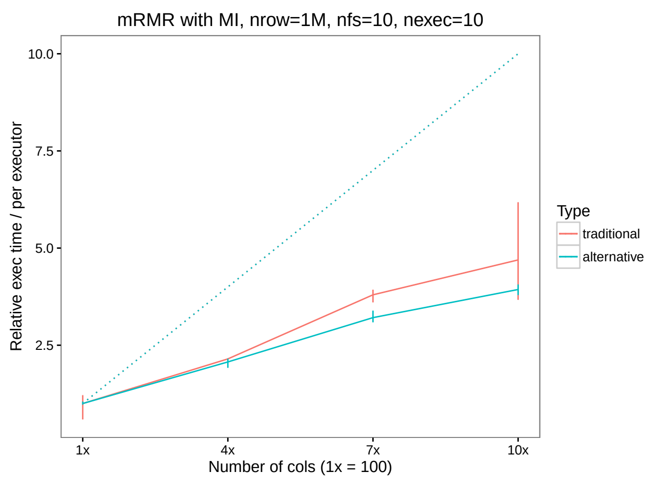

Fig. 6. mRMR scalability across the number of columns.

By comparing the linear scalability (dotted line) with the actual performances, results show that the scalability of mRMR in MapReduce is linear with respect to the number of rows, as expected by MapReduce design; superlinear with respect to the number of columns; sublinear with respect to the number of selected features and nodes, as expected by our iterative algorithm design and the increasing amount of data exchanged in the network with the increasing of nodes, respectively.

In studying mRMR with traditional and alternative layouts, we chose to use as independent variable the number of rows (columns) instead of the number of observation (features) for the following reason: while in the traditional layout we are able to scale across a very large number of rows, in the alternative layout we are strictly constraint by the amount of memory available in the mapper task to scale across the number of columns. In Figures 5 and 6, we tested up to 10 million rows and up to one thousand columns, because very high-dimensional S/W tables raises memory errors in the cluster. Hence, even though we show the relative execution time, it would be incorrect to plot performances by increasing the number of observation (features).

The absolute execution time of mRMR MapReduce jobs with alternative encoding is generally 4-6x faster than the respective jobs with traditional encoding.

VI. Conclusion

In this work we investigated the design and scalability of mRMR algorithm in MapReduce. We proposed two implementations depending on the data layout and we provide an interface in order to customize the feature score function in the alternative encoding scenario. Despite hadoop limitations on handling data with a large number of columns, the alternative data layout is a solution to store data from a phenomenon that has a very large number of features. In both traditional and alternative data layouts, we studied the scalability of mRMR in different settings: the number of rows, columns, selected features and nodes. The results give an overview of the performance of the algorithm implemented in MapReduce.

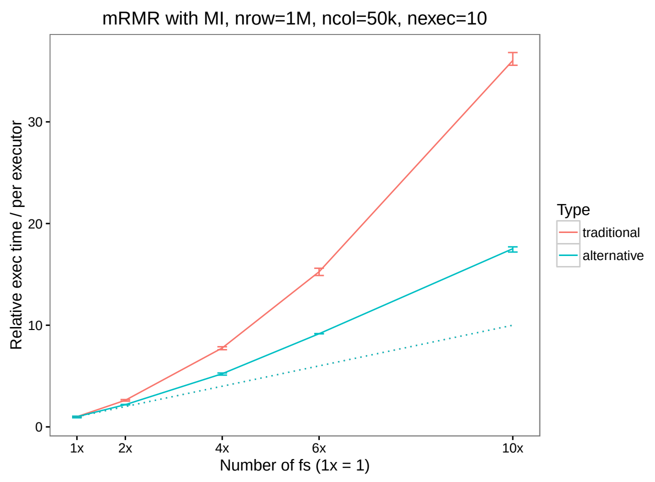

Fig. 7. mRMR scalability across the number of selected features.

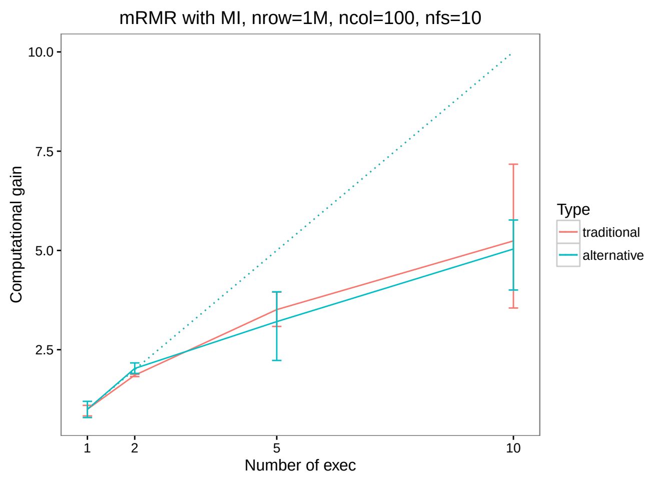

Fig. 8. mRMR scalability across the number of nodes.

Currently, the traditional encoding works only with discrete values; its extension to continuous features would require either additional MapReduce jobs to extimate the binning strategy or additional parameters as input of the jobs. MapReduce jobs with traditional encoding perform slower than those in alternative encoding; the latter is also by design inherently flexible to work with both discrete and continuous features.

Storing the data in alternative encoding better fit our objective of feature selection with high-dimensional datasets. While the limited amount of observations can be stored as columns without raising memory errors in the cluster, the high number of features can scale across the nodes. Hence, in the future, we intend to provide a portfolio of built-in feature selection algorithms that work with the alternative enconding. While we design and implement known FSA for MapReduce, novel algorithms that directly takes advantage of the distributed nature of the data will be investigated as well. It might be interesting to extend the scalability study to classification tasks and network inference.

Finally, the source code and example of the usage of the scala library is available on a public repository (https://github.com/creggian/mrmr).

REFERENCES

[1] V. Bolón-Canedo, N. Sánchez-Maroño, 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.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] J. H. Yeung, C. Tsang, K. H. Tsoi, B. S. Kwan, C. C. Cheung, A. P. Chan, and P. H. Leong, "Map-reduce as a programming model for custom computing machines," in *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on*. IEEE, 2008, pp. 149–159.

[4] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[5] C. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," *Advances in neural information processing systems*, vol. 19, p. 281, 2007.

[6] A. Mahout, "Scalable machine learning and data mining," 2012.

[7] X. Meng, J. Bradley, B. Yuvaz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *JMLR*, vol. 17, no. 34, pp. 1–7, 2016.

[8] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *J. Mach. Learn. Res.*, vol. 3, pp. 1157–1182, Mar. 2003.

[9] R. Kohavi and G. H. John, "Wrappers for feature subset selection," *Artif. Intell.*, vol. 97, no. 1-2, pp. 273–324, Dec. 1997.

[10] F. G. López, M. G. Torres, B. M. Batista, J. A. M. Pérez, and J. M. Moreno-Vega, "Solving feature subset selection problem by a parallel scatter search," *European Journal of Operational Research*, vol. 169, no. 2, pp. 477–489, 2006.

[11] N. Melab, S. Cahon, and E.-G. Talbi, "Grid computing for parallel bioinspired algorithms," *Journal of parallel and Distributed Computing*, vol. 66, no. 8, pp. 1052–1061, 2006.

[12] J. T. de Souza, S. Matwin, and N. Japkowicz, "Parallelizing feature selection," *Algorithmica*, vol. 45, no. 3, pp. 433–456, 2006.

[13] D. J. Garcia, L. O. Hall, D. B. Goldgof, and K. Kramer, "A parallel feature selection algorithm from random subsets," in *Proceedings of the 17th European Conference on Machine Learning and the 10th European Conference on Principles and Practice of Knowledge Discovery in Databases, Berlin, Germany*, 2006.

[14] A. Guillén, A. Sorjamaa, Y. Miche, A. Lendasse, and I. Rojas, "Efficient parallel feature selection for steganography problems." in *IWANN (1)*, ser. Lecture Notes in Computer Science, J. Cabestany, F. S. Hernández, A. Prieto, and J. M. Corchado, Eds., vol. 5517. Springer, 2009, pp. 1224–1231. [Online]. Available: http://dblp.uni-trier.de/db/conf/iwann/iwann2009-1.html#GuillenSMLR09

[15] S. Singh, J. Kubica, S. Larsen, and D. Sorokina, "Parallel large scale feature selection for logistic regression." in *SDM*. SIAM, 2009, pp. 1172–1183. [Online]. Available: http://dx.doi.org/10.1137/1.9781611972795.100

[16] D. Peralta, S. Río, S. Ramírez, I. Triguero, J. M. Benítez, and F. Herrera, "Evolutionary feature selection for big data classification: A mapreduce approach," *Mathematical Problems in Engineering*, vol. 2015, 2015. [Online]. Available: http://dx.doi.org/10.1155/2015/246139

[17] Z. Zhao, R. Zhang, J. Cox, D. Duling, and W. Sarle, "Massively parallel feature selection: an approach based on variance preservation," *Machine Learning*, vol. 92, no. 1, pp. 195–220, Jul. 2013.

[18] Z. Sun, "Parallel feature selection based on mapreduce," in *Computer Engineering and Networking*. Springer, 2014, pp. 299–306.

[19] B. Ordozgoiti, S. Gómez Canaval, and A. Mozo, "Massively parallel unsupervised feature selection on spark," in *New Trends in Databases and Information Systems: ADBIS 2015 Short Papers and Workshops, BigDap, DCSA, GID, MEBIS, OAIS, SW4CH, WISARD, Poitiers, France, September 8-11, 2015. Proceedings*, T. Morzy, P. Valduriez, and L. Bellatreche, Eds. Cham, Switzerland: Springer International Publishing, 2015, pp. 186–196. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-23201-0_21

[20] V. Bolón-Canedo, N. Sánchez-Maroño, and A. Alonso-Betanzos, "Distributed feature selection: An application to microarray data classification," *Applied Soft Computing Journal*, vol. 30, pp. 136–150, 2015.

[21] H. Peng, F. Long, and C. Ding, "Feature selection based on mutual information: criteria of max-dependency, max-relevance, and min-redundancy," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 8, 2005.

[22] P. E. Meyer, F. Lafitte, and G. Bontempi, "minet: A r/bioconductor package for inferring large transcriptional networks using mutual information." *BMC Bioinformatics*, vol. 9, no. 461, 2008.

[23] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in mapreduce," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York: ACM, 2010, pp. 975–986. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807273

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

[25] V. Bolón-Canedo, N. Sanchez-Marono, and A. Alonso-Betanzos, *Feature Selection for High-Dimensional Data*, ser. Artificial Intelligence: Foundations, Theory, and Algorithms. Cham, Switzerland: Springer International Publishing, 2015.

PLACE PHOTO HERE

Claudio Reggiani received his M.Eng degree in computer engineering from the Polytechnic University of Milan in 2013. He is currently a PhD candidate of the Interuniversity Institute of Bioinformatics in Brussels, (IB)2, Belgium. His research interests include machine learning, big data and integration of bioinformatics data.

PLACE
PHOTO
HERE

Yann-Aël Le Borgne Biography text here.

PLACE
PHOTO
HERE

Gianluca Bontempi Biography text here.