Journal of Parallel and Distributed Computing 169 (2022) 106–116

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Parallel-FST: A feature selection library for multicore clusters

Bieito Beceiro *, Jorge González-Domínguez, Juan Touriño

CITIC, Computer Architecture Group, Universidade da Coruña, Campus de Elviña s/n, 15071 A Coruña, Spain

ARTICLE INFO

Article history:
Received 12 November 2021
Received in revised form 20 June 2022
Accepted 20 June 2022
Available online 27 June 2022

Keywords:
Feature selection
Mutual information
MPI
HyperThreading
High performance computing

ABSTRACT

Feature selection is a subfield of machine learning focused on reducing the dimensionality of datasets by performing a computationally intensive process. This work presents Parallel-FST, a publicly available parallel library for feature selection that includes seven methods which follow a hybrid MPI/multithreaded approach to reduce their runtime when executed on high performance computing systems. Performance tests were carried out on a 256-core cluster, where Parallel-FST obtained speedups of up to 229x for representative datasets and it was able to analyze a 512 GB dataset, which was not previously possible with a sequential counterpart library due to memory constraints.

1. Introduction

The Big Data phenomenon has become popular in recent years due to the continuous increase of data stored in different fields such as bioinformatics, marketing, physics or engineering. However, these data are only valuable if we can extract useful information from them. This increase of data slows down their analysis, and occasionally it does not provide useful information due to the presence of redundant or irrelevant data.

Feature Selection (FS) is the Machine Learning (ML) procedure to remove these redundant and irrelevant data from the datasets, making them smaller and thus more feasible to analyze without losing relevant information. There exist many FS methods [2,22], each one with its advantages and drawbacks which make them useful for different scenarios. However, most FS methods present quadratic complexity related to the number of features, which makes them impractical for large datasets. In this work we present Parallel-FST,[1] a novel library that includes parallel implementations of seven highly employed FS methods, all of them based on Mutual Information (MI). More concretely, Parallel-FST includes the same methods as FEAST,[2] a broad suite of FS methods implemented in C and Matlab that is widely used by researchers from different fields of computational science. The FS methods included in FEAST (and thus Parallel-FST) are theoretically described in [4].

Parallel-FST allows us to execute all these FS methods in a High Performance Computing (HPC) system and thus complete the analyses of large datasets in a reasonable time. Specifically, it was developed using a hybrid approach with Message Passing Interface (MPI) [32] and C++ threads, for distributed- and shared-memory support, respectively.

The rest of the paper is organized as follows. Section 2 summarizes the related work and state of the art. Section 3 explains the background about FS necessary to understand the rest of the work, such as the base library FEAST and its FS methods. The parallel implementation and optimization of the methods included in Parallel-FST is described in Section 4. Section 5 provides the experimental evaluation in terms of runtime and scalability. Finally, concluding remarks and future work are presented in Section 6.

2. Related work

There exist some ML libraries in the state of the art, such as WEKA [10] or MAST [14], which include FS methods that can be executed on parallel systems. Nevertheless, these parallel implementations are only valid for shared-memory systems, which are quite limited in terms of scalability as they only include tens of cores. If more resources are needed, scientists can resort to distributed-memory systems such as clusters or supercomputers but, up to our knowledge, there was no FS library available that could exploit these architectures prior to Parallel-FST.

Regardless, the idea of using parallel computing to speed up a certain FS method has been extensively applied, using clusters, supercomputing facilities, hardware accelerators (such as GPUs), or

* Corresponding author.
E-mail addresses: bieito.beceiro.fernandez@udc.es (B. Beceiro), jgonzalezd@udc.es (J. González-Domínguez), juan@udc.es (J. Touriño).

[1] Publicly available at https://gitlab.com/bieito/parallel-fst.

[2] https://github.com/Craigacp/FEAST.

https://doi.org/10.1016/j.jpdc.2022.06.012

Table 1
State of the art related to parallel FS.

Name	Available	Framework	Year	Ref.
PE-EFS	No	CUDA	2021	[11]
CUDA-JMI	Yes	CUDA	2020	[9]
parallel M-FS	No	Spark	2020	[34]
Hadoop-Voting	No	Hadoop	2020	[37]
fast-mRMR-MPI	Yes	MPI	2019	[8]
DiCFS	Yes	Spark	2019	[24]
PAGreedy	No	Threads	2019	[21]
MR-GAFS	No	Hadoop	2018	[28]
PAJMI	No	Threads	2017	[20]
mRMR-MR	Yes	Spark	2017	[27]
Asy-OS	No	MPI	2016	[35]
Parallel Filter	No	Threads	2016	[29]
Fast-mRMR	Yes	CUDA/Spark	2016	[26]
DWFS	Web	MPI	2015	[31]
VLSRF	No	CUDA	2015	[16]

clouds. Table 1 summarizes the state of the art related to the use of HPC to accelerate FS algorithms.

Focusing on distributed-memory systems (the target of Parallel-FST), several approaches are based on the message-passing paradigm. Specific examples include DWFS [31], with a parallel genetic algorithm, fast-mRMR-MPI [8], based on a variant of the popular mRMR method [26], and the parallel implementation of an online FS algorithm presented in [35].

Furthermore, Big Data frameworks such as Hadoop or Spark are gaining attention in recent years and becoming more popular to develop FS codes that can be executed not only on clusters, but also on cloud environments. We can cite, among others, parallel implementations of FS methods based on random forests [34], genetic algorithms [37,28], the mRMR algorithm [27], and Correlation-based Feature Selection (CFS), either isolated [24] or combined with methods [12,30].

However, after a thorough analysis of the literature, we can assert that in all these previous works either the code is not publicly available or it is reduced to a single algorithm that is not widely employed by the research community. An integral library such as Parallel-FST that provides several FS methods is a must, so that researchers can adapt their analyses to the characteristics of the data, as required.

3. Background: feature selection with mutual information

Parallel-FST is based on FEAST, and it includes parallel implementations of the FS methods that are available in the original library (see [4] for further theoretical details). FEAST has been chosen as basis as it is highly cited and has been widely used and tested by numerous researchers. In fact, it has been used for studies in diverse areas such as medicine [3,15], genetics [33], electronics [7], or transportation [18,6].

As previously mentioned, FS algorithms try to select only those features that are interesting for the problem, discarding irrelevant or redundant ones. Nevertheless, relevance and redundancy cannot be directly measured, so they must be approximated. All FS algorithms included in Parallel-FST are based on MI for this estimation.

Entropy, as the fundamental information unit of a random variable, is necessary to calculate MI. It is denoted by $H(X)$, and quantifies the uncertainty present in the distribution of X. It is defined as:

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

where x is any value that the random variable X can take. The entropy of a variable will be lower if its distribution is biased towards a particular event, and higher when all events present the same probability to occur.

Entropy can be conditioned by other events, and the conditional entropy of a random variable X, given another variable Y, can be calculated with the following expression:

$$H(X|Y) = -\sum_{y \in Y} p(y) \sum_{x \in X} p(x|y) \log p(x|y) \tag{2}$$

Conditional entropy can be understood as the amount of uncertainty that X holds after the result of Y becomes known.

MI measures the amount of information shared between two variables, and it can be derived from entropy. The MI between two random variables X and Y is computed as:

$$\begin{aligned} MI(X;Y) &= H(X) - H(X|Y) \\ &= \sum_{x \in X} \sum_{y \in Y} p(xy) \log \frac{p(xy)}{p(x)p(y)} \end{aligned} \tag{3}$$

Since it is the difference between two entropies, MI can also be understood as the amount of uncertainty that is removed once Y is known. Alternatively, a simpler definition could be the amount of information that one variable provides over the other.

As with entropy, MI can also be conditional. That is, the amount of information that is still shared between two variables after a third one becomes known. The MI between X and Y conditioned by Z is computed as follows:

$$\begin{aligned} MI(X;Y|Z) &= H(X|Z) - H(X|YZ) \\ &= \sum_{z \in Z} p(z) \sum_{x \in X} \sum_{y \in Y} p(xy|z) \log \frac{p(xy|z)}{p(x|z)p(y|z)} \end{aligned} \tag{4}$$

The following subsections provide a basic introduction to the algorithms included in Parallel-FST. It should be noted that the notation $J_m(X)$ refers to the score of a random variable (or feature) X when using the algorithm m. High scores mean high relevance and low redundancy.

3.1. MIM - Mutual Information Maximisation

A first approach for computing the score of each feature could be some kind of correlation metric between the feature and the class label, and MI is a metric that can be used for this purpose. This way, the MIM score for a feature X_k and a class Y is computed as follows:

$$J_{MIM}(X_k) = MI(X_k;Y) \tag{5}$$

This heuristic has often appeared in the literature, for example in [17]. The MIM score assumes independence between the features of the dataset, so it only considers the relevance of a feature, but not the redundancy. For this reason, the score only needs to be computed once for each feature of the dataset, which makes MIM the least computationally complex method of FEAST and Parallel-FST.

The main disadvantage appears when features are not independent. For example, if the feature with the highest score appears twice in the dataset, it will be selected more than once, and the set of selected features will hold a high level of redundancy.

3.2. CondMI - Conditional Mutual Information

The CondMI criterion is an optimization derived from the formulation of the FS problem as a conditional likelihood problem [4]. The score for a feature X_k, a class Y and the set of already selected features S is computed as:

$$J_{CondMI}(X_k) = MI(X_k;Y|S) \tag{6}$$

That is, the MI between the feature and the class, conditioned by the features selected so far.

3.3. Methods in the beta-gamma space

3.3.1. MIFS - Mutual Information Feature Selection

The MIFS criterion was presented in [1] and introduces enhancements over MIM in order to reduce redundancy. The computation of the score of a feature X_k, a class Y and a set of selected features S follows the formula:

$$J_{MIFS}(X_k) = MI(X_k; Y) - \beta \sum_{X_j \in S} MI(X_k; X_j) \tag{7}$$

where β is a user-defined parameter that can be understood as the disagreement with the assumption of dependency among features.

3.3.2. CIFE - Conditional Infomax Feature Extraction

This criterion, which was proposed in [19], can be derived from several transformations of CondMI. The score for a feature X_k, a class Y and a set of selected features S is computed with the following expression:

$$J_{CIFE}(X_k) = MI(X_k; Y) - \sum_{X_j \in S} MI(X_k; X_j) + \sum_{X_j \in S} MI(X_k; X_j|Y) \tag{8}$$

Three main terms can be identified: the MI with the class $MI(X_k; Y)$, which suggests relevance; the MI with the already selected features $\sum MI(X_k; X_j)$, which suggests redundancy, and the MI with the already selected features conditioned by the class $\sum MI(X_k; X_j|Y)$, which can be understood as conditional redundancy.

3.3.3. Beta-gamma space

As can be seen, the formulas of the MIFS and CIFE criteria have a similar shape. If we parametrize the terms for redundancy and conditional redundancy, we can define a two-dimensional space in which both criteria could be expressed as a linear combination of information theory terms. That is, with two parameters (β and γ) we can set a weight for redundancy and conditional redundancy.

This way, any criterion in this space (named "Beta-Gamma") can be defined as:

$$J_{BetaGamma}(X_k) = MI(X_k; Y) - \beta \sum_{X_j \in S} MI(X_k; X_j) + \gamma \sum_{X_j \in S} MI(X_k; X_j|Y) \tag{9}$$

From this expression, some already explained criteria can be found:

- MIM: $\beta = 0, \gamma = 0$
- MIFS: $\beta \in [0, 1], \gamma = 0$
- CIFE: $\beta = 1, \gamma = 1$

Since both MIFS and CIFE can be derived from the Beta-Gamma space, they were merged into a single method "Beta-Gamma" rather than implementing a different method for each of them. Meanwhile, although MIM can be computed with $\beta = 0$ and $\gamma = 0$, it was implemented in a single method in order to avoid the computation of information measurements that would be otherwise multiplied by zero.

3.4. JMI - Joint Mutual Information

The JMI criterion is an alternative approach to MIFS presented in [36], which aims to increase complementary information among features rather than minimizing redundancy. Given a feature X_k, the class Y and the set of already selected features S, the JMI score is computed as follows:

$$J_{JMI}(X_k) = \sum_{X_j \in S} MI(X_k X_j; Y) \tag{10}$$

That is, the sum of the information between the class and a random joint variable $X_k X_j$ is defined by joining the candidate X_k with every already selected feature. This criterion is based on selecting a candidate feature when it contributes with new information complementary to the other selected features.

3.5. mRMR - Max-Relevance Min-Redundancy

Another criterion that varies with the number of selected features is mRMR [25]. However, it does not take conditional redundancy into account unlike CIFE. The mRMR score of a feature X_k according to the class Y and a set of selected features S can be computed as:

$$J_{mRMR}(X_k) = MI(X_k; Y) - \frac{1}{|S|} \sum_{X_j \in S} MI(X_k; X_j) \tag{11}$$

3.6. ICAP - Interaction Capping

This criterion was proposed in [13] and, unlike the previous methods, it makes use of order operators. The score of a feature X_k, given a class Y and a set of selected features S, is computed as:

$$J_{ICAP}(X_k) = MI(X_k; Y) - \sum_{X_j \in S} \max\left[0, \{MI(X_k; X_j) - MI(X_k; X_j|Y)\}\right] \tag{12}$$

The usage of order operators complicates a probabilistic interpretation of the ICAP criterion. However, it can be seen that features with higher redundancy will achieve lower scores.

3.7. DISR - Double Input Symmetrical Relevance

Finally, DISR is a modification of JMI proposed in [23]. The score of a feature X_k, given the class Y and the set of selected features S, is calculated by dividing the MI value by the conditional entropy, with the following expression:

$$J_{DISR}(X_k) = \sum_{X_j \in S} \frac{MI(X_k X_j; Y)}{H(X_k X_j|Y)} \tag{13}$$

3.8. Inclusion of weights

Some of the described methods (MIM, CondMI, JMI and DISR) have weighted versions. These approaches take an additional vector of weights that is used to specify the importance of each sample, so that the computation of the entropy or MI can be manually biased.

For instance, the computation of the MI for two variables X and Y would be calculated as follows, where $w(xy)$ is the weight of the co-occurrence of two values of each feature:

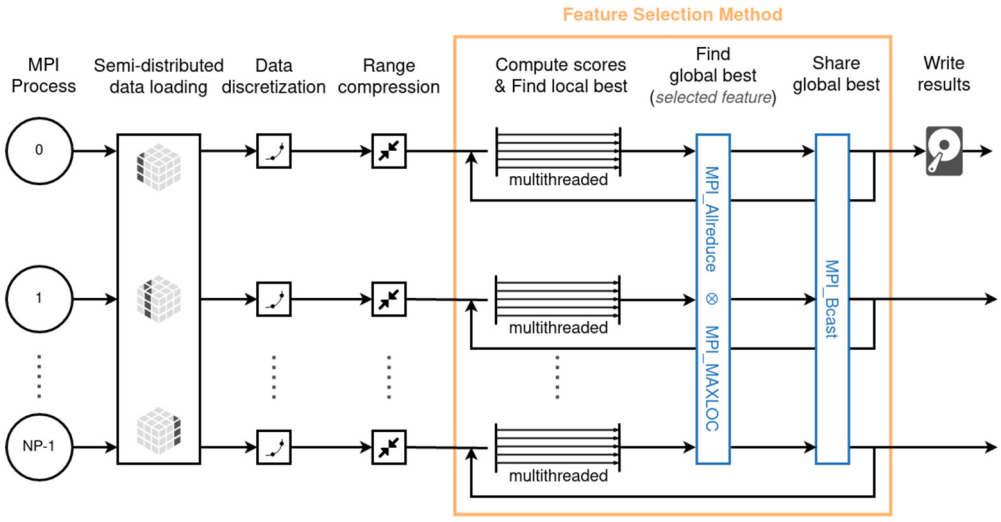


Fig. 1. Diagram of the distributed execution for all methods included in Parallel-FST.

Table 2
Example of dividing a continuous range between 0.6 and 3.1 into five bins.

Bin	Range
0	[0.6, 1.1)
1	[1.1, 1.6)
2	[1.6, 2.1)
3	[2.1, 2.6)
4	[2.6, 3.1]

Table 3
Assignment of a discrete value to a continuous one using the bins of the previous table.

Feature	Value				
f	1.4	0.6	0.9	3.1	2.8
f'	1	0	0	4	4

$$MI(X;Y) = \sum_{x \in X} \sum_{y \in Y} w(xy)p(xy) \log \frac{p(xy)}{p(x)p(y)} \tag{14}$$

This makes it possible to introduce knowledge and to give a meaning to certain samples.

4. Methodology

The Parallel-FST library includes the explained FS methods, written in C++ with MPI routines and threads to accelerate the execution on multicore clusters. Information about system requirements, as well as instructions to install, compile, and execute the library are included in its website: https://gitlab.com/bieito/parallel-fst.

Fig. 1 illustrates the general workflow of the approach followed by Parallel-FST. Besides the appropriate work distribution among processes and threads, with the necessary synchronizations to find the best candidate features, the workflow also includes a preprocessing step (data discretization) and two optimizations (semi-distributed data loading and range compression). Each of these phases are detailed in the following subsections.

4.1. Data discretization

MI, and consequently all the methods explained in Section 3, work with discrete data. As many real datasets contain continuous data, Parallel-FST also includes an auxiliary tool to discretize the input datasets as a preprocessing step. Specifically, it applies a *binning* approach where the continuous range is divided into n bins or fragments of equal length, and each value in the input dataset is replaced by the identifier of the fragment that contains it.

Tables 2 and 3 illustrate, for an example with five continuous data between 0.6 and 3.1, the procedure to convert them into discrete values using five bins. First, the range is divided into five fragments of equal size (0.5) and the bins are created (see Table 2). Then, the continuous values are replaced by the identifier of the bin (Table 3).

4.2. Data and workload distribution

Parallel-FST uses MPI to exploit the computational capability of HPC clusters, by distributing data and work among the available cluster nodes and cores within each node. Algorithm 1 shows the structure of the FS methods included in Parallel-FST. They all start with the selection of the first feature as the one having the highest MI with the class (Lines 5 to 8). Then, the rest of the features are selected according to the metric used by the FS method (Line 15), as seen in Section 3. As previously explained, most metrics depend not only on the class, but also on the already selected features. Note that Line 13 avoids calculating the score of previously selected features, as they cannot be chosen again.

An analysis of the pseudocode shows that most of the work is performed in the nested loops of Lines 10 and 12. The outer loop (Line 10) cannot be parallelized, as an iteration cannot start until the previous one has finished (the information of the previously selected feature is necessary to choose the next one). Nevertheless, the inner loop is a suitable target for parallelization as the computation of the score for each feature is independent of the other computations. Therefore, the FS implementations in Parallel-FST divide the input dataset into NP blocks with the same number of features per block, NP being the number of MPI processes (Lines 6 and 12). Each process computes the score of all the features in the block. This distribution of features is performed once at the beginning of the execution and all processes work over the same features in all the iterations of the outer loop. This static data distribution with equal-sized blocks is suitable for the FS methods included in Parallel-FST as the workload (computation of the score) is similar for all features (the complexity depends on the number of samples, which is the same for all features). Consequently, the workload is balanced among the MPI processes. The only imbalance can be generated by those features that have already been

Algorithm 1: General structure of the FS methods included in Parallel-FST for each process (MPI routines in red and loops shared among threads in blue).

```
1  Input: Discrete subdataset with M/NP features, N samples, and one class Y;
     Number of features to select NS
2  Output: Vectors selIds and selScores, of length NS, with the ids of the
     selected features and the scores obtained by the metric, respectively
3  Initialize featScore as a vector of length M/NP
4  Initialize selData as a vector of length NS
5  // Compute scores of local features (multithread)
6  for every local feature F_i with 0 ≤ i < M/NP do
7  |  featScore[i] = MI(F_i; Y)
8  selIds[0], selScore[0] = FindAndShareBest (featScore, selData[0])
9  // Select the other features
10 for every k with 0 < k < NS do
11 |  // Compute scores for unselected local features
12 |  for every local feature F_i with 0 ≤ i < M/NP do
13 |  |  if i is not in selIds then
14 |  |  |  // Depends on the selected metric
15 |  |  |  featScore[i] = calcScore(F_i; Y; selData)
16 |  selIds[k], selScore[k] = FindAndShareBest (featScore, selData[k])

   Procedure FindAndShareBest (featScore, selData_k)
17 |  // Find id and score of best local feature
18 |  localId = argMax(featScore)
19 |  localScore = max(featScore)
20 |  // Find best global score and owner process
21 |  globalScore, oRank = Allreduce_MAXLOC(localScore, pRank)
22 |  if oRank == pRank then
23 |  |  globalId = localId + pRank * M/NP
24 |  |  selData_k = F_localId
25 |  |  featScore[localId] = 0
26 |  globalId = Bcast_oRank→WORLD(globalId)
27 |  selData_k = Bcast_oRank→WORLD(selData_k)
28 |  return globalId, globalScore
```

selected and thus do not need to have their score calculated again (Line 13). The worst scenario would be that all selected features fall in the same block. However, we must take into account that in a real world analysis the percentage of selected features must be low in order to obtain useful information. Therefore, the impact of this workload imbalance is almost negligible even in the worst-case scenario.

Each process has the data corresponding to its block of features stored into its memory prior to the nested loop, through the procedure that will be explained in Subsection 4.4. Therefore, the only point of synchronization among processes is the choice of the feature with the highest score (procedure `FindAndShareBest` in Algorithm 1). Due to distributing the loops of Lines 6 and 12, each process has found the most promising feature of its block, but the algorithm must select only the best one among them. This is performed by an `MPI_Allreduce` collective with the `MPI_MAXLOC` operator (Line 21). Once the feature is selected, the owner process sends its data to the other processes with `MPI_Bcast` routines, as this information is necessary in the next iterations of the outer loop to compute the new scores (Line 15).

4.3. Hybrid MPI/multithreaded implementation

The previous subsection has explained how the data and workload are distributed among different MPI processes. This approach would be sufficient to execute the FS methods on distributed-memory systems by creating one MPI process per core. However, Parallel-FST includes a second level of parallelism, where each process can launch several C++ threads that collaborate in the computation of the scores of the block.

Consequently, the work of the loops in Lines 6 and 12 is distributed in two levels: first, the features are divided into equal-sized blocks, with one block per MPI process; and, second, the features of each single block are distributed among the threads launched by the owner process.

One typical use of this hybrid MPI/multithreaded approach on modern multicore clusters consists in creating one process per node, and the same number of threads as cores in the node, but intermediate configurations with different number of processes and threads can be applied. This approach has the following advantages:

- Creation, synchronization and destruction of threads is lighter than for processes. Therefore, reducing the number of processes per node in each execution should decrease runtime.
- It allows to exploit the HyperThreading technology currently available in most processors, where two logical threads can share the resources of a single physical core.
- As explained in the previous subsection, the data of the lastest selected feature must be sent from its owner process to the others. The use of threads allows replacing costly explicit message-passing communications (broadcasts in this case) with implicit shared-memory communications at node level.

4.4. Semi-distributed data loading

As mentioned in Subsection 4.2, the input datasets are partitioned and distributed among MPI processes so that they can work in parallel over their assigned blocks of features. A naive approach for this data distribution would consist in the root process reading the whole dataset from disk and storing it into a buffer, which would later be scattered among all processes using the appropriate MPI routines. However, that approach is limited by the memory of the node where the root process runs, so it would not be possible to straightforwardly analyze datasets larger than this memory size.

Since the amount of information that is collected has significantly grown in the latest years, most extremely large datasets are nowadays stored using a sparse format, i.e., some values (usually zeros) are not explicitly stored in order to reduce disk storage. Parallel-FST needs a solution that allows an efficient processing of these huge sparse datasets. For instance, the sequential implementations in FEAST need the data as dense matrices so that a preprocessing step to convert them is required. Our solution consists of three steps, following the diagram of Fig. 2:

1. The root process reads the dataset as a sparse matrix.
2. The `MPI_Bcast` collective is used to send the sparse matrix to all processes.
3. Each process "expands" as a dense matrix its assigned block of features exclusively.

In this way, the memory size limitation is overcome, and it is possible to analyze datasets that fulfill these two conditions:

- They fit in the memory of one node in sparse format.
- The block of features assigned to each MPI process fits in its available memory.

These conditions are significantly less restrictive than the original one (the whole matrix in dense format should fit in the memory of only one node) and thus this semi-distributed data loading allows working over large datasets. Furthermore, as the broadcast is performed with the data in sparse format, its impact on the total runtime remains limited.

It should be noted that other approaches could be developed for this purpose. For instance, the root process could iteratively read blocks of data from disk, expand them into dense format,

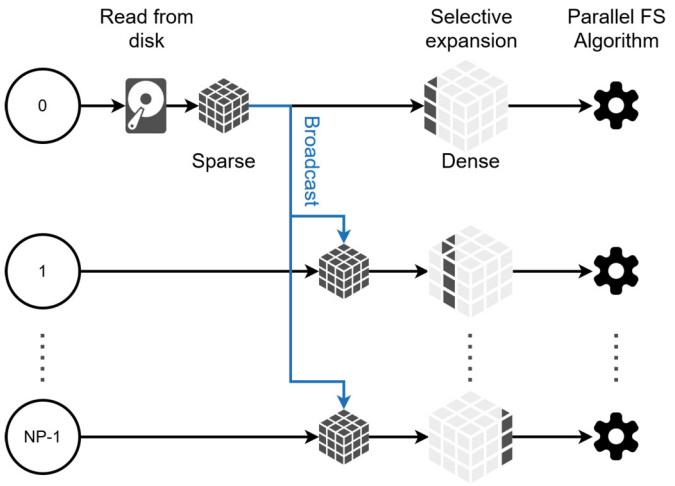


Fig. 2. Diagram of the semi-distributed data loading procedure.

Feature	Values				
f_1	0	2	2	0	1
f_2	1	3	3	2	1

$f_1 \setminus f_2$	**0**	**1**	**2**	**3**
0	0	1	1	0
1	0	1	0	0
2	0	0	0	2

Fig. 3. Features and co-occurrence matrix (dimension 3x4) when the number of different values is close to the difference between the minimum and the maximum sample.

and send each block of features to the process that will use it. However, this approach would perform all the format translation sequentially, with high impact on performance, while our semi-distributed data loading performs the expansion in parallel. Moreover, communication would be more expensive as blocks would be sent in dense format. A similar alternative that sends the data to the other processes in sparse format could also be implemented. However, many widely employed sparse formats store the datasets in a sample-major fashion (for instance, LIBSVM [5]), while the FS algorithms work with feature-major matrices. This would mean that each row (i.e. sample) would be split into blocks, and each block would be sent to a different process, so there would be a large number of small size communications, and therefore high overhead.

Finally, it is important to note that Parallel-FST is flexible enough to work with different types of files as input. In scenarios where the dataset is stored in dense format, it is loaded with the naive approach explained at the beginning of this subsection. The semi-distributed data loading is only applied to sparse formats, which is the common case for large datasets.

4.5. Range compression

The original sequential FS methods are based on the usage and combination of information metrics, which need to calculate co-occurrences between values instead of focusing on the values themselves. For instance, the values {100, 200, 100} are treated similarly to {1, 2, 1}. In order to compute the information metrics between two features, bidimensional histograms are created. However, when intermediate values are missing in the feature, these histograms may contain some rows or columns in which all values are zero. These rows and columns imply a waste of memory and computation time since they have no impact in the results of the information metrics. Two examples are shown in Figs. 3 and 4. The former illustrates a common case, where only the first column is completely full of zeros, while the latter shows an extreme example where memory requirements are huge.

Feature	Values				
f_3	0	1000	1000	0	1000
f_4	1000	2000	2000	1000	1000

$f_3 \setminus f_4$	**0**	…	**1000**	…	**2000**
0	0	…	2	…	0
…	…	…	…	…	…
1000	0	…	1	…	2

Fig. 4. Features and co-occurrence matrix (dimension 1001x2001) when the number of different values is much lower than the difference between the minimum and the maximum sample.

Some datasets hold features that take values from a wide range of numbers, so this issue might slow the computation or even completely fill the memory of the system. For example, a dataset about CPUs in which there are features for minimum and maximum frequency (in kHz) with ranges [0, 700000] and [500000, 4000000] respectively, would need a co-occurrence matrix of 10 TB.

Parallel-FST includes an efficient solution to this problem, which has been named as "range compression". It basically performs a translation of the values of the feature so that the intermediate values are suppressed. This way, the creation and computation over rows and columns that do not affect the results of the metrics are avoided.

Algorithm 2 shows a pseudocode of the procedure. Each feature (i.e. row) is processed independently, and Parallel-FST attempts to find new values so that the range of the resulting feature is minimal. This is achieved through $featMap$ and $mapCounter$: the former an array used to keep the relationship between new and old values, and the latter an accumulator that counts the number of distinct values found so far.

Algorithm 2: Range compression procedure.

```
1  Input: Matrix A with M rows (features) and N columns (samples)
2  Output: Updated matrix A
3  for every feature F_j in A with 0 ≤ j < M do
4      maxState ← max(F_j)
5      featMap ← zeros(maxState)
6      mapCounter ← 1
7      for every i with 0 ≤ i < N do
8          old_s ← F_j[i]
9          if featMap[old_s] = 0 then
10             featMap[old_s] ← mapCounter
11             mapCounter ← mapCounter + 1
12         F_j[i] ← featMap[old_s] − 1
```

This procedure presents linear complexity, thus its execution is very fast. Consequently, the overhead of performing the range compression is almost negligible, but the impact on the reduction of the FS execution time can be huge. Furthermore, as range compression is a per-feature procedure, it can be executed in parallel by the MPI processes, thus further reducing the overhead.

The impact of the range compression can be observed in the examples of Figs. 5 and 6, which show the results of its application to the features of Figs. 3 and 4, respectively. Note that the columns and rows that were composed of zeros, which were useless to the information metrics, do not appear anymore. In the extreme scenario (Figs. 4 and 6) the dimension of the co-occurrence matrix is reduced from two million elements to only four.

Feature	Values				
f'_1	0	1	1	0	2
f'_2	0	1	1	2	0

$f'_1 \setminus f'_2$	**0**	**1**	**2**
0	1	0	1
1	0	2	0
2	1	0	0

Fig. 5. Features and co-occurrence matrix after range compression (dimension 3x3 instead of 3x4) when the number of different values is close to the difference between the minimum and the maximum sample.

Feature	Values				
f'_3	0	1	1	0	1
f'_4	0	1	1	0	0

$f'_3 \setminus f'_4$	**0**	**1**
0	2	0
1	1	2

Fig. 6. Features and co-occurrence matrix after range compression (dimension 2x2 instead of 1001x2001) when the number of different values is much lower than the difference between the minimum and the maximum sample.

As a final remark note that the range compression is a general optimization technique that can also be directly applied to the implementations available in FEAST or other sequential libraries in order to improve their performance.

5. Experimental evaluation

Parallel-FST has been extensively compared to the sequential C version of FEAST,[3] whose FS methods have been presented in [4]. First, it was proved that the output results of the methods available both in FEAST and Parallel-FST are identical. The rest of this section compares both libraries in terms of performance.

The comparison with the state of the art was focused on FEAST as it is the only library in the literature that includes all these methods, its C implementations are fast for sequential computation, it is widely employed by scientists, and provides exactly the same results as Parallel-FST. Although FEAST does not include any support for parallel computing the runtime and speedups presented in this section are sufficient to prove the quality of the hybrid MPI/multithreaded implementation used in Parallel-FST. Weka [10], another highly employed library with FS and multithreaded support, was also considered for comparison. However, it was discarded for two reasons. On the one hand, the FS methods included in Weka differ from those implemented in Parallel-FST. As will be seen in this section the runtime for FS significantly depends on the method, so it would be unfair to compare the speed of different algorithms. On the other hand, Weka is implemented with Java, employing its multithreaded support for shared-memory systems. As Java executions are based on a virtual machine, they are usually slower than those of C/C++ and thus not comparable with the FEAST/Parallel-FST implementations in order to obtain proper conclusions.

5.1. Experimental configuration

A multicore cluster with 16 nodes, each one with two octa-core Intel Xeon E5-2660 processors and 64 GB of memory, has been used. It means that our experiments could be executed on up to 256 cores (16 cores per node). Moreover, as this architecture provides HyperThreading, up to 512 threads can be used (two logical threads per core). The 16 nodes are connected through an InfiniBand FDR network with high bandwidth and low latency.

Regarding software, both FEAST and Parallel-FST used the GNU C/C++ compiler v8.3.0, while the latter was also linked to the OpenMPI library v3.1.4. Finally, all the experiments were run with the nodes in exclusive mode, i.e. the hardware was never shared by other jobs.

Table 4
Characteristics of the datasets (the size is calculated using 8B per value).

	#Features	#Samples	#Classes	Size
Epsilon	2,000	400,000	2	6.96GB
RCV1	47,236	20,242	2	7.12GB
News20	62,061	15,935	20	7.37GB
SVHN	3,072	531,131	10	12.16GB
E2006	4,272,227	16,087	-	512.06GB

Five publicly available datasets with different characteristics (summarized in Table 4) have been used for the evaluation. They were all obtained from the LIBSVM collection [5], which stores them in sparse format in order to reduce disk and memory consumption. Therefore, the semi-distributed data loading technique presented in Subsection 4.4 is very useful in this case. On the one hand, two of the datasets (`Epsilon` and `SVHN`) are used as examples for scenarios with more samples than features. The first one, with only two classes, is an artificial dataset created for the "Pascal large scale learning challenge". The second dataset, which is multiclass, stores pictures of house plates with 32x32 resolution, with the features representing the RGB values for each pixel. On the other hand, the rest of datasets contain more features than samples. `RCV1` is a dataset with two classes that includes news categorized by hand for research purposes. `News20` also contains documents about news, divided into 20 classes. Finally, `E2006` was used as an example of a huge dataset that does not fit into the memory of one node. It contains information of reports obtained from several US companies between the years 1996 and 2006. This dataset is usually employed for regression, thus not having a specific feature used as class.

All the experiments shown in this section have been obtained after applying a discretization with 128 bins and by fixing the number of selected features to 200. This number is high enough to show whether there is workload imbalance because the already selected features do not require work, but not too high to avoid selecting too many features, which will never be the case in a real scenario. Fig. 7 shows (in logarithmic scale) the runtime of the different FS methods when using the sequential version available in FEAST (and applying the range compression technique). `E2006` could not be analyzed as loading it into memory requires more than the 512 GB available in a single node of the cluster. This figure shows that the runtime is extremely variable among the FS methods. For instance, as explained in Section 3, MIM is a very fast and simple method that takes into account the amount of information shared between the feature and the class, but not the redundancy among features. On the contrary, CondMI is the most expensive one, especially when working on datasets with a large number of samples (`Epsilon` and `SVHN`).

Table 5 shows the runtime of the FEAST version of the most computationally expensive method (CondMI) with and without range compression, in order to give insights about the benefits that this technique can provide to sequential FS methods. Range compression is beneficial in all scenarios but, as explained in Section 4.5, its impact on performance largely depends on the variety of data in the input dataset.

[3] https://github.com/Craigacp/FEAST.

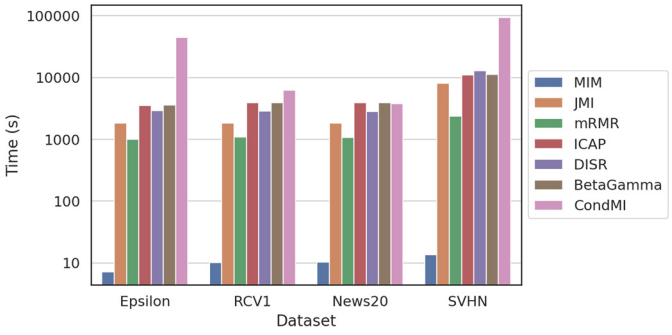


Fig. 7. Runtime (in seconds) needed by the original methods in FEAST.

Table 5
Runtime (in seconds) of the sequential CondMI method in FEAST with and without range compression (RC), as well as the time required to complete this technique.

	Without RC	With RC	Time for RC
Epsilon	46,214	44,430	1.91
RCV1	41,596	6,221	2.12
News20	27,611	3,799	2.21
SVHN	96,113	93,246	3.68

5.2. Performance analysis

The performance evaluation started by searching for the best combination of threads and MPI processes within each node. As each node contains two octa-core processors (16 physical cores) and allows for HyperThreading, all configurations launching a total of 32 threads per node were tested (i.e., one process with 32 threads, two processes with 16 threads, four processes with 8 threads and so on). The configuration with two processes per node and 16 threads per process obtained the best results in all cases, and thus it has been used for all the scalability experiments. This is a reasonable result, as each node has two processors, each one with its own memory module. Thus, this configuration maps one MPI process per processor and it guarantees that threads only access the memory module of the processor where they are launched.

The graphs in Fig. 8 show the speedups obtained by the hybrid MPI/threads implementation of the different FS methods present in Parallel-FST from one node (16 physical cores) to 16 nodes (256 physical cores), when compared to the original FEAST counterparts (using range compression in all the experiments). `E2006` is not included as, due to memory constraints, it could only be analyzed when using the whole cluster, and thus there is no base sequential runtime to calculate the speedup. The following conclusions can be drawn:

- In general, the speedups are higher for those datasets with more features than samples (`RCV1` and `News20`). The main reason is that the complexity of the methods depends on the number of features, and the higher the complexity the more opportunities of parallelism. Moreover, as explained in Subsection 4.2, the data of the feature selected in each iteration must be broadcast from its owner to the other MPI processes. When increasing the number of samples the weight of the communications, and thus the performance overhead, is higher.
- JMI, ICAP, DISR and the methods of the Beta-Gamma space achieve high scalability for the four datasets, reaching parallel efficiencies higher than 80% even for the whole cluster. They also present superlinear speedups for some experiments with two, four and eight nodes.
- mRMR obtains lower speedups than the four previous methods, due to its lower complexity, but it still achieves good scalability for the four datasets. It also presents superlinear speedups for some experiments with two nodes (32 cores).

Table 6
Runtimes of the Parallel-FST methods using the whole cluster (16 nodes) to analyze the `E2006` dataset.

Method	Time (s)
MIM	17
JMI	788
mRMR	328
ICAP	1,463
DISR	1,123
BetaGamma	1,505
CondMI	1,783

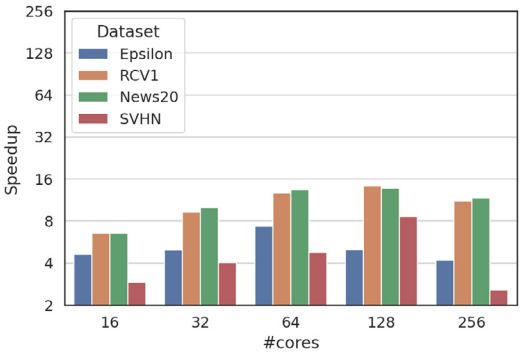
- CondMI is the method with the highest variability depending on the analyzed dataset. It is able to obtain an acceleration of 229x over the sequential implementation available in FEAST when working with the `News20` dataset, but the speedups are not higher than 55 for the two datasets with more samples than features (`Epsilon` and `SVHN`). In general, the performance of this FS method is really dependent on the number of samples, as was already remarked when analyzing the runtime of the original implementations (see Fig. 7).
- The only method that presents a limited scalability is MIM. The reason is not an inefficient parallel implementation but the high speed of this method in the sequential library FEAST, as it only has to compute one MI calculation per feature (see Subsection 3.1). In fact, the original implementation of MIM requires less than 15 seconds for the four datasets that it can analyze (see Fig. 7).

The results presented in Fig. 8 prove the large increase in speed that can be achieved thanks to Parallel-FST. For instance, the FS methods based on the Beta-Gamma space, as well as ICAP and DISR, require more than three hours to analyze the `SVHN` dataset with FEAST, and this time is reduced to around one minute with the implementations available in Parallel-FST. Another relevant example is the `News20` dataset, where FEAST can require up to one hour to select the 200 features, while all methods in Parallel-FST finish in less than 18 seconds. The only exception to these gains is the MIM method, where there is no room for improvement using parallel computing techniques, as its FEAST implementation is already very fast.

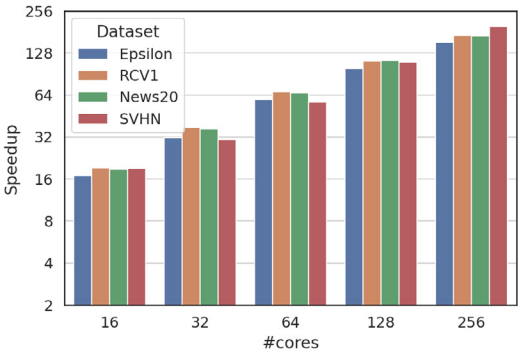
Finally, but not less important, we should remark that Parallel-FST not only reduces the runtime compared to FEAST, but it also allows us to complete FS analyses on scenarios previously not feasible for the sequential library. For instance, as mentioned earlier, every method in FEAST fails when trying to analyze the `E2006` dataset, as it would need around 512 GB of memory to be loaded, which is too much for almost any shared-memory system. Nevertheless, our parallel implementation distributes the features among the MPI processes (see Section 4.2) and thus it can aggregate the memory of the whole cluster (64 GB per node, 1 TB in total) to complete the FS analysis using the seven methods. Runtimes for this dataset are shown in Table 6, ranging from 17 seconds with MIM to approximately 30 minutes with CondMI.

6. Conclusion

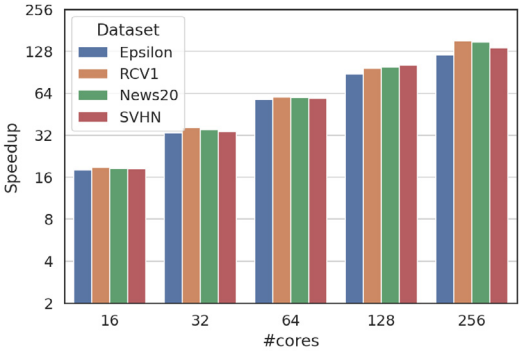
Feature selection has become a key step in ML due to the continuous increase of the average dataset sizes in different fields such as text mining, genetics or bioinformatics. This technique discards those features that are irrelevant or redundant, and whose inclusion in the ML analyses would lead to very high runtimes or even inaccurate conclusions. Nevertheless, the high computational and memory requirements prevent the use of most FS methods for large datasets.



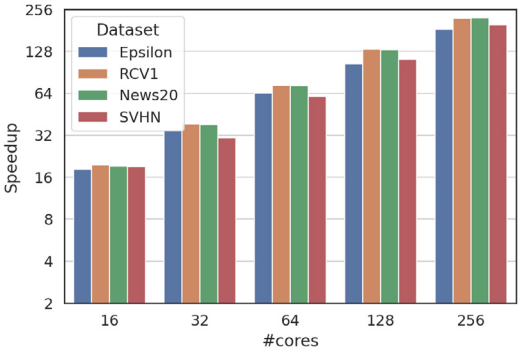
(a) MIM



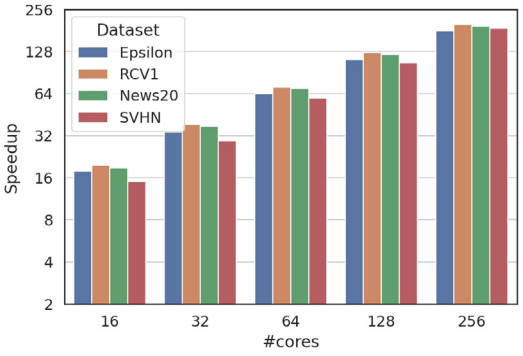
(b) JMI



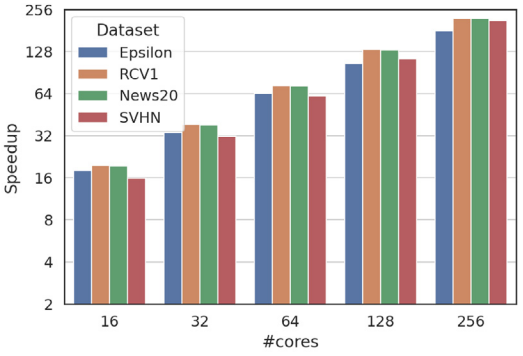
(c) mRMR



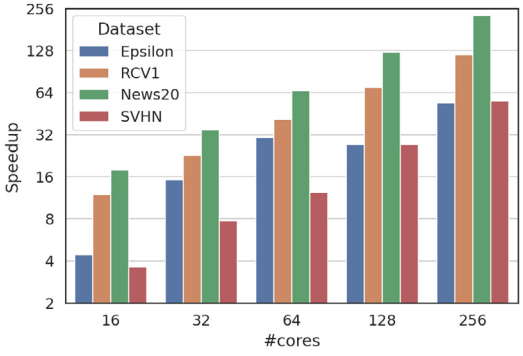
(d) ICAP



(e) DISR



(f) Method based on Beta-Gamma space



(g) CondMI

Fig. 8. Speedups of the FS methods included in Parallel-FST for a varying number of nodes, using as basis the sequential implementations available in FEAST with range compression. Each node contains 16 cores and allows for HyperThreading (two MPI processes per node, each one with 16 threads).

In this work we have presented Parallel-FST, a novel library whose aim is to accelerate the FS procedure by providing hybrid MPI/multithreaded implementations of seven FS methods (each one with two versions, with and without weights). All the methods included in the library follow the same parallel approach. Data and workload are distributed among MPI processes and C++ threads to exploit multicore clusters, including the HyperThreading technology. Furthermore, two optimization techniques (semi-distributed data loading and range compression) were implemented to improve performance and reduce memory requirements. Parallel-FST is publicly available to download under open source license at https://gitlab.com/bieito/parallel-fst.

The experimental evaluation proved that the features discarded by the methods of Parallel-FST are exactly the same as those of a widely employed sequential counterpart (FEAST), but the analysis is completed in significantly less time. The scalability of most methods is high, reaching speedups of up to 229 on a multicore cluster with 16 nodes (256 cores, 89% of efficiency) and even obtaining superlinear speedups for experiments with two, four and eight nodes (32, 64 and 128 cores, respectively). For instance, all parallel methods are able to select the most appropriate 200 features of a 7 GB dataset in less than 18 seconds when working on the whole cluster. The parallel implementations presented in this work distribute the features among MPI processes and thus they can exploit the memory of several nodes within a cluster. It means that Parallel-FST can complete FS analyses for datasets that do not fit in the memory of one single system or node and therefore cannot be computed by sequential libraries such as FEAST.

Future work can continue in three directions. First, attempt to further improve the performance of Parallel-FST by adding SIMD support with AVX directives. Second, extend the library to include parallel versions of other FS methods not based on MI (e.g. CFS). And third, work on the development of a similar parallel library focused on GPUs, so that researchers could also exploit these widely spread architectures.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

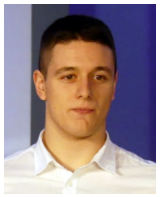
Acknowledgment

This research was supported by the Ministry of Science and Innovation of Spain (PID2019-104184RB-I00/AEI/10.13039/501100011033), by the Ministry of Universities of Spain under grant FPU20/00997, and by Xunta de Galicia and FEDER funds of the EU (CITIC, Centro de Investigación de Galicia accreditation 2019-2022, ref. ED431G 2019/01; Consolidation Program of Competitive Reference Groups, ED431C 2021/30). Funding for open access charge: Universidade da Coruña/CISUG.

References

[1] R. Battiti, Using mutual information for selecting features in supervised neural net learning, IEEE Trans. Neural Netw. 5 (4) (1994) 537–550.
[2] V. Bolón-Canedo, N. Sánchez-Maroño, A. Alonso-Betanzos, Feature Selection for High-Dimensional Data, Springer, 2015.
[3] N.M. Braman, M. Etesami, P. Prasanna, C. Dubchuk, H. Gilmore, P. Tiwari, D. Plecha, A. Madabhushi, Intratumoral and peritumoral radiomics for the pretreatment prediction of pathological complete response to neoadjuvant chemotherapy based on breast DCE-MRI, Breast Cancer Res. 19 (1) (2017) 1–14.
[4] G. Brown, A. Pocock, M.-J. Zhao, M. Luján, Conditional likelihood maximisation: a unifying framework for information theoretic feature selection, J. Mach. Learn. Res. 13 (2012) 27–66.
[5] C.-C. Chang, C.-J. Linn, LIBSVM: a library for support vector machines, ACM Trans. Intell. Syst. Technol. 2 (3) (2011) 27.
[6] W. Choi, H.J. Jo, S. Woo, J.Y. Chun, J. Park, D.H. Lee, Identifying ecus using inimitable characteristics of signals in controller area networks, IEEE Trans. Veh. Technol. 67 (6) (2018) 4757–4770.
[7] A. Das, N. Borisov, M. Caesar, Tracking mobile web users through motion sensors: attacks and defenses, in: Proceedings of the 23rd Annual Network and Distributed System Security Symposium, 2016.
[8] J. González-Domínguez, V. Bolón-Canedo, B. Freire, J. Touriño, Parallel feature selection for distributed-memory clusters, Inf. Sci. 496 (2019) 399–409.
[9] J. González-Domínguez, R.R. Expósito, V. Bolón-Canedo, CUDA-JMI: acceleration of feature selection on heterogeneous systems, Future Gener. Comput. Syst. 102 (2020) 426–436.
[10] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I.H. Witten, The WEKA data mining software: an update, ACM SIGKDD Explor. Newsl. 11 (1) (2009) 10–18.
[11] N.M. Hijazi, H. Faris, I. Aljarah, A parallel metaheuristic approach for ensemble feature selection based on multi-core architectures, Expert Syst. Appl. 182 (2021) 115290.
[12] V.J. Hodge, S. O'Keefe, J. Austin, Hadoop neural network for parallel and distributed feature selection, Neural Netw. 78 (2016) 24–35.
[13] A. Jakulin, Machine Learning Based on Attribute Interactions, Ph.D. thesis, University of Ljubljana, Slovenia, 2005.
[14] A. Kleerekoper, M. Pappas, A. Pocock, G. Brown, M. Lujan, A scalable implementation of information theoretic feature selection for high dimensional data, in: Proceedings of the 2015 IEEE International Conference on Big Data, 2015, pp. 339–346.
[15] I.O. Korolev, L.L. Symonds, A.C. Bozoki, Predicting progression from mild cognitive impairment to Alzheimer's dementia using clinical, MRI, and plasma biomarkers via probabilistic pattern classification, PLoS ONE 11 (2) (2016) e0138866.
[16] K.-Y. Lee, P. Liu, K.-S. Leung, M.-H. Wong, Very large scale ReliefF algorithm on GPU for genome-wide association study, in: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), 2015, pp. 78–84.
[17] D.D. Lewis, Feature selection and feature extraction for text categorization, in: Proceedings of the 1992 Workshop on Speech and Natural Language, 1992, pp. 212–217.
[18] G. Li, S.E. Li, B. Cheng, P. Green, Estimation of driving style in naturalistic highway traffic using maneuver transition probabilities, Transp. Res., Part C, Emerg. Technol. 74 (2017) 113–125.
[19] D. Lin, X. Tang, Conditional infomax learning: an integrated framework for feature extraction and fusion, in: Proceedings of the 2006 European Conference on Computer Vision, 2006, pp. 68–82.
[20] H. Liu, G. Ditzler, Speeding up joint mutual information feature selection with an optimization heuristic, in: Proceedings of the 2017 IEEE Symposium Series on Computational Intelligence, 2017, pp. 1–8.
[21] H. Liu, G. Ditzler, A semi-parallel framework for greedy information-theoretic feature selection, Inf. Sci. 492 (2019) 13–28.
[22] H. Liu, H. Motoda, Feature Selection for Knowledge Discovery and Data Mining, Springer Science & Business Media, 2012.
[23] P.E. Meyer, G. Bontempi, On the use of variable complementarity for feature selection in cancer classification, in: Proceedings of the Workshop on Applications of Evolutionary Computation, 2006, pp. 91–102.
[24] R.-J. Palma-Mendoza, L. de Marcos, D. Rodriguez, A. Alonso-Betanzos, Distributed correlation-based feature selection in Spark, Inf. Sci. 496 (2019) 287–299.
[25] H. Peng, F. Long, C. Ding, Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy, IEEE Trans. Pattern Anal. Mach. Intell. 27 (8) (2005) 1226–1238.
[26] S. Ramírez-Gallego, I. Lastra, D. Martínez-Rego, V. Bolón-Canedo, J.M. Benítez, F. Herrera, A. Alonso-Betanzos, Fast-mRMR: fast minimum redundancy maximum relevance algorithm for high-dimensional big data, Int. J. Intell. Syst. 32 (2) (2017) 134–152.
[27] C. Reggiani, Y.-A. Le Borgne, G. Bontempi, Feature selection in high-dimensional dataset using MapReduce, in: Proceedings of the 29th Benelux Conference on Artificial Intelligence, 2017, pp. 101–115.
[28] R. Saidi, W.B. Ncir, N. Essoussi, Feature selection using genetic algorithm for big data, in: Proceedings of the International Conference on Advanced Machine Learning Technologies and Applications, 2018, pp. 352–361.
[29] A. Salmerón, A.L. Madsen, F. Jensen, H. Langseth, T.D. Nielsen, D. Ramos-López, A.M. Martínez, A.R. Masegosa, Parallel filter-based feature selection based on balanced incomplete block designs, in: Proceedings of the 22nd European Conference on Artificial Intelligence, 2016, pp. 743–750.
[30] C.K. Sarumathiy, K. Geetha, C. Rajan, Improvement in Hadoop performance using integrated feature extraction and machine learning algorithms, Soft Comput. 24 (1) (2020) 627–636.
[31] O. Soufan, D. Kleftogiannis, P. Kalnis, V.B. Bajic, DWFS: a wrapper feature selection tool based on a parallel genetic algorithm, PLoS ONE 10 (2) (2015) e0117988.
[32] The MPI Forum, MPI: a message passing interface (version 3.1), http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, 2015.

[33] E.R. Velazquez, C. Parmar, Y. Liu, T.P. Coroller, G. Cruz, O. Stringfield, Z. Ye, M. Makrigiorgos, F. Fennessy, R.H. Mak, et al., Somatic mutations drive distinct imaging phenotypes in lung cancer, Cancer Res. 77 (14) (2017) 3922–3930.
[34] L. Venkataramana, S.G. Jacob, R. Ramadoss, A parallel multilevel feature selection algorithm for improved cancer classification, J. Parallel Distrib. Comput. 138 (2020) 78–98.
[35] H. Yang, R. Fujimaki, Y. Kusumura, J. Liu, Online feature selection: a limited-memory substitution algorithm and its asynchronous parallel variation, in: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, pp. 1945–1954.
[36] H.H. Yang, J. Moody, Data visualization and feature selection: new algorithms for nongaussian data, in: Proceedings of the 12th International Conference on Neural Information Processing Systems, 1999, pp. 687–693.
[37] S.-F. Zhang, J.-H. Zhai, S. Tian, X. Zhou, Y. Li, Feature selection for big data based on MapReduce and voting mechanism, in: Proceedings of the 2020 International Conference on Machine Learning and Cybernetics, 2020, pp. 213–218.



Bieito Beceiro received the B.S. in computer science and the M.S. in High Performance Computing (HPC) from the Universidade da Coruña (UDC), Spain, in 2020 and 2021, respectively. He is currently a Ph.D. student at the Computer Architecture Group of the UDC. His work is focused on the acceleration of machine learning methods for computational science using HPC techniques.

Jorge González-Domínguez received the B.S., M.S., and Ph.D. degrees in computer science from the Universidade da Coruña (UDC), Spain, in 2008, 2009, and 2013, respectively. He is currently an Associate Professor with the Department of Computer Engineering, UDC. His main research interests include the development of parallel applications on multiple fields, such as bioinformatics, data mining, and machine learning, focused on different architectures.

Juan Touriño is a Full Professor with the Department of Computer Engineering, Universidade da Coruña, where he also leads the Computer Architecture Group. He has extensively published in the area of High Performance Computing (HPC): HPC & AI convergence, programming languages and compilers for HPC, high-performance architectures and networks, parallel algorithms and applications in computational science and engineering. He is coauthor of more than 170 papers on these topics in international conferences and journals.