Fast-mRMR: Fast Minimum Redundancy Maximum Relevance Algorithm for High-Dimensional Big Data

Sergio Ramírez-Gallego,[1,*] Iago Lastra,[1] David Martínez-Rego,[2] Verónica Bolón-Canedo,[2] José Manuel Benítez,[2] Francisco Herrera,[1] Amparo Alonso-Betanzos[2]
[1]*Department of Computer Science and Artificial Intelligence, CITIC-UGR, University of Granada, 18071, Granada, Spain*
[2]*Department of Computer Science, University of A Coruña, 15071, A Coruña, Spain*

With the advent of large-scale problems, feature selection has become a fundamental preprocessing step to reduce input dimensionality. The minimum-redundancy-maximum-relevance (mRMR) selector is considered one of the most relevant methods for dimensionality reduction due to its high accuracy. However, it is a computationally expensive technique, sharply affected by the number of features. This paper presents *fast-mRMR*, an extension of mRMR, which tries to overcome this computational burden. Associated with *fast-mRMR*, we include a package with three implementations of this algorithm in several platforms, namely, CPU for sequential execution, GPU (graphics processing units) for parallel computing, and Apache Spark for distributed computing using big data technologies. © 2016 Wiley Periodicals, Inc.

1. INTRODUCTION

In the past few years, the dimensionality of data sets in many domains, like bioinformatics or text analysis,[1] has severely increased. This fact has raised an interesting challenge in the research community, since many machine learning (ML) methods cannot deal with a high number of input features efficiently. In fact, if we analyze the data sets posted in the popular libSVM Database,[2] we can observe that the maximum dimensionality of the data has increased to more than 29 million. Similarly, some of these algorithms also suffer when they face large sample sizes. In this new scenario, it is common now to deal with data sets that are too big both

*Author to whom all correspondence should be addressed; e-mail: sramirez@decsai.ugr.es.

INTERNATIONAL JOURNAL OF INTELLIGENT SYSTEMS, VOL. 32, 134–152 (2017)

View this article online at wileyonlinelibrary.com. • DOI 10.1002/int.21833

in terms of number of features and number of samples, so the existing learning methods need to be adapted.

To confront this problem, dimensionality reduction techniques can be applied to reduce the number of features and improve the performance of a subsequent learning process. One of the most widely used dimensionality reduction strategies is feature selection (FS), which achieves dimensionality reduction by removing irrelevant and redundant features.[3] Since FS maintains the original features, it is especially useful for applications where model interpretation and knowledge extraction are important. However, existing FS methods are not expected to scale well when dealing with large-scale problems (both in number of features and instances), due to the fact that their efficiency may significantly deteriorate or they may even become inapplicable.

Among the broad suite of available FS methods, the mRMR filter method[4] has been one of the most frequently used in the past few years. Authors in Ref. 5 report its great popularity (thanks to its accuracy), despite being a computationally expensive method. This method scales quadratically with the number of features and grows linearly with respect to the sample size. On the other hand, mRMR has been criticized for not including conditional redundancy in its computations. However, in Ref. 6, the authors show its good performance and the uselessness of this term in many problems.

In this paper, we present *fast-mRMR*, an extension of mRMR that includes several efficiency optimizations to be able to tackle the big dimensionality problem. These optimizations are based on a set of techniques (like caching relevant information or a new data-access pattern) introduced with the aim of boosting the overall performance of mRMR. To our knowledge, this is the first work in the literature that tries to address the mRMR's weaknesses. In addition to this redesign, we provide a package that includes three versions of *fast-mRMR* for different platforms[a]: a sequential version in C++ for small data sets, a GPU-CUDA (compute unified device architecture) parallel version,[7] and a distributed version using Apache Spark[8] for Big Data. All of them implement the aforementioned optimizations but with some important changes derived from each platform (e.g., the use of broadcasting in the distributed implementation).

Comprehensive experiments on different platforms and real-world data sets (up to $O(10^7)$ instances and features) has been carried out to assess the performance of this improved version. The subsequent results show that our approach greatly improves the performance of mRMR on all platforms tested.

The rest of this paper is organized as follows: In Section 2, we describe the main concepts related to this work and present the mRMR method and its importance in several real-world problems. Section 3 introduces the list of optimizations proposed, as well as the different implementations included in our package. Section 4 explains the experimental results carried out to compare our proposal with the standard version of mRMR. Finally, Section 5 concludes the paper.

[a]The source code can be downloaded from the following repository: https://github.com/sramirez/fast-mRMR.

2. PRELIMINARIES

This section introduces the main concepts related to our proposal and the associated package, such as FS, Big Data, and GPU processing. We also present mRMR and highlight the wide range of real-world problems that benefit from this technique.

2.1. Feature Selection

FS can be defined as the process of detecting the relevant features and discarding the irrelevant and redundant ones with the goal of obtaining a subset of features that describes properly the given problem with minimum performance degradation. It has several advantages, [9] such as improving the performance of the machine learning algorithms or the possibility of using simpler models and gaining speed.

Formally, a standard FS problem can be defined as follows: Let $\boldsymbol{e_r}$ be an instance $\boldsymbol{e_r} = (e_{1r}, \ldots, e_{nr}, e_{cr})$, where e_{ir} corresponds to the ith feature value of the rth sample and e_{cr} with the value of the output class c. Let us assume a training data set $\boldsymbol{D}$ with m examples, which instances $\boldsymbol{e_r}$ consist of a set $\boldsymbol{X}$ of n features, and a testing set $\boldsymbol{DT}$. Let us define $\boldsymbol{S} \subseteq \boldsymbol{X}$ as a subset of selected features generated by an FS algorithm. Those features in S are considered as the most relevant among the entire set of input features.

Feature selection methods can be divided according to two different approaches: *individual evaluation* and *subset evaluation.* [10] Individual evaluation is also known as feature ranking and assesses individual features by assigning them weights according to their degrees of relevance. On the other hand, subset evaluation produces candidate feature subsets based on a certain search strategy. Each candidate subset is evaluated by a certain evaluation measure and compared with the previous best one with respect to this measure. While the individual evaluation is incapable of removing redundant features because redundant features are likely to have similar rankings, the subset evaluation approach can handle feature redundancy with feature relevance. However, methods in this framework can suffer from an inevitable problem caused by searching through all feature subsets required in the subset generation step.

Aside from this classification, three major approaches can be distinguished based upon the relationship between a FS algorithm and the inductive learning method used to infer a model [9]:

- *Filters*, which rely on the general characteristics of training data and carry out the FS process as a preprocessing step independently to the induction algorithm. This model is advantageous for its low computational cost and good generalization ability.
- *Wrappers*, which involve using a learning algorithm as a black box and consist of using its prediction performance to assess the relative usefulness of subsets of variables. In other words, the FS algorithm uses the learning method as a subroutine with the computational burden that comes from calling the learning algorithm to evaluate each subset of features. However, this interaction with the classifier tends to give better performance results than filters.
- *Embedded methods*, which perform FS in the process of training and are usually specific to given learning machines. Therefore, the search for an optimal subset of features is

built into the classifier construction and can be seen as a search in the combined space of feature subsets and hypotheses. This approach is able to capture dependencies at a lower computational cost than wrappers.

In this work, we will be focused on a method called minimum redundancy maximum relevance (mRMR), which belongs to the category of filter methods that return an ordered ranking of all the features. More details about this method are given in the next subsection.

2.2. mRMR: Formulation and Applications

The mRMR method, first developed by Peng et al. in Ref. 4, is considered as one of the most powerful filters among the ML community as shown by its high citation count. At first, this method was mainly intended to deal with the classification of DNA microarray data,[11] which is a challenging field for ML researchers due to the extremely large number of features and the small number of samples. The authors stated that the genes (features) selected via mRMR provide a more balanced coverage of the space and capture broader characteristics of DNA phenotypes. Nowadays, the method has been used more extensively in other fields, such as anomaly detection in cooling fans,[12] eye-movement analysis,[13] gender classification,[14] or analysis of multispectral satellite images.[15] There are several high-dimensional problems in which mRMR is used as a preprocessing step, like text or image analysis.[16,17]

Scalability of mRMR cannot be deemed as negligible, but on the contrary is of major importance. In Ref. 5, an analysis of scalability of mRMR is presented, in which its frequent use in several fields due to its accuracy is reported, but also states its high complexity in computational terms. The complexity scales quadratically with the number of features and linearly with the number of samples. mRMR has also been criticized for not considering class-conditional redundancy in its selection process. However, Brown et al.[6] demonstrate in their experiments that this term is useless in many problems, and that mRMR offers one of the best trade-offs between stability and accuracy.

mRMR is used to rank the importance of a set of features for a given classification task. This method can rank features based on their relevance to the target, and, at the same time, the redundancy of features is also penalized. The main objective is to find the *maximum dependency* between a set of features $\mathbf{X}$, and the class c, using mutual information (MI) (represented by I). The MI between a pair of features is defined in Equation 1 and can be obtained effectively once the marginal probabilities $p(a)$ and $p(b)$, and the joint probability $p(a, b)$ between these two features are known.

$$I(A;B) = \sum_{b\in B}\sum_{a\in A} p(a,b)\log\left(\frac{p(a,b)}{p(a)p(b)}\right) \tag{1}$$

Implementing the maximum dependency criterion is not an easy-to-solve task in high-dimensional spaces. Namely, the number of samples is often insufficient and, moreover, estimating the multivariate density usually implies expensive

computations. An alternative is to determine the *maximum relevance* criterion. Maximum relevance consists of searching for those features which satisfy the following equation:

$$\max D(\mathbf{X}, c) ; D = \frac{1}{|\mathbf{X}|} \sum_{X_i \in \mathbf{X}} I(X_i; c) \tag{2}$$

Selecting features according to the maximum relevance criterion can bring a large amount of redundancy. Therefore, the following criterion of *minimum redundancy* must be added, as suggested by Ref. 4:

$$\min R(\mathbf{X}) ; R = \frac{1}{|\mathbf{X}|} \sum_{X_i, X_j \in \mathbf{X}} I(X_i, X_j) \tag{3}$$

The combination and optimization of both criteria D and R leads to the criterion mRMR. In practice, a greedy algorithm can be employed, where $\mathbf{S}$ is the set of selected features:

$$\max_{X_i \notin \mathbf{S}} [I(X_i; c) - \frac{1}{|\mathbf{S}|} \sum_{X_j \in \mathbf{S}} I(X_j; X_i)] \tag{4}$$

The pseudocode for original version of mRMR is presented in Algorithm 1. In this algorithm, the main bottleneck is related to the computation of the MI between two elements: either being a given feature with the class (line 4), or a pair of input features (line 7). This function is computed between each pair of features despite many pairs of features being irrelevant for the final result, which is inefficient in high-dimensional problems.

Algorithm 1 mRMR: original algorithm

```
INPUT: candidates, numFeaturesWanted
// candidates is the set of initial features
// numFeaturesWanted is the number of selected features
OUTPUT: selectedFeatures // The set of selected features.
for  feature fi in candidates do
   relevance = mutualInfo(fi, class);
   redundancy = 0;
   for  feature fj in candidates do
      redundancy += mutualInfo(fi, fj);
   end for
   mrmrValues[fi] = relevance - redundancy;
end for
selectedFeatures = sort(mrmrValues).take(numFeaturesWanted);
```

This criterion has proven to be one of the most relevant in the literature. Its optimization is the main focus of the following sections since, if directly implemented, mRMR can be slow, and its scalability might be compromised.

2.3. High-Dimensional Feature Selection in Big Data

Extracting valuable information from huge collections of data has become one of the most important and complex challenges in data analytics research. This concept, known as Big Data, has caused many knowledge extraction algorithms turn into obsolete methods when they face such vast amounts of data. As a result, the need for new methods, capable of managing such a large amount of data efficiently with similar performance, arises. Gartner[18] introduced the 3Vs concept by defining Big Data as high volume, velocity, and variety information that require a new large-scale processing. Afterward, this list was extended with two additional Vs: veracity and value.

From the beginning, data scientists have generally focused on only one side of Big Data, which in early days referred to the huge number of instances; paying less attention to the feature side.[1] This phenomenon, also known as the "Big Dimensionality," arose as a result of the explosion of features from new incoming data where thousands or even millions of features are present. Big Dimensionality, though, calls for new FS strategies and methods that are able to deal with the combinatorial effects derived from this phenomenon. The most famous public data set repositories in computational intelligence (like UCI or libSVM)[2,19] already reflect this phenomenon in most of their new data sets.

Not only the amount of features but also the great variety of feature types and combinations are becoming a standard in many applications nowadays. Since not all the features contribute equally to the results, FS is thus required by the learning and prediction processes for a fast and cost-effective performance, now more than ever. Selecting remarkable features from the original set of input features (potentially irrelevant, redundant, and noisy), while maintaining the requirements in measurement and storage, is one of the most important challenges in Big Data research.

Many platforms for large-scale processing have emerged in the Big Data environment in past years. Apache Spark,[8] as one of the most powerful engines in this environment, is aimed at performing faster distributed computing on Big Data by using in-memory primitives. This platform allows user programs to load data into memory and query it repeatedly, making it a well suited tool for online and iterative processing (specially for ML algorithms). In Spark, the driver (main program) is in charge of controlling multiple workers (slaves) and collecting information from them, whereas worker nodes read data blocks (partitions) from a distributed file system, perform some computations and save the resulting partitions.

Spark is based on distributed data structures called resilient distributed data sets (RDDs). Operations on RDDs automatically place tasks into partitions, maintaining the locality of persisted data. Beyond this, RDDs are a versatile tool that let programmers persist intermediate results into the memory/disk for reusability purposes and customize the partitioning to optimize data placement.

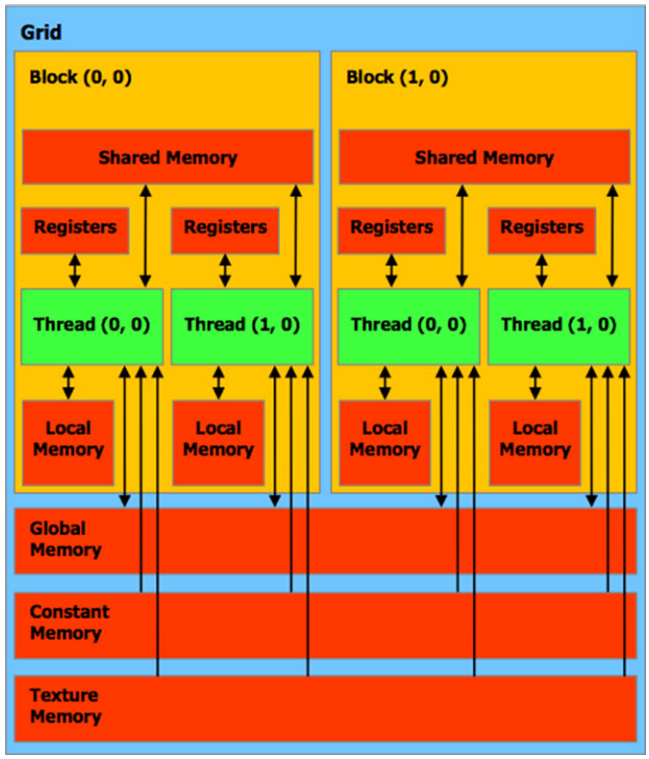


Figure 1. Nvidia scheme, with two blocks in a grid.

2.4. GPU Background

The use of GPUs for rendering is well known, but their power for general parallel computation (like machine learning or numerical calculus) has only recently been explored.[20,21] Parallel algorithms running on GPUs can often achieve up to 100 × speedup over similar CPU algorithms, with many existing applications for physics simulations, signal processing, financial modeling, neural networks, and countless other fields. For that reason, it is worthwhile adapting mRMR to this parallel environment, specially when the data size exceeds the capability of the CPU version.

CUDA is a parallel computing platform and programming model created by NVIDIA[7] and implemented in its GPUs. CUDA gives direct access to the virtual instruction set and the parallel memory in GPUs. In CUDA, the computation is distributed across a grid of thread blocks, with all the blocks containing the same number of threads (see Figure 1). A kernel is a special program designed to be executed in parallel on multiple threads, which are grouped in blocks. Only threads in the same block can communicate directly and synchronize each other.

The kernels can make use of registers, shared and global memory to make their calculations and communicate with other threads. The first two levels can be

accessed very quickly but, unfortunately are very scarce. If the program uses too many of these resources, many processing units are to remain unemployed during a kernel execution due to a lack of resources. On the other hand, global memory is usually not limited for practical purposes, but its access is too slow. Thus, the kernel must be carefully designed to balance (a) the use of shared memory and registers, (b) the active processing units, and (c) the number of global memory accesses and their access patterns. A typical CUDA program consists of the following stages:

- allocate GPU global memory region,
- move data from random access memory (RAM) to global memory,
- run the GPU kernel(s),
- bring the results back from global memory to RAM, and
- free GPU global memory.

For mRMR, this means that we have to move each involved feature from RAM to GPU global memory, run the histogramming kernel, and then return the result back to RAM to continue with mRMR's main loop.

3. FAST-MRMR: AN EXTENSION OF MRMR FOR BIG DIMENSIONALITY

In this section, we describe the extension of mRMR proposed, as well as the list of optimizations associated with this method. Furthermore, a software package with different implementations of *fast-mRMR* is presented. The parallel and distributed versions of this package are explained into detail since they imply important changes to the original extension.

3.1. Fast-mRMR: Optimizations

In the original version of mRMR, the main bottleneck is related to the computation of the MI between two elements: either a given feature with the class or a pair of input features. To cope with this problem, we propose a greedy approach for mRMR described in Algorithm 2. This algorithm uses the number of features to select to limit the number of comparison between features. As mRMR is considered as a ranking method, the greedy search will not affect the final result as the algorithm aims at selecting a reduced number of features (specified as input). In this manner, the original complexity will be transformed into an iterative process (linear order), limited by a small number of iterations (the number of features selected).

Fast-mRMR first computes the relevance values for all input features and caches them (lines 3–7) to be reused in the following steps. The best feature according to relevance is selected and set as reference for the redundancy phase (lines 9–11). A loop is then started to select the remaining features according to the mRMR criterion. For each selected feature, the algorithm computes the MI between this feature and the rest of nonselected ones (lines 13–21). The resulting values are cached and accumulated in variables associated with the candidate features in each iteration. Finally, the best feature according to mRMR is added to the final set and

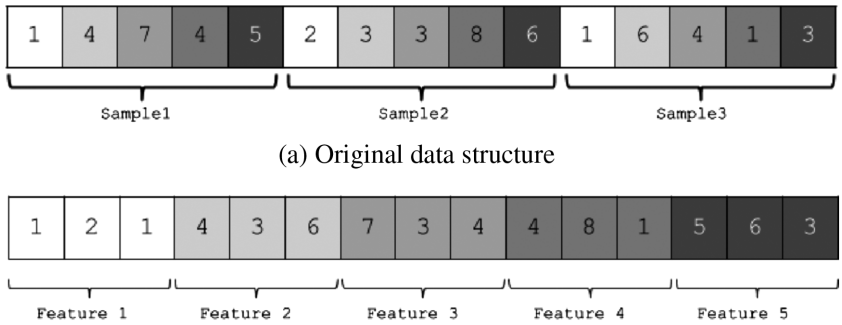
marked as the new maximum (lines 22 and 23). The loop ends when features are selected.

Algorithm 2 fast-mRMR: main algorithm

```
1:  INPUT: candidates, numFeaturesWanted
2:  // candidates is the set of initial features
3:  // numFeaturesWanted is the number of selected features
4:  OUTPUT: selectedFeatures // The set of selected features.
5:  selectedFeatures = ();
6:  for feature f in candidates do
7:     relevancesVector[f] = mutualInfo(f, class);
8:     acumulatedRedundancy[f] = 0; //Begin with no redundancy.
9:  end for
10: selected = getMaxRelevance(relevancesVector);
11: lastFeatureSelected = selected
12: selectedFeatures.add(selected);
13: candidates.remove(selected);
14: while selectedFeatures.size() <numFeaturesWanted do
15:    for feature fc in candidates do
16:       relevance = relevancesVector[fc]
17:       acumulatedRedundancy[fc] += mutualInfo(fc, lastFeatureSelected);
18:       redundancy = acumulatedRedundancy[fc] / selectedFeatures.size();
19:       mrmr = relevance - redundancy;
20:       if mrmr is maximun then
21:          fc = lastFeatureSelected;
22:       end if
23:    end for
24:    selectedFeatures.add(lastFeatureSelected);
25:    candidates.remove(lastFeatureSelected);
26: end while
```

Furthermore, we apply a list of optimizations designed to alleviate the complexity of these costly operations. These optimizations are described in the following list:

- *Accumulating redundancy*: Computing the MI between every pair of features can be expensive. A possible optimization is to accumulate the redundancy in each iteration. Thus, only the MI between the set of nonselected features and the last selected feature needs to be calculated. As mentioned before, we use a greedy approach to tackle this problem (see Equation 4).
- *Caching marginal computations*: To avoid computing marginal probabilities in each iteration, some marginal proportions are calculated only once at the start of the program and cached to be reused in the following iterations.
- *Data-access pattern*: The data access pattern of mRMR is thought to be featurewise, in contrast to many other ML algorithms, in which the access pattern is mainly rowwise (see Figure 2a). Although being a low-level technical nuance, this aspect can significantly degrade its performance since random access has a much greater cost than featurewise access. This is specially important for GPUs, where the data have to be transferred from RAM to GPU global memory. Here, the way in which the data are stored in memory is reorganized to be in a columnar format (as shown in Figure 2).



(a) Original data structure

(b) Refactored data structure

Figure 2. Example with three samples and five features, with each feature represented by a different color.

To assure that the original algorithm does not include the aforementioned optimizations, we have studied the source code of mRMR from the authors' webpage[b] in depth. Notice that the aforementioned optimizations only affect the performance of the original and do not alter the final result at all. Therefore, our approach can be deemed as an optimized and nonapproximative version of mRMR.

Associated with *fast-mRMR*, we provide a package that includes several implementations, designed to give support to the ML community: a sequential version in C++ for CPU, a parallel version in CUDA for GPUs ,[7] and a distributed version in Scala for Apache Spark framework.[8] The sequential version is a direct implementation of the previous list of optimizations in C++. The following sections will describe in detail the other versions since they imply some nontrivial changes to the original extension.

3.2. Distributed Fast-mRMR for Apache Spark

In this section, the procedure used to adapt *fast-mRMR* to the distributed paradigm is explained. Most of the code has been adapted to this new paradigm, except the main loop (Algorithm 2), which has remained unchanged. This new version consists of the following steps:

1. *Columnar transformation*: The idea behind this transformation is to transpose the local matrix provided by each partition of the original data, so that the resulting data can be cached (in memory) in this new format, and reused in the following steps (leveraging for data locality). After applying this operation, new matrices are yielded in each partition/block with one row per feature, instead of one row per instance. To put together all the blocks for the same feature, the algorithm collects and groups them in the same set of partitions. This operation preserves the original partitioning scheme of the data and, at the same time, the data locality. Figure 3 explains the aforementioned process using a small example with two partitions and four features.

[b]Peng's webpage: penglab.janelia.org/proj/mRMR/.

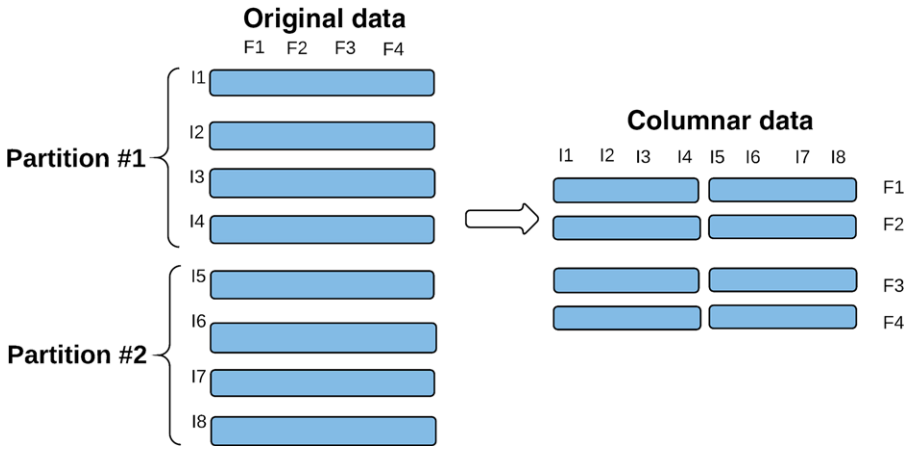


Figure 3. Columnar transformation scheme. *F* indicates features and *I* instances. Each rectangle on the left represents a single register in the original data set. Each rectangle on the right represents a transposed feature block in the new columnar format.

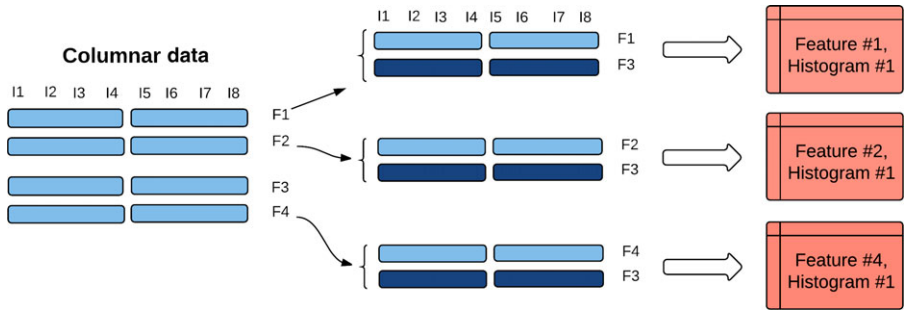


Figure 4. Histograms creation scheme. *F* indicates features and *I* instances. Each lighter rectangle represents a single feature block in the columnar format. Darker rectangles represent the distributed copies of the secondary variable.

2. *Relevance and redundancy*: Once the data are in a columnar format, the algorithm computes the histograms for all the candidate features (see Figure 4). These histograms count the number of occurrences between each candidate feature and the secondary one (the class or the last selected feature), according to Algorithm . This operation is performed in an isolated way and keeping the data locality, so that each feature has all the information needed to compute the relevance or redundancy independently. To do that, the algorithm replicates the secondary feature across all the nodes, so that all the candidate features have access to this variable. Roughly, the algorithm iterates over the blocks derived from the columnar transformation, generating a local histogram per partition and feature (for high-dimensional problems, only one histogram per feature). Each local histogram is generated by counting the combinations between the candidate feature and the secondary one. Finally, all the partial matrices are aggregated by feature to obtain the final histograms.
3. *Mutual information*: This phase is aimed at computing the MI values used to rank the candidate features. As a preliminary step, the algorithm replicates the marginal proportions and the values of the secondary variable across the cluster. An independent

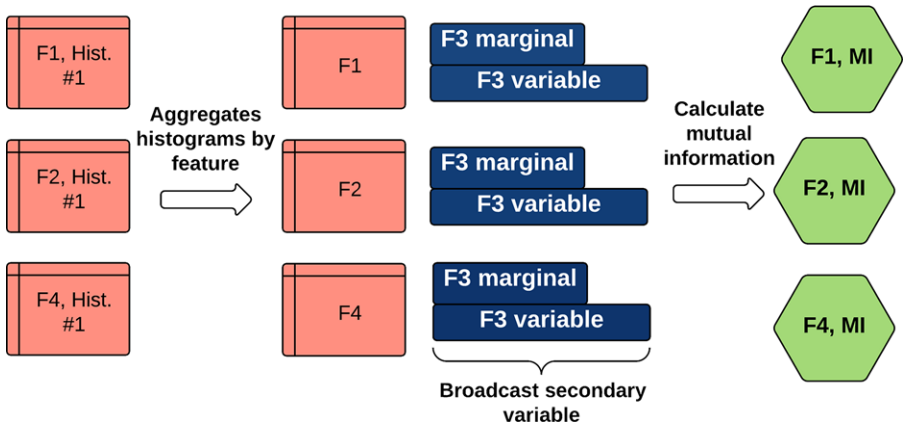


Figure 5. MI computations. *F* indicates features. The matrices represent the histograms for each feature. Darker rectangles represent the broadcasted variables associated with the secondary variable.

procedure is then started on each final histogram. This procedure preserves the data locality and is performed on each feature in an independent way. It starts by calculating the marginal proportions associated with each histogram/feature, as well as the joint probability with respect to the secondary variable. Then, with all the information needed, the results for each combination are computed and aggregated to get the final MI value per feature (see Equation 1). Figure 5 presents an scheme of this operation.

3.3. Parallel Fast-mRMR for GPU Processing

For applying fast-mRMR to GPUs, we have adopted a hybrid approach for MI and marginal probability calculations based on previous studies. This hybrid strategy follows this logic:

1. If the number of possible outcomes is below 64, current GPUs can make use of the full set of processing units without using global memory. So, the kernel in Algorithm 3 is used.
2. If the number of possible outcomes is greater than 64, and less than 256, the kernel shown in Algorithm 3 is no longer valid, since a lack of shared memory would produce a sharp decrease in the processing units usage. In this case, the kernel presented in Algorithm 4 is used.
3. If the number of possible outcomes exceeds 256, then the shared memory is no longer an option by itself, so the calculations should be partitioned. This strategy is detailed in Algorithm 5.

3.3.1. Per-Thread Kernel

This kernel is designed for those cases in which there are fewer than 64 values. Given the current memory constraints, there is enough memory to have a local histogram for each thread (maximum performance). No memory access conflicts are thus present in this case as the histograms are computed independently. Notice that in subsequent kernels a slow atomic operation called *atomicAdd* is required to

cope with these conflicts, whereas in Algorithm a simple addition is performed. In this situation, each thread does not have to worry about serializing memory writes and a significant speedup is consequently achieved.

Algorithm 3 Per thread Kernel

```
1: INPUT: data //The vector with the feature values.
2: OUTPUT: histogram // A vector containing the count for each possible value.
3: //Initialize one localHistogram per thread to zeros.
4: i = threadId;
5: while i < data.size() do
6:     localhistogram[data[threadId]]++;
7:     i += totalActiveThreads;
8: end while
9: reduceLocalHistograms(histogram) //partial histogram are merged in histogram.
10: return histogram;
```

3.3.2. Per Warp-Kernel

This kernel is designed to calculate histograms up to 256 possible values and is mainly used to calculate the marginal probabilities (see Algorithm 4). CUDA offers 48 Kb of memory for every streaming multiprocessor (SMX), and there are at most 64 warps (a set of 32 consecutive threads) for each SMX. This means that if all the warps were running, each warp would have 768 bytes of shared memory available. Since 768 bytes are not enough to have a histogram of 256 values per warp, a small percentage (much smaller than if a per-thread kernel was used) of processing units should remain idle. Each warp will have its local histogram in shared memory and will perform a linear pass through the corresponding positions. Once local histograms are calculated, they are finally merged into the global memory and sent back to RAM.

Algorithm 4 Per warp Kernel

```
1: INPUT: data //The vector with the feature values.
2: OUTPUT: histogram // A vector containing the count for each possible value.
3: //Initialize one sharedHistogram per warp to zeros.
4: i = threadId;
5: while i < data.size() do
6:     atomicAdd(sharedHistogram[data[threadId]], 1);
7:     i += totalActiveThreads;
8: end while
9: reduceSharedHistograms(histogram) //partial histogram are merged in histogram.
10: return histogram;
```

3.3.3. Joint Kernel

This last kernel is designed to calculate the joint probabilities and allows for the computation of histograms with up to 65,536 values. Since, in these cases, shared memory is completely surpassed, the calculations are partitioned balancing

the number of passes through the data set and the usage of shared memory. With a current limit of 768 values per warp, [7] the kernel would take three passes to compute a histogram of 1756 possible outcomes. In real cases, better computing times can be obtained by keeping idle warps and by computing more values of the histogram at each pass. Although in this manner each pass is slower, the final execution time could be lower because less synchronizations are needed. This kernel is accomplished by adding a lap counter to the main loop (see Algorithm 5), to know which subhistogram is calculated in each iteration.

Algorithm 5 Joint Kernel Main Loop

```
1: INPUT: data //The vector with the values from two features.
2: OUTPUT: histogram // A vector containing the count for each possible combination.
3: totalBins = getTotalBins(data);
4: for i = 0; i <totalBins / maxBinsPerStep; i++ do
5:    kernelThird(data, lap, globalHistogram)
6: end for
```

Algorithm 6 Joint Kernel

```
1: INPUT: data, lap, globalHistogram
2: // data is The vector with the feature values.
3: // globalHistogram is the vector where shared histograms will be merged.
4: OUTPUT: histogram // A vector containing the count for each possible value.
5: if lap = 0 then
6:    Initialize globalHistogram to zeros.
7: end if
8: Initialize sharedHistogram per warp to zeros.
9: i = threadId;
10: while i < data.size() do
11:    bin = data[threadId]];
12:    if bin ≥ lap * maxBinsPerStep and bin <(lap + 1) * maxBinsPerStep then
13:       atomicAdd(sharedHistogram[bin], 1);
14:    end if
15:    i += totalActiveThreads;
16: end while
17: reduceSharedHistograms(globalHistogram) //partial histogram are merged.
18: return globalHistogram;
```

4. EXPERIMENTAL ANALYSIS

This section is aimed at evaluating the performance of the different implementations of *fast-mRMR*. First, we present the experimental framework used to evaluate the different versions of *fast-mRMR*. The results derived from these experiments are also analyzed below.

4.1. Experimental Framework

For sequential experiments, a standard computer with the following features has been used: Intel Core i5-4690S (6 M cache, up to 3.90 GHz), 4 GB RAM DDR3

Table I. mRMR versus CPU fast-mRMR (in seconds) for 200 features selected.

Data sets	Number of features	Number of instances	mRMR	CPU	Speedup
Lung	326	73	23.27	0.06	387.83
NCI	9,173	60	39.41	2.02	19.51
Colon	2,001	62	40.24	0.37	108.76
Leukemia	7,071	72	43.54	1.51	28.83
Lymphoma	4,027	96	50.81	0.95	53.48

DIMM, and 1 TB HDD (7200 rpm S-ATA). In this comparison, all the data sets used in Ref. 4 have been considered in our experiments.[c]

For the comparison of GPU and CPU versions, a set of synthetic data sets with different sampling ratios were generated. These values were randomly generated in the range [0, 30] following an uniform distribution. Although being uniformly distributed, they were generated to prove the efficiency of fast-mRMR, which is not influenced by the specific distribution of data. Finally, a NVIDIA GeForce GTX 780M device was used as a reference for GPU executions.

Finally, the distributed version implemented on Spark is evaluated using three large-scale data sets: The first one (henceforth called ECBDL14) was used as a reference at the GECCO-2014 international conference.[d] For this imbalanced problem, the MapReduce version of the Random OverSampling (ROS) algorithm presented by Ref. 22 was used. The other data sets (*epsilon* and *kddb*) are part of the LibSVM repository.[2] For these experiments, a cluster composed of 18 computing nodes and one master node was used. The computing nodes hold the following characteristics: two processors $\times$ Intel Xeon CPU E5-2620, six cores per processor, 2.00 GHz, 15 MB cache, QDR InfiniBand network (40 Gbps), 2 TB HDD, 64 GB RAM.

4.2. Results and Analysis

Table I and Figure 6 depict the comparison performed between the sequential version of *fast-mRMR* and mRMR in terms of selection time. The results show that *fast-mRMR* outperforms the original version in all the cases studied, achieving a maximum speedup (mRMR time/fast-mRMR time) value of 387.83.

In Table II, the GPU and CPU versions of *fast-mRMR* are compared. As shown in this table, the GPU version obtains better results for data sets with a high number of instances, whereas the CPU version is slightly better for the small ones.

The performance of the distributed version is assessed in Table III. The results show how the Spark version achieves good speedup values (defined as CPU time/Spark time), proving that the computational burden is fairly distributed over the cluster. Note that, for *ECBDL14*, CPU time was estimated using a subset; and for *kddb*, this was not possible because the data set is in sparse format, which is

[c] All of them can be downloaded from Peng's webpage: http://penglab.janelia.org/proj/mRMR/#data.

[d] http://cruncher.ncl.ac.uk/bdcomp/.

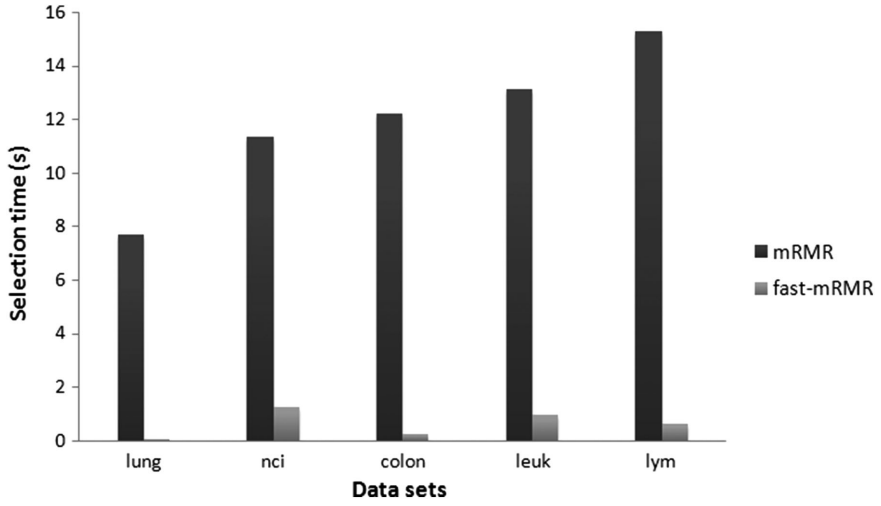


Figure 6. mRMR versus sequential fast-mRMR. 200 features selected.

Table II. CPU versus GPU version (in seconds). 1000 features and 100 features selected.

Number of instances	CPU	GPU	Speedup
160	0.08	0.50	0.16
1,600	0.28	0.67	0.42
16,000	0.53	0.75	0.71
160,000	2.64	1.04	2.54
1,600,000	22.49	5.28	4.26
3,200,000	42.34	7.96	5.32

Table III. CPU versus Spark version (in seconds). 100 features selected.

Data sets	Number of features	Number of instances	CPU	Spark	Speedup
ECBDL14	631	65,003,913	11,281.27	2,420.94	4.65
epsilon	2,000	400,000	2,553.79	542.05	4.71
kddb	29,890,095	19,264,097	-	2,789.55	-

not accepted by the original version. Note that the classification performance of *fast-mRMR* has not been evaluated since it selects the same set of features as the original mRMR for all cases.

An additional study has been carried out to show the performance of the algorithm for a different amount of resources, in this case, varying the number of cores used in the selection process. Here, the ECBDL14 data set was used as a reference, using the same parameters as in the previous study. Figure 7 depicts the time spent by our algorithm when selecting 100 features, and with different numbers of cores from 10 to 100. Figure 7 shows a logarithmic curve as the number of cores is increased. Notice that for 10 cores, there is not enough memory

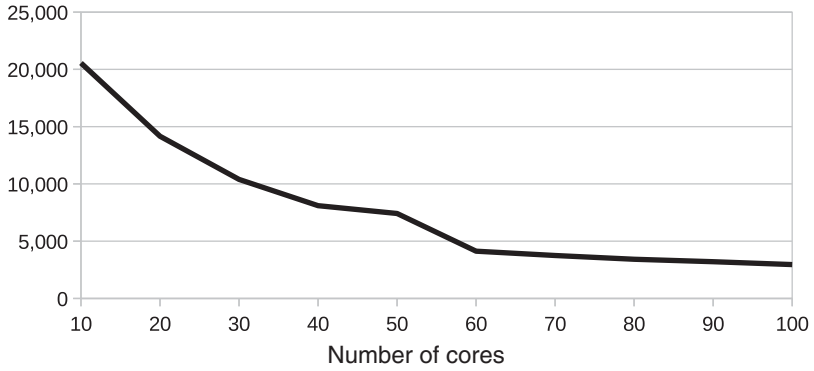


Figure 7. Time spent (in seconds) selecting top-100 features versus number of cores used.

to persist the whole data set in memory, which affects the performance of the algorithm.

According to the previous results, we can draw some conclusions about the differences and the use of the aforementioned versions. As shown in Table II, it is worthwhile to use the parallel implementation (GPU) in cases where the number of instances is big enough ($> 100,000$ approximately). In these cases, the CPU version starts to run slower than the GPU version and a parallel execution becomes the better choice. The main factor that sharply affects the performance of the aforementioned versions is the number of features to be processed. In Table III, we can observe how the CPU version does not deal well with a high number of features (≥ 2000 approximately), In these situations, the Spark version is the best choice. Similarly, when the number of instances becomes huge ($> 20,000,000$ approximately), a distributed execution is needed. Finally, the Spark version is required again for sparse problems where millions of features are present. Note that the classification performance of the optimized version has not been evaluated in this work since this version generates by definition the same results (in terms of selected features) as the original mRMR.

5. CONCLUDING REMARKS

In this paper, we have presented an extension of mRMR, called *fast-mRMR*. This extension includes several optimizations that avoid unnecessary calculations and optimize the data-access pattern held by the original algorithm. Through these optimizations, the original quadratic complexity of mRMR has been transformed into an efficient greedy process. Similarly, the performance of MI computations has been improved drastically. We also provide a package with three implementations of *fast-mRMR*: a sequential one for C++, a parallel one for GPUs, and a distributed one for Apache Spark. The last two versions have entailed a complete redesign of the algorithm through the native primitives of each platform.

The experimental results show that our proposal is capable of outperforming the original version of mRMR up to 374 times in some cases. Additional experiments with large data sets (up to $O(10^7)$ instances and features) show a clear improvement when using the parallel and distributed versions over the sequential one in the largest cases. Additionally, the aforementioned results demonstrate the necessity of an optimized version for mRMR at the current time, where the size of real-world data sets is always increasing. This is specially important for Big Data, where our approach enables the resolution of cases that were not practical with the classical approach.

Acknowledgments

This work is supported by the Spanish National Research Project TIN2013-47210-P, TIN2014-57251-P, and TIN-2015-65069-C2-1-R, and the Andalusian Research Plan P11-TIC-7765 and P12-TIC-2958, and by the Xunta de Galicia through the research project GRC 2014/035 (all projects partially funded by FEDER funds of the European Union). S. Ramírez-Gallego holds a FPU scholarship from the Spanish Ministry of Education and Science (FPU13/00047). D. Martínez-Rego and V. Bolón-Canedo acknowledge support of the Xunta de Galicia under postdoctoral grant codes POS-A/2013/196 and ED481B 2014/164-0.

References

1. Zhai Y, Ong Y, Tsang IW. The emerging "big dimensionality". IEEE Comput Intell Mag 2014;9(3):14–26.
2. Chang C-C, Lin C-J. LIBSVM: a library for support vector machines. ACM Trans Intell Syst Technol 2011;2:27:1–27:27. Datasets available at http://www.csie.ntu.edu.tw/∼cjlin/libsvmtools/datasets/.
3. Liu H, Motoda H. Feature selection for knowledge discovery and data mining. Norwell, MA: Kluwer; 1998.
4. Peng H, Long F, Ding C. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. IEEE Trans Pattern Anal Mach Intell 2005;27(8):1226–1238.
5. Rego-Fernández D, Bolón-Canedo V, Alonso-Betanzos A. Scalability analysis of mRMR for microarray data. In: Proc 6th Int Conf on Agents and Artificial Intelligence, March 6–8, 2014, in ESEO, Angers, Loire Valley, France. pp 380–386.
6. Brown G, Pocock A, Zhao M, Luján M. Conditional likelihood maximisation: a unifying framework for information theoretic feature selection. J Mach Learn Res 2012;13:27–66.
7. NVIDIA-CUDA. Programming guide. http://docs.nvidia.com/cuda/index.html. Accessed April 2016.
8. Apache Spark: Lightning-fast cluster computing. Apache Spark; 2015. http://shop.oreilly.com/product/0636920028512.do. Accessed April 2016.
9. Guyon I. Feature extraction: foundations and applications. Studies in Fuzziness and Soft Computing, Vol. 207. Berlin: Springer; 2006.
10. Yu L, Liu H. Efficient feature selection via analysis of relevance and redundancy. J Mach Learn Res 2004;5:1205–1224.
11. Ding C, Peng H. Minimum redundancy feature selection from microarray gene expression data. J Bioinform Comput Biol 2005;3(2):185–205.
12. Jin X, Ma E, Chang L, Pecht M. Health monitoring of cooling fans based on Mahalanobis distance with mRMR feature selection. IEEE Trans Instrum Meas 2012;61(8):2222–2229.
13. Bulling A, Ward J, Gellersen H, Troster G. Eye movement analysis for activity recognition using electrooculography. IEEE Trans Pattern Anal Mach Intell 2011;33(4):741–753.

14. Tapia J, Perez C. Gender classification based on fusion of different spatial scale features selected by mutual information from histogram of lbp, intensity, and shape. IEEE Trans Inform Forensics Secur 2013;8(3):488–499.
15. Bratasanu D, Nedelcu I, Datcu M. Interactive spectral band discovery for exploratory visual analysis of satellite images. IEEE J Sel Top Appl Earth Obs Remote Sens 2012;5(1):207–224.
16. Hu Y, Milios EE, Blustein J. Interactive feature selection for document clustering. In: Proc 2011 ACM Symp on Applied Computing (SAC '11), March 21–24, 2011, in Tunghai University, Taichung, Taiwan. pp 1143–1150.
17. Guo D. Coordinating computational and visual approaches for interactive feature selection and multivariate clustering. Inform Visual 2003;2(4):232–246.
18. Laney D. 3D data management: controlling data volume, velocity and variety; 2001. http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf. Accessed April 2016.
19. Lichman M. UCI machine learning repository; 2013.
20. Krüger J, Westermann R. Linear algebra operators for GPU implementation of numerical algorithms. ACM Trans on Graph 2003;22:908–916.
21. Catanzaro B, Sundaram N, Keutzer K. Fast support vector machine training and classification on graphics processors. In: Proc 25th Int Conf on Machine Learning (ICML '08), July 5–9, 2008, in Helsinki, Finland. pp 104–111.
22. del Río S, López V, Benítez JM, Herrera F. On the use of MapReduce for imbalanced big data using Random Forest. Inform Sci 2014;285(285):112–137.