Fast-mRMR: an optimal CPU/GPU implementation of minimum Redundancy Maximum Relevance algorithm

Iago Lastra Rodriguez, David Martinez-Rego, Verónica Bolón-Canedo, and Amparo Alonso-Betanzos

Department of Computer Science Faculty of Informatics, University of A Coruna
Campus de Elvina s/n, 15071 A Coruna, Spain
{iago.lastra@gmail.com,dmartinez@udc.es,veronica.bolon@udc.es,ciamparo@udc.es}

Abstract. With the advent of large-scale applications, feature selection has become a fundamental preprocessing step in order to reduce input dimensionality. Among the broad suite of feature selection algorithms available, the minimum-Redundancy-Maximum-Relevance (mRMR) method is widely-used in different applications due to its high accuracy. However, it is a computationally expensive method, sharply affected by the number of features. In this paper several optimizations based both on CPU and GPU (Graphics Card Unit) capabilities are presented to overcome its computational burden. Experimental results over different synthetic and real datasets have shown that the proposed algorithm is sound and speeds up execution time up to 600 times.

Keywords: Feature Selection, Optimization, High-performance computing, GPU

1 Introduction

In the last years, the dimensionality of datasets in many domains such as bioinformatics or text analysis has severely increased. This fact has brought an interesting challenge to the research community, since for machine learning methods it is difficult to deal with a high number of input features. To confront the problem of a high number of input features, dimensionality reduction techniques can be applied to reduce the dimensionality of the original data and improve learning performance.

One of the most widely used dimensionality reduction strategies is *feature selection*, which achieves dimensionality reduction by removing irrelevant and redundant features [1]. It is applied in data mining applications like DNA microarray analysis [2], intrusion detection [3, 4], text categorization [5, 6] or information retrieval [7], including image retrieval [8] or music information retrieval [9]. Since feature selection maintains the original features, it is especially useful for applications where these are important for model interpretation and knowledge extraction Among the broad suite of feature selection methods available,

the mRMR filter method has been one of the most frequently used in the last years. At the beginning, this method was mainly intended to deal with the classification of DNA microarray data [10], which is a challenging field for machine learning researchers due to the extremely large number of features and small sample size. 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 extended its use to other fields such as anomaly detection in cooling fans [11], eye movement analysis [12], gender classification [13] or analysis of multispectral satellite images [14]. In [15], an analysis of scalability of the mRMR method is presented, in which it is reported its frequent use in several fields due to its accuracy, but also states that its application is computationally expensive. mRMR's execution time scales quadratically with the number of features and grows linearly with respect to the sample size. There are several applications in which mRMR is used as a prepossessing step with datasets that exhibit high dimension such as text and image analysis[14, 16, 17]. Scalability of mRMR cannot be deemed as negligible, but on the contrary is of main importance.

In this paper we present several optimizations of the mRMR algorithm which significantly reduce its computational burden. First, a dynamic programming approach is taken in its main loop, which avoids all unnecessary computations. Further, information theoretic calculations are transferred to GPU giving a final algorithm which can run 600 times faster than the original version. Thanks to the proposed GPU implementation, mRMR can give performances close to a cluster in an usual workstation.

The rest of this paper is organized as follows. In Section 2, the mRMR formulation is described and the CPU and GPU optimizations are presented in Section 3 and Section 4 respectively. Finally, experimental results in which the proposed algorithms are compared with the standard and widely used mRMR original version [18] are presented in Section 5. The behaviour of the new proposed implementations demonstrates their interest, especially for heavy datasets.

2 Algorithm description

The mRMR method, first developed by Peng, Long and Ding [18], 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. Features that have the best trade-off between maximum relevance to target and minimum redundancy are considered as informative features.

Feature selection basic purpose is to find *maximum dependency* between a feature set S with m features $\{x_i\}$ and the target class c. It is calculated as the maximum mutual information (I), described by the authors [18] as:

$$max\ Dp(S,c)\ ; Dp = I(\{x_i, i = 1, ..., m\}; c) \tag{1}$$

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 on searching those features which satisfy the following equation:

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

Selecting the 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 [18] :

$$min\ R(S)\ ; R = \frac{1}{|S|^2} \sum_{x_i, x_j \in S} I(x_i, x_j) \tag{3}$$

Combining the above two criteria and trying to optimize D and R at the same time, the criterion called minimum redundancy maximum relevance (mRMR) arises.

$$\Phi = D - R \tag{4}$$

In practice, a greedy algorithm can be employed, where S_{m-1} is the selected feature set with $m-1$ features.

$$max_{x_j \in X - S_{m-1}} [I(x_j; c) - \frac{1}{m-1} \sum_{x_i \in S_{m-1}} I(x_j; x_i)] \tag{5}$$

This criterion has been proven optimal in [18] and its optimization is the main focus of the next sections since, if directly implemented, mRMR can be slow and its scalability might be compromised.

3 Optimizations

Algorithm 1 shows the main steps of the proposed mRMR implementation. In its first step (line 3), relevance of all the candidate features to the class is determined, which is achieved by calculating mutual information between them (line 4). Also the redundancy between each feature and the selected features set (S) is initialised to 0 (line 5). After that, the feature which maximizes the relevance (line 7) is added to S (line 9) (every time that a feature is selected it is removed from the candidate features set (line 10)).

In the main loop (line 11), the accumulated redundancy for each feature f is updated adding the mutual information between the latest feature added to S and f (line 14), thus the total redundancy can be calculated doing: $\frac{accumulatedRedundancy}{|S|}$. At the end of this loop (line 21) the feature which maximizes relevance minus redundancy is selected.

Algorithm 1 mRMR Optimized

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

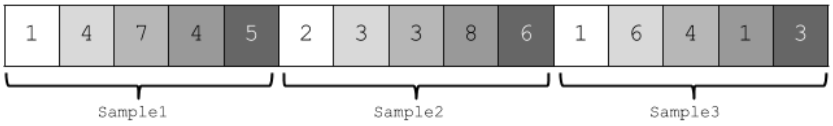
Since the main calculation is to compute the mutual information between two columns (either features or class), (lines 7 and 14) our goal is to make this calculation as fast as possible. Mutual information is defined in equation (6) and can be obtained effectively once knowing marginal probabilities $p(x)$ and $p(y)$ and joint probability $p(x, y)$ for the features X and Y.

$$I(X;Y) = \sum_{y \in Y} \sum_{x \in X} p(x,y) \log(\frac{p(x,y)}{p(x)p(y)}) \tag{6}$$

The following optimizations are applied in order to speed up the mutual information calculation.

- **Cache marginal probabilities:** Instead of obtaining the marginal probabilities in each calculation of mutual information (line 4 and line 14), those are computed only once at the beginning of the program, and cached.
- **Accumulating redundancy:**This is the most important step of the optimization. Instead of calculating the mutual information between each candidate feature and every selected feature in S, now redundancy is accumulated in each iteration, in a dynamic programming style (line 14).
- **Data access pattern:** As can be observed in Algorithm 1, the access pattern of mRMR to the dataset is by feature, in contrast to many other ML (machine learning) algorithms, in which access pattern is pattern-wise. Al-

though being a low-level technical nuance, this aspect can significantly degrade mRMR performance since random access has a much greater cost than block-wise access. This is specially important in the case of GPU, since data has to be transferred from CPU memory to GPU global memory. Thus, the manner in which data is stored in memory is altered to allow access to all the values of a feature in a block-wise memory operation. Data is received in a sample-wise manner as it is shown in figure 1a. After reorganizing the data, all values are reorganized by feature (figure 1b).



(a) Original Data Structure

(b) Refactored Data Structure

Fig. 1: Example with three samples and five features, with each feature represented with a different color.

4 GPU Optimization

In the previous section we presented the optimization of the main loop of mRMR. The latter only optimizes mRMR in terms of feature dimension, avoiding unnecessary calculations and optimizing data access patterns. In this section, we deal with the second source of computational burden, data set size, which influences the computational cost of mutual information and marginal probabilities calculations.

To further increase performance, GPU units can be used to accelerate these calculations. This problem can be mapped to the well known *histogramming problem* [19] in GPU community. In this section we detail the issues regarding GPU mutual information and marginal probability calculation and the solutions that can be adopted. We start with a short summary of GPU computational model which helps to understand the issues involved the *histogramming problem*.

Calculating the marginal probabilities and joint probabilities (see equation (6)) is equivalent to computing an histogram with the absolute frequency of each possible outcome in the dataset and eventually normalizing that count.

It has been proven that GPUs can be used to efficiently compute image histograms [20, 21] and there are also some works that address acceleration of mutual information in GPU [22]. Most of these methods are designed for image processing, a domain that can handle up to 256 possible values, a number too low in some ML real cases. The challenge of the proposed fast-mRMR is to create kernels that allow for a higher number of possible values without compromising performance. Three kernels optimized for 64 [23], 256 and 65536 possible values are proposed. All kernels are based on the same idea, the input dataset is distributed among thousands of threads, each one responsible of calculating a partial histogram. These partial histograms are lastly merged to obtain the final global one.

4.1 GPU Background

The use of Graphics Processing Units (GPUs) for rendering is well known, but their power for general parallel computation has only recently been explored. Parallel algorithms running on GPUs can often achieve up to 100x 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 to adapt the mRMR to be executed in GPU, so as to be employed when data size exceeds the size in which the CPU algorithm does not behave adequately.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA and implemented by the graphics processing units (GPUs) that they manufacture [24]. CUDA gives direct access to the virtual instruction set and memory of the parallel computational elements in GPUs.

In CUDA, the computation is distributed in a grid of thread blocks, with all blocks containing the same number of threads (see fig 2) that execute a special program called kernel on the device. A kernel is 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 others.

A kernel can make use of registers, shared memory and global memory to make its calculations and communicate with other threads. The first two levels have very fast access but, unfortunately are very scarce. If you were to use too much 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 pruposes, but its access is too slow, degrading the overall performance. Thus, the kernel must be carefully designed to balance: (a) the use of shared memory and registers, (b) active processing units and (c) reducing the number of global memory accesses and optimizing these access patterns.

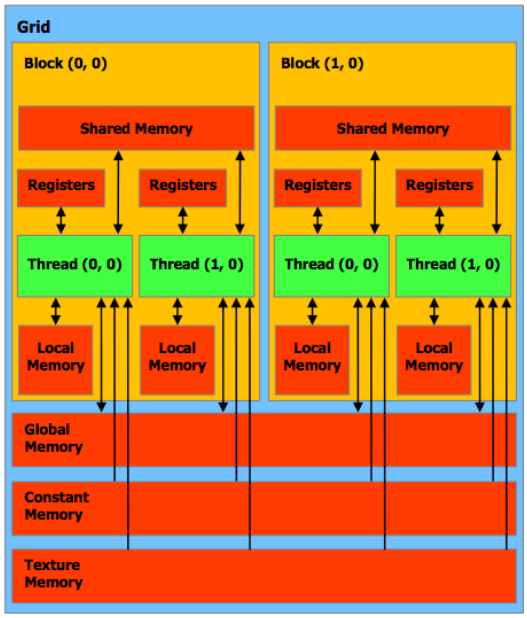


Fig. 2: Nvidia Scheme, with two Blocks in a grid

A typical CUDA program consists of the following stages:

- Allocate GPU global memory region.
- Move data from RAM to global memory.
- Run the GPU kernel(s)
- Bring the results back from global memory to CPU memory.
- Free GPU global memory.

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

4.2 Fast-mRMR GPU

In fast-mRMR, we have adopted a hybrid approach for mutual information 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 making use of global memory. So the kernel in Algorithm 2 is used.

2. If the number of possible outcomes is greater than 64 and less than 256, the kernel shown in algorithm 2 is no longer valid, since a lack of shared memory would produce a sharp decrease in processing units usage. So the kernel in Algorithm 3 is used.
3. If the number of possible outcomes exceeds 256, then shared memory is no longer an option by itself, so the calculations should be partitioned. This strategy is detailed in Algorithms 4.

Per-thread Kernel This kernel is designed for those cases in which there are fewer than 64 values. Given the current memory constraints, enough memory is available for having a local histogram with this size for each thread giving the maximum performance. The benefit of having a local histogram per thread, is that there are not memory access conflicts. Notice that in subsequent kernels, p.e. Algorithm 4 line 6, a slow atomic operation called *atomicAdd* is required in order to cope with these conflicts while in Algorithm 2 line 6 a simple addition is performed. In this situation, each thread does not have to worry about serializing memory writes, and a significant acceleration is consequently achieved.

Algorithm 2 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;
```

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 3). CUDA offers 48Kb of memory for every Streaming Multiprocessor (SMX), and there are at most 64 warps (a warp is a set of 32 consecutive threads) for each SMX. This means that if we have all the warps running, each warp has 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 per-thread kernel was used) of processing units should remain idle. Each warp will have its local histogram in shared memory and performs a linear pass through the corresponding positions. Once local histograms are calculated, they are finally merged on global memory and sent back to CPU.

Algorithm 3 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;
```

Joint kernel This last kernel is designed to calculate the joint probabilities, and allows to compute histograms up to 65536 values. Since, in these cases, shared memory is completely surpassed, the calculations are partitioned balancing the number of passes through the dataset and shared memory usage. With a current limit of 768 values per warp [24], if we would like, for example, to compute a histogram of 1756 possible outcomes, we would need 3 passes. In real cases, better computing times can be obtained by having idle warps and calculating more values of the histogram at each pass. Although in this manner each pass is slower, the final execution time could be better due to the need of less synchronizations. This kernel is accomplished by adding a lap counter to the main loop (see Algorithm 4), so as to know which sub-histogram is calculating each time.

Algorithm 4 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
```

5 Experimental Settings

The aim of the experimental settings devised is showing that the implementations of the algorithm scale appropriately and run faster than the previous version. Several scenarios were employed.

Performance First, in order to compare the proposed version with the original one, the datasets described in table 1a (the same ones used by Peng and available in the mRMR webpage [25]) have been employed.

Algorithm 5 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;
```

Scalability To test the scalability of the proposed implementation in terms of number of values for the features, number of samples and number of features, various synthetic datasets were created by fixing the number of possible values, patterns and features for the datasets. These datasets contain random integer values between 0 and the maximum possible values number given, uniformly generated for each value of each feature for each sample. Although being uniformly distributed, the aim of these datasets is to prove the efficiency of the proposed implementation, which is not influenced by the specific distribution of data. Five datasets of each type are generated which are described below.

The first type is used to check how the algorithm scales in terms of number of samples. The datasets contain 1000 features with 251 possible values with 50, 500, 5000, 50000, and 500000 samples, respectively, as shown in table 1b. The second kind of datasets is used to test the scalalability of the algorithm regarding the number of possible values of each feature varying this number to 10, 50, 100, 150 and 200. All these datasets have 80 samples and 5000 features (see table 1c). The third type of datasets have fixed 80 samples and 10 possible values for each feature and are used to test the scalability in terms of number of features. Again, there are 5 datasets starting with 100 features and up to 1 million features (see table 1d).

GPU implementation To test the GPU performance regarding the number of possible values, synthetic datasets like 1b were used but those were discretized in 30 and 200 possible values. Also two real datasets were used in order to test the algorithm under real conditions. The KDD Cup 99 dataset [26] is a dataset derived from the DARPA dataset which has about 5 million samples and was used for the KDD (Knowledge Discovery and Data Mining

Table 1: Datasets used in the tests.

Dataset	Samples	Features	Values
Lung	73	326	3
Lymphoma	96	4027	3
Leukemia	72	7071	3
Colon	62	2001	3
NCI	60	9173	3

(a) Datasets proposed by Peng [25]

Dataset	Samples	Features	Values
a-samples	50	1000	251
b-samples	500	1000	251
c-samples	5000	1000	251
d-samples	50000	1000	251
f-samples	500000	1000	251

(b) Synthetic datasets to test the influence of the number of samples

Dataset	Samples	Features	Values
a-bins	80	5000	10
b-bins	80	5000	50
c-bins	80	5000	100
d-bins	80	5000	150
f-bins	80	5000	200

(c) Synthetic datasets to test the influence of the number of possible values

Dataset	Samples	Features	Values
a-feat	80	100	10
b-feat	80	1000	10
c-feat	80	10000	10
d-feat	80	100000	10
f-feat	80	1000000	10

(d) Synthetic datasets to test the influence of the number of features

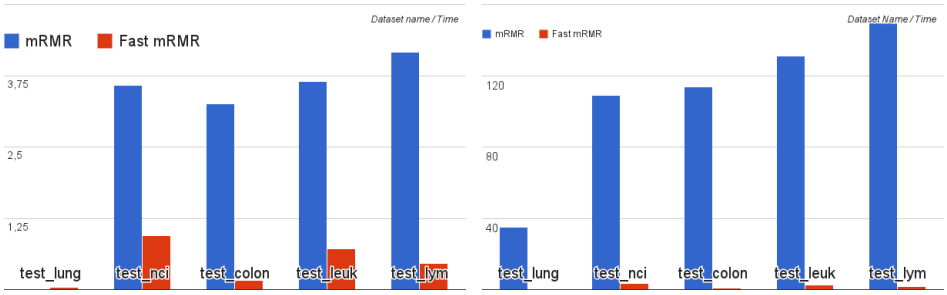
Dataset	Samples	Features	Values
kddcup99	4000000	41	255
Higgs	11000000	21	255

(e) Real datasets to test GPU implementation.

Tools Conference) Cup 99 Competition. Each record represents a TCP/IP connection that is composed of 41 features that are both qualitative and quantitative in nature. The HIGGS Dataset, available on the UCI repository [27], contains 11 million samples with 28 features containing physical data produced using Monte Carlo simulations. The first 21 features are kinematic properties measured by the particle detectors in the accelerator. The last seven features are functions of the first 21 features; these are high-level features derived by physicists to help discriminate between two classes.

5.1 Experimental Results: Comparison of Fast-mRMR and mRMR

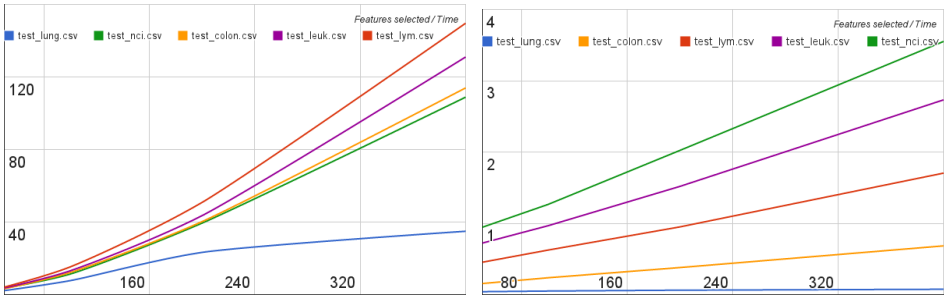
The comparison between our implementation, called Fast-mRMR, and the original mRMR was performed using the datasets in table 1a. Since mRMR is a ranker method which returns an ordered ranking of the features, a threshold needs to be established, and we have opted for retaining the top 50, 100, 200 and 400 features. For space reasons, only the two extreme values are shown in Figure 3. As can be seen, fast-mRMR obtained much better performance results for the 5 datasets, with an average improvement of 20 times faster for 50 features, 50 for 100 features, 116 for 200 features and 159 for 400 features.



(a) 50 Features selected (b) 400 Features selected

Fig. 3: Time vs number of selected features between the proposed implementation and the original mRMR

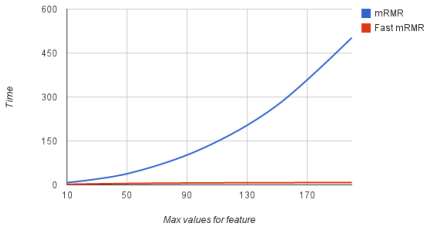
In terms of complexity, as the number of features grows, the time increases linearly in the case of the fast-mRMR while the original mRMR increases polynomially, as can be seen in Figure 4.



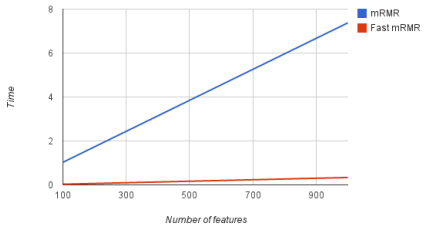
(a) mRMR Complexity (b) Fast mRMR Complexity

Fig. 4: Time versus number of features for each data set in table 1a, for the proposed implementation (fastmRMR) and the original mRMR implementation

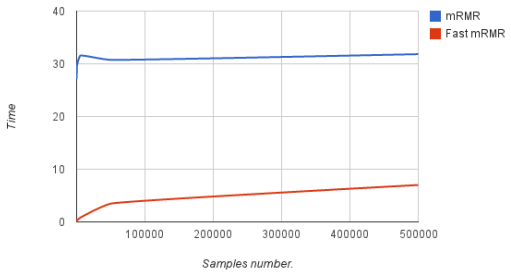
Finally, we analyzed the scalability of the methods using synthetic datasets, as explained above. Figure 5 shows the results obtained by varying the number of possible values (fig. 5a), features (fig. 5b), and samples (fig. 5c). As can be seen, the complexity of Fast-mRMR increases linearly, whilst that of mRMR does it pollynomicaly for the first case, that is, regarding the number of possible values for a feature. As a result of this, finer discretizations can be made on the original data when using our proposed implementation, without degrading the performance in terms of time and accuracy.



(a) Time vs. possible values



(b) Time vs. features



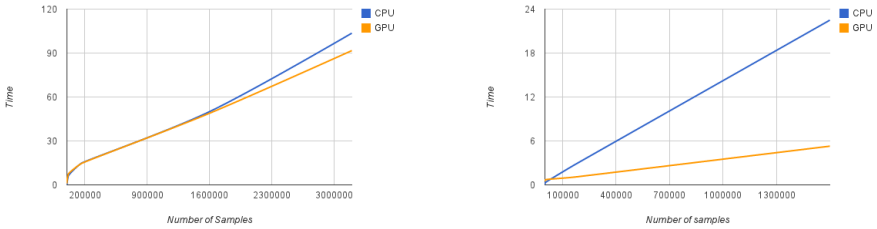
(c) Time vs. samples

Fig. 5: Time versus number of possible values, features and samples for each synthetic dataset in tables 1d 1c 1b, for the proposed implementation (fast-mRMR) and the original mRMR implementation

When trying to compare the scalability of both implementations regarding the number of features, we have used datasets with 80 samples and 10 possible values for each feature. The original implementation imposes an artificial limit for the number of features, for which an error message is displayed. Thus, and in order to be fair, figure 5b shows the comparison between both implementations only up to 1000 features, although fast-mRMR does not have this upper limit.

Figure 5c shows the results in terms of the number of patterns. Both implementations have similar behaviour, although fast-mRMR requires much lower times. In figure 6, it is shown the time complexity versus the number of patterns for the case in which the dataset has a high number of possible values (200, which is the worst case for GPU implementation, see fig. 6a) , and the case in which the dataset has a small number of possible values (30, which is the best case for GPU implementation, see fig. 6b). These figures show that it is worth using the GPU implementation from a million patterns on.

Finally, and in order to show the benefits of using the GPU implementation with real datasets, in figure 7 we show the time needed for both fast-mRMR



(a) Worst situation, near 256 possible values. (b) Best situation, low number of possible values.

Fig. 6: Results of comparying GPU and CPU implementations of fast-mRMR for the worst and best cases, regarding GPU.

implementations (GPU and CPU) for KDDCup dataset and HIGGS dataset. In those cases, the GPU version runs arround 1.6 times faster than the CPU optimized implementation, thus accelerating up to 600 times the original algorithm in the best case.

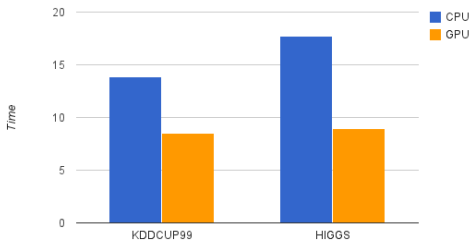


Fig. 7: Time taken for CPU and GPU implementations for KDDCup-99 and Higgs

6 Conclusions

In this paper we have proposed several optimizations of the well-known mRMR feature selection algorithm with the aim of reducing its computational burden. Firstly, we have optimized the main loop of mRMR in terms of feature dimension, avoiding unnecessary calculations and optimizing data accessing patterns. The

experimental results showed that our proposed implementation, fast-mRMR, was able to speed up the process up to 465 times when retaining 400 features. As a result of the optimization, the time complexity of fast-mRMR increases linearly when the number of features grows, instead of exponentially, as it was the case of the original implementation by Peng.

Then, we have dealt with the second source of computational burden, the dataset size. In this case, we have used GPU capabilities to compute information theoretic calculations. With this implementation, that we have called fast-mRMR GPU, the execution time has been accelerated up to 600 times, obtaining a performance close to a cluster in a standard computer.

Acknowledgements. This research has been economically supported in part by the Secretaría de Estado de Investigación of the Spanish Government through the research project TIN 2012-37954, partially funded by FEDER funds of the European Union. V. Bolón-Canedo acknowledges the support of Xunta de Galicia under *Plan I2C* Grant Program.

References

1. Zheng Alan Zhao and Huan Liu. *Spectral feature selection for data mining*. Chapman & Hall/CRC, 2011.
2. Lei Yu and Huan Liu. Redundancy based feature selection for microarray data. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 737–742. ACM, 2004.
3. Verónica Bolón-Canedo, Noelia Sánchez-Maroño, and Amparo Alonso-Betanzos. Feature selection and classification in multiple class datasets: An application to kdd cup 99 dataset. *Expert Systems with Applications*, 38(5):5947–5957, 2011.
4. Wenke Lee, Salvatore J Stolfo, and Kui W Mok. Adaptive intrusion detection: A data mining approach. *Artificial Intelligence Review*, 14(6):533–567, 2000.
5. George Forman. An extensive empirical study of feature selection metrics for text classification. *The Journal of machine learning research*, 3:1289–1305, 2003.
6. Juan Carlos Gomez, Erik Boiy, and Marie-Francine Moens. Highly discriminative statistical features for email classification. *Knowledge and information systems*, 31(1):23–53, 2012.
7. Ofer Egozi, Evgeniy Gabrilovich, and Shaul Markovitch. Concept-based feature generation and selection for information retrieval. In *AAAI*, pages 1132–1137, 2008.
8. Jennifer G. Dy, Carla E. Brodley, Avi Kak, Lynn S. Broderick, and Alex M. Aisen. Unsupervised feature selection applied to content-based retrieval of lung images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 25(3):373–378, 2003.
9. Pasi Saari, Tuomas Eerola, and Olivier Lartillot. Generalizability and simplicity as criteria in feature selection: application to mood classification in music. *Audio, Speech, and Language Processing, IEEE Transactions on*, 19(6):1802–1812, 2011.
10. Chris Ding and Hanchuan Peng. Minimum redundancy feature selection from microarray gene expression data. *Journal of bioinformatics and computational biology*, 3(02):185–205, 2005.

11. Xiaohang Jin, E.W.M. Ma, L.L. Chang, and M. Pecht. Health monitoring of cooling fans based on mahalanobis distance with mrmr feature selection. *Instruimentation and Measureemnt, IEEE Trans. on*, 61(8):2222–2229, 2012.
12. A. Bulling, J.A. Ward, H. Gellersen, and G. Troster. Eye movement analysis for activity recognition using electrooculography. *Pattern Analysis and Machine Intelligence, IEEE Trans. on*, 33(4):741–753, 2011.
13. J.E. Tapia and C.A. Perez. Gender classification based on fusion of different spatial scale features selected by mutual information from histogram of lbp, intensity, and shape. *Information Forensics and Security, IEEE Trans. on*, 8(3):488–499, 2013.
14. D. Bratasanu, I. Nedelcu, and M. Datcu. Interactive spectral band discovery for exploratory visual analysis of satellite images. *Selected Topics in Applied Earth Observations and Remote Sensing, IEEE Journal of*, 5(1):207–224, 2012.
15. Diego Rego-Fernández, Verónica Bolón-Canedo, and Amparo Alonso-Betanzos. Scalability analysis of mrmr for microarray data. In *Proceedings of the 6th International Conference on Agents and Artificial Intelligence*, pages 380–386, 2014.
16. Yeming Hu, Evangelos E. Milios, and James Blustein. Interactive feature selection for document clustering. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1143–1150, New York, NY, USA, 2011. ACM.
17. Diansheng Guo. Coordinating computational and visual approaches for interactive feature selection and multivariate clustering. *Information Visualization*, 2(4):232–246, 2003.
18. Hanchuan Peng, Fulmi Long, and Chris Ding. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27(8):1226–1238, 2005.
19. Ugljesa Milic, Isaac Gelado, Nikola Puzovic, Alex Ramirez, and Milo Tomasevic. Parallelizing general histogram application for cuda architectures. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, pages 11–18. IEEE, 2013.
20. Ramtin Shams and RA Kennedy. Efficient histogram algorithms for nvidia cuda compatible devices. In *Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS)*, pages 418–422, 2007.
21. Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Bart Mesman. High performance predictable histogramming on gpus: exploring and evaluating algorithm trade-offs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 1. ACM, 2011.
22. Ramtin Shams and Nick Barnes. Speeding up mutual information computation using nvidia cuda hardware. In *Digital Image Computing Techniques and Applications, 9th Biennial Conference of the Australian Pattern Recognition Society on*, pages 555–560. IEEE, 2007.
23. Victor Podlozhnyuk. Histogram calculation in cuda. `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/histogram64/doc/histogram.pdf`, 2007.
24. CUDA Nvidia. Programming guide, 2008.
25. Hanchuan Peng. mrmr (minimum redundancy maximum relevance feature selection). `http://penglab.janelia.org/proj/mRMR/#data`, 2014.
26. The UCI KDD Archive. Kdd cup 1999 data. `https://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`, 1999.
27. A. Asuncion and D.J. Newman. UCI machine learning repository. `http://www.ics.uci.edu/mlearn/MLRepository.html`, 2007.