DOI 10.3233/IDA-150375
IOS Press

A multi-core computing approach for large-scale multi-label classification

Juan Manuel Rodriguez[*], Daniela Godoy, Cristian Mateos and Alejandro Zunino
ISISTAN Research Institute, Unicen University, Campus Universitario, Tandil (B7001BBO), Argentina

Abstract. Large scale multi-label learning, i.e. the problem of determining the associated set of labels for an instance, is gaining relevance in recent years due to the emergence of several real-world applications. Most notably, the exponential growth of the Social Web where a resource can be labeled by millions of users using one or more tags, i.e. a resource can be associated to several labels at the same time. A well-known approach for multi-label classification is the Binary Relevance (BR) algorithm which trains a binary classifier for each label independently. However, the serial implementation of BR is not suitable for medium or large datasets due to the time and computational resources required for training. For example, training classifiers for mid-size datasets using MULAN implementation of BR might take several weeks. This paper discusses a parallel implementation of the MULAN BR technique that harnesses the computational power of nowadays multi-core processors. Our implementation presents a speed-up in the training phase of up to 12 times when compared to the original MULAN implementation. In addition, the cross-validation technique of MULAN had huge RAM requirements, making it unusable with large datasets. Therefore, we have overcome this limitation by using compact data structures and taking advantage of disk caching. We have also compared our implementation against scikit-learn, a popular tool for data mining and data analysis, showing significant improvements in speed-up.

Keywords: Binary relevance classification, multi-core programming, parallel classification, multi-label classification

1. Introduction

Supervised learning techniques often deal with the problem of assigning a single class label to instances or examples based on previously known instances-class associations. However, there are many domains in which a particular instance can be associated with a set of labels. This kind of classification problems is known as multi-label classification (MLC) [47]. A classical example of MLC can be found in textual data, since documents or Web pages frequently belong to multiple topics [29]. For example, a text document about scientific contributions in medicine by applying artificial intelligence techniques may belong to both *medicine* and *artificial intelligence* categories. Compared to other known techniques such as clustering or self-organizing maps, MLC differs in the very nature of data as the classes examples belong to are pre-defined in advance, and it does not involve an unsupervised search plus appropriate grouping.

Nowadays, a number of applications, especially those from massively open environments like the Web, require MLC in the presence of large data. Examples of such applications are folksonomies, data

[*]Corresponding author: Juan Manuel Rodriguez, ISISTAN Research Institute, Unicen University, Campus Universitario, Tandil (B7001BBO), Argentina. Tel.: +54 249 4385682 ext. 35; Fax: +54 249 4385681; E-mail: juanmanuel.rodriguez@isistan.unicen. edu.ar.

structures resulting from collaborative or social tagging systems. In sites such as Delicious,[1] Fickr[2] or BlogCatalog,[3] users annotate resources, e.g. Web pages, photos or videos, using open-ended tags. Usually, users assign more than one tag to a single resource. Hence, tag recommendation can be casted to a very large-scale MLC problem [17,20,23,42,43]. Moreover, MLC is not only used for classifying Web resources in social networks, but also in other contexts associated with very large domains, such as predicting gene functions [51].

This sort of applications has challenged existing MLC algorithms, especially because the label space usually grows with the exponential growth of the instance space [44]. For example, in a folksonomy the number of labels is typically very high as users annotate resources as they like. This presents a challenge because the computational cost of training a multi-label classifier heavily depends on the number of labels for most of the existing algorithms. While typical MLC algorithms have linear complexity in the number of labels, they need to maintain multiple models in memory, severely compromising their ability to handle large label spaces [44]. Besides, the greater the number of instances is, the greater the computational requirement for learning models is.

In order to scale up current MLC algorithms, some researchers have proposed to preprocess labels so as to apply a pruning technique. For instance, [46] proposes a dataset preprocessing technique to automatically determine the relevant set of labels for each instance, while [10] proposes a pruning approach designed to deal with many labels in the presence of a small number of sample instances per label. Although these methods enable MLC algorithms to scale up to millions of instances, they require reducing label specificity. In response, some researchers have combined MLC algorithms with general-purpose optimization techniques. For example, Rank-CVM [53,54] uses the binary core vector machine (CVM) to boost Rank-SVM [11] execution time, an MLC algorithm based on SVM. Moreover, handling large training datasets is an issue because not only the training times are long, but also the required memory for storing the generated model is huge. This problem has been already identified and tackled in traditional classification algorithms [9]. Furthermore, this problem also affects MLC algorithms that transform the problem into traditional classification problems. Notice that its severity depends on the dataset properties as well as the traditional classifier memory requirements. Even more, the number of traditional classifiers required depends on the selected multi-label classifier. For instance, Binary Relevance (BR) requires a binary classifier per label, but Ranking by Pairwise Comparison further transforms the problem and requires $\frac{l(l-1)}{2}$ binary classifiers, where l is the number of labels [47].

Although nowadays multi-core processors are ubiquitous, a recent study [50] has shown that, until now, parallelism in the area of Machine Learning (ML) is mostly circumscribed to data mining algorithms, particularly frequent itemset mining and association rule mining. Since using several cores with shared memory has been proved effective to accelerate the processing of large datasets [3,27], we propose a novel multi-core approach to train and evaluate multi-label classifiers. The main objective of this work is accelerating MLC training by means of taking full advantage of currently available parallel computing capabilities in multi-core computers.

Since training and evaluating MLC can be a time consuming task, fully using computational resources of nowadays multi-core processors is of utmost importance. For example, in [26], the authors could not fully assess the effectiveness of the algorithm BR as implemented due to the long training times. This work focuses on the BR sequential implementation provided by the well-known library MULAN [49]. In

[1] Delicious: http://delicious.com/.

[2] Fickr: http://www.flickr.com/.

[3] BlogCatalog: http://www.blogcatalog.com.

this context, our goal is to improve this implementation to run in a parallel manner. However, parallelizing the training phase might increase RAM memory requirements. This is an issue because MULAN extensively uses a dataset copying mechanism for transforming the training dataset without affecting the input dataset. Taking this into account, parallelizing might imply having different dataset copies in memory at the same time. Our approach does not only consider parallelization, but also proposes a disk swapping technique to overcome RAM requirement limitations.

In brief, our paper presents an adaptation of the well-known MULAN library BR implementation to effectively harness nowadays multi-core overcoming RAM limitations by means of disk swapping. Therefore, our approach makes it possible to use MULAN BR on datasets that cannot be analyzed by original MULAN BR implementation. Our experimental results have shown that our approach can be used for effectively analyzing datasets with up to 250,000 instances, 1,000 features and 81 labels. Additionally, we have also evaluated our approach against the well-known scikit-learn library.[4] Although scikit-learn supports multi-core processors, this Python library requires large amount of RAM due to its parallelism is based on a multi-process scheme, which is imposed by Python multi-thread limitations. Consequently, scikit-learn cannot effectively harness multi-thread processors capabilities.

The rest of the paper is organized as follows. Section 2 discusses previous work in multi-core and distributed classifier training and evaluation. Section 3 introduces our approach to train a multi-label classifier taking advantage of multi-core processors. Section 4 discusses the advantages and issues of using parallel multi-label classification evaluation methods. In addition, this section also introduces a disk swapping approach to overcome RAM memory requirements during the evaluation process. Then, Section 5 presents the empirical evaluation of our approach in order to train and evaluate multi-label classifiers with large datasets in a multi-core environment. This section also presents an evaluation of the approach against other commonplace BR implementation that supports parallelism, namely an implementation provided in the scikit-learn library. Finally, Section 6 concludes the paper and outlines future research in this topic. It is worth noting that both the source code implementing our approach and the experimental material are available upon request.

2. Related work

In the past few years, several works exploiting parallel computing techniques have entered the scene. While the main motivation in MLC research is still to increase classification efficacy in terms of common metrics –precision, recall, Hamming loss, and so on–, many researchers [26,44,47] have already identified training and testing time as an important concern, especially in applications dealing with large datasets and label sets.

From an algorithmic standpoint, there are mainly two methods for MLC: algorithm adaptation and problem transformation [35]. Algorithm adaptation ML-learners extend existing single label learning algorithms to handle multi-label problems. Examples of extended algorithms are Adaboost.MH [40], Rank-SVM [11], Multi-label C4.5 [8], and ML-KNN [57]. Transformation based ML-learners, on the other hand, focus on transforming multi-label problems into several single label problems, whose outputs are then aggregated into a prediction to the whole classification problem. Examples of methods in this category are BR, ranking via pair-wise comparison, and Calibrated Label Ranking (CLR) [47].

[4] Scikit-learn: http://scikit-learn.org.

One of the most commonly used implementation of MLC is MULAN [49]. MULAN is a Java library that implements several MLC algorithms, such as BR, HOMER [48], Multi-label C4.5 and ML-KNN. MULAN relies on a well-known machine learning Java library called WEKA [15]. Although WEKA is vastly used, it has been shown that it does not take advantage of parallel processing environments [4]. Similarly to WEKA, MULAN does not take advantage of parallel processing environments, which negatively affects the training times of MLC algorithms because current multi-core processors can perform more computation when running different tasks at the same time. Therefore, this inability to harness the computational power of multi-core processors has severely hindered the applicability of ML-learners. For example, a recent study on MLC effectiveness [26] could not complete a full assessment due to MLC training times. Particularly, this problem affected transformation based MLC algorithms, such as BR and HOMER. Although MLC were performed on a high-end computer (Intel Quad-Core running at 2.5 GHz and 64 GB of RAM), some of the experiments took more than a week so they were discarded. Although it is not reported in the paper, during the experimentation the processor of the machine was underused because multi-core processors are designed for running several tasks in parallel. Since MULAN runs only one task at the time, most of the processors capabilities are simply wasted. These facts evidence the necessity of providing parallel versions of MLC algorithms.

Efficient and scalable MLC algorithms are required for nowadays large amount of data. Taking this into account, Read et al. [36] have studied some of the techniques discussed above and proposed a new one for MLC in streams of data. This scenario is different from the typical machine learning in which the data for training the classifiers is available a priori. Although the paper states the importance of scalability, all the experiments are mostly focused on evaluating the algorithms in terms of classification effectiveness rather than scalability and performance. Moreover, the paper conclusions remark that a future work is to evaluate MLC algorithms with even large data streams.

Another example of MLC applied to Big Data is presented in [56] that analyzes the Classifier Chain Algorithm, a transformation method, with different datasets. Although the authors highlight the importance of Big Data, all the evaluation is performed using medium-size datasets. In fact, the two datasets used are the Reuters-21578 text collection and a set of 2000 natural scene images, which can hardly be considered as Big Data. Furthermore, the paper does not discuss taking advantage of any parallel environment for speeding-up the training or the evaluation of these classifiers.

Furthermore, the need for parallel algorithms for classification has been identified even in more traditional scenarios. In [12], the authors propose a MapReduce approach for classifying large amounts of text into a single category out of many. In this work, the authors uses SVM classifiers that are known to be effective for classifying text, but very complex to train with large datasets. The approach uses Hadoop[5] for speeding up the training of a one-vs.-one classifier. Notice that for K classes, a one-vs-one classifier requires $K(K-1)/2$ binary classifier. For speeding up the training, Hadoop maps the training of each binary SVM classifier into different nodes, i.e., each node is assigned the responsibility of training a subset of $K(K-1)/2$ binary SVM classifiers. This is particularly effective for speeding up the multi-class classifier training when the number of possible classes is high and, since the approach is intended for text about traditional Chinese medicine, it is very effective.

Currently, there are some parallel implementations of extended MLC algorithms. For example, in [38] a parallel custom implementation of a decision tree classification algorithm is proposed. The support is aimed at text document classification and runs on top of GridGain,[6] a commercial platform for Grid

[5] Apache Hadoop: http://hadoop.apache.org/.

[6] GridGain: http://www.gridgain.com.

Computing. Moreover, MultiSVM [16] is a serial implementation of a MLC classifier based on SVM that runs on CUDA-enabled GPUs. CUDA (Compute Unified Device Architecture) is a parallel programming model created by NVIDIA to exploit the physical cores of graphics cards. According to the authors, MultiSVM outperforms a classical SVM solver (LIBSVM[7]) by an order of magnitude in terms of training and testing times. Particularly, SVM is known to suffer from serious scalability problems in both memory use and computational time even for single label classification problems, for which parallel solutions have been proposed. A representative example is PSVM [5,55]. In the same line, but in a more algorithm-agnostic way, Weka-Parallel [4] is an extension of the well-known Weka toolkit [15] that provides parallel and distributed support for cross-validation tasks, but does not parallelize training tasks. Lastly, MOA[8] exploits Weka to support data streaming applications with large scale data. To this end, MOA uses Hadoop, a well-known parallel platform for large scale data processing in clusters, which is the reference implementation of the Google's MapReduce [22] parallel programming model.

In general, irrespective of whether parallel/distributed programming is used or not, MLC is handled via problem transformation methods [27]. Within these, the most well used problem transformation method is the BR method [27,30,37,48], which transforms an MLC problem into one binary classification problem for each label. Therefore, the BR method creates one binary classifier for each label in the dataset. Then, each binary classifier is trained independently for each label. From a parallel programming perspective, binary relevance methods shape MLCs into master-worker applications [28], whereby several workers (single classifiers) can be executed independently to compute subproblems, and the master (binary relevance logic) combines task outputs into a classification output. This idea has gave birth to several efforts, whose shared goal is to promptly execute these tasks and efficiently transfer the required data so that the overall MLC problem runs faster.

First, [27] implements a framework for MLC based on multi-threaded MapReduce for text document classification. The framework is also based on binary relevance and supports a variety of binary classifiers. A weak point of the work is that the computational requirements of the framework (in terms of memory and execution time) have not been assessed yet. Furthermore, the dataset used by the authors consists only of 120 documents and an undisclosed number of features, so it is very likely that training the classifiers do not require large amount of main memory. Finally, the authors do not evaluate speed-up when comparing their approach to sequential multi-label classifiers, such as MULAN.

HOMER [48] proposes an algorithm whereby a set of labels are split into disjoint subsets so that similar labels are placed together and dissimilar ones are placed apart. HOMER starts with all the labels (and instances) at the root and recursively creates a hierarchy in a top-down depth-first fashion. For each node, N children nodes are created and the labels of the current node are evenly distributed into N disjoint (children) subsets. This facilitates the execution of the associated binary classifiers for these sets in parallel. Another machine learning library that includes MLC algorithms is scikit-learn [34]. Some of the machine learning algorithms provided by scikit-learn can take advantage of multi-core processors. However, all the processing and evaluation is done in-memory, severely limiting the applicability of the library to small or medium sized datasets.

In [52], the authors propose a MapReduce approach for labeling Youtube videos. Unlike feature based approaches, this approach bases its predictions on relationships between the instances, as they can provide relevant information about labels [21]. Youtube videos, such as other post in social media, have relationships between one another, so this approach considers videos and their relationships as graph

[7] LIBSVN: http://www.csie.ntu.edu.tw/cjlin/libsvm/.

[8] MOA: http://moa.cms.waikato.ac.nz/overview/.

and uses an Iterative Multi-label Propagation (IMP) algorithm for predicting labels. Although this approach can be used for data in social network or having explicit relationship, it cannot be used for data with unknown relationships or independent data points.

Sometimes fast prediction is more important than fast training, like when advertising based on search engine queries [1]. In their approach, Agrawal et al. [1] are more interested in generating a fast prediction rather than fast training. They proposed to use Multi-Label Random Forest for predicting whether an advertisement is likely to be relevant based on its labels. The paper claims that a prediction can be achieved within milliseconds using just a single machine with 10 Gb of RAM. However, since the training data was huge, approximately 90 million data points, they trained the model using a MapReduce approach with thousand 2 Gb-or-RAM nodes. As the model did not fit within 2 GB of RAM, the authors employed a modified version of the PLANET approach [33], used for training trees' ensembles in a massively distributed manner.

In [31], the authors present a distributed matrix completion method for large-scale MLC for recommendation systems, like the ones used for Netflix challenge or in music ranking prediction. For classification, matrix completion is used for guessing the missing values in a matrix in which each column represents an instance. This matrix consists of the training features ($X^T_{f,i}$), training labels ($Y^T_{f,i}$), and the features of instances to predict ($X^P_{f,i}$). In such matrix, the labels to predict ($Y^?_{f,i}$) are the missing values.

The work reported in [31] uses a Map-Reduce approach to solve the MLC problem. The approach was evaluated using a single computer with a multi-core Intel I7 processor and 8 Gb of RAM. Under these conditions, the reported speed-up was about 4 times. Finally, a limitation of this approach is that all operations are in memory, restricting the size of the datasets to the available RAM.

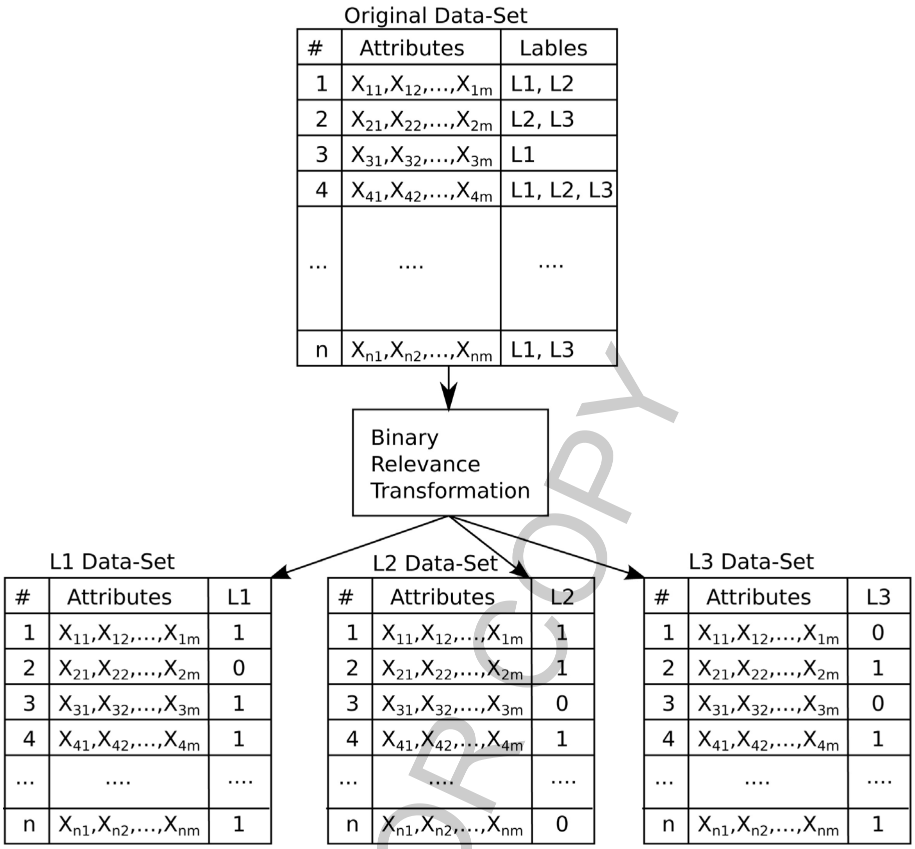
Although most of these works aim at exploiting parallel or distributed environments and programming techniques, only a few of them actually evaluate the impact of using such environments and techniques. Most of the works are more interesting in evaluating traditional MLC metrics, such as Hamming loss, F1-Measure, or Accuracy. Partially, this is because many of them are domain specific. For instance, [52] is focused on classifying Youtube videos, [1] aims at predicting advertisement, and [12] is for classifying traditional Chinese medicine text. In contrast, this work is focused on measuring the advantage of leveraging multi-core processors for MLC rather than evaluating traditional MLC metrics.

3. Multi-threaded binary relevance

BR [47] is one of the most popular transformation based ML learners. It uses a one-against-all strategy by learning one classifier for each label, using all the instances associated to the label as positive examples and all remaining instances as negative ones. Each binary classifier predicts whether an instance should be assigned with the label. BR outputs the union of the relevant labels predicted by all classifiers.

The BR algorithm consists in transforming an input dataset that conveys the instances and their labels into several datasets, one per label. Each new dataset is constructed to associate the instances to a particular label. This means that each new dataset has all the attributes of the original instances plus a binary class. Figure 1 depicts how a multi-label dataset with three labels is transformed into three binary class datasets. After that, a single-label classifier is trained for each new dataset. Therefore, classifying a new instance consists in using the different classifiers to determine whether an instance should be labeled as belonging to a given class.

Figure 2 depicts the execution flow of training a MULAN BR algorithm. In this diagram, "Generate *i*-labeled dataset" and "Train classifier using *i*-labeled dataset" are not only the most time expensive



Original Data-Set

#	Attributes	Lables
1	$X_{11},X_{12},...,X_{1m}$	L1, L2
2	$X_{21},X_{22},...,X_{2m}$	L2, L3
3	$X_{31},X_{32},...,X_{3m}$	L1
4	$X_{41},X_{42},...,X_{4m}$	L1, L2, L3
...	....	....
n	$X_{n1},X_{n2},...,X_{nm}$	L1, L3

L1 Data-Set

#	Attributes	L1
1	$X_{11},X_{12},...,X_{1m}$	1
2	$X_{21},X_{22},...,X_{2m}$	0
3	$X_{31},X_{32},...,X_{3m}$	1
4	$X_{41},X_{42},...,X_{4m}$	1
...	....	....
n	$X_{n1},X_{n2},...,X_{nm}$	1

L2 Data-Set

#	Attributes	L2
1	$X_{11},X_{12},...,X_{1m}$	1
2	$X_{21},X_{22},...,X_{2m}$	1
3	$X_{31},X_{32},...,X_{3m}$	0
4	$X_{41},X_{42},...,X_{4m}$	1
...	....	....
n	$X_{n1},X_{n2},...,X_{nm}$	0

L3 Data-Set

#	Attributes	L3
1	$X_{11},X_{12},...,X_{1m}$	0
2	$X_{21},X_{22},...,X_{2m}$	1
3	$X_{31},X_{32},...,X_{3m}$	0
4	$X_{41},X_{42},...,X_{4m}$	1
...	....	....
n	$X_{n1},X_{n2},...,X_{nm}$	1

Fig. 1. Binary relevance transformation.

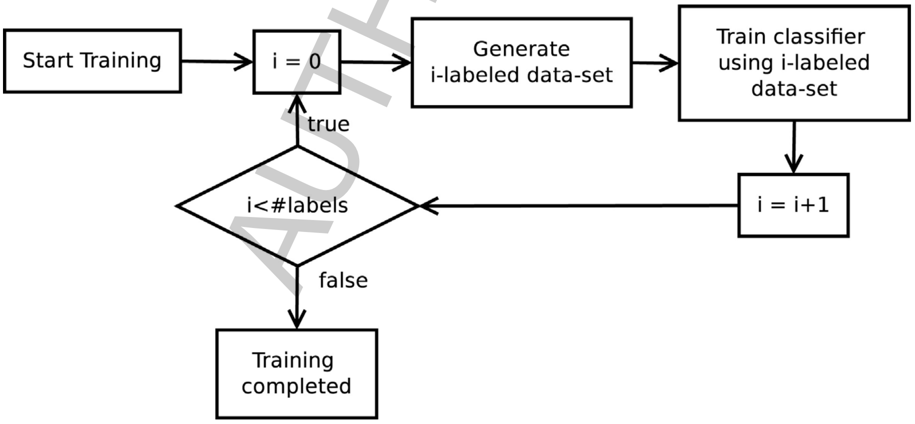

Start Training
i = 0
Generate i-labeled data-set
Train classifier using i-labeled data-set
i = i+1
i<#labels
true
false
Training completed

Fig. 2. Original MULAN execution flow.

task, but also are executed as many times as labels the dataset has. Although these tasks are independent, MULAN performs them sequentially [49]. As a result of this, multi-core processors are underused [19, 41] and execution time cannot be reduced by adding processing cores. Therefore, training a BR classifier using MULAN implementation, even using small datasets, might take too long.

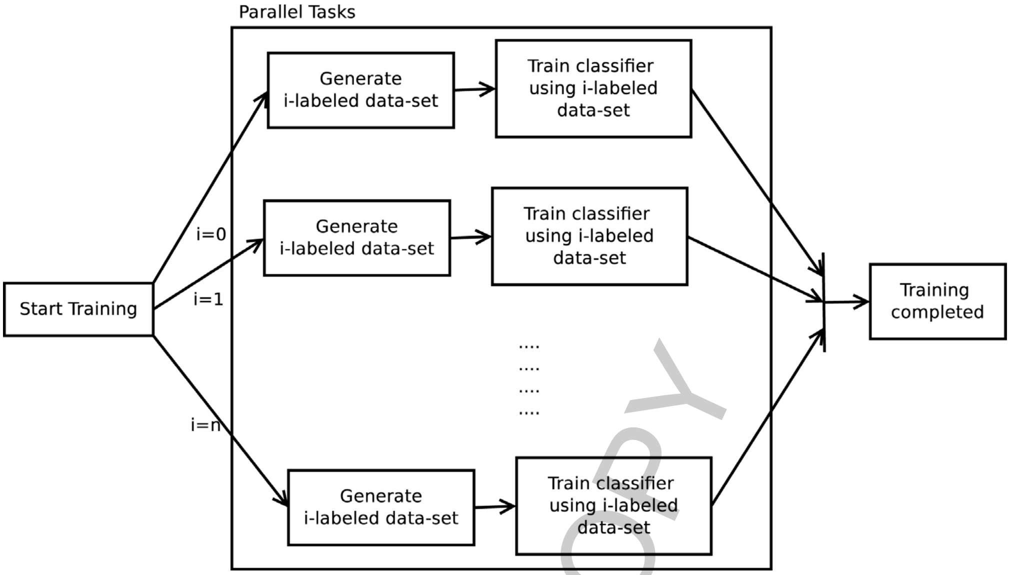

Parallel Tasks
Start Training
i=0
i=1
i=n
Generate i-labeled data-set
Train classifier using i-labeled data-set
Generate i-labeled data-set
Train classifier using i-labeled data-set
....
Generate i-labeled data-set
Train classifier using i-labeled data-set
Training completed

Fig. 3. Parallel MULAN execution flow.

To take advantage of multi-core processors, we propose to execute both tasks, namely dataset generation and classifier training, in parallel for each label. Figure 3 shows the new organization of the algorithm. Notice that the parallel tasks might not run all at the same time. Since they require RAM to store the transformed dataset during the training process, running all at the same time might not be possible, but it might be possible to run a subset in parallel. Therefore, it is important to determine how many tasks can be run to maximize the use of the available CPU cores without consuming all available RAM, which might cause slowdowns due to the increase of disk activity when the RAM is nearly fully used.

To implement a multi-threaded version of the MULAN BR algorithm, we used the Java Executor Service mechanism[9] and thread pools. Basically, a thread pool is a set of pre-created threads that are used to execute tasks, reusing the threads once the assigned task is finished. Thread pool is a vastly used design pattern [2], and in this case it is used to limit the number of tasks executing at the same time, reducing the necessity of using ad-hoc synchronization structures – such as semaphores – in the multi-threaded code. In Java, Executor Services are the classes that implement different administration policies for thread pools hiding to the developer the logic of how the threads are created and how tasks are assigned to them. Therefore, the Execution Service reduces the amount of code overhead required to manage parallelism. For our implementation, we selected an Executor Service that provides a fixed size thread pool. Basically, this Executor Service limits the growth of the thread pool to a predefined size. When there are more tasks than threads to execute them, this Executor Service simply enqueues the tasks until a thread becomes available, i.e. it finishes its previously assigned tasks. In consequence, to implement the MULAN BR multi-thread algorithm, we decided that the parallel tasks to be responsible of transforming the training dataset and train a binary classifier. Each task is associated with a particular label, so the tasks are independent of one another.

[9] Java Executors: http://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html.

Listing 1: Original binary relevance training code

```
//INITIALIZE CLASSIFIERS ARRAY
ensemble = new Classifier[numLabels];
//INITIALIZE dataset TRANSFORMATION
brt = new BinaryRelevanceTransformation(train);
for (int i = 0; i < numLabels; i++) {
    //CREATE A CLASSIFIER FROM A SAMPLE
    ensemble[i] = AbstractClassifier.makeCopy(baseClassifier);
    //CREATE THE TRANSFORM dataset FOR THE LABEL i
    Instances shell = brt.transformInstances(i);
    //TRAIN THE CLASSIFIER i
    ensemble[i].buildClassifier(shell);
}
```

Listing 2: Multi-threaded binary relevance training code

```
//INITIALIZE CLASSIFIERS ARRAY
ensemble = new Classifier[numLabels];
//INITIALIZE dataset TRANSFORMATION
brt = new BinaryRelevanceTransformation(train);
//INITIALIZE FUTURES
Future<?>[] futures=new Future<?>[numLabels];
//INITIAlIZE THREAD POOL
ExecutorService exec=Executors.newFixedThreadPool(
        Runtime.getRuntime().availableProcessors());
//LAUNCH TRAINING TASKS
for (int i = 0; i < numLabels; i++) {
    futures[i]=
        exec.submit(
            new BinaryRelevanceRunner(ensemble,
                    i, brt, baseClassifier,
                    numLabels));
}
//CLOSE POOL
exec.shutdown();
//WAIT FOR ALL TRAINING TASK TO FINISH
for (int i = 0; i < numLabels; i++) {
    futures[i].get();
}
```

Listing 1 shows a simplified version of the code in the original BR `buildInternal` method, which defines how the BR classifier is trained. Notice that the presented code is simplified because some initialization and debug/logging code was removed, but the code is completely functional. Listing 2 presents the multi-thread version of the code. In this case, the 3 lines presented in Listing 1 for-block were moved to the `run` method in the `BinaryRelevanceRunner` class.

The first difference between the original version and the multi-threaded one is the creation of the `Future` array and the `ExecutorService`. The `Future` array is created as a synchronization mechanism to wait for all the classifiers to be trained. This synchronization is shown in Listing 2, second sentence, where the `get` method in the `Future` blocks the thread until the associated task is finished. In this case, each `Future` is associated with the task of training one of the classifiers. Regarding the `ExecutorService`, our approach uses one with a fixed number of threads, which was initialized to be equal to the number of logical processors. This is because each task requires creating a copy of the dataset, which is a memory consuming task that in large datasets might lead to memory problems. Furthermore, having several threads running at the same time might require extra time for context-switching, which could lead to performance degradation. Since training WEKA classifiers is a CPU-bound task, it is most likely that all the CPU would be near full-load when training the multi-threaded version of the BR classifier.

Finally, the original implementation of the BR transformation uses WEKA filters to create the new datasets. These filters were designed for modifiability, so that a new filter can be easily implemented and combined with others to transform a dataset. However, this design requires creating partially transformed copies of the dataset, requiring lots of memory. Also, this might affect the filters performance because creating temporal structures generates objects that have to be deleted by the Java Garbage Collector,[10] which is a time consuming activity. Therefore, we proposed two alternative implementations.

The first one creates two matrices of `double`, which is a primitive type in Java, one for the features and one for the labels. Then, when a new dataset is created, it creates a WEKA `DenseInstance` copying the features and adding the desired label value from the label matrix. This implementation was expected to be the fastest because using a matrix of primitive types requires less indirections that using objects and accessing the data, i.e. the value for one attribute, in `DenseInstances` has constant temporal complexity $\theta(1)$.

The second implementation uses WEKA `SparseInstance` for both storing the data and generating the transformed datasets. `SparseInstances` require less memory to store sparse data, but the temporal complexity to access a particular feature value is $\theta(\log_2(n))$, where n is the number of attributes with non-zero values in a particular instance. The temporal complexity is associated with the representation of the `SparseInstances`. They are represented via a sorted list of attribute-value for non attributes with non-zero values. Therefore, determining an attribute value requires a binary search that has the aforementioned temporal complexity. Lastly, the original transformation uses the dataset representation and WEKA transformation, as a result of this, the efficiency, required memory and the temporal complexity of accessing the data depends on the original dataset representation.

4. Multi-thread and low-memory classifier cross-validation

Cross-validation is a common technique for estimating the performance of a predictive model. The basic form of cross-validation is k-fold cross-validation, in which the input dataset is randomly partitioned into k equal size subsamples or folds. Then, k independent tests are executed such that for each test a different fold is held out for validation while the remaining $k - 1$ folds are used for learning. Finally, results are aggregated to obtain a statistically fair assessment of the classifier quality.

The current implementation of the cross-validation method provided by MULAN is sequential and in-memory, e.g. the k independent tests are executed one at the time and the partial results are kept in main memory until the evaluation is completed. This presents two issues when evaluating large datasets: low performance in multi-core environments [19,41] and high memory consumption.

The first issue can be overtaken by reimplementing the cross-validation to run each independent test in a different thread. As a result, several classifiers are trained at the same time and, then, their predictions are contrasted against the expected output or ground truth. MULAN provides implementations for different metrics used to evaluate the multi-label learners' effectiveness. Mainly, these metrics compares the output of the classifier against the real labels for each instance. The metrics differs in how these values are compared. For instance, Hamming loss calculates the average normalized Hamming distance

[10] Java Garbage Collector: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-140228.html.

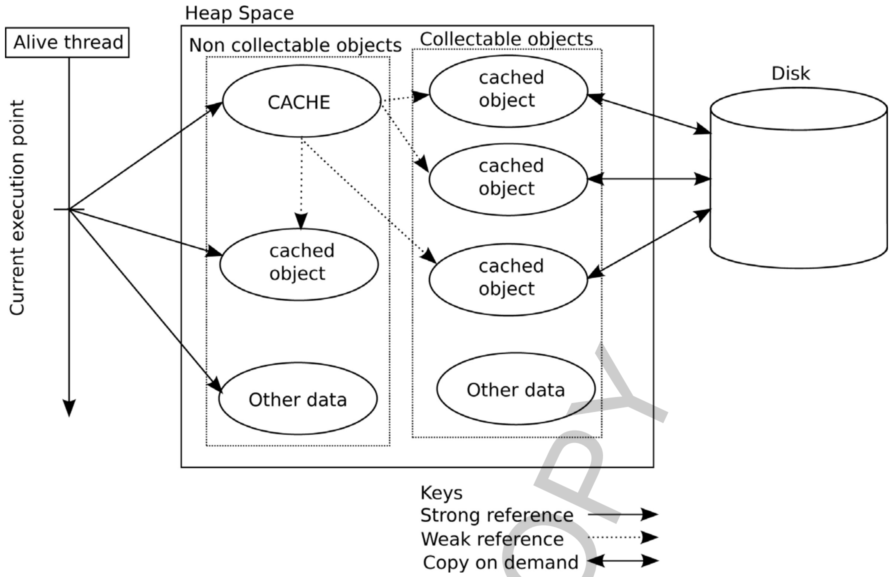


Fig. 4. Garbage collector based cache.

between the classifier output and the real instances labels. In contrast, Subset Accuracy calculates the percentage of instances that had all their labels correctly assigned. To do so, MULAN defines a class, called `MeasureBase`, that defines the interface for a new measure. The key method of this interface is the `update` method that allows to incrementally constructing the measures by adding information of one instance at a time.

Although most of the measures can be easily updated, some of them require storing the information of each classification to be calculated at the end. This introduces the second issue, because it can require several megabytes of RAM to keep the information necessary to evaluate a large dataset [14,25,32,45]. To avoid this problem, we swap these measure objects between RAM and disk on-demand. The swap mechanism was implemented as a layer that keeps the objects as Java weak references[11] so as to enable the Garbage Collector to reclaim that memory when needed. Our layer adds a hook that writes the information that is not currently needed to disk when its memory is reclaimed, so the information can be reloaded when the object is needed again. This layer works as an associative cache in which the stored Java objects are associated to keys that allow retrieving the objects when needed regardless if they are in main memory or if they were evicted and stored in disk.

Figure 4 depicts how our cache works. At a given execution time only a part of the objects are used by the running threads, which means that those objects need to be in memory. Other objects can remain in memory as long as no more memory is required. If the execution requires allocating more memory, the Garbage Collector can claim memory of all objects that are not strongly referenced, i.e. garbage and cached objects. The difference between garbage and our cached objects is that the cached objects are written to disk, so that they can be reloaded when needed.

In order to make our solution efficient, we had to refactor the code from MULAN that performs each independent test. The original mechanism of performing the tests consists in training the classifier first. Then, for each instance in the test dataset, MULAN calculates the output of the classifier and updates

[11] Java WeakReference: http://docs.oracle.com/javase/7/docs/api/java/lang/ref/WeakReference.html.

each measure. As a result of this data access pattern, the current implementation of the cross-validation algorithm has low temporal locality. Under this condition, simply adding the swap layer might result in several swaps to disk, causing an unacceptable performance loss because hard-drives are several orders of magnitude slower than RAM. To reduce the number of swaps, our implementation firstly obtains the prediction and the expected results for all the instances in the test dataset, which are stored using the swap layer in blocks of several results. Then, each measure is updated using all the results, which means that each measure instance would be swapped at most twice. Basically, this changes the data access pattern to increase the temporal locality, thus reducing the number of swaps [6].

5. Evaluation

We evaluated the speedup of our approaches to train and validate BR classifiers separately. Firstly, in Section 5.1, we assess the speedup of using multi-threading to train the BR classifier. Secondly, in Section 5.2 we evaluated scikit-learn [34], an open source tool that takes advantage of multi-core processors for in-memory machine learning. Finally, in Section 5.3, we evaluate the speedup of the BR classifiers using the MULAN implementation and the multi-threaded implementation, both without caching and with caching. Each of these approaches was assessed using the different training implementations discussed in Section 2. Since the BR classifier uses a binary class classifier, we selected the WEKA Naïve-Bayes classifier [18].

5.1. Multi-threaded training experiments

The first experiment was designed to assess the speedup obtained by harnessing multi-core processors by means of multi-threaded programming in BR classifier training, i.e. the approach discussed in Section 3. To do this, we used different large datasets and compared the times required to train the BR classifier using MULAN implementation and our implementation variations. Since the resulting models were equivalent, common MLC performance metrics were not affected. As a result, there is no trade-off between training time and classifier effectiveness. Hence, this section is focused on the required training times only.

This experiment consisted in comparing the time required to train a BR classifier using multi-threading with the different transformation techniques discussed in Section 3 versus the time required to train the MULAN BR classifier implementation. This resulted in four implementations of the BR classifier, three multi-threaded versions using the different transformation implementation, e.g. the MULAN transformation implementation, the one based in matrices of double-precision float point numbers and the last using sparse instances, and the original version of the classifier. For the sake of simplicity we called the different implementations as follows:

- Single Thread Binary Relevance (STBR), which is the MULAN implementation.
- Multi-thread Binary Relevance with MULAN transformation implementation (MTBRO).
- Multi-thread Binary Relevance with data Matrix transformation implementation (MTBRM).
- Multi-thread Binary Relevance with Sparse transformation implementation (MTBRS).

To assure the representativity of the experiments, we used seven distinct datasets. Each dataset had 269,648 instances and 81 possible labels for each instance. The datasets were constructed using the different features and tags obtained from a real-life images dataset called NUS-WIDE [7]. Instances correspond to images crawled from Flickr and the ground-truth of 81 concepts for these images. The

Table 1
Datasets' characteristics

Dataset	Features	Type of feature	Non-zero values	Zero values	Dense size	Sparse size
LLF normalized BoW	500	Bag of visual words	39.70%	60.30%	1203 Mb	723 Mb
LLF normalized CH	64	Color histogram	45.43%	54.57%	306 Mb	214 Mb
LLF normalized CM55	225	Block-wise color moments (LAB)	74.14%	25.86%	637 Mb	711 Mb
LLF normalized CORR	144	Color auto-correlogram (HSV)	64.83%	35.17%	470 Mb	461 Mb
LLF normalized EDH	73	Edge direction histogram	48.62%	51.38%	324 Mb	242 Mb
LLF normalized WT	128	Eavelet texture	62.14%	37.86%	437 Mb	412 Mb
All tags	1002	User assigned tags	0.86%	99.14%	2236 Mb	40 Mb

different datasets describe instances using different set of attributes or features, including low-level image features and associated tags in the folksonomy. The different characteristics of the datasets are summarized in Table 1. In terms of noise and missing values, instances in the NUS-WIDE dataset were originally annotated manually by a group of students with carefully chosen concepts or labels [7]. Thus, there are no missing labels and the level of noise is reduced as a consequence of the label selection and annotation process. Note that in the BR approach dealing with missing labels is straightforward as they can be simply ignored, training a classifier only with the available positive and negative examples for the corresponding label.

The first column of the table names the datasets. The second column shows the number of features describing each instance. The third column depicts what the features represent. The fourth and fifth columns display the percentage of non-zero and zero values, respectively. Notice that the number of values in a dataset is the number of instances multiplied by the number of features and labels, e.g. the number of values in LLF_Normalized_BoW is 269,648 $*$ (500 + 81), which means that dataset has 156,665,488 values. Finally, the two last columns show the size of the dumps of the datasets instantiated in WEKA when using `DenseInstances` and `SparseInstances`, respectively.

In addition, we created two versions of each dataset using a different representation supported by WEKA to assess how the data representation affects the classifiers' performance. One version uses `DenseInstances` to represent the datasets. This version is expected to provide fast access to the data at the expense of the required memory. The other version uses `SparseInstances` to represent the datasets. This version requires less memory if the dataset is sparse at the cost of access time. However, if dataset is not sparse enough, the required memory to contain the dataset using the `SparseInstances` representation might be even higher that the required memory to contain the `DenseInstances` representation.

We ran this experiment using two different hardware/software configurations. The first configuration consisted of an AMD Phenom(tm) II X6 1055T Processor, which is a 6 core CPU running at 2.8 Ghz, with 8 Gb of RAM running Ubuntu 11.04 and Java 6. The other configuration was an Intel(R) Core(TM) i7-3820, which is a 4 core CPU with Hyper-Threading[12] running at 3.6 Ghz, with 32 Gb of RAM running Ubuntu 13.04 and Java 7. These machines were dedicated to run the experiments to avoid interference of other processes. Besides, we run each classifier for all the input datasets in each machine three times to assess the variation of the execution time across different runs. All the presented values in the figures are the average of the runs, and no run deviates more than 2.5% from the average.

Table 2 presents the training times (in hours) required by the MULAN library to train BR classifiers for each dataset. These times are the baselines for our experiment and all the speedups were calculated

[12] Intel Hyper-Threading Technology: http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html.

Table 2
Training times using MULAN

Dataset	Dataset representation	First configuration	Second configuration
LLF normalized BoW	Sparse	14:01:11.53	10:13:15.53
	Dense	4:34:17.84	1:58:13.04
LLF normalized CH	Sparse	1:39:52.89	1:04:26.64
	Dense	0:33:10.71	0:14:23.69
LLF normalized CM55	Sparse	7:27:16.42	5:14:17.45
	Dense	2:12:52.74	0:58:36.18
LLF normalized CORR	Sparse	3:57:47.24	2:45:39.96
	Dense	1:12:56.76	0:31:11.12
LLF normalized EDH	Sparse	1:57:20.76	1:16:31.95
	Dense	0:37:54.81	0:16:47.10
LLF normalized WT	Sparse	3:42:56.40	2:32:00.35
	Dense	1:10:08.52	0:30:24.89
All tags	Sparse	11:21:4.65	7:36:19.16
	Dense	N/A	4:17:23.88

using these times as base-time. Notice that it was not possible to train a BR classifier for the All tags dataset using the dense representation in the computers of the first configuration due to the high memory requirements. Another characteristic is that the training times were always lower when using dense representations. Finally, training times might be notably higher when using more time consuming base classifiers, such as SMO.

Figure 5 depicts the results of classifying the datasets using the `DenseInstances` representation. Although all classifiers were tested with all the datasets, some classifiers have no value for some datasets because their classifiers failed to finish due to lack of RAM memory. The All tags dataset, which is the largest dataset, could not be classified using the first configuration. In addition, the multi-thread versions that do not use sparse data representation, namely MTBRO and MTBRM, failed to run even with 32 Gb of RAM. However, MTBRS ran 2.27 times faster than STBR. There is an analogous case of this in the Fig. 5(a) for the LLF Normalized BoW dataset. This implies that in the case of large sparse datasets, the MTBRS classifier is the fastest.

For the rest of the datasets in Fig. 5(a), MTBRO was slightly faster than MTBRM, and both of them were significantly faster than MTBRS and STBR. In addition, MTBRS was between 10% and 22% faster than STBR for all the cases. Similarly, in the the second configuration results, which are outlined by Fig. 5(b), MTBRO and MTBRM were remarkably faster than MTBRS and STBR. However, MTBRO was slightly slower than MTBRM and MTBRS performed up to 24% slower than STBR.

Figure 6 presents the results of using sparse representation as input for the different classifiers' implementation. Similarly to when we used `DenseInstances`, the All tags datasets could be classified by the classifiers which use `SparseInstances` as intermediate representation of the transformations. Yet, in the case of LLF normalized BoW for the first configuration (Fig. 6(a)), MTBRO, which was not expected to fail because in this case it uses sparse data representation, actually failed. This probably stems from the memory overhead of WEKA Filters, and the fact that the dataset is not as sparse as the All tags dataset. Regarding the other datasets in the first configuration, all the classifiers show the same behavior: both MTBRO and MTBRS are between 3.75 and 4.23 times faster than STBR, while there is no significant speed variation between them. Finally MTBRM is between 9.90 and 10.28 times faster than STBR.

Regarding the second configuration, MTBRO and MTBRS are between 3.6 and 4.23 times faster than STBR, which is similar to what was observed in the other configuration. In this case, MTBRS finished

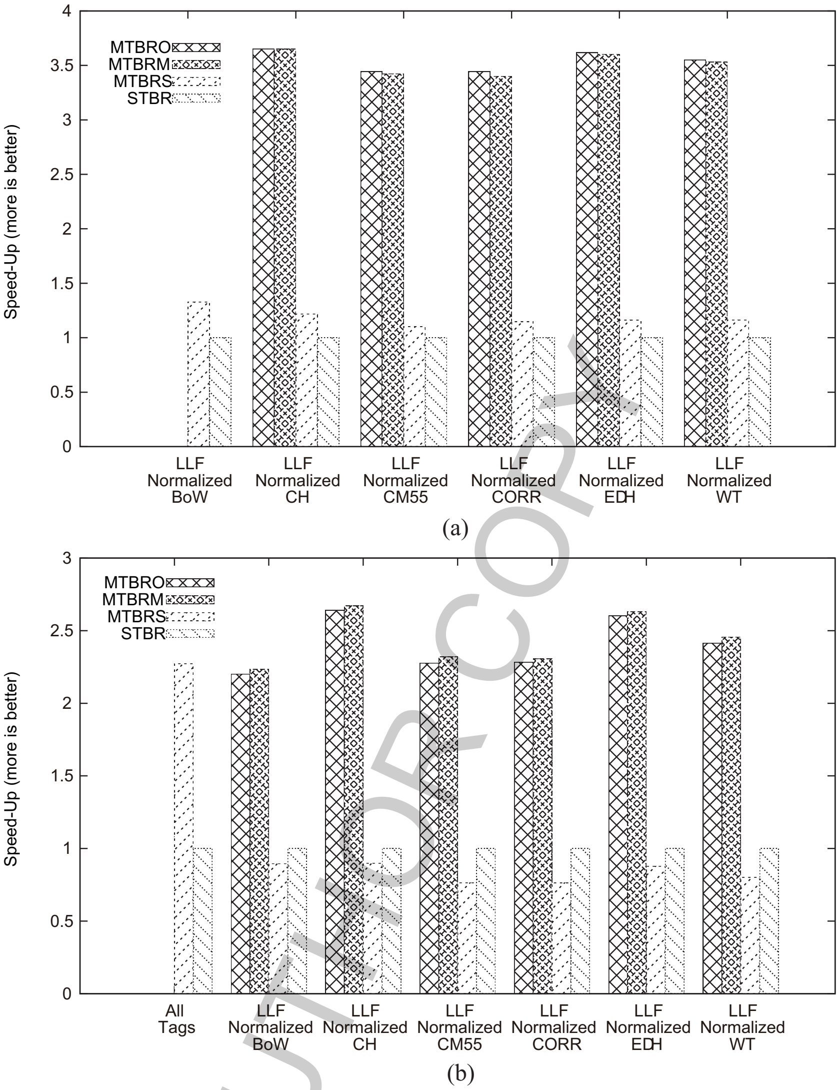
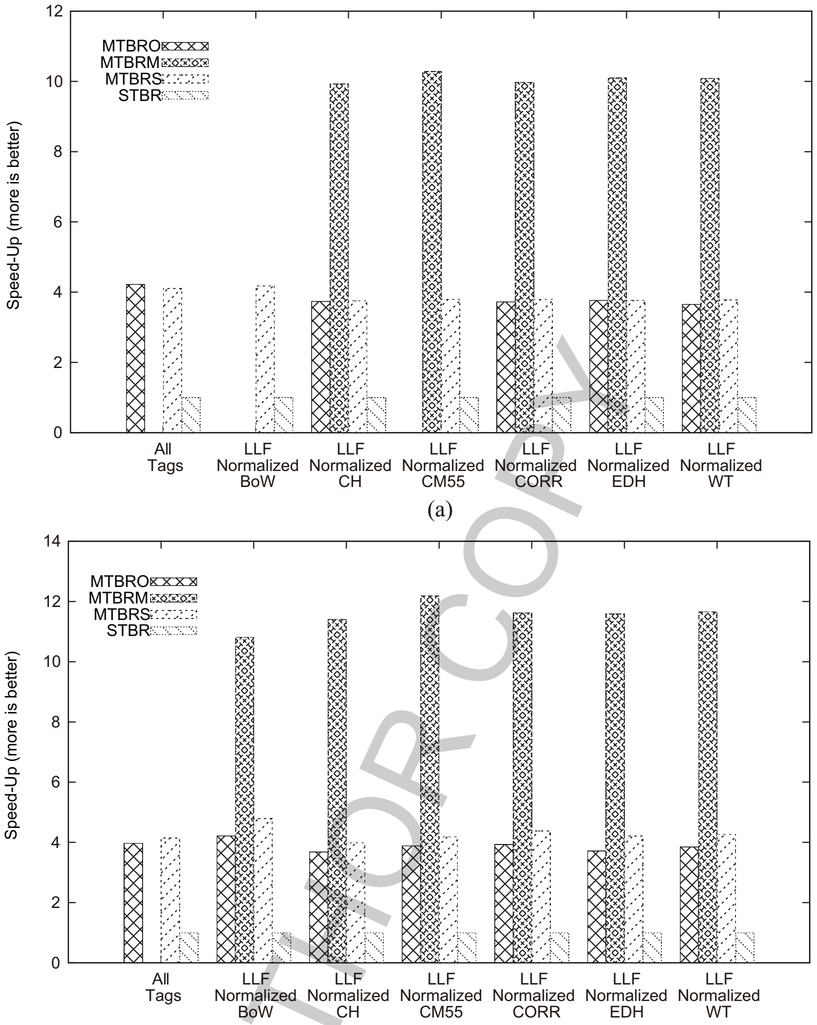


Fig. 5. DenseInstances results. (a) First configuration; (b) Second configuration.

marginally faster than MTBRO. Also consistent with the first configuration, MTBRM outperformed STBR, in this case by more than 10 times.

Figure 7 presents the speedups of all the runs in all the datasets normalized at the slowest classifier, which in all the cases is STBR with `SparseIntances` representation as input. It can be noticed that STBR always performed better, if it did not failed, when the input is given in `DenseInstances` representation. This is even true for the All Tags dataset which is the sparsest with 99.14% of the values being zeros. As shown in Fig. 7(b), STBR with `DenseInstances` as input performed 1.78 times faster than STBR with `SparseInstances`, while for all the other cases in both configurations the former performed more than three times faster than the later. A similar difference can be appreciated for MTBRO for all the cases in which both versions did not failed. For the other cases, namely MTBRM and MTBRS, there is no significant speedup when using `DenseInstances` or `SparseInstances`. This is prob-



(a)

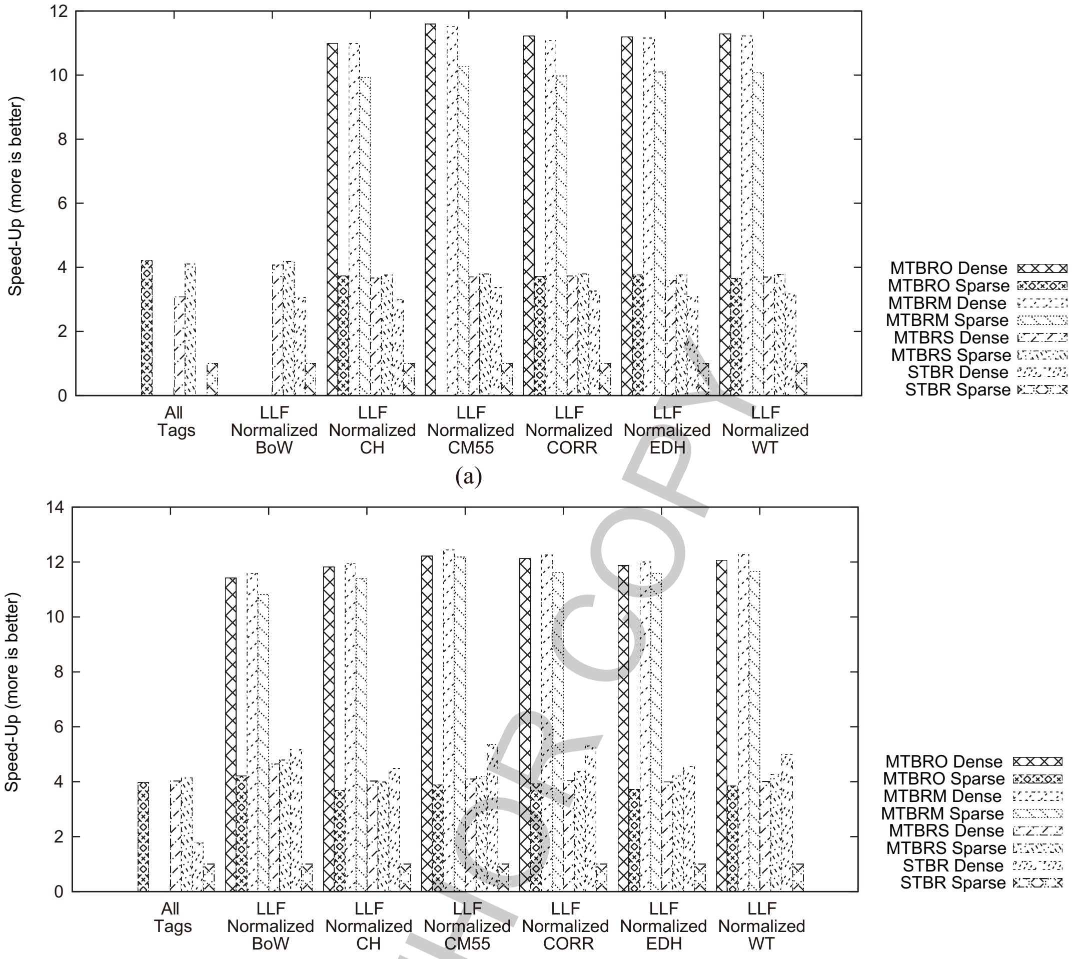
(b)

Fig. 6. SparseInstances results. (a) First configuration; (b) Second configuration.

ably because using one or the other representation only impacts on the creation of the transformation that changes the data representation regardless the input representation.

5.2. Multi-process training using scikit-learn

For this experiment, we used the first configuration only, scikit-learn version 0.15, and CPython 2.7.9. The experimental methodology followed for this evaluation was the same followed in previous section. This means that we trained a classifier using only one process, i.e. we performed a single core training, and then we performed the same task using concurrent processing. In this context, it is important to notice that scikit-learn takes advantage of multi-core processors in a different manner compared to our solution. Scikit-learn uses multi-process parallelization instead of multi-thread parallelization, being this latter a lightweight scheme. This is because CPython, which is the most common Python implementation, cannot execute Python bytecode concurrently in a multi-threaded manner due to its global interpreter



(a)

(b)

Fig. 7. Complete results. (a) First configuration; (b) Second configuration.

lock.[13] As a result, scikit-learn does not fully take advantage of processors that have some thread-level parallelism, such as Intel processors with Hyper-Threading. Then, we wanted to test the potential gains of our approach to parallel processing compared to that of scikit-learn.

We run the training phase using two different techniques as base classifiers, namely Gaussian Naïve Bayes (NB)[14] and Support Vector Classifier (SVC)[15] with a polynomial kernel (which is the kernel type that Weka uses by default), and used the BR multi-label classifier, which in scikit-learn is called One-Vs-the-Rest (OvR) multi-label.[16] OvR scikit-learn implementation allows multi-core execution by defining how many classifiers can be trained concurrently. In this context, we set OvR to train one classifier at the

[13] Global Interpreter Lock: https://wiki.python.org/moin/GlobalInterpreterLock.

[14] NB scikit-learn class: sklearn.naive_bayes.GaussianNB. URL: http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html.

[15] SVC scikit-learn class: sklearn.svm.SVC. URL: http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.

[16] OvR scikit-learn class: sklearn.svm.SVC. URL: http://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsRestClassifier.html.

Table 3
Scikit-learn training times and speed-up

Dataset	NB SP time	SVC SP time	NB MP speed-up	SVC MP speed-up
LLF normalized BoW	00:03:46	–	–	–
LLF normalized CH	00:00:30	72:03:36	2.31	3.27
LLF normalized CM55	00:01:32	120:23:44	2.36	–
LLF normalized CORR	00:01:00	77:51:36	2.31	2.98
LLF normalized EDH	00:00:34	91:56:13	2.27	3.02
LLF normalized WT	00:00:54	66:14:53	2.25	3.04
All tags	00:06:34	112:23:50	–	–

Table 4
Scikit-learn vs. MTBRO

Dataset	Scikit-learn	MTBRO	MTBRO improvement
LLF_Normalized_CH	3.27	3.65	11.60%
LLF_Normalized_CORR	2.98	3.44	15.39%
LLF_Normalized_EDH	3.01	3.62	19.98%
LLF_Normalized_WT	3.04	3.55	16.80%
Average	*3.07*	*3.57*	*15.94%*

time for the single-core version, and as many classifiers as cores are reported by the operating system for the multi-core version. The experiments were carried out using a dense representation of the datasets because not all classifier implementations in scikit-learn, e.g. the NB classifier implementation, support a sparse representation.

Table 3 presents the execution times obtained during the experiments using the OvR in a single core mode, and the speed-up when OvR is used in multi-core mode. Missing values in the table indicate experiments which failed because of RAM memory restrictions. The table reveals that the speed-up for training SVC models using the multi-core version is, in average, 3.08, with a lower bound at 2.98 and a higher bound at 3.27. Roughly, this value is slightly lower than the average obtained using the same configuration and MTBRO approach, i.e. the approach that only parallelize MULAN BR that runs without optimizing the training dataset representation (as specified above, MTBRM and MTBRS force specific data representation for the binary classifiers). In fact, the average speed-up of MTBRO was 3.54, which is 14% greater than the speed-up obtained by scikit-learn, and the speed-up bounds of MTBRO are 3.44 and 3.65. If for the sake of comparison we only consider the dataset for with both approaches could be applied successfully, MTBRO average speed-up was 3.57, while scikit-learn speed-up was 3.08, i.e. our approach outperformed scikit-learn by a 15.94%. Table 4 presents the speed-ups for sciki-learn and MTBRO for the comparable cases, as well as how MTBRO performed compared with scikit-learn.

We do not compare the single core time because we used SVC for scikit-learn experiments and SMO for MULAN experiments. Although both are based on support vector machines, there are many implementations differences, such as how the dataset is stored and accessed in main memory, making difficult to compare the absolute training times. Regardless, the main goal is not evaluating absolute times, but measuring the speed-up when taking advantage of multi-core capabilities. Regarding the speed-up difference between our approach and scikit-learn, it is likely that using a multi-process (such as scikit-learn) scheme instead of multi-thread one negatively impacts on the performance. A multi-thread scheme has several well-known advantages, e.g. sharing memory across threads is done by default because a multi-threaded application runs within one process, but sharing information between processes requires inter-process communication, which yields extra time costs.

Table 5
Time and relative speedup for dense representation

Dataset	Times	SC STBR	SC MTBRO	SC MTBRM	SC MTBRI	SC MTBRS	MC MTBRO
LLF normalized	Time	N/A	N/A	N/A	N/A	*35:16:14*	N/A
BoW	Speedup	N/A	N/A	N/A	N/A	*1*	N/A
LLF normalized	Time	6:13:28	2:41:53	2:40:31	*2:36:50*	5:17:23	N/A
CH	Speedup	1	2.3069	2.3265	*2.3811*	1.1767	N/A
LLF normalized	Time	21:39:17	*7:35:18*	N/A	N/A	19:45:17	N/A
CM55	Speedup	1	*2.8536*	N/A	N/A	1.0962	N/A
LLF normalized	Time	12:13:32	4:40:52	4:34:28	*4:28:02*	11:34:22	N/A
CORR	Speedup	1	2.6472	2.7090	*2.7739*	1.0782	N/A
LLF normalized	Time	7:01:02	3:01:22	2:56:23	*2:49:23*	6:03:39	6:28:35
EDH	speedup	1	2.3215	2.3870	*2.4855*	1.1578	1.0835
LLF normalized	Time	11:52:00	4:27:21	4:22:01	*4:12:57*	10:23:00	N/A
WT	Speedup	1	2.6632	2.7173	*2.8146*	1.1429	N/A

5.3. Multi-thread cached cross-validation experiments

The third experiment consisted in running different versions of the cross-validation algorithm and then comparing the execution times. The experiments were executed on the machines with the first configuration described above. Basically, we run the following versions of the cross-validation algorithm using a 10-fold validation:

- Original MULAN cross-validation (OR).
- Single threaded cross-validation with cached results, i.e. store partial results to disk for reducing RAM consumption (SC).
- Multi-threaded cross-validation with cached results (one thread per test) (MC).

Each cross-validation was tested using the four binary classifier implementations described in Section 5.1, namely STBR, MTBRO, MTBRM, MTBRS. We have also added a fifth implementation that extends MTBRM, which was the implementation that achieved the best performance in the experiments discussed in the previous section. This extension did not modify the training algorithm, but the prediction algorithm. Since predicting the labels for a new instance required to run different independent classifiers, the new version of MTBRM, which we call MTBRI ("I" stands for improved), uses different threads for the different classifiers. This might boost the performance of the validation part in which instances with known labels are classified for comparing the output against the actual labels.

We had also planned to run a multi-thread version of the MULAN cross-validation, i.e. without disk caching, but the MULAN cross-validation always failed due to lack of RAM. Therefore, it is not possible to run a multi-threaded version. In addition, several other experiment configurations could not be executed successfully because of RAM restrictions. The configurations that were run successfully are presented in two tables. Table 5 presents the results of using the dense representation of the datasets, while Table 6 shows the results obtained using the sparse representation.

As Table 5 shows, it was not possible to successfully finish any configuration that involves the All tags dataset when using dense representation. The case of the LLF Normalized BoW dataset was successfully finalized using SC with MTBRS. Another particular case was LFF Normalized CM55, in which the configurations SC with MTBRI and SC with MTBRS failed to execute. In these cases, the best performance was obtained using SC with MTBRM. In the rest of the cases, the best performance was obtained using SC with MTBRI. In all these cases, SC with MTBRS had a slightly worse performance, while SC with MTBRO was a close third. All in all, SC with MTBRS could be always executed. However, in the cases

Table 6
Time and relative speedup for sparse representation.

Dataset	Times	SC STBR	SC MTBRO	SC MTBRM	SC MTBRI	SC MTBRS	MC STBR	MC MTBRO	MC MTBRS
All tags	Time	108:49:43	31:07:56	N/A	N/A	N/A	*29:48:31*	30:11:48	N/A
	Speedup	1	3.4957	N/A	N/A	N/A	*3.6509*	3.6040	N/A
LLF normalized	Time	N/A	N/A	N/A	N/A	*32:29:42*	N/A	N/A	N/A
BoW	Speedup	N/A	N/A	N/A	N/A	*1*	N/A	N/A	N/A
LLF normalized	Time	16:20:32	5:35:51	2:50:14	*2:47:31*	5:08:14	N/A	N/A	11:32:26
CH	Speedup	1	2.9195	5.7595	*5.8531*	3.1812	N/A	N/A	1.4161
LLF normalized	Time	N/A	N/A	N/A	N/A	*18:45:18*	N/A	N/A	N/A
CM55	Speedup	N/A	N/A	N/A	N/A	*1*	N/A	N/A	N/A
LLF normalized	Time	37:07:08	11:03:29	4:56:45	*4:44:47*	10:30:03	N/A	N/A	N/A
CORR	Speedup	1	3.3567	7.5048	*7.8200*	3.5348	N/A	N/A	N/A
LLF normalized	Time	18:36:57	6:07:11	3:01:04	*2:57:43*	5:47:04	N/A	N/A	N/A
EDH	Speedup	1	3.0418	6.1684	*6.2849*	3.2182	N/A	N/A	N/A
LLF normalized	Time	34:15:37	10:18:58	4:51:44	*4:38:26*	9:47:23	N/A	N/A	N/A
WT	Speedup	1	3.3211	7.0462	*7.3827*	3.4996	N/A	N/A	N/A

in which SC with MTBRI or SC with MTBRO could be executed, they were more than two times faster than SC with MTBRS.

It was also not possible to successfully run all the experiments using the sparse dataset. As shown in Table 6, the combinations with the greatest performance were SC with MTBRI and SC with MTBRM, in the cases that was possible to run them. Then, SC with MTBRS and SC with MTBRO were between 1.8 to 2.25 times slower than SC with MTBRI and SC with MTBRM. Finally, it was possible to successfully evaluate the BR classifier using the All tags dataset. In this case, MC STBR was slightly faster than MC MTBRO and SC MTBRO. However, they were between 3.45 to 3.66 times faster that the slowest combination.

From the results presented in Tables 5 and 6, it is worth noticing that for the combination that force a similar representation of the transformed dataset, i.e. MTBRM, MTBRO and MTBRS, the execution times are fairly similar. However, MTBRO was substantially slower for the dataset sparse representation. However, sparse representation showed to be effective for processing large, but sparse dataset, e.g. the All tags dataset.

6. Conclusion

The multi-threaded approach for training a BR classifier has been shown to be effective in terms of reducing the execution time. According to our experiments, adding parallelism to the training process can bring a speed-up of up to 4.3 times. Such results are consistent with reported results of parallelizing other MLC algorithms [31]. In addition, changing the internal data representation of the dataset, when using a sparse representation, resulted in a speed-up of more than 10 times for some datasets. Our experiments showed that the sparse representation is useful only when the dataset has a large proportion of zero values. In our experiments, the only dataset that could take advantage of the sparse representation was the All Tags dataset that has 99.14% of zero values. Furthermore, under similar conditions, i.e. dense representation and same machine configuration, our MULAN based BR classifier implementation provided a 15.94% higher average speed-up versus the scikit-learn multi-core approach.

Another important finding of our experiments was that the original MULAN was unable to cross-validate the BR algorithm in any case. Furthermore, our experiments have shown that to achieve non-prohibitive memory requirements it is necessary to store part of the evaluation data into the hard-drive for freeing RAM. Our approach achieves this using the Java garbage collector to decide whether it is necessary to free RAM. This is done to minimize the times that objects should be swapped into the hard-drive because reading and writing operations in hard-drives are several orders of magnitude slower than in RAM.

In future work, we will study how our disk-caching approach can be adapted to train other complex transforming problems that require training a much larger number of traditional classifiers. For example, CRL and QWeighted approach to multi-label learning (QWML) [13] train a BR classifier and one versus one classifier for each pair of labels, i.e. the number of binary classifiers trained is $n + n * (n - 1)/2$ classifiers, where n is the number of labels. Since each binary classifier might require a large amount of memory [9], training CLR or QWML can require a substantial amount of memory. Considering the datasets used in this paper (81 labels), training BR requires training only 81 binary classifiers, but training CLR or QWML requires training 3,321 classifiers. Another type of MLC that can take advantage of this is ensemble MLC [39]. Ensemble MLC trains different classifiers at the same time and combines their results to improve prediction quality.

Also, we will evaluate the possibility of achieving further speed-ups by using a distributed computing environment, such as cluster, Grid and Cloud computing, to increase the number of tasks that can be run simultaneously. Furthermore, to address the issue of the required RAM, we will consider using in-memory distributed databases [25,45]. We also will investigate the combination of in-memory distributed databases for shared information with our disk-caching approach. Another issue that should be addressed is how to assign tasks in order to minimize the network usage resulting from data transfer. Otherwise, the network latency might have a great negative impact on the execution times [24]. Finally, we will analyze how to train multi-label classifiers by means of partitioning datasets in distributed environments. This would allow training multi-label classifiers with very large datasets that cannot be handled within a single computer [1,14,32]. In this context, our BR parallelization can scale in computational nodes linearly with the number of labels. However, other transformation approaches, such as CLR or QWML, can scale better because the number of classifiers required for such methods is quadratic with respect to the labels. For instance, in the 81-label dataset used, CLR and QWML methods would train 3,321 classifiers. Furthermore, this would require 3,321 transformed datasets that poses memory requirements which cannot be meet by a single computer. This means that, although a distributed approach might not be very beneficial for a BR classifier, it will provide several advantages for training CLR or QWML classifiers using large datasets because these approaches are both memory and CPU intensive.

Acknowledgements

We acknowledge the financial support by ANCPyT (grant PICT-2012-0045) and CONICET (grant PIP 2013-2015, code 11220120100185CO).

References

[1] R. Agrawal, A. Gupta, Y. Prabhu and M. Varma, Multi-label learning with millions of labels: Recommending advertiser bid phrases for web pages, in: *Proceedings of the 22nd International Conference on World Wide Web (WWW '13)*, Republic and Canton of Geneva, Switzerland, (2013), 13–24. URL http://dl.acm.org/citation.cfm?id=2488388.2488391.

[2] P. Basanta-Val and M. García-Valls, Resource management policies for real-time Java remote invocations, *Journal of Parallel and Distributed Computing* **74**(1) (2014), 1930–1944.

[3] G. Capannini, F. Silvestri and R. Baraglia, Sorting on GPUs for large scale datasets: A thorough comparison, *Information Processing & Management* **48**(5) (2012), 903–917. URL http://www.sciencedirect.com/science/article/pii/S0306457310001007.

[4] S. Celis and D.R. Musicant, Weka-parallel: Machine learning in parallel, Technical report, Carleton Collegue, 2003.

[5] E.Y. Chang, PSVM: Parallelizing support vector machines on distributed computers, in: *Foundations of Large-Scale Multimedia Information Management and Retrieval*, Springer Berlin Heidelberg (2011), 213–230.

[6] F.-C. Chang and H.-C. Huang, A refactoring method for cache-efficient swarm intelligence algorithms, *Information Sciences* **192** (2012), 39–49, swarm Intelligence and Its Applications.

[7] T.-S. Chua, J. Tang, R. Hong, H. Li, Z. Luo and Y. Zheng, NUS-WIDE: A real-world Web image database from National University of Singapore, in: *Proceedings of the ACM International Conference on Image and Video Retrieval (CIVR '09)*, ACM, Santorini, Fira, Greece (2009), 48:1–48:9.

[8] A. Clare and R.D. King, Knowledge discovery in multi-label phenotype data, in: *Proceedings of the 5th European Conference on Principles of Data Mining and Knowledge Discovery (PKDD '01)*, Springer-Verlag, London, UK, UK (2001), 42–53.

[9] R. Collobert, S. Bengio and C. Williamson, SVMTorch: Support vector machines for large-scale regression problems, *Journal of Machine Learning Research* **1** (2001), 143–160.

[10] O. Dekel and O. Shamir, Multiclass-multilabel classification with more classes than examples, *Journal of Machine Learning Research – Proceedings Track* **9** (2010), 137–144.

[11] A. Elisseeff and J. Weston, A kernel method for multi-labelled classification, in: *Advances in Neural Information Processing Systems 14*, MIT Press (2001), 681–687.

[12] X. Fei, X. Li and C. Shen, Parallelized text classification algorithm for processing large scale tcm clinical data with mapreduce, in: *Information and Automation, 2015 IEEE International Conference on*, (2015), 1983–1986.

[13] J. Fürnkranz, E. Hüllermeier, E.L. Mencía and K. Brinker, Multilabel classification via calibrated label ranking, *Machine Learning* **73**(2) (2008), 133–153.

[14] L. Georgopoulos and M. Hasler, Distributed machine learning in networks by consensus, *Neurocomputing* **124** (2014), 2–12. URL http://www.sciencedirect.com/science/article/pii/S0925231213003639.

[15] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann and I.H. Witten, The WEKA data mining software: An update, *ACM SIGKDD Explorations Newsletter* **11**(1) (2009), 10–18.

[16] S. Herrero-Lopez, J.R. Williams and A. Sanchez, Parallel multiclass classification using SVMs on GPUs, in: *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU '10)*, ACM, Pittsburgh, PA, USA (2010), 2–11.

[17] J. Illig, A. Hotho, R. Jäschke and G. Stumme, A comparison of content-based tag recommendations in folksonomy systems, in: *Proceedings of the 1st International Conference on Knowledge Processing and Data Analysis (KONT'07/KPP'07)*, volume 6581 of LNCS, Springer-Verlag, Novosibirsk, Russia (2011), 136–149.

[18] G.H. John and P. Langley, Estimating continuous distributions in bayesian classifiers, in: *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence (UAI'95)*, Morgan Kaufmann Publishers Inc., Montreal, Canada (1995), 338–345.

[19] M. Kalin and D. Miller, Clojure for number crunching on multicore machines, *Computing in Science Engineering* **14**(6) (2012), 12–23.

[20] I. Katakis, G. Tsoumakas and I. Vlahavas, Multilabel text classification for automated tag suggestion, in: *Proceedings of the ECML/PKDD-08 Workshop on Discovery Challenge*, Antwerp, Belgium **75** (2008).

[21] X. Kong, B. Cao and P.S. Yu, Multi-label classification by mining label and instance correlations from heterogeneous information networks, in: *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '13, ACM*, New York, NY, USA (2013), 614–622. URL http://doi.acm.org/10.1145/2487575.2487577.

[22] R. Lammel, Google's MapReduce programming model – Revisited, *Science of Computer Programming* **70**(1) (2008), 1–30.

[23] Z. Li, Z. Shi, W. Zhao, Z. Li and Z. Tang, Learning semantic concepts from image database with hybrid generative/discriminative approach, *Engineering Applications of Artificial Intelligence* **26**(9) (2013), 2143–2152. URL http://www.sciencedirect.com/science/article/pii/S0952197613001322.

[24] T.-Y. Liang, C.-Y. Wu, C.-K. Shieh and J.-B. Chang, A grid-enabled software distributed shared memory system on a wide area network, *Future Generation Computer Systems* **23**(4) (2007), 547–557.

[25] K.W. Lin and Y.-C. Lo, Efficient algorithms for frequent pattern mining in many-task computing environments, *Knowledge-Based Systems* **49** (2013), 10–21.

[26] G. Madjarov, D. Kocev, D. Gjorgjevikj and S. Džeroski, An extensive experimental comparison of methods for multi-label learning, *Pattern Recognition* **45**(9) (2012), 3084–3104.

[27] P. Malarvizhi and R.V. Pujeri, Multilabel classification of documents with MapReduce, *International Journal of Engineering and Technology* **5** (2013), 1260–1267.

[28] C. Mateos, A. Zunino and M. Hirsch, EasyFJP: Providing hybrid parallelism as a concern for divide and conquer Java applications, *Computer Science and Information Systems* **10**(3) (2013), 21–21.

[29] A.K. McCallum, Multi-label text classification with a mixture model trained by EM, in: *AAAI 99 Workshop on Text Learning*, Orlando, FL, USA (1999), 1–7.

[30] E.L. Mencía, S.-H. Park and J. Furnkranz, Efficient voting prediction for pairwise multilabel classification, *Neurocomputing* **73**(7–9) (2010), 1164–1176.

[31] E.A. Mosabbeb and M. Fathy, Distributed matrix completion for large-scale multi-label classification, *Intelligent Data Analysis* **18**(6) (2014), 1137–1151. URL http://dx.doi.org/10.3233/IDA-140688.

[32] M. Naldi and R. Campello, Evolutionary k-means for distributed data sets, *Neurocomputing* **127** (2014), 30–42. URL http://www.sciencedirect.com/science/article/pii/S0925231213007674.

[33] B. Panda, J.S. Herbach, S. Basu and R.J. Bayardo, PLANET: Massively parallel learning of tree ensembles with mapreduce, *Proc VLDB Endow* **2**(2) (2009), 1426–1437. URL http://dx.doi.org/10.14778/1687553.1687569.

[34] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot and E. Duchesnay, Scikit-learn: Machine learning in Python, *Journal of Machine Learning Research* **12** (2011), 2825–2830.

[35] P. Prajapati, A. Thakkar and A. Ganatra, A survey and current research challenges in multi-label classification methods, *International Journal of Soft Computing and Engineering* **2** (2012), 248–252.

[36] J. Read, A. Bifet, G. Holmes and B. Pfahringer, Scalable and efficient multi-label classification for evolving data streams, *Machine Learning* **88**(1) (2012), 243–272.

[37] J. Read, B. Pfahringer, G. Holmes and E. Frank, Classifier chains for multi-label classification, *Machine Learning* **85**(3) (2011), 333–359.

[38] M. Samovsky and T. Kacur, Cloud-based classification of text documents using the Gridgain platform, in: *Proceedings of the 7th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI 2012)*, Timisoara, Romania (2012), 241–245.

[39] C. Sanden and J.Z. Zhang, Enhancing multi-label music genre classification through ensemble techniques, in: *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '11)*, ACM, Beijing, China (2011), 705–714.

[40] R. Schapire and Y. Singer, BoosTexter: A boosting-based system for text categorization, *Machine Learning* **39**(2–3) (2000), 135–168.

[41] S. Schiele, H. Blaar, M. Muller-Hanneman, D. Thurkow and M. Moller, Parallelization strategies to speed-up computations for terrain analysis on multi-core processors, in: *ARCS Workshops (ARCS 2012)*, Muenchen, Germany (2012), 1–6.

[42] C. Shen, J. Jiao, Y. Yang and B. Wang, Multi-instance multi-label learning for automatic tag recommendation, in: *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC 2009)*, San Antonio, TX, USA (2009), 4910–4914.

[43] Y. Song, L. Zhang and C.L. Giles, A sparse gaussian processes classification framework for fast tag suggestions, in: *Proceedings of the 17th ACM Conference on Information and Knowledge Management (CIKM '08)*, ACM, Napa Valley, CA, USA (2008), 93–102.

[44] M.S. Sorower, A literature survey on algorithms for multi-label learning, Technical report, Corvallis, OR, Oregon State University, 2010.

[45] F. Stahl and M. Bramer, Computationally efficient induction of classification rules with the PMCRI and J-PMCRI frameworks, *Knowledge-Based Systems* **35** (2012), 49–63.

[46] L. Tang, S. Rajan and V.K. Narayanan, Large scale multi-label classification via metalabeler, in: *Proceedings of the 18th International Conference on World Wide Web (WWW '09)*, ACM, Madrid, Spain (2009), 211–220.

[47] G. Tsoumakas, I. Katakis and I. Vlahavas, Mining multi-label data, in: *Data Mining and Knowledge Discovery Handbook*, Springer (2010), 667–685.

[48] G. Tsoumakas, I. Katakis and I.P. Vlahavas, Effective and efficient multilabel classification in domains with large number of labels, in: *ECML/PKDD 2008 Workshop on Mining Multidimensional Data*, (2008), 30–44.

[49] G. Tsoumakas, E. Spyromitros-Xioufis, J. Vilcek and I. Vlahavas, MULAN: A Java library for multi-label learning, *Journal of Machine Learning Research* **12** (2011), 2411–2414.

[50] S.R. Upadhyaya, Parallel approaches to machine learning – A comprehensive survey, *Journal of Parallel and Distributed Computing* **73**(3) (2013), 284–292.

[51] P. Vateekul, M. Kubat and K. Sarinnapakorn, Hierarchical multi-label classification with SVMs: A case study in gene function prediction, *Intelligent Data Analysis* **18**(4).

[52] P.K. Wojciech Indyk, Tomasz Kajdanowicz, Relational large scale multi-label classification method for video categorization, *Multimedia Tools and Applications* **65**(1) (2013), 63–74.

[53] J. Xu, An efficient multi-label support vector machine with a zero label, *Expert Systems with Applications* **39**(5) (2012), 4796–4804.

[54] J. Xu, Fast multi-label core vector machine, *Pattern Recognition* **46**(3) (2013), 885–898.

[55] I. Yélamos, G. Escudero, M. Graells and L. Puigjaner, Performance assessment of a novel fault diagnosis system based on support vector machines, *Computers & Chemical Engineering* **33**(1) (2009), 244–255.

[56] Z. Yu, Q. Wang, Y. Fan, H. Dai and M. Qiu, An improved classifier chain algorithm for multi-label classification of big data analysis, in: *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, (2015), 1298–1301.

[57] M.-L. Zhang and Z.-H. Zhou, ML-KNN: A lazy learning approach to multi-label learning, *Pattern Recognition* **40**(7) (2007), 2038–2048.