Supervised Papers Classification on Large-Scale High-Dimensional Data with Apache Spark

Leonidas Akritidis, Panayiotis Bozanis, Athanasios Fevgas
Data Structuring & Engineering Lab
Department of Electrical and Computer Engineering
University of Thessaly, Volos, Greece
Email: {leoakr,pbozanis,fevgas}@e-ce.uth.gr

***Abstract*—The problem of classifying a research article into one or more fields of science is of particular importance for the academic search engines and digital libraries. A robust classification algorithm offers the users a wide variety of useful tools, such as the refinement of their search results, the browsing of articles by category, the recommendation of other similar articles, etc. In the current literature we encounter approaches which attempt to address this problem without taking into consideration important parameters such as the previous history of the authors and the categorization of the scientific journals which publish the articles. In addition, the existing works overlook the huge volume of the involved academic data. In this paper, we expand an existing effective algorithm for research articles classification, and we parallelize it on Apache Spark –a parallelization framework which is capable of sharing large amounts of data into the main memory of the nodes of a cluster– to enable the processing of large academic datasets. Furthermore, we present data manipulation methodologies which are useful not only for this particular problem, but also for most parallel machine learning approaches. In our experimental evaluation, we demonstrate that our proposed algorithm is considerably more accurate than the supervised learning approaches implemented within the machine learning library of Spark, whereas it outperforms them in terms of execution speed by a significant margin.**

I. Introduction

During the past years, the academic search engines and digital libraries have played a crucial role in the evolution of scientific research. The scientists who desire to develop novel solutions or address new problems, usually consult them with the aim of obtaining a detailed picture of the current literature in a specific topic. Hence, they are able to review the state-of-the-art articles for a particular problem and organize their research accordingly.

To allow this functionality, the aforementioned services classify their articles into one or more areas of science. Some of them employ static, pre-defined hierarchies of categories and ask from the authors to classify their articles themselves (e.g., the IEEE and ACM digital libraries). In a well-controlled environment, where the number of the published articles is limited, this approach is feasible. However, in the context of academic search engines (such as Microsoft Academic and Google Scholar), which index hundreds of millions of articles from different publishers and with diverse categorization, this method apparently cannot be applied. Consequently, the issue of the automatic classification of research articles is of remarkable importance, since it allows the users to perform searches by focusing on only a specific portion of the indexed documents, thus increasing both effectiveness and efficiency. Additional benefits include similar documents recommendations, collaborative filtering, query expansion facilities, expert identification, and so on.

In the current literature there are several methods which attempt to address the problem of the automatic papers classification. The traditional approaches employ keyword extraction algorithms which identify repeated patterns of words and discover the most representative keywords of an article. In the sequel, they employ standard supervised classification approaches (such as kNN and naive Bayes) to label the article. The model proposed in [1] introduced rich feature vectors and included additional information such as authorship, co-authorship, and journals history to enhance the accuracy of the classification. Other methods adopt citation analysis procedures which focus on the recognition of various properties of the incoming and outgoing citations. However, since a portion of the citations of a paper is usually unavailable, these methods cannot construct a complete graph of papers and, consequently, they miss valuable information. A second drawback of the link-based approaches is that a reference to an article does not necessarily reveal thematic affinity.

Nevertheless, none of the aforementioned approaches takes into consideration the huge volume of the involved data. The current scientific literature contains at least hundreds of millions of articles and its size increases constantly at high rates, as more articles get published. The Open Academic Graph, an experimental dataset which was recently made publicly available[1], contains more than 167 million publications and occupies approximately 300GB in uncompressed form [2], [3]. It is obvious that such big data with so frequent updates cannot be handled efficiently by a single workstation.

In this paper we introduce a parallel algorithm to confront the problem of the automatic paper classification in such large-scale datasets. Our proposed method is based on the supervised model of [1], and it has been developed by using a relatively new parallelization framework, the *Apache Spark*. This framework, in contrast to its predecessor, MapReduce, has been designed to allow the sharing of large amounts of data into the main memory of the workstations of a cluster.

[1] https://www.openacademic.ai/oag

Consequently, it avoids redundant disk reads and disk writes, a problem which was common in MapReduce. Spark can be many times faster than MapReduce (up to 100 times), especially for iterative and machine learning algorithms similar to the one we present here. In addition, we capitalize on a careful combination of data manipulation techniques, like feature vectorization and dimensionality reduction.

More specifically, our method initially converts the dataset by constructing the appropriate feature vectors. Since the number of features is huge and the vectors are very sparse, we apply a mixture of dimensionality reduction techniques to shrink the feature space. After the preparation of the dataset, the training process builds in parallel a statistical model which is based on a dictionary data structure for storing the keywords, the authors, and the journal of an article. Next, the model is broadcasted to all processing nodes of the cluster and it is used to classify the articles of the testing set by computing a score for each category of the given taxonomy. Finally, each article is classified in the highest-scoring category.

The rest of the paper is organized as follows: In Section II we refer to some of the most significant works in the literature related to the problem of paper classification and the parallel algorithms for Apache Spark. In Section III we establish the theoretical background and we present the details of our introduced model. The experimental evaluation of the proposed method is discussed in Section IV, where we also describe the dataset filtration and pre-processing steps. Finally, Section V contains some useful remarks and final conclusions about the paper and the presented methodology.

II. Related Work

The problem of research articles classification has attracted numerous scientists in the past and there is a significant number of works which attempted to address it. These works introduced methods which can be divided into two categories: link-based and text-based methods.

In the former category, we mainly encounter approaches which employ link-analysis strategies to discover useful information hidden behind the references of the articles and the linkage between them. For instance, in [4] the authors introduce a statistical framework for modeling link distributions. In the sequel, they identify the category of a document according to the category its links belong to. Furthermore, the link-based classification strategies have been proved effective in categorizing graph nodes (for example, labeling the nodes of a graph [5], or in networked data classification [6]). Similar approaches have been also applied to Web data, where the document interlinking can be used for a variety of purposes. An important survey which studies these methods in depth is provided in [7].

Regarding the text-based algorithms, there is a surge of research relevant to the supervised text categorization problem [8]. A systematic survey of the most effective machine-learning approaches is presented in [9]. Moreover, [10] and [11] provide detailed evaluations of the primary statistical and machine learning approaches to text categorization. Finally, Joachims employed support vector machines (SVM) to address the problem [12], whereas [13] introduced AdaBoost.MH, a multi-class extension of the traditional AdaBoost binary classifier.

However, none of the aforementioned approaches study the additional features which are important for the specific problem that we examine here. Therefore, they miss valuable information such as the previous work of the authors of an article, co-authorship information and the history of the publishing journal. Additionally, they do not take into consideration the huge volume of the involved data and are not designed to operate in parallel across the nodes of a computational cluster.

In this paper we employ Apache Spark [14], a robust framework for big data processing, to parallelize the algorithm introduced in [1]. Spark has been designed to address the weaknesses of its predecessor, the well-known MapReduce introduced by Google [15]. In [16] the authors provide a tutorial for Spark and its characteristics, whereas [17], [18] and [19] discuss its performance in comparison to MapReduce. Finally, a number of works introduce parallel algorithms for Spark in order to confront problems such as distributed matrix computations [20] and graph partitioning [21].

III. Articles Classification Algorithm

In this Section we describe the details of the proposed algorithm for research articles classification with the Apache Spark framework. Initially, we present some basic preliminary elements and, in the sequel, we describe the characteristics of the model, the training and classification methods, and the development of the algorithm in Spark.

A. Theoretical Background

Before we proceed to the presentation of the algorithm, let us initially establish the necessary theoretical background which shall provide the basis for the analysis that follows. We begin with Y, a set which includes all the research fields (also mentioned as categories, or labels, or *fields of study - FOS*) of the dataset. The items of Y can be organized by employing a hierarchical tree structure, or a graph, or they may not be explicitly connected to each other.

In addition, we introduce X, a set which contains all the publications of the dataset, and another set J, which includes the journals (or the conference proceedings, magazines, books, etc.) where the items of X have been published. Since each paper cannot be published by more than one journal, each entry $x \in X$ is uniquely correlated with a single element $j \in J$. In addition, we use A to symbolize the set which includes all the distinct authors who have contributed to the publication of the articles of P.

A significant portion of the published articles also includes a limited number of keywords; that is, special representative words which are selected by the authors to briefly describe the content of their work. A part of the proposed algorithm is based on these keywords, therefore, it is also required that we define K, a set which contains all the keywords encountered in all papers of X. In the same set K we also include the

keywords extracted from the titles of the articles, since these words represent the contents of the documents as well.

The keywords, the authors and the publishing journals represent the feature space F. We can now introduce the following groups of subsets:

- $F_x \subset F$: These subsets contain all the features of an article x. For instance, $K_x \subset K$ and $A_x \subset A$ include all the keywords and the authors of x, respectively.
- $X_f \subset X$: Another group of subsets comprised by the articles which have the feature f. In particular, $X_k \subset X$ and $X_a \subset X$ include all the publications which contain the keyword k and have been authored by a, respectively. Notice that $|X_f|$ represents the frequency of the feature f across the entire document collection.
- $X_{f,y} \subset X_f$: They include the articles which both have the feature f and belong to the category y. In this case, $|X_{f,y}|$ is a counter which represents how many times f has been correlated with y.

Finally, the supervised machine learning algorithms split their input dataset $\mathcal{D}$ into two parts: the training set $\mathcal{T}$, and the testing set $\mathcal{R}$. The former consists of samples, that is, labeled articles whose properties are used to build the classification model. The latter provides a number of unlabeled records to attest the effectiveness of the model.

B. Model Training

In this phase we process the training set $\mathcal{T}$ and we construct our model with respect to the set of labels Y. The objective of this procedure is to correlate the features included in $\mathcal{T}$ to one or more labels from Y and also, to quantify the strength of these correlations.

Initially, let us describe the types of the features which we use to build our model and justify our decision to include them in our algorithm. Clearly, we desire to focus on elements which are strong indicators of the label of a research article. Consequently, the first feature we utilize is the explicit keywords, that is, the words which are used by the authors to quickly describe the content of their article. Moreover, the words occurring in the titles are also considered representatives of the document's content and for this reason we shall treat them as keywords.

The previous activity of the authors who contribute to a research paper can also provide an indication of the research field a paper discusses. Learning the areas of expertise of a scholar is important, since it can be exploited to classify his/her unlabeled articles. Therefore, the article authors become our second feature. Additionally, the publishing journal is also indicative of the research area that a paper belongs to. This is due to the fact that journals are also categorized and do not publish articles with subjects unrelated to their area(s) of interest.

The correlation of a feature with a research field is quantified by employing the two last groups of subsets of the previous subsection. More specifically, the classification model is based on a data structure $\mathcal{M}$, where we store all the distinct features of the dataset. For each feature $f \in \mathcal{M}$, this structure also maintains:

Algorithm 1: Model Training

```
1  initialize dictionary M;
2  for each sample x ∈ T with known label y do
3      extract all features F_x of x;
4      for each feature f ∈ F_x do
5          if M.search(f) == false then
6              M.insert(f);
7              set |X_f| ← 1;
8              M.insertRDV(f,y);
9              set |X_{f,y}| ← 1;
10         else
11             set |X_f| ← |X_f| + 1;
12             if M.searchRDV(f,y) == false then
13                 M.insertRDV(f,y);
14                 set |X_{f,y}| ← 1;
15             else
16                 set |X_{f,y}| ← |X_{f,y}| + 1;
17             end
18         end
19     end
20 end
```

- A weight w_f, which reflects the importance of f. The features of the same type are assigned identical weight values. Therefore, all the keywords, authors, and publishing journals are assigned the same weights w_k, w_a, and w_j, respectively.
- A value $|X_f|$, which represents the number of articles having the feature f –in other words, the overall frequency of the feature in the dataset.
- A vector of labels, which have been correlated with f, called *relevance description vector*, or *RDV*. Each entry in the RDV is accompanied by a frequency value $|X_{f,y}|$ which counts how many times f has been correlated with a label y; that is, the number of papers which both possess f and are labeled as y.

As we will see shortly in the following subsection, the idea is to determine the strength of the correlation of f with y through the calculation of the ratio $|X_{f,y}|/|X_f|$. Algorithm 1 shows the basic steps for training $\mathcal{M}$. For each paper x with label y in the training set, we initially identify all the features F_x. For each of these features, we perform a lookup in $\mathcal{M}$. In case the search is unsuccessful, the feature is stored in $\mathcal{M}$ with frequency equal to 1. We also store y in the RDV of f and we set $|X_{f,y}| = 1$. In case f already exists in $\mathcal{M}$ (successful search), we increase its global frequency $|X_f|$ and we perform another search for y in its RDV. If this second search is not successful, we insert y in the RDV and we set $|X_{f,y}| = 1$; otherwise, we merely increase $|X_{f,y}|$ by one.

C. Research Articles Classification

We can now employ the trained model to classify an unlabeled article $x \in \mathcal{R}$. The process begins with the initialization of an empty list of candidate labels Y. Each entry $y \in Y$ is assigned a score S_y, whose value is initially set equal to zero and it will later be updated by using the model $\mathcal{M}$.

In the sequel, we extract all the features F_x out of x and, for each of these features, we perform a search within the

Algorithm 2: Research Articles Classification

```
1  initialize list of candidate labels Y;
2  for each unlabeled article x ∈ R do
3      extract all features F_x of x;
4      for each feature f ∈ F_x do
5          if M.search(f) == true then
6              for each label y in the RDV of f do
7                  set S_y ← S_y + w_f|X_{f,y}|/|X_f|;
8                  Y.insert(y, S_y);
9              end
10         end
11     end
12     Y.sortByScore();
13     set label of x ← Y[0];
14 end
```

Algorithm 3: Driver program

```
1  Function main()
2      Configure and Initialize Spark;
3      set RDD<LabeledPoint> D ← readLIBSVM(location);
4      set RDD<LabeledPoint> [] W ← D.split(N, 1 − N);
5      set RDD<LabeledPoint> T ← W[0];
6      set RDD<LabeledPoint> R ← W[1];
7      Initialize Model M;
8      M.train(T);
9      broadcast M;
10     M.classify(R);
11     Shutdown Spark;
12 end
```

model $\mathcal{M}$. In case this search is successful, we retrieve the RDV of f with the associated research areas along with their respective $|X_{f,y}|$ values. Each entry of this vector is inserted in the candidate labels list, provided that it is not already present there. At next, the score value in the candidate list for each label is updated by a quantity $w_f|X_{f,y}|/|X_f|$.

At the end of this process, we obtain a list of candidate labels (research areas); each label is accompanied by a score value given by the following formula:

$$S_y = \sum_{f \in F_x} w_f \frac{|X_{f,y}|}{|X_f|} \tag{1}$$

which reflects how strong is the connection of this area with all the features of the paper. All we have to do now is to sort this list in decreasing score order and select the highest scoring label as the representative research area for this article. In Algorithm 2 we present the basic steps of this procedure.

D. Algorithm Parallelization & Deployment in Spark

After the establishment of the main parts of the algorithm, we now proceed to its preparation for parallel execution in Spark. In Algorithm 3 we present the skeleton of the driver program, i.e., the program which controls the execution flow of the assigned job. Initially, in step 2, we determine the details of the job by tuning a several parameters, such as the number of the executors, the number of threads which will be created inside each executor, the memory limits for both the driver and the executors, the maximum allowed size of the output data, and numerous others.

Then, in step 3, we read the dataset $\mathcal{D}$ from its location and we store it into a *Resilient Distributed Dataset (RDD)*. The Spark RDDs are fault-tolerant object collections for handling data, and may be assigned various persistence levels. The input dataset has been previously processed and each research article was converted to a *labeled point* by a carefully designed combination of feature vectorization and dimensionality reduction techniques as described in Section IV. A labeled point is a Spark abstraction which is used to represent a record of the dataset and consists of two parts: i) the label of the record, and ii) a *sparse vector* of feature-value (f, w_f) pairs. This collection of labeled points is formatted according to the *LIBSVM* format, a schema which is supported by all machine learning algorithms of Spark. More details about how the original dataset was converted to the LIBSVM format are provided in the experimental evaluation, in Subsection IV-B.

In the next three steps, the training set $\mathcal{T}$ and the testing set $\mathcal{R}$ are created from $\mathcal{D}$. Spark offers a simple and powerful dataset *split* method, which initially shuffles $\mathcal{D}$ and splits it into two parts stored in an array $\mathcal{W}$. The parameter N is expressed as a percentage of the size of $\mathcal{D}$ and controls the sizes of the training and test sets. In our experiments, we set $N = 0.6$, that is, the sizes of $\mathcal{T}$ and $\mathcal{R}$ are equal to the 60% and 40% of the size of $\mathcal{D}$, respectively. Afterwards, the model $\mathcal{M}$ is initialized and then trained by using the training set $\mathcal{T}$ (step 8). After the completion of the training phase, a special *broadcast* call transmits $\mathcal{M}$ to all the executors of the cluster to allow them classify, in parallel and independently of each other, the unlabeled records of the testing set $\mathcal{R}$. The driver program is finally terminated by shutting down the Spark session and by releasing all the allocated resources (step 11).

In Algorithm 4 we present a brief pseudo-implementation of our classification model in Spark. We have already discussed the properties of $\mathcal{M}$: it is a standard dictionary data structure which stores all the features, accompanied by with their frequency $|X_f|$ and their RDV. The model implements two basic functions, *train* and *classify*. The former is comprised of two phases: In the first phase we perform a ***flatMap*** transformation, which, for each row (i.e., labeled point) of $\mathcal{T}$, returns a list λ of (f, y, w_f) tuples (steps 4–16). After the input has been exhausted, all the lists λ are sent back to the driver, where they are *collected* (step 16) and their contents (i.e., the (f, y, w_f) tuples), are stored together in a large list l. During the second phase (steps 17–32), we iterate through l and we populate the features and their respective RDVs in $\mathcal{M}$, likewise to the procedure of Subsection III-B. Notice the similarity of the steps 17–32 with the steps 4–19 of Algorithm 2.

Regarding the classification function, the procedure begins with the initialization of an empty list of candidate labels. A *map* transformation of the testing set $\mathcal{R}$ creates one (y', y) pair per input row, where y and y' represent the actual and the predicted labels respectively. The predicted label is computed by searching $\mathcal{M}$ for each feature f of each labeled point of

Algorithm 4: Model implementation in Spark

```
Model
    Features Dictionary M
    Function train(RDD<LabeledPoint> T)
        List<Tuple(f, y, w_f)> l ← T.flatMap(LabeledPoint)
            for each row ∈ T do
                initialize list λ;
                set y ← row.label;
                set V ← row.featureVector;
                for each entry v ∈ V do
                    set featureID f ← v.f;
                    set weight w_f ← v.w_f;
                    λ.insert(f, y, w_f);
                end
                return λ;
            end
        collect
        for each tuple (f, y, w_f) ∈ l do
            if M.search(f) == false then
                M.insert(f);
                set |X_f| ← 1;
                M.insertRDV(f,y);
                set |X_{f,y}| ← 1;
            else
                set |X_f| ← |X_f| + 1;
                if M.searchRDV(f,y) == false then
                    M.insertRDV(f,y);
                    set |X_{f,y}| ← 1;
                else
                    set |X_{f,y}| ← |X_{f,y}| + 1;
                end
            end
        end
    end
    Function classify(RDD<LabeledPoint> R)
        List<Tuple(y', y)> L ← R.map(LabeledPoint)
            for each row ∈ R do
                initialize candidates list C;
                set y ← row.label;
                set V ← row.featureVector;
                for each entry v ∈ V do
                    set featureID f ← v.f;
                    set weight w_f ← v.w_f;
                    if M.search(f) == true then
                        for each label c in the RDV of f do
                            set S_c ← S_c + w_f|X_{f,c}|/|X_f|;
                            C.insert(c, S_c);
                        end
                    end
                end
                C.sortByScore();
                set y' ← C[0];
                return Tuple(y', y);
            end
        collect
    end
end
```

$\mathcal{R}$. In case the search is successful, $|X_f|$ and the RDV of f are retrieved. For each entry y in the RDV, the score of y is updated according to equation 1. In the sequel, the list of the candidate labels is sorted in decreasing score order, and y' is set equal to the first, highest scoring element of the list.

IV. Experiments

In this section, we experimentally demonstrate the effectiveness and the efficiency of the proposed algorithm which we named *Paper Classifier (PC)*. The code for all algorithms was developed in Java 1.8, and it was executed on the latest stable version 2.3.0 of Spark.

The experiments were conducted on the cloud infrastructure of our Department. In particular, our cluster included 8 nodes with 16 CPUs and 64GB of memory each. In this environment, Spark was deployed on top of a YARN cluster with the Hadoop Distributed File System installed. YARN and Spark require a careful configuration of the available resources to achieve maximum performance. To ensure fair results, all methods were tested on the same configuration. Spark was allowed to launch 15 executors, that is, 2 executors per node, except from one node which hosted one executor plus the driver process. Each executor was allowed to use 2 processing cores and 28GB of memory. This setup complies with [22], which indicates that the performance gains are linear to the number of cores for a number of 24–32 cores. For greater numbers of cores, the benefits are decreased by a significant margin.

The machine learning library *MLlib* of Spark provides implementations of the most common supervised classification algorithms. Our approach is compared against these implementations and, more specifically, the three ones which support classification with multiple classes: *Logistic Regression*, *Decision Trees*, and *Random Forests*. The binary classifiers of MLlib (Gradient Boosted Trees and Multilayer Perceptron) could also be applied, but they require the application of the one-vs-all strategy, which in turn requires the repetition of the experiments multiple times (once per label). Since here we are dealing with massive data where the processing requires many expensive resources, we consider the binary classifiers as a non viable solution and we exclude them from our evaluation.

A. Dataset Characteristics and Initial Filtration

The experiments of this article were conducted by using the Open Academic Graph (OAG), a large-scale document collection which consists of roughly 167 million research articles from 19 major scientific areas[2]. These areas include Biology, Medicine, Geology, Chemistry, Psychology, Philosophy, Sociology, Art, Engineering, Economics, Computer Science, Physics, History, Political Science, Materials Science, Mathematics, Geography, Business, and Environmental Science.

The dataset is organized in 167 plain text files and each file contains the metadata of 1 million articles expressed in JSON format. The metadata provides a wealth of information about each paper including titles, abstracts, author names, keywords, venues, publishers, fields of study (FOS, or categories, or labels) and numerous others.

OAG includes more than 40 thousand FOS descriptors, however, there is no information on how these labels are connected, or why they sometimes appear together and some

[2]https://www.microsoft.com/en-us/research/project/academic/articles/microsoft-academic-increases-power-semantic-search-adding-fields-study/

Algorithm 5: Dataset filtration with SparkSQL (steps 1–3) and conversion to a collection of LIBSVM-formatted labeled points (steps 4–26)

```
1  RDD<Row> OAG ← readJSON(location);
2  RDD<Row> OAG1 ← filter(OAG, FOS ∈ Y);
3  RDD<Row> FilteredOAG ← filter(OAG1, lang="en");
4  RDD<String> ConvertedOAG ← FilteredOAG.map<Row>
5      for each row x ∈ FilteredOAG do
6          initialize a new string container line;
7          y_x ← extract label of x;
8          line.append(y_x);
9          K_x ← extract all keywords and title words from x;
10         for each keyword k ∈ K_x do
11             line.append(hash(k):w_k);
12         end
13         A_x ← extract all authors from x;
14         for each author a ∈ A_x do
15             line.append(hash(a):w_a);
16             for each author a' ∈ A_x after a do
17                 line.append(hash(a+a'):w_a);
18             end
19         end
20         j ← extract publishing journal of x;
21         line.append(hash(j):w_j);
22         return line;
23     end
24 end
25 ConvertedOAG.repartition(8);
26 ConvertedOAG.save(location);
```

times they do not. For instance, we have encountered the label *optical engineering* in combination with multiple different labels such as *Physics* and *Biology*, whereas it occasionally occurs isolated. Since this situation is repeated in many articles, we decided to avoid the ambiguity by preserving only the 19 aforementioned labels and by eliminating all others.

Notice that a significant portion of the articles of OAG are not labeled at all. Examples of such articles include those which are written in a non-English language. Of course, these samples cannot be used neither for training the model, nor for testing; consequently, they were removed from our experiments. Another set of articles is labeled, but none of these labels belong to the 19 used basic fields of study; these articles were also excluded from our experiments.

Spark includes a powerful filtration mechanism which refines a dataset by using SQL-like statements. The application of this mechanism, which is known as *SparkSQL*, is performed by each executor, after the distribution of the dataset by the driver program. In Algorithm 5 (steps 1–3) we show the filtration process of the OAG dataset with Spark. Initially, we read the original OAG files from the distributed storage. In the sequel, we apply the first filter with the aim of keeping only the samples whose FOS list includes at least one of 19 basic fields of study. Finally, in step 3 we apply the second filter and we discard all the non-English papers of the dataset.

The application of the first three steps of Algorithm 5 leads to a new filtered dataset (FilteredOAG) which consists of approximately 73.5 million research articles.

B. Dataset Preprocessing

After the initial filtration of the dataset, it is required that we apply two more transformations to it before it becomes suitable for processing. The first one dictates that we convert it to an appropriate form so that the adversary algorithms of MLlib can access it. In Subsection IV-B1 we describe the conversion of our dataset to a collection of labeled points. In this form, the dataset contains roughly 83.3 million features. Although our model is able to handle this huge feature space, the algorithms of MLlib cannot. Consequently, it is necessary to apply a second transformation to reduce the size of the feature space. More details about our dimensionality reduction methodology are presented in Subsection IV-B2.

1) Dataset Conversion: Most datasets are usually distributed to the public in structured or unstructured raw text format and OAG is not an exception; it has its data organized in a series of JSON-formatted records. However, the algorithms of MLlib accept their input in LIBSVM, a schema which dictates that each row represents a labeled point formatted according to the following form:

$$\text{Label id1:value1 id2:value2 ...} \tag{2}$$

where the label is represented by a number with double precision, followed by a specially formatted feature vector. The components of this vector are pairs of integer feature identifiers accompanied by their respective values. The feature identifiers must be one-based and in ascending order, whereas the feature values are the aforementioned weights for keywords, authors and journals. To facilitate the comparison of our algorithm with the MLlib implementations, we converted OAG to the LIBSVM format.

The common method for feature vectorization is to incrementally build a dictionary data structure (e.g., a hash table or a trie) during the record parsing, and use it to map features to indices. However, here we preferred to employ the *hashing trick* of [23], which applies a hash function directly to the features and uses the generated hash value to assign IDs to the features. This method is much faster than the dictionary-based approach, and it also saves the space occupied by the additional data structure. A side-effect of the hashing trick is the introduction of collisions, that is, different features obtain identical hash values. However, as stated in [24], the impact of these collisions to the statistical performance of a classifier is infinitesimal.

In our experiments, we employed the 32-bit version of the MurmurHash3 function[3], due to its high speed and its capability to uniformly distribute the input keys to the available target space (provided that the size of the target space is a power of 2). Since our filtered dataset includes more than 83.3 million features, we utilized a table of 2^{27} buckets, and we hashed the features into these buckets. MurmurHash3 is also the function which is used in some feature extraction implementations of Spark, such as TF-IDF[4].

[3]https://github.com/aappleby/smhasher/wiki/MurmurHash3

[4]https://spark.apache.org/docs/2.2.0/ml-features.html

Features	Algorithm	Accuracy
$83.3 \cdot 10^6$	Paper Classifier	79.1
	Logistic Regression	–
	Decision Trees	–
	Random Forests	–
4181	Paper Classifier	52.1
	Logistic Regression	46.9
	Decision Trees	18.6
	Random Forests	24.9

TABLE I
COMPARISON OF THE PAPERS CLASSIFICATION ACCURACY FOR VARIOUS ALGORITHMS

Algorithm/Method	Duration (min)
Paper Classifier ($83.3 \cdot 10^6$ features)	92
Paper Classifier	17
Logistic Regression	182
Decision Trees	239
Random Forests	263
Dataset preprocessing (Algorithm 5)	51
Singular Value Decomposition	–
Principal Component Analysis	–
Sparse Random Projection	44

TABLE II
COMPARISON OF THE ALGORITHMS EXECUTION TIMES (IN MINUTES)

The steps 4–26 of Algorithm 5 illustrate the parallel application of the hashing trick in our dataset. A *map* transformation on the FilteredOAG dataset generates for each research article, a string formatted according to eq. 2. Initially, we extract all keywords, title words, authors, co-authors pairs, and the journal of each research article by parsing the respective JSON entry. Each of these features is hashed by the MurmurHash3 function and the generated hash value accompanied by the corresponding feature weight are concatenated with the aforementioned string. Finally, the converted dataset is repartitioned to the 8 machines of our cluster, and it is written to the distributed file system.

2) Dimensionality Reduction: Now the dataset has been converted to a collection of labeled points formatted according to the LIBSVM schema, and includes more than 83.3 million features, hashed to a target space of 2^{27} buckets. Despite the high dimensionality of the feature space, our algorithm managed to complete the papers classification task normally. Nonetheless, it was impossible to execute any of the adversary MLlib implementations; in all cases, the execution failed by returning errors related to memory shortage problems. Consequently, it was necessary to apply a dimensionality reduction technique for comparison purposes.

Spark includes implementations of two such dimensionality reduction methods: Singular Value Decomposition (SVD) and Principal Component Analysis (PCA). Unfortunately, both of them failed to operate on our dataset. The former returned an error regarding the size of the matrix which stored the involved feature vectors, whereas the Spark implementation of the latter cannot operate on datasets which have more than 2^{16} features.

To overcome this problem, we implemented Sparse Random Projection (SRP), a dimensionality reduction method which is computationally simpler and faster compared to SVD and PCA. SRP projects the original feature space into a lower dimensional subspace through the origin, using a random matrix whose columns have unit lengths [25]. The application of SRP to the original dataset led to a lower dimensional space with only 4181 features. Notice here that the size of the new dimensional space was not set manually by us; it was determined by the application of the Johnson-Lindenstrauss lemma, which states that if the points in a vector space are projected onto a randomly selected subspace, then the distances between the points are approximately preserved [26].

C. Classification Accuracy

In this subsection we compare our algorithm to the three multi-class classifiers of MLlib in terms of classification accuracy. All methods were given the same training set to build their models and the same test set for the performance measurements. As mentioned earlier, these sets were randomly generated from the original dataset; their sizes were set equal to the 60% and 40% of the initial dataset, respectively.

Moreover, in all cases, the measurements were conducted by adopting the best-performing feature weights reported in [1], that is, $w_k = 0.3$, $w_a = 0.2$, and $w_k = 0.5$. Notice that the explicit keywords and the title words receive the same weight w_k; similarly, both authors and co-author pairs are assigned the same weight w_a and all journals the same weight w_k.

In Table I we report the accuracy measurements for our proposed Paper Classifier (PC) algorithm against the MLlib multi-class supervised classifiers. PC was the only method which achieved to classify the research articles of the dataset by operating on the original feature space. Its accuracy reached a remarkable 79.1%, that is, it classified correctly approximately 8 out of 10 papers. On the other hand, the MLlib implementations failed with errors regarding inadequate memory; the dash symbols in rows 3, 4 and 5 indicate the failure of the corresponding experiments.

Regarding the accuracy of the algorithms on the reduced feature space, we observe –as expected– a degradation of the performance of PC. Nevertheless, our method outperformed the MLlib algorithms by classifying correctly the 52.1% of the input entries. The best-performing adversary classification approach was Logistic Regression, whose accuracy was measured at 46.9%. On the other hand, the accuracy of the other two methods, Decision Trees and Random Forests, was surprisingly low; 18.6% and 24.9%, respectively.

D. Execution Times

Now let us compare the efficiency of our PC algorithm against the multi-class classification approaches of MLlib. In Table II we present the running times (in minutes) of the various algorithms which we executed (or attempted to execute) in our cluster. The dashes symbolize the failed experiments. The rows of Table II are organized in 3 groups. In the first group (rows 2–6) we present the examined classification methods, that is, PC and the adversary classification methods of MLlib.

The presented durations include the model training process and the classification of the records of the test set.

Regarding PC, we present two execution times. The first one (row 2) concerns the high dimensional space with the $83.3 \cdot 10^6$ features, and it was 92 minutes. The second one represents the running time of PC on the low dimensional space of the 4181 features, and it was approximately 5.4 times lower, at 17 minutes. In comparison to the fastest MLlib algorithm, that is, Logistic Regression, our two runs were roughly 2 and 10.7 times faster. The other two algorithms, Decision Trees and Random Forests were considerably slower; their execution times were 239 and 263 minutes respectively. Remarkably, PC was faster than the MLlib implementations even when it operated on a much larger dataset.

In row 7 we present the execution time of the dataset filtration and preprocessing (Algorithm 5). According to the Spark job profiler, the procedure was completed in 51 minutes. Finally, in the third group (last three rows) of Table II we include the three dimensionality reduction techniques. The dash symbol in the execution times of SVD and PCA symbolizes the failure of these two algorithms in reducing the dimensionality of our dataset. Regarding SRP, the algorithm consumed approximately 44 minutes to project the 83.3 million features into a lower dimensional space of 4181 features.

V. Conclusion

In this paper we presented an algorithm for classifying research articles in large-scale document collections by using the Apache Spark framework. The problem which was examined here is significant for both the academic search engines and the digital libraries, since a robust solution provides improved functionality and performance benefits. In contrast to the existing approaches, the method which we introduced here takes into consideration not only the keywords, but also the history of the authors, co-authorship information, and the areas of science published by each journal. The proposed algorithm was compared against the multi-class classifiers of the Spark machine learning library (MLlib), that is, Logistic Regression, Decision Trees and Random Forests, which dictated a careful application of a dimensionality reduction method. The experiments indicated that our approach achieved higher classification accuracy, accompanied by substantially better execution times. Moreover, it was the only method which handled efficiently the huge dimensionality of the dataset, without requiring an additional dimensionality reduction technique.

References

[1] L. Akritidis and P. Bozanis, "A Supervised Machine Learning Classification Algorithm for Research Articles," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC)*, 2013, pp. 115–120.

[2] J. Tang, J. Zhang, L. Yao, J. Li, L. Zhang, and Z. Su, "Arnetminer: Extraction and Mining of Academic Social Networks," in *Proceedings of the 14th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2008, pp. 990–998.

[3] A. Sinha, Z. Shen, Y. Song, H. Ma, D. Eide, B.-j. P. Hsu, and K. Wang, "An Overview of Microsoft Academic Service (MAS) and Applications," in *Proceedings of the 24th International Conference on World Wide Web (WWW)*, 2015, pp. 243–246.

[4] L. Getoor, "Link-Based Classification," in *Advanced Methods for Knowledge Discovery from Complex Data*, 2005, pp. 189–207.

[5] M. Taheriyan, "Subject Classification of Research Papers Based on Interrelationships Analysis," in *Proceedings of the 2011 Workshop on Knowledge Discovery, Modeling and Simulation*, 2011, pp. 39–44.

[6] H. Nanba, N. Kando, and M. Okumura, "Classification of Research Papers using Citation Links and Citation Types: Towards Automatic Review Article Generation." *Advances in Classification Research Online*, vol. 11, no. 1, pp. 117–134, 2011.

[7] X. Qi and B. D. Davison, "Web Page Classification: Features and Algorithms," *ACM Computing Surveys*, vol. 41, no. 2, p. 12, 2009.

[8] X. Pang, B. Wan, H. Li, and W. Lin, "MR-LDA: An Efficient Topic Model for Classification of Short Text in Big Social Data," *International Journal of Grid and High Performance Computing (IJGHPC)*, vol. 8, no. 4, pp. 100–113, 2016.

[9] F. Sebastiani, "Machine Learning in Automated Text Categorization," *ACM Computing Surveys*, vol. 34, no. 1, pp. 1–47, 2002.

[10] Y. Yang and J. O. Pedersen, "A Comparative Study on Feature Selection in Text Categorization," in *Proceedings of the 14th International Conference on Machine Learning (ICML)*, vol. 97, 1997, pp. 412–420.

[11] Y. Yang, "An Evaluation of Statistical Approaches to Text Categorization," *Information Retrieval*, vol. 1, no. 1-2, pp. 69–90, 1999.

[12] T. Joachims, "Text Categorization with Support Vector Machines: Learning with many Relevant Features," in *Proceedings of the 10th European Conference on Machine Learning (ECML)*, 1998, pp. 137–142.

[13] T. Hastie, S. Rosset, J. Zhu, and H. Zou, "Multi-Class Adaboost," *Statistics and its Interface*, vol. 2, no. 3, pp. 349–360.

[14] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, "Apache Spark: a Unified Engine for Big Data Processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[15] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[16] J. G. Shanahan and L. Dai, "Large Scale Distributed Data Science Using Apache Spark," in *Proceedings of the 21th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2015, pp. 2323–2324.

[17] K. Wang and M. M. H. Khan, "Performance Prediction for Apache Spark Platform," in *Proceedings of the 17th IEEE International Conference on High Performance Computing and Communications*, 2015, pp. 166–173.

[18] D. Chen, H. Chen, Z. Jiang, and Y. Zhao, "An Adaptive Memory Tuning Strategy with High Performance for Spark," *International Journal of Big Data Intelligence*, vol. 4, no. 4, pp. 276–286, 2017.

[19] S. Gopalani and R. Arora, "Comparing Apache Spark and Map Reduce with Performance Analysis using k-means," *International Journal of Computer Applications*, vol. 113, no. 1, 2015.

[20] R. Bosagh Zadeh, X. Meng, A. Ulanov, B. Yavuz, L. Pu, S. Venkataraman, E. Sparks, A. Staple, and M. Zaharia, "Matrix Computations and Optimization in Apache Spark," in *Proceedings of the 22nd ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2016, pp. 31–38.

[21] E. Carlini, P. Dazzi, A. Esposito, A. Lulli, and L. Ricci, "Balanced Graph Partitioning with Apache Spark," in *Proceedings the 2014 European Conference on Parallel Processing*, 2014, pp. 129–140.

[22] A. Gounaris, G. Kougka, R. Tous, C. T. Montes, and J. Torres, "Dynamic Configuration of Partitioning in Spark Applications," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 7, pp. 1891–1904, 2017.

[23] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature Hashing for Large Scale Multitask Learning," in *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*, 2009, pp. 1113–1120.

[24] G. Forman and E. Kirshenbaum, "Extremely Fast Text Feature Extraction for Classification and Indexing," in *Proceedings of the 17th ACM Conference on Information and Knowledge Management (CIKM)*, 2008, pp. 1221–1230.

[25] E. Bingham and H. Mannila, "Random Projection in Dimensionality Reduction: Applications to Image and Text Data," in *Proceedings of the 7th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2001, pp. 245–250.

[26] W. B. Johnson and J. Lindenstrauss, "Extensions of Lipschitz mappings into a Hilbert space," *Contemporary mathematics*, vol. 26, no. 1, pp. 189–206, 1984.