A Scalable Short-Text Clustering Algorithm Using Apache Spark

Leonidas Akritidis
School of Science and Technology
Int'l Hellenic University
Thessaloniki, Greece
Email: lakritidis@ihu.gr

Miltiadis Alamaniotis
Department of Electrical and Computer Engineering
University of Texas at San Antonio
San Antonio, USA
Email: Miltos.Alamaniotis@utsa.edu

Athanasios Fevgas
Department of Electrical and Computer Engineering
University of Thessaly
Volos, Greece
Email: fevgas@e-ce.uth.gr

Panayiotis Bozanis
School of Science and Technology
Int'l Hellenic University
Thessaloniki, Greece
Email: pbozanis@ihu.gr

***Abstract*—Short text clustering deals with the problem of grouping together semantically similar documents with small lengths. Nowadays, huge amounts of text data is being generated by numerous applications such as microblogs, messengers, and services that generate or aggregate entitled entities. This large volume of highly dimensional and sparse information may easily overwhelm the current serial approaches and render them inefficient, or even inapplicable. Although many traditional clustering algorithms have been successfully parallelized in the past, the parallelization of short text clustering algorithms is a rather overlooked problem. In this paper we introduce pVEPHC, a short text clustering method that can be executed in parallel in large computer clusters. The algorithm draws inspiration from VEPHC, a recent two-stage approach with decent performance in several diverse tasks. More specifically, in this work we employ the Apache Spark framework to design parallel implementations of both stages of VEPHC. During the first stage, pVEPHC generates an initial clustering by identifying and modelling common low-dimensional vector representations of the original documents. In the sequel, the initial clustering is improved in the second stage by applying cluster split and merge operations in a hierarchical fashion. We have attested our implementation on an experimental Spark cluster and we report an almost linear improvement in the execution times of the algorithm.**

***Index Terms*—short text clustering, clustering, machine learning, big data, parallel algorithms, Spark**

I. Introduction

The problem of short text clustering concerns the unsupervised grouping of semantically similar documents into distinct, non-overlapping clusters. It is an emerging field of research with an enormous number of important applications. Indicative examples include the clustering of entitled entities (e.g. news articles, scientific articles, products, etc.) for retrieval and aggregation purposes, the discovery of knowledge from microblogs, the sentimental analysis of user comments and/or reviews in online communities, and numerous others.

In the majority of the aforementioned cases, the volume of data that must be processed is huge and increases, or changes at high rates. In such challenging conditions, a serial algorithm running on a single workstation can be quickly overwhelmed and become unable to handle the input data efficiently. For this reason, the development of distributed machine learning algorithms that can be executed in parallel by multiple processing units is a strong and straightforward solution. Here the term "processing unit" is rather abstract and it may concern a standard CPU process, a CPU/GPU thread, a virtual or a physical workstation, etc.

In the past years, there has been a significant amount of research towards the introduction of parallel data clustering algorithms. These efforts primarily focused on the traditional algorithms such as k-Means [1], [2], hierarchical clustering [3], [4], [5], spectral clustering [6], [7], DBSCAN [8], [9] and so on. Conforming to the present requirements for efficiently processing large volumes of text data, some recent works introduced state-of-the-art parallel algorithms for clustering massive document collections, [10], [11], [12].

In this paper we present a parallel short text clustering algorithm which originates from VEPHC (the name derives from Vector Projection - Hierarchical Clustering), a recently introduced method for grouping similar short documents [13], [14]. The new algorithm, called *parallel VEPHC* (or simply *pVEPHC*), was designed to render VEPHC capable of efficiently processing large document collections by employing the Spark framework [15]. Spark was chosen over the well-established Hadoop MapReduce [16] due to its ability to handle massive amounts of data in main memory, its flexibility in programming multiple successive operations in a single job, its state-of-the-art job scheduler that organizes the execution plan via a directed acyclic graph, and several other features that render it faster and more attractive than its predecessor.

The original VEPHC algorithm includes two phases. The first one processes the dataset, and for each input document, it constructs a representative vectorial representation called the *Dominant Reference Vector (DRV)* [14]. This is performed by forming and scoring all possible word combinations (called *Projection Vectors (PVs)*) of the document, under the constraint of a hyper-parameter k that determines their maximum length. Then, the PV with the highest score becomes the DRV of that document and its dimensional space is subsequently utilized to form a new space where the document is projected. At the end of the first phase, all the documents that have been projected onto the same dimensional space are grouped together into the same cluster. It is noticeable that this design allows VEPHC to automatically estimate the number of clusters without requiring prior knowledge of this number.

The parallelization of this procedure seems straightforward. However, the scoring function of the PVs requires several diverse pieces of information (e.g. word inverse document frequencies, the PV frequency in the corpus, and others), therefore, a carefully designed execution plan is required. This plan includes i) the vectorization of the documents by using a previously constructed word dictionary, ii) the construction and the partial scoring of PVs, iii) the final scoring of PVs (by integrating the PV frequency into the scoring function) and the declaration of the DRV, and iv) the projection of the documents onto the space that is determined by that DRV.

The second phase continues from the point where the first phase stopped. Its role is to improve the original clusterings by applying a hierarchical procedure where the clusters are merged together (if they are similar enough), or new clusters may be formed. The parallelization of hierarchical clustering is achieved by solving the Minimal Spanning Tree problem on a fully connected graph of the involved data points [5].

The rest of the paper is organized as follows: Section II contains an overview of the relevant literature on parallel clustering algorithms. Section III provides a brief description of the two stages of the serial VEPHC algorithm, whereas the two subsections of Section IV describe the parallelization strategy for each of these stages. The contributions of this paper were experimentally attested on a Spark cluster; the results are analyzed in Section V. Finally, Section VI summarizes the conclusions of this work and provides some future insights.

II. Related Work

Clustering is one of the most well-studied problems in the data mining and machine learning disciplines. The literature contains a variety of solutions based on different strategies.

The matrix decomposition and factorization methods constitute one of the most popular techniques in the area [17], [18]. In particular, Non-negative Matrix Factorization (NMF) is a generic mathematical method that applies well to the problem of text clustering [19]. In the context of short text, NMF was enriched with additional features (e.g. term co-occurrence [20], and word weighting [21]) to confront the native problem of data sparseness. Furthermore, DNMF is another notable extension of NMF that takes into consideration the geometric form of both the underlying data and feature manifolds [22].

Another category of text clustering methods is based on topic modelling approaches such as Latent Dirichlet Allocation (LDA) [23], [24]. Among the numerous models of this family, GSDMM constitutes a state-of-the-art method that iteratively performs Gibbs sampling on data with Dirichlet Multinomial Mixture distribution [25]. On the other hand, BTM (BiTerm Model) achieves clustering by identifying and modelling the word co-occurrence patterns that exist in the corpus [26].

This article introduces a parallel version of VEPHC, a short-text clustering algorithm that was introduced in [13] and further extended in [14]. VEPHC draws its origins from an unsupervised entity matching method that focused on product entities in online comparison platforms [27], [28]. It is a two-stage algorithm that initially generates clusters by identifying common dominant lower-dimensional representations of the documents. In the sequel, the clustering is improved firstly by removing the most dissimilar items from the clusters, and then, by merging the derived clusters in an hierarchical fashion.

The recent information explosion on the Web, the growth of popular online communication networks and the emergence of novel applications rendered the parallelization of the text clustering methods particularly important. For this reason, a large number of researchers focused on this problem and devised solutions for the traditional algorithms, based on the well-established MapReduce and Spark frameworks. In addition, another portion of relevant works addressed the problem by studying techniques based on the modern parallelization hardware such as shared/distributed memory CPUs, GPUs, multi-threaded processes, and so on.

In this context, [2] presented a GPU accelerated k-Means method, whereas [1] and [29] introduced variants of k-Means for the MapReduce framework. Furthermore, [30] and [31] provided parallel implementations of the algorithm for MPI and shared memory multiprocessors respectively. Regarding hierarchical clustering, a parallel algorithm for distributed memory multiprocessor architectures was studied in [4]. Also, in [5] the authors proposed an interesting Spark algorithm that formulates the problem as a minimal spanning tree problem; we adopted this approach during the agglomeration of the clusters in the last part of pVEPHC. A survey on the most important previous works on parallel hierarchical clustering is provided in [3]. The literature also contains parallel variants of spectral clustering [6], [7], DBSCAN [8], [9], affinity propagation [32], OPTICS [33], mean shift [34], and others.

Although the parallelization of the traditional algorithms is a well-studied problem, this does not apply to the short text clustering algorithms. In particular, [35] studied a Hadoop-based k-Means variant for short documents. In [10] the authors introduced another parallel k-Means algorithm based on neighbors and devised a parallel pair-generating technique to construct the neighbor matrix. Moreover, [36] adapted the Jarvis-Patrick algorithm in MapReduce and conducted experiments which demonstrated the efficiency of their work. Finally, [12] proposed a Spark parallel method with the aim of addressing the problem of high dimensionality through the implementation of a novel document hashing strategy.

III. Preliminary Elements

Here we briefly describe the two stages of VEPHC and we introduce all the necessary background elements that will be employed during the design of the parallel algorithm.

Given a document vector $\mathbf{x}$ and an integer hyperparameter $k \geq 2$, a *projection vector (PV)* $\mathbf{p_x}$ of $\mathbf{x}$ is defined as a vector with components that derive from an arbitrary combination of *at most* k components of $\mathbf{x}$. Moreover, we introduce the set $P_{\mathbf{x},k}$ that contains all the PVs of $\mathbf{x}$ with respect to k. Hence, for a document vector $\mathbf{x} = (\textit{black, cats, white, dogs})$ and a setting $k = 3$, the set $P_{\mathbf{x},k}$ will include the following PVs: (*black, cats*), (*black, white*), (*black, dogs*), (*cats, white*), (*cats, dogs*),

(*white, dogs*), (*black, cats, white*), (*black, cats, dogs*), (*black, white, dogs*), and (*cats, white, dogs*).

According to [14], each PV $\mathbf{p_x} \in P_{\mathbf{x},k}$ is assigned a score that is computed by applying the following function:

$$S_{\mathbf{p_x}} = \frac{1}{l_{\mathbf{p_x}}} \log(f_{\mathbf{p_x}} + 1) \sum_{\mathbf{p_x}} p_{x,i}^k. \quad (1)$$

Equation 1 incorporates the length of the projection vector $l_{\mathbf{p_x}}$ and its frequency in the collection $f_{\mathbf{p_x}}$. In addition, it sums up the coefficients of its components $p_{x,i}$ raised in the kth power. The simplest approach for specifying the values of $p_{x,i}$ is to adopt the well-known $tf - idf$ model:

$$p_{x,i} = tf_i \cdot \text{IDF}(w_i), \quad (2)$$

where tf_i represents the number of the occurrences of the ith word of $\mathbf{x}$ in $\mathbf{x}$, and $\text{IDF}(w_i)$ is its inverse document frequency:

$$\text{IDF}(w_i) = \log(n/tf_i), \quad (3)$$

Apart from $tf-idf$, [14] introduced $tp-idf$, a technique that additionally takes into consideration the position(s) $r_{i,j}$ of a word w_i in a document $\mathbf{x}$:

$$p_{x,i} = \text{IDF}(w_i) \sum_{\forall w_i \in \mathbf{x}} \frac{1}{\alpha + \log(r_{i,j} + 1)} \quad (4)$$

where $\alpha \in [0, 1]$ is a decimal hyperparameter that regulates the importance of the position(s) of a word in the final score. Observe that the summation term in Eq. 4 rewards the potential multiple occurrences of a word w_i in a document $\mathbf{x}$.

After the scoring of each PV, the algorithm selects the one that achieved the highest score and declares it as the *Dominant Reference Vector (DRV)* $\mathbf{p}_\mathbf{x}^*$ of $\mathbf{x}$. In the sequel, the unit components of $\mathbf{p}_\mathbf{x}^*$ are used to set up a lower dimensional space where $\mathbf{x}$ is eventually projected. The first stage of VEPHC concludes by grouping together all the documents that have been projected onto the same (lower) dimensional space.

The second stage introduces a post-processing procedure with the aim of enhancing the clustering quality of the previous phase. Initially, the algorithm traverses the existing cluster universe U and for each cluster $C \in U$ it computes its clustroid element π_C (namely, the element that exhibits the highest similarity with the rest of the elements of C). Simultaneously, all the elements of C that are not similar enough to π_C are removed from C and they form an equal number of new clusters. These new clusters are appended to U and in the sequel, an agglomerative procedure is executed iteratively in order to merge the most similar clusters. The process terminates when no clusters can be further merged, with respect to a pre-defined similarity threshold T.

IV. Parallel Short Text Clustering

This section presents in details the key elements of the proposed parallel Spark algorithm. It is organized in two subsections, where the two stages of pVEPHC are analyzed and the design choices are explained.

Spark supports three types of data structures to safely distribute data across a cluster of interconnected nodes (called executors): i) Resilient Distributed Datasets (RDDs), ii) Dataframes, and iii) Datasets[1]. Without loss of generality, throughout this presentation we use the term RDD for two reasons: i) to avoid the confusion that may be caused by the usage of "Dataset" (since in the majority of cases the term "dataset" is used to describe the original input data), and ii) RDDs can be very easily transformed to either Dataframes or Datasets, and vice versa. Consequently, the presented logic can be directly applied to these data structures too.

A. Stage 1: Initial Cluster Generation

The first stage of the algorithm focuses on the parallel construction of suitable, low-dimensional representations of the input documents. As mentioned earlier, the logic behind this approach is to group together all the documents that share such common vector representations.

1) Document vectorization and initial word scoring: This process requires the existence of a dictionary that contains all the unique words in the collection. For each entry, this dictionary must also store:

- a unique integer identifier (WordID) that will be used to convert the documents into vectors. We choose the document vectors to be dense (with no zero elements), to limit the effects of the well-known Curse of Dimensionality. Also, the usage of numerical vectors will accelerate the computation of similarity scores, since numerical comparisons are much faster than string comparisons.
- its inverse document frequency (IDF). This value is essential for computing the PV scores according to either the $tf - idf$ (Eq. 2), or the $tp - idf$ (Eq. 4) model.

Algorithm 1 illustrates the basic steps of the dictionary construction process. The utilized notation adopts the established Python syntax. More specifically, a set of brackets $[a, b, \ldots]$ denotes a list, whereas a set of parentheses $(a, b, \ldots)$ represents a tuple. For the reader convenience, we also use Table I to report the form of an RDD after a transformation or an action. In this table, the dagger symbol (†) denotes a key that is unique in the RDD, so duplicate keys are not possible.

Initially, the input files are read from HDFS. The dataset is stored in an RDD called rdd_dataset and distributed in the executors of the cluster (line 1). The number of entries in this RDD provides the number of documents n in the corpus; n will be employed later to compute the inverse document frequencies of the words (line 2).

In line 3 we apply a sequence of transformations and actions on rdd_dataset. Hence, the flatMap operation of line 4 invokes a tokenize() function to create a bag of words for each input document. Each generated word w is subsequently passed through a set of filters that apply casefolding, punctuation removal, stopword removal, etc. Eventually, flatMap emits a list of $(w, 1)$ tuples for each document. In the sequel, the reduceByKey action in line 5 sums up all the 1 values associated with the same word, and thus, it computes its frequency tf_w in the corpus. The map transformation that

[1]Datasets are not supported by PySpark.

Algorithm 1: Word dictionary construction

```
1  rdd_dataset ← read from HDFS;
2  n ← |rdd_dataset|;
3  rdd_words ← rdd_dataset
4        .flatMap(d → tokenize(d))
5        .reduceByKey(x, y → x + y)
6        .map(x → (x[0], log(n/x[1])));
7  rdd_dic ← rdd_words
8        .zipWithIndex()
9        .map(x → (x[0][0], (x[1], x[0][1])));
10 dictionary ←rdd_dic.collectAsMap();
11 D ← broadcast(dictionary);
12  -OR-
13 dictionary_file ← write(rdd_dic);
```

TABLE I
INPUT AND OUTPUT RDDS FOR ALGORITHM 1

Line	Operation	Output RDD
1	read	rdd_dataset=[$(docID^{\dagger}, Text), \ldots$]
4	flatMap	[$(w, 1), \ldots$]
5	reduceByKey	[$(w^{\dagger}, tf_w), \ldots$]
6	map	rdd_words=[$(w^{\dagger}, \text{IDF}(w)), \ldots$]
8	zipWithIndex	[$((w^{\dagger}, \text{IDF}(w)), wID), \ldots$]
9	map	rdd_dic=[$(w^{\dagger}, (wID, \text{IDF}(w)), \ldots$]

follows, simply computes the IDF(w) of each word according to Eq. 3. The produced data structure is a list of $(w, \text{IDF}(w))$ tuples stored in a new RDD called rdd_words.

After the IDFs of the words have been calculated, we focus on the assignment of a unique integer identifier to each word. Line 8 calls the native zipWithIndex() Spark operation to assign a monotonically increasing integer to each row in rdd_words. In the next line, the RDD is rearranged by performing another map transformation so that the word alone becomes the key of the tuples contained within.

The generated rdd_dic contains all the necessary information: each word in the collection is accompanied by a unique integer identifier and its respective IDF. Nevertheless, before we proceed with document vectorization and PVs construction, we must make a critical choice. Our dictionary is stored in a distributed data structure; in case the input dataset is large, the aforementioned operations will perform millions of look-ups against rdd_dic. However, the submission of a huge number of requests against a data structure that is distributed across hundreds, or even thousands of executors will significantly degrade the performance. This problem dictates that all executors possess a local copy of the *entire* word dictionary.

We now distinguish the two following cases:

- The dictionary is relatively small (e.g. a few million words) and it fits in the executor memory. In this case, we can collect the contents of rdd_dic in the Driver program and build an in-memory data structure from these contents. Then, the Driver can broadcast this data structure D to all the executors (lines 10–11).

Algorithm 2: Projection vector construction

```
1  function vectorize(x)
2      docID ← x[0];
3      text ← x[1];
4      W ← tokenize text;
5      dataList ← [];
6      for each w ∈ W do
7          w ← liguistic processing (case folding, etc.);
8          (wID, IDF(x)) ← D.search(w);
9          dataList ← dataList + (wID, IDF(x));
10     end
11     return (docID, dataList);
12 end
13 rdd_vec ← rdd_dataset
14       .map(x ← vectorize(x));
15 function construct_pv(x)
16     docID ← x[0];
17     V_x ← x[1];
18     pos ← 0;
19     tempList ← [];
20     for each tuple (wID, IDF(w)) ∈ V_x do
21         pos ← pos + 1;
22         p_w ← Eq. 4;
23         tempList ← tempList + (wID, p_w)
24     end
25     P ← create combinations of all wIDs of length k
         by using tempList;
26     dataList ← [];
27     for each combination (PV) p ∈ P do
28         S'_p ← Eq. 7;
29         dataList ← dataList + (p, (docID, S'_p));
30     end
31     return dataList;
32 end
33 rdd_pv ← rdd_vec
34       .flatMap(x ← construct_pv(x));
```

TABLE II
INPUT AND OUTPUT RDDS FOR ALGORITHM 2 ($V_{\mathbf{x}}$ IS GIVEN BY EQ. 5)

Line	Operation	Output RDD
13	map	rdd_vec=[$(docID^{\dagger}, V_{\mathbf{x}}), ..$]
33	flatMap	rdd_pv=[$(\mathbf{p_x}, (docID, S'_{\mathbf{p_x}})), ...$]

- In the opposite case, the dictionary is prohibitively large to fit in the executor memory. Consequently, we must persist the contents of rdd_dic in HDFS (line 13). In this way, each executor can copy the dictionary file/s on its local filesystem and perform disk-based look-ups.

2) Projection vectors construction and partial scoring: Algorithm 2 displays the basic steps of the construction of the projection vectors. Similarly to the previous discussion, Table II presents the form of the output RDDs after the execution of either a transformation, or an action.

In line 13 we apply a map transformation to our starting RDD, i.e. rdd_dataset (see row 2 of Table I). The invoked vectorize() function employs the previously broadcasted dictionary D to perform one look-up per word w, per document of rdd_dataset. In this way, the identifier wID and the inverse document frequency IDF(w) of the word w are retrieved and a tuple $(wID, \text{IDF}(w))$ is formed. All the created tuples for a document $\mathbf{x}$ are stored in a list:

$$V_{\mathbf{x}} = [(wID, \text{IDF}(w)), ...], \tag{5}$$

and then, $V_{\mathbf{x}}$ is associated with the $docID$. The form of the created RDD, rdd_vec, is shown in the second row of Table II.

The second part of Algorithm 2 concerns the construction and scoring of the required projection vectors (PVs). To achieve this goal, a flatMap transformation accompanied by a call to the construct_pv function (lines 15–32) is performed on rdd_vec. For each row of the input RDD, we traverse the attached list $V_{\mathbf{x}}$ and we create a new temporary list of (wID, p_w) tuples, where p_w is the $tp - idf$ score of Eq. 4.

In the sequel, the aforementioned temporary list is used to create all possible word combinations with lengths that do not exceed k (line 25). At this point, we possess all the required parameters to compute the score of each projection vector (according to Eq. 1), apart from its frequency in the collection. For this reason, we split Eq. 1 into two parts, as follows:

$$S_{\mathbf{p_x}} = S'_{\mathbf{p_x}} \log(f_{\mathbf{p_x}} + 1), \tag{6}$$

where $S'_{\mathbf{p_x}}$ is the partial score of the projection vector, that is:

$$S'_{\mathbf{p_x}} = \frac{1}{l_{\mathbf{p_x}}} \sum_{\mathbf{p_x}} p^k_{x,i}. \tag{7}$$

The partial score $S'_{\mathbf{p_x}}$ can be directly computed inside the present flatMap operation, since the parameters it involves (i.e. the length $l_{\mathbf{p_x}}$ and the $tp - idf$ score) are already known. The full score of the projection vector will be calculated later, when the frequency $f_{\mathbf{p_x}}$ becomes available. The details of the partial score computation are shown in lines 27–30.

3) Initial clustering: The last part of Stage 1 includes the full scoring of PVs, the identification of the DRV (i.e., the PV with the highest score) for each document, and the grouping of the documents with common DRVs into the same cluster.

Algorithm 3 presents the steps of the aforementioned procedure. Initially, a combineByKey action on rdd_pv gathers all the values (i.e. the $(docID, S'_{\mathbf{p_x}})$ tuples) that are associated with equal keys (i.e. the PV $\mathbf{p_x}$) to the same executor. Hence, the number of these values represents the frequency $f_{\mathbf{p_x}}$ that we were missing. Consequently, we are now able to compute the scores of all projection vectors by applying Eq. 6.

Let us discuss in details this action. At first, combineByKey sends all the values that are associated with the same key to the same executor. In contrast to reduceByKey, this one allows the output RDD to receive a completely different form than the input RDD. Second, combineByKey accepts three methods:

- make_lists: it converts all the values of an RDD to lists. Here we multiply the partial scores with $\log 2$ (lines 1–3).

Algorithm 3: PV scoring and initial clustering

```
1  function make_lists(A)
2      return [(A[0], A[1] · log 2)];
3  end
4  function append(A, B)
5      return A.append(B);
6  end
7  function extend(A, B)
8      A.extend(B);
9      f_p ← 0;
10     for each tuple (docID, S'_px) in A do
11         f_p ← f_p + 1;
12     end
13     dataList ← [];
14     for each tuple (docID, S'_px) in A do
15         S_px ← f_p · S'_px / log 2;
16         dataList ← dataList + (docID, S_px);
17     end
18     return dataList;
19 end
20 function reorder_byDocID(x)
21     p ← x[0];
22     dlist ← x[1];
23     dataList ← [];
24     for each tuple (docID, S_px) in dlist do
25         dataList ← dataList + (docID, (S_px, p));
26     end
27     return dataList;
28 end
29 rdd_drv ← rdd_pv
30     .combineByKey(make_lists, append, extend)
31     .flatMap(x ← reorder_byDocID(x))
32     .reduceByKey(max)
33     .map(x ← (x[0], x[1][1]))
34 rdd_clusters ← rdd_drv.map(x ← (x[1], x[0]))
```

TABLE III
INPUT AND OUTPUT RDDS FOR ALGORITHM 3

Line	Operation	Output RDD
30	combineByKey	$[(\mathbf{p}^{\dagger}_{\mathbf{x}}, (docID, S_{\mathbf{p_x}})), \ldots]$
31	flatMap	$[(docID, (S_{\mathbf{p_x}}, \mathbf{p_x})), \ldots]$
32	reduceByKey	$[(docID, (S_{\mathbf{p_x}}, \mathbf{p}^*_{\mathbf{x}})), \ldots]$
33	map	rdd_drv=$[(docID, \mathbf{p}^*_{\mathbf{x}})), \ldots]$
34	map	rdd_clusters=$[(\mathbf{p}^*_{\mathbf{x}}, docID)), \ldots]$

In this way, we calculate correctly the final scores of the PVs that appear only once in the collection. Regarding the ones with more occurrences, this is erroneous, since Eq. 6 dictates that we should multiply with $\log(f_{\mathbf{p_x}} + 1)$. We will correct this error in the extend function.

- append: a merging function that appends a value into the previously collected values.
- extend: it combines the merged values together. Notice how the scores of PVs are computed in line 15; we divide with $\log 2$ to correct the previous deliberate error.

The flatMap transformation that follows combineByKey just rearranges the tuples, so that the score of the PVs is placed first. This is useful for the next action, reduceByKey(max), that among all the entries with the same key, keeps the one the highest value and discards the rest of them. In other words, this reduceByKey(max) action directly identifies the Dominant Reference Vector of a document. Now, the DRV score is no longer needed, so we remove it via a new map call obtaining a new RDD, called rdd_drv.

The components of the DRV are used to form a new low-dimensional space where the document is projected. All the documents that lie on the same dimensional space (i.e. they share common DRVs) are grouped together into the same cluster. This operation is performed in the last map operation, where each DRV (i.e. cluster) is associated with a document.

B. Stage 2: Improvement of the Initial Clustering

The second stage of pVEPHC consists of two phases. In phase 1, the most dissimilar documents of each cluster are removed and each one of them is placed into a new singleton cluster. In phase 2, an agglomerative algorithm merges the highly similar clusters until no clusters can be merged further.

1) Removal of the Dissimilar Elements: This process requires a method for computing the similarity of an element with a cluster. According to [13], [14], this is achieved by computing the similarity of an element with the clustroid π_C of a cluster C. Recall that π_C is a member of C that has the highest similarity with the rest of the items of C. An element $\mathbf{x}$ of C is called *dissimilar* to C, if its cosine similarity with π_C is smaller than a pre-defined similarity threshold T, i.e:

$$\cos(\mathbf{x}, \pi_C) < T \tag{8}$$

Such elements are removed from their respective clusters and they are placed into new, singleton clusters (that is, clusters with only one element). Unfortunately, rdd_clusters is not in the appropriate form to support this operation because: i) it does not contain the document vectors, so the computation of similarities is not possible, and ii) it includes associations of a cluster with a single document; however, what we require here is to associate each cluster with the list of its contained documents, so that we can determine the clustroid elements.

For this reason, we perform a series of transformations and actions, as shown in lines 1–5 of Algorithm 4. At first, the union of line 2 merges rdd_drv and rdd_vec. Consequently, the output RDD contains for each document two tuples of different forms, as shown in second row of Table IV. These two different tuples are subsequently merged via a reduceByKey action (line 3), leading into a new RDD that includes unique docIDs, accompanied by a tuple of their cluster and their vectorial representation $V_{\mathbf{x}}$. The purpose of the following map and combineByKey operations is to rearrange the generated RDD so that their key is the RDV of the cluster, followed by the list of documents that fall into that cluster.

The second part of Algorithm 4 (lines 6–22) includes a flatMap transformation (line 22) that for each input cluster may generate multiple clusters: i) the input cluster itself excluding the removed documents, and ii) one singleton cluster for each one of these removed documents. More specifically, the invoked findDissimilarElements() function performs two operations. First, it computes the clustroid element π_C of a cluster C (line 8), and ii) it removes all the elements of C whose cosine similarity is smaller than the value of a threshold T (lines 11–18). For these elements, one singleton cluster is created and appended in the $ClusterList$ object (line 15). In this object we also store the input cluster C, after the removal of the dissimilar elements (line 17). Notice that, at this point, we no longer need the DRV to be present, so we do not so we do not include it in the cluster RDD.

All that is left now is to merge the most similar clusters. This is a challenging task, since the agglomerative clustering algorithm exhibits inherent data dependency. Specifically, it requires pairwise comparisons among all the clusters in the RDD, a procedure that does not scale well in a distributed environment. Nevertheless, [5] confronted the problem by formulating it as a Minimum Spanning Tree problem with satisfactory results. Consequently, in this part we adopted this promising solution to obtain the final output of the algorithm.

Algorithm 4: Removal of dissimilar elements

```
1  rdd ← rdd_drv
2      .union(rdd_vec)
3      .reduceByKey(x, y ← (x, y))
4      .map( x ← (x[1][0], (x[0], x[1][1])))
5      .combineByKey(make_lists_2, append, extend_2);
6  function findDissimilarElements(x)
7      docList ← x[1];
8      π_C ← ComputeClustroid(docList);
9      cDocs ← [];
10     dataList ← [];
11     for each (docID, V_x) tuple in docList do
12         if cos(π_C, V_x) > T then
13             cDocs ← cDocs + (docID, V_x);
14         else
15             dataList ←
                 dataList + (V_x, [(docID, V_x)]);
16         end
17         dataList ← dataList + (π_C, cDocs);
18     end
19     return dataList;
20 end
21 rdd ← rdd
22     .flatMap(x ← findDissimilarElements(x));
```

TABLE IV
INPUT AND OUTPUT RDDS FOR ALGORITHM 4($V_{\mathbf{x}}$ IS GIVEN BY EQ. 5)

Line	Operation	Output RDD
2	union	$[(docID, \mathbf{p}^*_{\mathbf{x}}), (docID, V_{\mathbf{x}}), ...]$
3	reduceByKey	$[(docID^{\dagger}, (\mathbf{p}^*_{\mathbf{x}}, V_{\mathbf{x}}), ...]]$
4	map	$[(\mathbf{p}^*_{\mathbf{x}}, (docID, V_{\mathbf{x}})), \ldots]$
5	combineByKey	rdd=$[(\mathbf{p}^*_{\mathbf{x}}, [(docID, V_{\mathbf{x}}), ...]), ...]$
22	flatMap	rdd=$[(\pi_{\mathbf{p}^*_{\mathbf{x}}}, [(docID, V_{\mathbf{x}}), ...]), ...]$

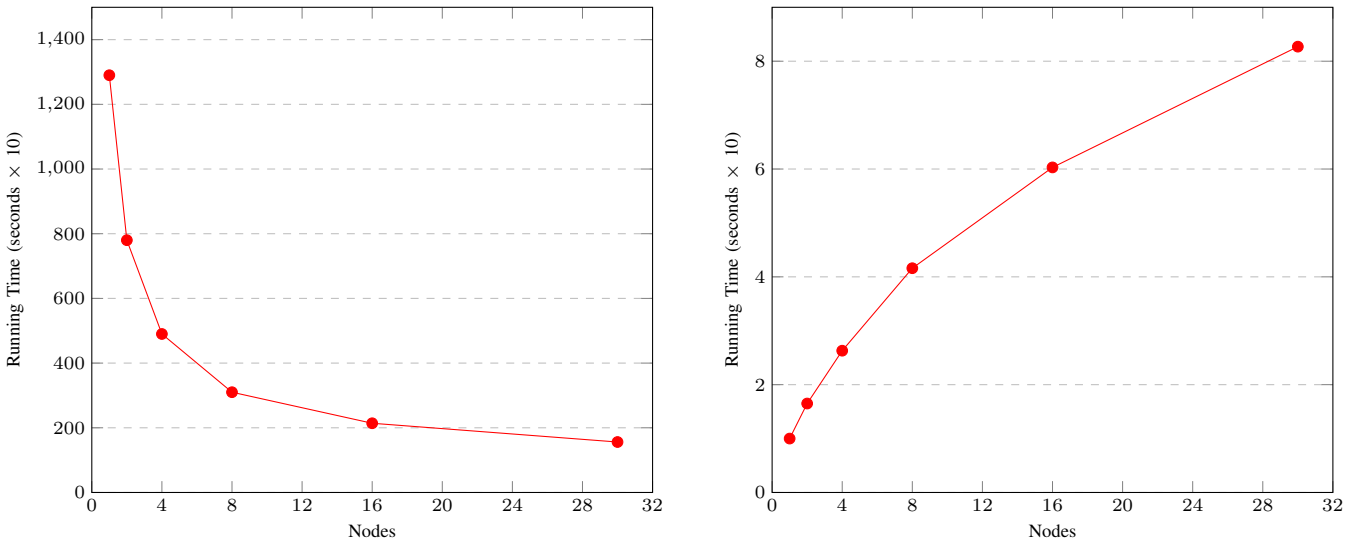

Fig. 1. Running times of pVEPHC for various cluster sizes (left), and acceleration factor vs. number of processing nodes (right).

V. Experiments

This section presents the experimental evaluation of pVEPHC. The parallel algorithm was implemented with Java 1.8 and was deployed on a cluster comprised on 8 virtual machines (VMs). Each VM was running Ubuntu 18.04 LTS and was equipped with 16 CPUs and 64 GB of RAM. The software that was installed on the cluster included Hadoop 3.2.2 and Spark 3.1.2 (the latest stable release). In addition, Hadoop was accompanied by the YARN resource manager and the Hadoop Distributed File System (HDFS).

Regarding the cluster configuration settings, each processing node was set up to launch two YARN containers and each one of these containers was configured to deploy a single Spark executor. This is translated into a total of 2 executors per VM. Furthermore, each executor was assigned 2 vCores and 24GB of RAM. In total, the entire Spark cluster included one driver process and 15 executors; or equivalently, an amount of 30 vCores and 360 GB of memory was allocated and evenly distributed to 15 executors.

The dataset we used was FakeNewsOnlyTitles[2], a large collection of news articles that were manually classified by special tags such as “reliable”, “unreliable”, “clickbait” and others. FakeNewsOnlyTitles consists of approximately 400 thousand titles, however, it requires some cleansing procedures before it becomes suitable for use.

The hyper-parameters of pVEPHC were tuned as follows: The maximum lengths of the projection vectors were limited to $k = 5$ components, the parameter that regulates the importance of word positions was set equal to $\alpha = 0.5$, whereas the similarity threshold in phase 2 was $T = 0.7$.

In the left diagram of Figure 1 we present the running times of pVEPHC for different cluster sizes (that is, number of processing nodes). In general, we observe a significant acceleration of the algorithm for moderate number of nodes. More specifically, the execution of the algorithm consumed roughly 1290 seconds on a single-node cluster. The insertion of one additional vCore led to a remarkable speedup of 65%; the duration of the execution was 780 seconds. Next, we doubled further the size of the cluster so that it included 4 vCores; this resulted in a drop of the execution time to 480 seconds. Finally, the durations for 8, 16, and 30 vCores were 310, 214, and 156 seconds respectively. The curve in the left diagram reveals that for this dataset, the performance benefits are starting to shrink as the number of processing nodes becomes greater than 16.

Furthermore, the right diagram of Figure 1 depicts the growth of the acceleration factor as the size of the cluster increases. The results reveal a satisfactory behaviour of pVEPHC, since its performance is proportional to the number of the processing nodes that participate in the execution. Hence, when the size of the cluster doubles, the speedup that we achieve ranges between 45% (for a number of vCores greater than 16) and 65% (for smaller clusters). This result reveals the scalability of pVEPHC and demonstrates the robustness of the proposed solution.

VI. Conclusions

In this paper we presented pVEPHC, a parallel short text clustering algorithm implemented on the Apache Spark framework. The proposed method parallelizes a recently published sequential variant, called VEPHC.

Similarly to its predecessor, pVEPHC includes two stages: the first stage processes the input data and for each document, it determines a low dimensional vector representation that reflects the content as accurately as possible. In the sequel, it projects the document in the dimensional space of this representative vector and in the last phase, it groups together all the documents that lie onto the same dimensional space. The parallelization of this stage was carefully orchestrated to

[2]https://www.kaggle.com/luizfkcunhautfpr/fakenewsonlytitles

allow the robust scoring scheme of VEPHC. More specifically, the full score of each projection vector is performed in multiple phases, and only when a new required parameter is computed. This maximizes the efficiency of the algorithm and speeds up its execution.

In the second stage, the clustering of the first stage is improved further. At first, we remove all the dissimilar documents from their clusters and we generate a new singleton cluster for each evicted element. Then, we apply an agglomerative clustering method with the aim of merging the most similar clusters. Since hierarchical clustering is challenging to parallelize due to its inherent data dependence, in this stage we adopted a state-of-the-art strategy that treats the problem as a typical Minimal Spanning Tree problem.

REFERENCES

[1] W. Zhao, H. Ma, and Q. He, "Parallel K-Means clustering based on MapReduce," in *Proceedings of the IEEE International Conference on Cloud Computing*, 2009, pp. 674–679.

[2] S. Cuomo, V. De Angelis, G. Farina, L. Marcellino, and G. Toraldo, "A GPU-accelerated parallel K-means algorithm," *Computers & Electrical Engineering*, vol. 75, pp. 262–274, 2019.

[3] C. F. Olson, "Parallel algorithms for hierarchical clustering," *Parallel Computing*, vol. 21, no. 8, pp. 1313–1325, 1995.

[4] M. Dash, S. Petrutiu, and P. Scheuermann, "Efficient parallel hierarchical clustering," in *Proceedings of the European Conference on Parallel Processing*, 2004, pp. 363–371.

[5] C. Jin, R. Liu, Z. Chen, W. Hendrix, A. Agrawal, and A. Choudhary, "A scalable hierarchical clustering algorithm using Spark," in *Proceedings of the 2015 IEEE First International Conference on Big Data Computing Service and Applications*, 2015, pp. 418–426.

[6] Y. Song, W.-Y. Chen, H. Bai, C.-J. Lin, and E. Y. Chang, "Parallel spectral clustering," in *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2008, pp. 374–389.

[7] W.-Y. Chen, Y. Song, H. Bai, C.-J. Lin, and E. Y. Chang, "Parallel Spectral Clustering in distributed systems," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 3, pp. 568–586, 2010.

[8] M. M. A. Patwary, D. Palsetia, A. Agrawal, W.-k. Liao, F. Manne, and A. Choudhary, "A new scalable parallel DBSCAN algorithm using the disjoint-set data structure," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11.

[9] M. Götz, C. Bodenstein, and M. Riedel, "HPDBSCAN: highly parallel DBSCAN," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, 2015, pp. 1–10.

[10] Y. Li, C. Luo, and S. M. Chung, "A parallel text document clustering algorithm based on neighbors," *Cluster Computing*, vol. 18, no. 2, pp. 933–948, 2015.

[11] T. H. Sardar and Z. Ansari, "An analysis of mapreduce efficiency in document clustering using parallel k-means algorithm," *Future Computing and Informatics Journal*, vol. 3, no. 2, pp. 200–209, 2018.

[12] M. A. B. HajKacem, C.-E. B. N'cir, and N. Essoussi, "A parallel text clustering method using Spark and hashing," *Computing*, pp. 1–25, 2021.

[13] L. Akritidis, M. Alamaniotis, A. Fevgas, and P. Bozanis, "Confronting sparseness and high dimensionality in short text clustering via feature vector projections," in *Proceedings of the 32nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, 2020, pp. 813–820.

[14] L. Akritidis, M. Alamaniotis, A. Fevgas, P. Tsompanopoulou, and P. Bozanis, "Improving hierarchical short text clustering through dominant feature learning," *International Journal on Artificial Intelligence Tools*, 2021.

[15] 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.

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

[17] W. Xu and Y. Gong, "Document clustering by concept factorization," in *Proceedings of the 27th International ACM SIGIR Conference on Research and Development in Information Retrieval*. Association for Computing Machinery, 2004, pp. 202–209.

[18] M. Chen, Q. Wang, and X. Li, "Adaptive projected matrix factorization method for data clustering," *Neurocomputing*, vol. 306, pp. 182–188, 2018.

[19] D. D. Lee and H. S. Seung, "Learning the parts of objects by non-negative matrix factorization," *Nature*, vol. 401, no. 6755, pp. 788–791, 1999.

[20] X. Yan, J. Guo, S. Liu, X. Cheng, and Y. Wang, "Learning topics in short texts by Non-Negative Matrix Factorization on term correlation matrix," in *Proceedings of the 2013 SIAM International Conference on Data Mining*, 2013, pp. 749–757.

[21] X. Yan, J. Guo, S. Liu, X.-q. Cheng, and Y. Wang, "Clustering short text using ncut-weighted non-negative matrix factorization," in *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, 2012, pp. 2259–2262.

[22] F. Shang, L. Jiao, and F. Wang, "Graph dual regularization non-negative matrix factorization for co-clustering," *Pattern Recognition*, vol. 45, no. 6, pp. 2237–2250, 2012.

[23] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[24] J. Kumar, J. Shao, S. Uddin, and W. Ali, "An online semantic-enhanced Dirichlet model for short text stream clustering," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, July 2020, pp. 766–776.

[25] J. Yin and J. Wang, "A Dirichlet multinomial mixture model-based approach for short text clustering," in *Proceedings of the 20th ACM International Conference on Knowledge Discovery and Data Mining*, 2014, pp. 233–242.

[26] X. Yan, J. Guo, Y. Lan, and X. Cheng, "A biterm topic model for short texts," in *Proceedings of the 22nd International Conference on World Wide Web*, 2013, pp. 1445–1456.

[27] L. Akritidis and P. Bozanis, "Effective unsupervised matching of product titles with k-combinations and permutations," in *Proceedings of the 14th IEEE International Conference on Innovations in Intelligent Systems and Applications*, 2018, pp. 1–10.

[28] L. Akritidis, A. Fevgas, P. Bozanis, and C. Makris, "A self-verifying clustering approach to unsupervised matching of product titles," *Artificial Intelligence Review*, pp. 1–44, 2020.

[29] Z. Lv, Y. Hu, H. Zhong, J. Wu, B. Li, and H. Zhao, "Parallel k-means clustering of remote sensing images based on MapReduce," in *Proceedings of the International Conference on Web Information Systems and Mining*, 2010, pp. 162–170.

[30] J. Zhang, G. Wu, X. Hu, S. Li, and S. Hao, "A parallel k-Means clustering algorithm with mpi," in *Proceedings of the 4th International Symposium on Parallel Architectures, Algorithms and Programming*, 2011, pp. 60–64.

[31] M. N. Joshi, "Parallel k-Means algorithm on distributed memory multiprocessors," *Computer*, vol. 9, 2003.

[32] L. Weiming, W. B. Du Chenyang, S. Chunhui, and Y. Zhenchao, "Distributed affinity propagation clustering based on MapReduce," *Journal of Computer Research and Development*, vol. 49, no. 8, p. 1762, 2012.

[33] M. A. Patwary, D. Palsetia, A. Agrawal, W.-k. Liao, F. Manne, and A. Choudhary, "Scalable parallel OPTICS data clustering using graph algorithmic techniques," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[34] F. Huang, Y. Chen, L. Li, J. Zhou, J. Tao, X. Tan, and G. Fan, "Implementation of the parallel mean shift-based image segmentation algorithm on a GPU cluster," *International Journal of Digital Earth*, vol. 12, no. 3, pp. 328–353, 2019.

[35] Q. Zhao, Y. Shi, and Z. Qing, "Research on Hadoop-based massive short text clustering algorithm," in *Proceedings of the 4th International Workshop on Pattern Recognition*, vol. 11198, 2019, p. 111980A.

[36] C. Zewen and Z. Yao, "Parallel text clustering based on mapreduce," in *Proceedings of the 2nd International Conference on Cloud and Green Computing*, 2012, pp. 226–229.