A Scalable Short-Text Clustering Algorithm Using Apache Spark

L. Akritidis[1], M. Alamaniotis[2], A. Fevgas[3], P. Bozanis[1]

[1]School of Science and Technology, International Hellenic University
[2]Department of Electrical and Computer Engineering, University of Texas at San Antonio
[3]Department of Electrical and Computer Engineering, University of Thessaly

33rd IEEE International Conference on Tools with Artificial Intelligence
ICTAI 2021
November 1-3, 2021, Virtual Event

Short text clustering

- It deals with the problem of grouping together semantically similar documents with small lengths.
- Nowadays, huge volumes of short documents are being produced:
  - Microblogs (Twitter).
  - Entitled entities (news articles, scientific documents, products, etc.).
  - User comments.
  - Messengers.
- There is an increasing requirement for efficient processing (including clustering) of short documents.

Short text clustering

- There is a vast number of clustering algorithms in the literature.
  - A portion of them may handle large amounts of data.
  - Some others just do not work well (i.e. they are slow).
  - Some others do not work *at all*.
- Traditional methods (k-Means, hierarchical, NMF, etc.):
  - parallelized in the past, but not very effective in this particular problem.
- Focused methods (GSDMM, BTM, VEPHC, etc.):
  - much more effective in this particular problem, but not parallelized yet.

pVEPHC

- In this work we introduce pVEPHC, a parallel clustering algorithm for large scale, short document collections.
- pVEPHC parallelizes a recent state-of-the-art short text clustering method, called VEPHC.
  - VEPHC: "VEctor Projection, Hierarchical Clustering". Presented at ICTAI 2020.
  - pVEPHC: parallel VEPHC.
- pVEPHC was designed, implemented, and tested on Apache Spark, a popular parallelization framework.

VEPHC – Stage 1 – Phase 1 – Projection vectors (PVs)

- VEPHC is a two stage short text clustering algorithm.
- **Stage 1 (VEP part)**: It includes three phases.
- **Phase 1**: We project the original feature vectors onto a lower dimensional space by generating all the feature combinations of the initial text vectors.
- Each feature combination corresponds to a Projection Vector (PV).
- The population of the PVs is limited by a positive integer hyper-parameter K.
- A projection vector may include at most K components.

VEPHC – Stage 1 – Phase 2 – PVs scoring

- **Phase 2**: Each PV is assigned a score which favors the completeness and the homogeneity of the cluster that will include it.
- Thus, we introduce the following scoring function to assign a score to the j-th projection vector $\mathbf{x}'_{ij}$ of an input document vector $\mathbf{x}_i$:

$$S_{\mathbf{x}'_{ij}} = \frac{1}{l_{\mathbf{x}'_{ij}}} \log f_{\mathbf{x}'_{ij}} \sum_{\forall x_{ij} \in \mathbf{x}'_{ij}} x_{ij}^{K}$$

where

- $l_{\mathbf{x}'_{ij}}$: the length of the projection vector $\mathbf{x}'_{ij}$.
- $f_{\mathbf{x}'_{ij}}$: the frequency in the collection of the projection vector $\mathbf{x}'_{ij}$.
- x_{ij}: the weight of the j-th element (term) in the original input vector $\mathbf{x}_i$.

VEPHC – Stage 1 – Phase 3 – Initial clustering

- **Phase 3**: For each input feature vector $\mathbf{x}_i$, we identify the Projection Vector $\mathbf{x}_i^*$ with the highest score.
- We call $\mathbf{x}_i^*$ as the Dominant Projection Vector (DPV).
- All documents that share common DPVs essentially lie into the same (lower) dimensional space.
- Finally: All documents whose feature vectors are projected onto the same dimensional space are grouped together into the same cluster.

VEPHC – Stage 1 – Example

Document	$\mathbf{x}$	$\mathbf{x}'$	DPV $\mathbf{x}^*$
$d_1 = \{t_1, t_2, t_3\}$	$\mathbf{x}_1 = (x_{11}, x_{12}, x_{13})$	(x_{11}), (x_{12}), (x_{13}) (x_{11}, x_{12}), (x_{11}, x_{13}),(x_{12}, x_{13})	$x_{11}\hat{\mathbf{t}}_1 + x_{12}\hat{\mathbf{t}}_2$
$d_2 = \{t_1, t_2, t_4\}$	$\mathbf{x}_2 = (x_{21}, x_{22}, x_{24})$	(x_{21}), (x_{22}), (x_{23}) (x_{21}, x_{22}), (x_{21}, x_{23}),(x_{22}, x_{23})	$x_{21}\hat{\mathbf{t}}_1 + x_{22}\hat{\mathbf{t}}_2$
$d_3 = \{t_1, t_2\}$	$\mathbf{x}_3 = (x_{31}, x_{32})$	(x_{31}), (x_{32}), (x_{31}, x_{32})	$x_{31}\hat{\mathbf{t}}_1 + x_{32}\hat{\mathbf{t}}_2$
$d_4 = \{t_3, t_4\}$	$\mathbf{x}_4 = (x_{41}, x_{42})$	(x_{41}), (x_{42}), (x_{41}, x_{42})	$x_{41}\hat{\mathbf{t}}_3 + x_{42}\hat{\mathbf{t}}_4$
$d_5 = \{t_3, t_4, t_5\}$	$\mathbf{x}_5 = (x_{51}, x_{52}, x_{53})$	(x_{51}), (x_{52}), (x_{53}) (x_{51}, x_{52}), (x_{51}, x_{53}),(x_{52}, x_{53})	$x_{51}\hat{\mathbf{t}}_3 + x_{52}\hat{\mathbf{t}}_4$

Table 3. Dominant Reference Vectors and clustering.

Document	DPV $\mathbf{x}^*$	DRV $\hat{\mathbf{x}}^*$	Cluster
$d_1 = \{t_1, t_2, t_3\}$	$x_{11}\hat{\mathbf{t}}_1 + x_{12}\hat{\mathbf{t}}_2$	$\hat{\mathbf{t}}_1 + \hat{\mathbf{t}}_2$	c_1
$d_2 = \{t_1, t_2, t_4\}$	$x_{21}\hat{\mathbf{t}}_1 + x_{22}\hat{\mathbf{t}}_2$	$\hat{\mathbf{t}}_1 + \hat{\mathbf{t}}_2$	c_1
$d_3 = \{t_1, t_2\}$	$x_{31}\hat{\mathbf{t}}_1 + x_{32}\hat{\mathbf{t}}_2$	$\hat{\mathbf{t}}_1 + \hat{\mathbf{t}}_2$	c_1
$d_4 = \{t_3, t_4\}$	$x_{41}\hat{\mathbf{t}}_3 + x_{42}\hat{\mathbf{t}}_4$	$\hat{\mathbf{t}}_3 + \hat{\mathbf{t}}_4$	c_2
$d_5 = \{t_3, t_4, t_5\}$	$x_{41}\hat{\mathbf{t}}_3 + x_{42}\hat{\mathbf{t}}_4$	$\hat{\mathbf{t}}_3 + \hat{\mathbf{t}}_4$	c_2

pVEPHC – Stage 1

Algorithm 1: Word dictionary construction

1 rdd_dataset $\leftarrow$ read from HDFS;
2 $n \leftarrow |$rdd_dataset$|$;
3 rdd_words $\leftarrow$ rdd_dataset
4   .flatMap($d \rightarrow$ tokenize(d))
5   .reduceByKey$(x, y \rightarrow x + y)$
6   .map$(x \rightarrow (x[0], \log(n/x[1])))$;
7 rdd_dic $\leftarrow$ rdd_words
8   .zipWithIndex()
9   .map$(x \rightarrow (x[0][0], (x[1], x[0][1])))$;
10 dictionary $\leftarrow$rdd_dic.collectAsMap();
11 $D \leftarrow$ broadcast(dictionary);
12 -OR-
13 dictionary_file $\leftarrow$ 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=$[(w^{\dagger}, (wID, \text{IDF}(w)), \ldots]$

Algorithm 2: Projection vector construction

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

Technical details

- There are too many technical details here.
- The interested reader may refer to the paper.
- We need a dictionary that for each entry (word) stores:
  - a unique identifier that will be used to convert the documents into vectors,
  - its inverse document frequency that will be used for PV scoring.
- We will use this dictionary to perform millions of word look-ups.
- If we store it into a distributed data structure (e.g. an RDD), we will suffer severe performance degradation.
- Better alternative: broadcast the dictionary to all executors.
- Make the distributed look-ups local.

pVEPHC – Stage 1

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	$[(\mathbf{p}_x^\dagger, (docID, S_{\mathbf{p}x})), \dots]$
31	flatMap	$[(docID, (S_{\mathbf{p}x}, \mathbf{p}_x)), \dots]$
32	reduceByKey	$[(docID, (S_{\mathbf{p}x}, \mathbf{p}_x^*)), \dots]$
33	map	rdd_drv=$[(docID, \mathbf{p}_x^*)), \dots]$
34	map	rdd_clusters=$[(\mathbf{p}_x^*, docID)), \dots]$

VEPHC – Stage 2

- Stage 2 (HC part) is a post-processing step that enhances the clustering of the previous phase.
- Initially, the most dissimilar elements are removed from their respective clusters.
- The removed elements form new, singleton clusters.
- Then, the most similar clusters are merged together by adopting an Hierarchical Agglomerative Clustering approach.

pVEPHC – Stage 2 – Removal of dissimilar elements

Algorithm 4: Removal of dissimilar elements

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

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

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

pVEPHC – Stage 2 – Hierarchical merging

- The parallelization of the second phase is challenging as it exhibits inherent data dependency during the hierarchical tree construction.
- However, it has been efficiently solved in the literature [1].
- By formulating the problem of Agglomerative clustering as a Minimal Spanning Tree problem.
  - Based on the theoretical finding that calculating the HC dendrogram is equivalent to finding the Minimum Spanning Tree (MST) of a complete weighted graph, where the vertices are the data points and the edge weights are the distances between any two points.
- "A Scalable Hierarchical Clustering Algorithm Using Spark", Proc. of the 1st International Conf. on Big Data Computing Service and Applications.

Experimental setup

- **Implementation**: Java 1.8.
- **Computer cluster**: 8 VMs with 16 CPUs and 64 GB of RAM each.
- **Configuration**: each node deployed 2 YARN containers. Each container deployed a single Spark executor.
  - Total: 2 executors per VM.
- **Executor setup**: 2 vCores and 24GB of RAM.
  - **Total cluster resources**: 1 driver, 15 executors / 30 vCores, 360 GB RAM.
- **Software**: Ubuntu 18.04 LTS, Hadoop 3.2.2, Spark 3.1.2, YARN, and HDFS.
- **Dataset**: FakeNewsOnlyTitles2. This is a large document collection of news articles. It includes approximately 400 thousand titles.

Experimental evaluation

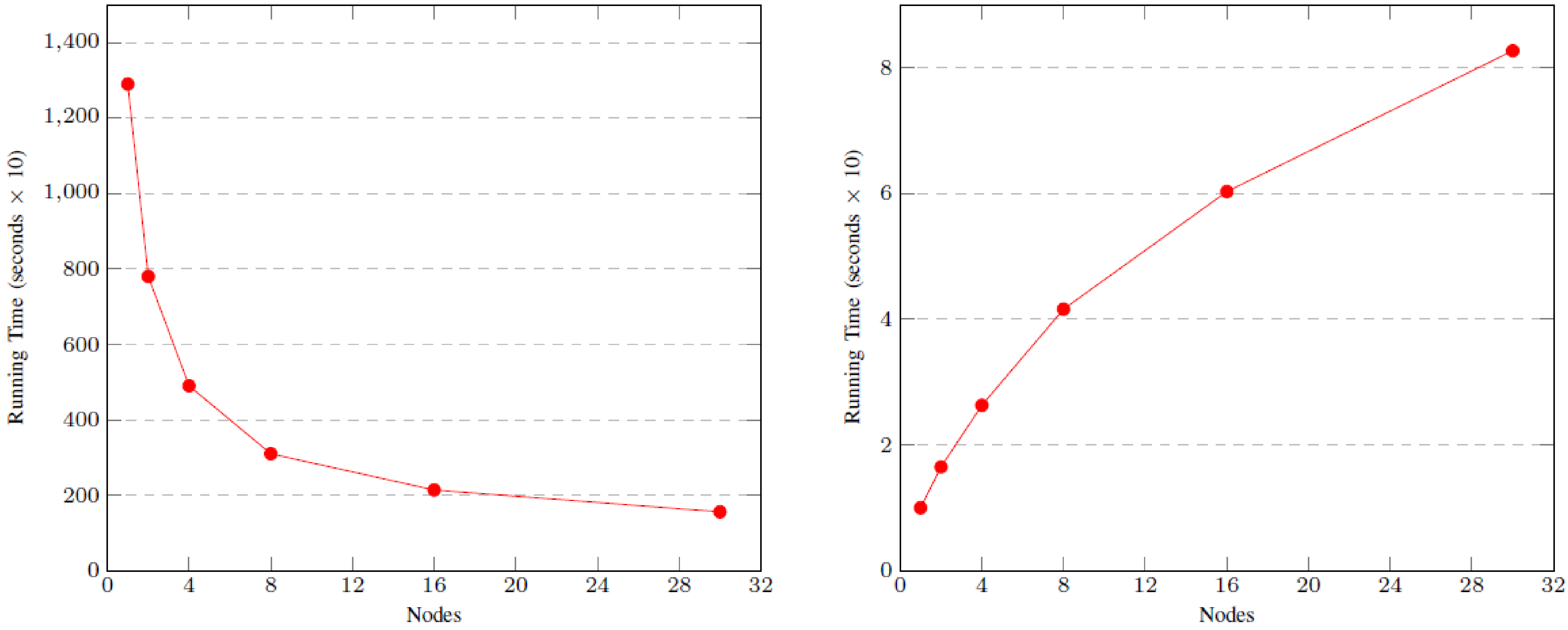

Fig. 1. Running times of pVEPHC for various cluster sizes (left), and acceleration factor vs. number of processing nodes (right).

Conclusions & future work

- In this work we presented pVEPHC, a parallel Spark algorithm for clustering massive collections of short documents.
- The algorithm builds on VEPHC, a two stage algorithm that:
  - Initially projects the input documents onto a lower-dimensional space, and
  - then, it performs a post-processing hierarchical cluster refinement process
- The experiments demonstrated a satisfactory scaling of the introduced algorithm.
- Presently, we are attempting to further extend the algorithm so that it works on large-scale collections of streaming documents.

Thank you for watching

Questions?