2015 IEEE Trustcom/BigDataSE/ISPA

A MapReduce-based k-Nearest Neighbor Approach for Big Data Classification

Jesús Maillo
Department of Computer Science
and Artificial Intelligence,
University of Granada, 18071
Granada, Spain
Email: jesusmh@correo.ugr.es

Isaac Triguero
Department of Respiratory Medicine,
Ghent University, 9000
Gent, Belgium
VIB Inflammation Research Center,
9052 Zwijnaarde, Belgium
Email: Isaac.Triguero@irc.vib-UGent.be

Francisco Herrera
Department of Computer Science
and Artificial Intelligence,
University of Granada, 18071
Granada, Spain
Email: herrera@decsai.ugr.es

***Abstract*—The k-Nearest Neighbor classifier is one of the most well known methods in data mining because of its effectiveness and simplicity. Due to its way of working, the application of this classifier may be restricted to problems with a certain number of examples, especially, when the runtime matters. However, the classification of large amounts of data is becoming a necessary task in a great number of real-world applications. This topic is known as big data classification, in which standard data mining techniques normally fail to tackle such volume of data. In this contribution we propose a MapReduce-based approach for k-Nearest neighbor classification. This model allows us to simultaneously classify large amounts of unseen cases (test examples) against a big (training) dataset. To do so, the map phase will determine the k-nearest neighbors in different splits of the data. Afterwards, the reduce stage will compute the definitive neighbors from the list obtained in the map phase. The designed model allows the k-Nearest neighbor classifier to scale to datasets of arbitrary size, just by simply adding more computing nodes if necessary. Moreover, this parallel implementation provides the exact classification rate as the original k-NN model. The conducted experiments, using a dataset with up to 1 million instances, show the promising scalability capabilities of the proposed approach.**

I. Introduction

The classification of big data is becoming an essential task in a wide variety of fields such as biomedicine, social media, marketing, etc. The recent advances in data gathering in many of these fields has resulted in an inexorable increment of the data that we have to manage. The volume, diversity and complexity that bring big data may hinder the analysis and knowledge extraction processes [1]. Under this scenario, standard data mining models need to be re-designed or adapted to deal with this data.

The k-Nearest Neighbor algorithm (k-NN) [2] is considered one of the ten most influential data mining algorithms [3]. It belongs to the lazy learning family of methods that do not need of an explicit training phase. This method requires that all of the data instances are stored and unseen cases classified by finding the class labels of the k closest instances to them. To determine how close two instances are, several distances or similarity measures can be computed. This operation has to be performed for all the input examples against the whole training dataset. Thus, the response time may become compromised when applying it in the big data context.

Recent cloud-based technologies fortunately offer an ideal environment to handle this issue. The MapReduce framework [4] highlights as a simple and robust programming paradigm to tackle large-scale datasets within a cluster of nodes. Its application is very appropriate for data mining because of its fault-tolerant mechanism (recommendable for time-consuming tasks) and its ease of use [5] as opposed to other parallelization schemes such as Message Passing Interface [6]. In recent years, several data mining techniques have been successfully implemented by using this paradigm, such as [7], [8]. Some related works utilize MapReduce for similar k-NN searches. For example, in [9] and [10] the authors apply kNN-join queries within a MapReduce process.

In this work we design a new parallel k-NN algorithm based on MapReduce for big data classification. In our particular implementation, the map phase consists of deploying the computation of similarity between test examples and splits of the training set through a cluster of computing nodes. As a result of each map, the k nearest neighbors together with their computed distance values will be emitted to the reduce stage. The reduce phase will determine which are the final k nearest neighbors from the list provided by the maps. Through the text, we will denote this approach as MR-kNN.

To test the performance of this model, we have carried experiments on a big dataset with up to 1 million instances. The experimental study includes an analysis of test accuracy, runtime and speed up. Moreover, several values of k and number of mappers will be investigated.

The paper is structured as follows. Section II provides background information about the k-NN algorithm and MapReduce. Section III describes the proposal. Section IV analyzes the empirical results. Finally, Section V summarizes the conclusions.

II. Preliminaries

In this section we introduce some background information about the main components used in this paper. Section II-A presents the k-NN algorithm as well as its weaknesses to deal with big data classification. Section II-B provides the description of the MapReduce paradigm and the implementation used in this work.

978-1-4673-7952-6/15 $31.00 © 2015 IEEE
DOI 10.1109/Trustcom-BigDataSe-ISPA.2015.577

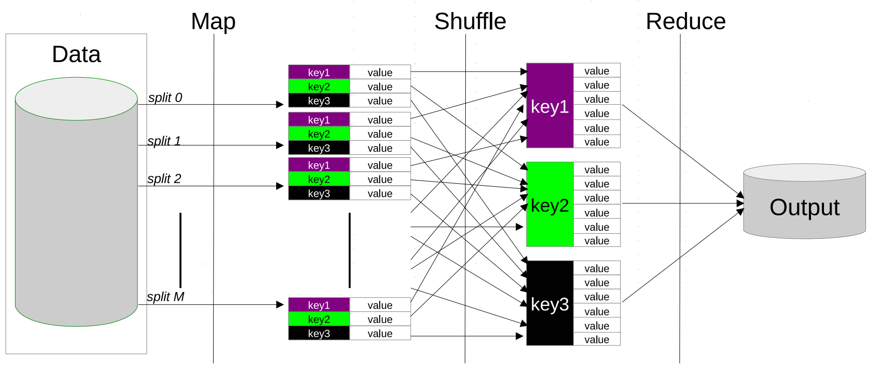

Fig. 1. The MapReduce workflow

A. k-NN and weaknesses to deal with big data

The k-NN algorithm is a non-parametric method that can be used for either classification and regression tasks. This section defines the k-NN problem, its current trends and the drawbacks to manage big data. A formal notation for the k-NN algorithm is the following:

Let TR be a training dataset and TS a test set, they are formed by a determined number **n** and **t** of samples, respectively. Each sample $\mathbf{x}_p$ is a tuple $(\mathbf{x}_{p1}, \mathbf{x}_{p2}, ..., \mathbf{x}_{pD}, \omega)$, where, $\mathbf{x}_{pf}$ is the value of the f-th feature of the p-th sample. This sample belongs to a class ω, given by $\mathbf{x}_p^{\omega}$, and a D-dimensional space. For the TR set the class ω is known, while it is unknown for TS. For each sample $\mathbf{x}_{test}$ contained in the TS set, the k-NN model looks for the k closest samples in the TR set. To do this, it computes the distances between $\mathbf{x}_{test}$ and all the samples of TR. The Euclidean distance is commonly used for this purpose. The k closest samples $(\mathbf{neigh}_1, \mathbf{neigh}_2, ..., \mathbf{neigh}_k)$ are obtained by ranking (ascending order) the training samples according to the computed distance. By using the k closest neighbors, a majority vote is conducted to determine which class is predominant among the neighbors. The selected value of k may influence the performance and the noise tolerance of this technique.

Despite the promising results shown by the k-NN in a wide variety of problems, it lacks of scalability to address big TR datasets. The main problems found to deal with large-scale data are:

- Runtime: The complexity of the traditional k-NN algorithm is $O((n \cdot D))$, where n is the number of instances and D the number of features.
- Memory consumption: For a rapid computation of the distances, the k-NN model may normally require to store the training data in memory. When TR is too big, it could easily exceed the available RAM memory.

These drawbacks motivate the use of big data technologies to distribute the processing of k-NN over a cluster a nodes. In this work, we will focus on the reduction of the runtime according to the number of instances.

B. MapReduce

MapReduce is a very popular parallel programming paradigm [4] that was developed to process and/or generate big datasets that do not fit into a physical memory. Characterized by its transparency for programmers, this framework enables the processing of huge amounts of data on top of a computer cluster regardless the underlying hardware or software. This is based on functional programming and works in two main steps: the map phase and the reduce phase. Each one has key-value ($< key, value >$) pairs as input and output. These phases are the only thing that the programmer must implement. The map phase takes each $< key, value >$ pair and generates a set of intermediate $< key, value >$ pairs. Then, MapReduce merges all the values associated with the same intermediate key as a list (shuffle phase). The reduce phase takes that list as input for producing the final values.

Figure 1 presents a flowchart of the MapReduce framework. In a MapReduce program, all map and reduce operations run in parallel. First of all, all map functions are independently run. Meanwhile, reduce operations wait until the map phase has finished. Then, they process different keys concurrently and independently. Note that inputs and outputs of a MapReduce job are stored in an associated distributed file system that is accessible from any computer of the used cluster.

Different implementations of the MapReduce framework are possible, depending on the available cluster architecture. In this paper we will use the Hadoop implementation [11] because of its performance, open source nature, installation facilities and its distributed file system (Hadoop Distributed File System, HDFS).

From a programmer's point of view, the Hadoop implementation of MapReduce divides the lifecycle of a map/reduce task as: (1) setup, (2) operation itself (map or reduce) and (3) cleanup. The setup procedure is normally devoted to read parameters from the configuration object. The cleanup method can be used to clean up any resources you may have allocated, but also, to flush out any accumulation of aggregate results.

A Hadoop cluster is formed by a master-slave architecture, where one master node manages an arbitrary number of slave nodes. The HDFS replicates file data in multiple storage nodes that can concurrently access to the data. As such cluster, a certain percentage of these slave nodes may be out of order temporarily. For this reason, Hadoop provides a fault-tolerant mechanism, so that, when one node fails, Hadoop restarts automatically the task on another node.

As we commented above, the MapReduce approach and

TABLE I. ENCODING THE RESULTING k NEAREST NEIGHBORS (CLASSES AND DISTANCES) FOR A CERTAIN MAPPER Map_j

	Neighbor 1	Neighbor 2	...	Neighbor k
$\mathbf{x}_{test,1}$	$< Class(neigh_1), Dist(neigh_1) >_1$	$< Class(neigh_2), Dist(neigh_2) >_1$	...	$< Class(neigh_k), Dist(neigh_k) >_1$
$\mathbf{x}_{test,2}$	$< Class(neigh_1), Dist(neigh_1) >_2$	$< Class(neigh_2), Dist(neigh_2) >_2$	...	$< Class(neigh_k), Dist(neigh_k) >_2$
...				
$\mathbf{x}_{test,t}$	$< Class(neigh_1), Dist(neigh_1) >_t$	$< Class(neigh_2), Dist(neigh_2) >_t$	...	$< Class(neigh_k), Dist(neigh_k) >_t$

its derivatives can be useful for many different tasks. In terms of data mining, it offers a propitious environment to successfully speed up these kinds of techniques. In fact, projects like Apache Mahout [12] and the MLlib library from Apache Spark [13] collect distributed and scalable machine learning algorithms implemented with MapReduce and further extensions (mainly, iterative MapReduce).

III. MR-kNN: A MapReduce implementation for k-NN

In this section we explain how to paralellize the k-NN algorithm based on MapReduce. As a MapReduce model, it organizes the computation into two main operations: the map and the reduce phases. The map phase will compute the classes and the distances to the k nearest neighbors of each test example in different splits of the training data. The reduce stage will process the distances of the k nearest neighbors from each map and will create a definitive list of k nearest neighbors by taking those with minimum distance. Afterwards, it will carry out the majority voting procedure as usual in the k-NN model to predict the resulting class. In what follows, we will detail the map and reduce phases, separately (Sections III-A and III-B, respectively). At the end of the section, Figure 2 illustrates a high level scheme of the proposed parallel system.

A. Map phase

Let TR a training set and TS a test set of a arbitrary sizes that are stored in the HDFS as single files. The map phase starts diving the TR set into a given number of disjoint subsets. In Hadoop, files are formed by h HDFS blocks that can be accessed from any computer of the cluster independently of its size. Let m be the number of map processes that will be defined by the end-user. Each map task ($Map_1, Map_2, ..., Map_m$) will create an associated TR_j, where $1 \leq j \leq m$, with the sample of each chunk in which the training set file is divided. It is noteworthy that this partitioning process is sequentially performed, so that, the Map_j corresponds to the j data chunk of h/m HDFS blocks. As a result, each map analyze approximately a similar number of training instances.

Note that splitting the data into several subsets, and analyze them individually, fits better with the MapReduce philosophy than with other parallelization schemes because of two reasons: Firstly, each subset is individually processed, so that it does not need any data exchange between nodes to proceed. Secondly, the computational cost of each chunk could be so high that a fault-tolerant mechanism is mandatory.

Since our goal is to obtain an exact implementation of the original k-NN algorithm, the TS file will not be split because we will need to have access to every single test sample in all the maps. In this way, when each map has formed its corresponding TR_j set, we will compute the distance of each $\mathbf{x}_t$ against the instances of TR_j. The class label of the closest k neighbors (minimum distance), for each test example, and the their distance will be saved. As a result, we will obtain a matrix CD_j of pairs $< class, distance >$ with dimension $n \cdot k$. Therefore, at row i, we will have the distance and the class of the k nearest neighbors. It is noteworthy that every row will be ordered in ascending order regarding the distance to the neighbor, so that, $Dist(neigh_1) < Dist(neigh_2) < .... < Dist(neigh_k)$. For sake of clarity, Table I formally defines how this matrix is formed.

It is important to point out that to avoid memory restriction problems, the test file is accessed line by line, so that, we do not load in memory the whole test file. Hadoop's primitive functions allows us to make this with no efficiency losses. Algorithm 1 contains the pseudo-code of the map function.

As each map finishes its processing the results are forwarded to a single reduce task. The output key in every map is established according to an identifier value of the mapper.

Algorithm 1 Map function

```
Require: $TS$; $k$
1: Constitute $TR_j$ with the instances of split $j$.
2: for $i = 0$ to $i < size(TS)$ do
3:     Compute k-NN ($\mathbf{x}_{test,i}$,$TR_j$,$k$)
4:     for $n = 0$ to $n < k$ do
5:         $CD_j(i, n) =< Class(neigh_n), Dist(neigh_n) >_i$
6:     end for
7: end for
8: $key = idMapper$
9: EMIT($< key, CD_j >$)
```

B. Reduce phase

The reduce phase consists of determining which of the tentative k nearest neighbors from the maps are the closest ones for the complete TS. Given that we aim to design a model that can scale to arbitrary size training sets and that it is independent of the selected number of neighbors, we carefully implemented this operation by taking advantage of the setup, reduce, and cleanup procedures of the reduce task (introduced in Section II-B). Algorithm 2 describes the reduce operation and Algorithm 3 the cleanup phase. A detailed explanation of all of them is as follows:

- **Setup**: apart from reading the parameters from the Hadoop configuration object, the setup operation will allocate a class-distance matrix $CD_{reducer}$ of fixed size ($size(TS) \cdot kneighbors$). As requirement, this function will need to know the size of TS, but it

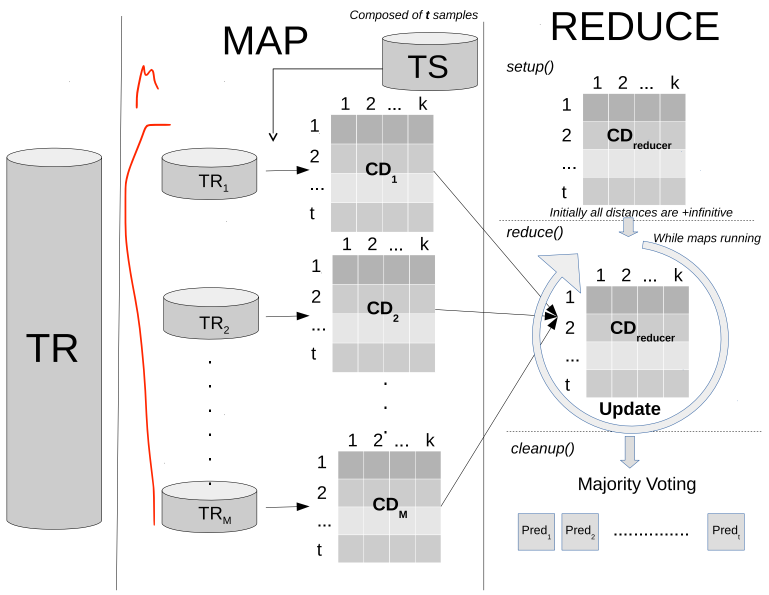

Fig. 2. Flowchart of the proposed MR-kNN algorithm

does not need to read the set itself. This matrix will be initialized with random values for the classes and positive infinitive value for the distances. Note that this operation is only run the first time that the reduce phase takes place. In MapReduce the reducer may start receiving data as the first mapper is finished.

- **Reduce**: when the map phase finishes, the class-distance matrix $CD_{reducer}$ is updated by comparing its current distances values with the matrix that comes from every Map_j, that is, CD_j. Since the matrices coming from the maps are ordered according to the distance, the update process becomes faster. It consists of the merging process of two sorted lists up to have k values (complexity O(k)). Thus, for each test example $\mathbf{x}_{test}$, we compare every distance value of the neighbors one by one, starting from the closets neighbor. If the distance is lesser than the current value, the class and the distance of this matrix position is updated with the corresponding values, otherwise we proceed with the following value (See Instructions 3-6 in Algorithm 2 for more details). In contradistinction to the setup operation, this is run every time a map is finished. Thus, it could be interpreted as an iterative procedure that aggregate the results provided by the maps. As such, this operation does not need to send any $< key, value >$ pairs. It will be performed in the next procedure.

- **Cleanup**: after all the maps' inputs have been processed by the previous reduce procedure, the cleanup phase carries out its processing. At this point, the $CD_{reducer}$ will contain the definitive list of neighbors (class and distance) for all the examples of TS. Therefore, the cleanup is devoted to perform the majority voting of the k-NN model and determine the predicted classes for TS. As a result, the predicted classes for all the TS set are provided as the final output of the reduce phase. The key is established as the number of instance in the TS set (See Instruction 3 in Algorithm 3).

As we have claimed before, MR-kNN only uses one single reducer. The use of a single reducer results in a computationally less expensive process in comparison to use more than one. Concretely, it helps us to reduce the network-related Mapreduce overhead. It also allows us to obtain a single output file.

Algorithm 2 Reduce operation

```
Require: $size(TS)$, $k$, $CD_j$
Require: Setup procedure has been launched.
1: for $i = 0$ to $i < size(TS)$ do
2:     cont=0
3:     for $n = 0$ to $k$ do
4:         if $CD_j(i, cont).Dist < CD_{reducer}(i, n).Dist$ then
5:             $CD_{reducer}(i, n) = CD_j(i, cont)$
6:             cont++
7:         end if
8:     end for
9: end for
```

Algorithm 3 Reduce cleanup process

```
Require: $size(TS)$, $k$
Require: Reduce operation has finished.
1: for $i = 0$ to $i < size(TS)$ do
2:     $PredClass_i = MajorityVoting(Classes(CD_{reduce}))$
3:     $key = i$
4:     EMIT($< key, PredClass_i >$)
5: end for
```

As summary, Figure 2 depicts a flowchart containing the main steps of the proposed model.

IV. Experimental study

In this section we present all the questions raised with the experimental study. Section IV-A establishes the experimental framework and Section IV-B presents and discusses the results achieved.

A. Experimental Framework

The following measures will be considered to assess the performance of the proposed technique:

- *Accuracy:* It counts the number of correct classifications regarding the total number of instances.
- *Runtime:* We will quantify the time spent by MR-kNN in map and reduce phases as well as the global time to classify the whole TS set. The global time includes all the computations performed by the MapReduce framework (communications).
- *Speed up:* It checks the efficiency of a parallel algorithm in comparison with a slower version. In our experiments we will compute the speed up achieved depending on the number of mappers.

$$Speedup = \frac{reference\_time}{parallel\_time} \quad (1)$$

where $reference\_time$ is the runtime spent with the sequential version and $parallel\_time$ is the runtime achieved with its improved version.

In our experiments, global times (including all MapReduce computations) are compared to those obtained with the sequential version to compute the speed up.

This experimental study is focused on analyzing the effect of the number of mappers (16, 32, 64, 128, and 256) and number of neighbors (1, 3, 5, and 7) in the proposed MR-kNN model. We will test its performance over the PokerHand dataset, taken from the UCI repository [14]. It contains a total of 1025010 instances, 10 features and 10 different classes. This dataset has been partitioned using a 5 fold cross-validation (5-fcv) scheme.

The experiments have been carried out on sixteen nodes in a cluster: a master node and fifteen compute nodes. Each one of these compute nodes has an Intel Core i7 4930 processor, 6 cores per processor, 3.4 GHz and 64GB of RAM. The network is Ethernet 1Gbps. In terms of software, we have used the Cloudera's open-source Apache Hadoop distribution (Hadoop 2.5.0-cdh5.3.2). Please note that the maximum number of map functions concurrently running for this cluster is setup to 128, so that, for experiments with 256 maps we cannot expect a linear speedup.

B. Results and discussion

This section presents and analyzes the results obtained in the experimental study. First, we focus on the results of the sequential version of k-NN as our baseline. Table II collects the average accuracy (AccTest) in the test partitions and the runtime (in seconds) results obtained by the standard k-NN algorithm, according to the number of neighbors.

Table III summarizes the results obtained by the proposed approach. It shows, for each number of neighbors and number of maps, the average time needed by the map phase (AvgMapTime), the average time spent by the reduce phase (AvgRedTime), the average total time (AvgTotalTime), the obtained average accuracy (AccTest) and the speed up achieved (Speedup).

TABLE II. Sequential k-NN performance

Number of Neighbors	AccTest	Runtime(s)
1	0.5019	105475.0060
3	0.4959	105507.8470
5	0.5280	105677.1990
7	0.5386	107735.0380

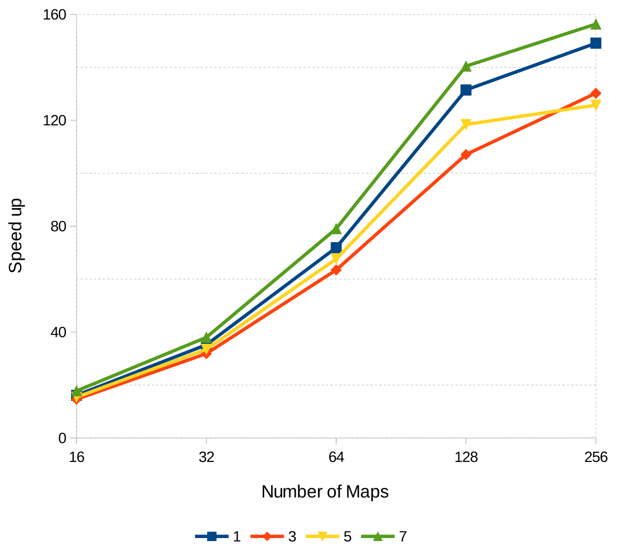

Fig. 3. Speedup

From these tables and figure we can highlight several factors:

- As we can observe in Table II, the required runtime for the sequential k-NN method is quite high. However, by using the proposed approach, a great reduction of the consumed computation time is shown when the number of mappers is increased. Because of the implementation performed, our proposal always provides the same accuracy than the original k-NN model, independently of the number of mappers. Nevertheless, even though the reduce phase is not very time-consuming, a higher number of mappers implies a slightly higher computation in the reduce phase because it is aggregating more results that come from the maps.
- For the considered problem, higher values of k yield better accuracy for both the original and the proposed parallel version. In terms of runtime, larger k values slightly increase the computation time in either parallel and sequential versions being that more computations have to be performed. In our particular

TABLE III. Results obtained by the MR-kNN algorithm

#Neighbors	#Maps	AvgMapTime	AvgRedTime	AvgTotalTime	AccTest	SpeedUp
1	256	356.3501	9.5156	709.4164	0.5019	149.1014
	128	646.9981	5.2700	804.4560	0.5019	131.4864
	64	1294.7913	3.1796	1470.9524	0.5019	71.9092
	32	2684.3909	2.1978	3003.3630	0.5019	35.2189
	16	5932.6652	1.6526	6559.5666	0.5019	16.1253
3	256	351.6059	20.3856	735.8026	0.4959	130.2356
	128	646.4311	11.0798	894.9308	0.4959	107.0783
	64	1294.3999	6.3078	1509.5010	0.4959	63.4830
	32	2684.9437	3.9628	3007.3770	0.4959	31.8642
	16	5957.9582	2.4302	6547.3316	0.4959	14.6361
5	256	371.1801	33.9358	793.4166	0.5280	125.7262
	128	647.6104	17.9874	842.3042	0.5280	118.4291
	64	1294.6693	10.0790	1474.5290	0.5280	67.6509
	32	2679.7405	5.6878	2977.1442	0.5280	33.5064
	16	5925.3548	3.5698	6467.3044	0.5280	15.4242
7	256	388.0878	49.3472	746.2892	0.5386	156.3386
	128	647.0609	23.6258	830.7192	0.5386	140.4492
	64	1295.0035	12.8888	1475.3190	0.5386	79.0838
	32	2689.1899	7.3508	3069.3328	0.5386	38.0128
	16	5932.6652	1.6526	6559.5666	0.5386	17.7868

implementation, it means larger matrices CD_j. However, due to the encoding we use, it does not result in very large runtimes.

- According to Figure 3, the achieved speed up is linear in most of the cases, except for the case with 256. As stated before, this case overtakes the maximum number of concurrent running map tasks. Moreover, we sometimes appreciate some superlinear speed ups that could be interpreted as memory-consumption problems of the sequential version.

V. Concluding remarks

In this contribution we have proposed a MapReduce approach to enable the k-Nearest neighbor technique to deal with large-scale problems. Without such a parallelization, the application of the k-NN algorithm would be limited to small or medium data, especially when low runtimes are a need. The proposed scheme is an exact parallelization of the k-NN model, so that, the precision remains the same and the efficiency has been largely improved. The experiments performed has shown the reduction of computational time achieved by this proposal compared to the utilization of the sequential version. As future work, we plan to carry out more extensive experiments as well as the use of more recent technologies such as Spark [13] to make the computation process even faster by using operations beyond the MapReduce philosophy.

Acknowledgment

This work was supported by the Research Projects TIN2014-57251-P and P11-TIC-7765. J. Maillo holds a research initialization scholarship from the University of Granada. I. Triguero holds a BOF postdoctoral fellowship from the Ghent University.

References

[1] M. Minelli, M. Chambers, and A. Dhiraj, *Big Data, Big Analytics: Emerging Business Intelligence and Analytic Trends for Today's Businesses (Wiley CIO)*, 1st ed. Wiley Publishing, 2013.

[2] T. M. Cover and P. E. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967.

[3] X. Wu and V. Kumar, Eds., *The Top Ten Algorithms in Data Mining*. Chapman & Hall/CRC Data Mining and Knowledge Discovery, 2009.

[4] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[5] A. Fernández, S. Río, V. López, A. Bawakid, M. del Jesus, J. Benítez, and F. Herrera, "Big data with cloud computing: An insight on the computing environment, mapreduce and programming frameworks," *WIREs Data Mining and Knowledge Discovery*, vol. 4, no. 5, pp. 380–409, 2014.

[6] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. MIT press, 1999, vol. 1.

[7] A. Srinivasan, T. Faruquie, and S. Joshi, "Data and task parallelism in ILP using mapreduce," *Machine Learning*, vol. 86, no. 1, pp. 141–168, 2012.

[8] I. Triguero, D. Peralta, J. Bacardit, S. García, and F. Herrera, "MRPR: A mapreduce solution for prototype reduction in big data classification," *Neurocomputing*, vol. 150, Part A, no. 0, pp. 331 – 345, 2015.

[9] T. Yokoyama, Y. Ishikawa, and Y. Suzuki, "Processing all k-nearest neighbor queries in hadoop," in *Web-Age Information Management*, ser. Lecture Notes in Computer Science, H. Gao, L. Lim, W. Wang, C. Li, and L. Chen, Eds. Springer Berlin Heidelberg, 2012, vol. 7418, pp. 346–351.

[10] C. Zhang, F. Li, and J. Jestes, "Efficient parallel knn joins for large data in mapreduce," in *Proceedings of the 15th International Conference on Extending Database Technology*, ser. EDBT '12. New York, NY, USA: ACM, 2012, pp. 38–49.

[11] A. H. Project, "Apache hadoop," 2015. [Online]. Available: http://hadoop.apache.org/

[12] A. M. Project, "Apache mahout," 2015. [Online]. Available: http://mahout.apache.org/

[13] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 1–14.

[14] A. Frank and A. Asuncion, "UCI machine learning repository," 2010. [Online]. Available: http://archive.ics.uci.edu/ml