A Nearest Neighbours-Based Algorithm for Big Time Series Data Forecasting

Ricardo L. Talavera-Llames[1], Rubén Pérez-Chacón[1],
María Martínez-Ballesteros[2], Alicia Troncoso[1],
and Francisco Martínez-Álvarez[1](✉)

[1] Division of Computer Science, Universidad Pablo de Olavide, 41013 Seville, Spain
{rltallla,ali,fmaralv}@upo.es, rpercha@alu.upo.es

[2] Department of Computer Science, University of Seville, Seville, Spain
mariamartinez@us.es

Abstract. A forecasting algorithm for big data time series is presented in this work. A nearest neighbours-based strategy is adopted as the main core of the algorithm. A detailed explanation on how to adapt and implement the algorithm to handle big data is provided. Although some parts remain iterative, and consequently requires an enhanced implementation, execution times are considered as satisfactory. The performance of the proposed approach has been tested on real-world data related to electricity consumption from a public Spanish university, by using a Spark cluster.

Keywords: Big data · Nearest neighbours · Time series · Forecasting

1 Introduction

Recent technological advances have led to a rapid and huge data storage. In fact, 90 % of existing data in the world have been generated in the last decade. In this context, the improvement of the current data mining techniques is necessary to process, manage and discover knowledge from this big volume of information. The modern term big data [10] is used to refer to this evolution. Sensor networks, typically associated with smart cities, are one of the main sources of big data generation and can be found in diverse areas such as energy, traffic or the environment.

The popular MapReduce paradigm [4] has been recently proposed by Google for big data parallel processing. This paradigm has been widely used by Apache Hadoop [15], which is an open source software implemented in Java and based on a distributed storage system called Hadoop Distributed File System (HDFS). However, the limitations of the MapReduce paradigm to develop iterative algorithms have promoted that other proposals emerge, such as Apache Spark [6]. Apache Spark is also an open source software project that allows the multipass computations, provides high-level operators, uses diverse languages (Java, Python, R) in addition to its own language called Scala, and finally, offers the machine learning library MLlib [5].

F. Martínez-Álvarez et al. (Eds.): HAIS 2016, LNAI 9648, pp. 174–185, 2016.
DOI: 10.1007/978-3-319-32034-2_15

In this work, an efficient forecasting algorithm for big data is introduced. The proposed method is based on the well-known nearest neighbours techniques [3] in machine learning. This choice is due to the good results reported when applied to datasets of small or moderate size. The algorithm has been developed in the framework Apache Spark under the Scala programming language. The algorithm has been tested on real–world big datasets, namely energy consumption, collected from a sensor network located in several buildings of a public university.

The rest of the paper is structured as follows. Section 2 describes a review of the existing literature related to the nearest neighbours algorithms for time series forecasting and to the different approaches of the nearest neighbours for big data published in recent years. In Sect. 3 the proposed method based on nearest neighbours to forecast big data time series is presented. Section 4 presents the experimental results corresponding to the prediction of the energy consumption coming from a sensor network of building facilities. Section 5 closes the paper giving some final conclusions.

2 Related Work

Predicting the future has fascinated human beings since its early existence. Accurate predictions are essential in economical activities as remarkable forecasting errors in certain areas may incur large economic losses. Therefore, many of these efforts can be noticed in everyday events such as energy management, natural disasters, telecommunications, pollution, and so forth.

The methods for time series forecasting can be roughly classified as follows: classical Box and Jenkins-based methods such as ARMA, ARIMA, ARCH or GARCH [1] and data mining techniques (the reader is referred to [9] for a taxonomy of these techniques applied to energy time series forecasting). Namely, data mining techniques based on the k nearest neighbours (kNN) have been successfully applied, providing competitive results [7,8,14]. However, these methods cannot be applied when big time series have to be predicted due to the high computational cost of the kNN.

Consequently, several MapReduce-based approaches to address the kNN algorithm in big data scenarios have been recently proposed. The authors in [17] study parallel kNN joins in a MapReduce programming model that involves both the join and the NN search to produce the k nearest neighbours of each point in a new dataset from an original dataset. In particular, both exact (H-BRJ) and approximate (H-zkNNJ) algorithms are proposed to perform efficient parallel kNN joins on big data. In [11], an algorithm is proposed to address the problem of the fast nearest neighbour approximate search of binary features in high dimensional spaces using the message passing interface (MPI) specification. A MapReduce-based framework focused on several instance reduction methods is proposed in [13] to reduce the computational cost and storage requirements of the kNN classification algorithm.

In the context of this work, the kNN query is usually required in a wide range of sensor network applications. In fact, authors in [16] propose a MapReduce-based algorithm to generalize the spatio-textual kNN join in large-scale data.

This algorithm aims at searching text-similar and k-nearest sensors to a query set containing more than one query point.

Furthermore, other distributed architectures such as GPU have been used to address parallel versions of the kNN algorithm [2], also allowing a fast and scalable meta-feature generation for highly dimensional and sparse datasets.

Alternatively, the MLlib does not include any Spark implementation of the kNN algorithm, in spite of providing several traditional algorithms such as k-means, decision tree, among others. Thus, several parallel implementations of the kNN algorithm have been proposed in the literature. For instance, the work presented in [12] provides a Spark implementation in Java of the kNN and the SVM-Pegasos algorithms to compare the scalability of this parallelization technology with the MPI/OpenMP on a Beowulf cluster architecture. This kNN Spark implementation maps the Euclidean distance from each training sample to a test sample.

3 Methodology

This section describes the methodology proposed in order to forecast big data time series. In particular, Sect. 3.1 introduces the methodology itself and in Sect. 3.2 how it is implemented to be used in Spark.

3.1 Time Series Forecasting Based on Nearest Neighbours

This section describes the technique applied to time series forecasting based on the kNN algorithm.

Given the electricity consumption recorded in the past, up to c_i, the problem consists in predicting the h consecutive measures for electricity consumption (note that h is the prediction horizon).

Let $C_i \in \mathbb{R}^h$ be a vector composed of the h values to be predicted:

$$C_i = [c_{i+1}, c_{i+2}, \ldots, c_{i+h}] \tag{1}$$

Then, the associated vector $CC_i \in \mathbb{R}^w$ is defined by gathering the consumption contained in a window composed of w consecutive samples, from values of the vector C_i backwards, as follows:

$$CC_i = [c_{i-w+1}, c_{i-w+2}, \ldots, c_{i-1}, c_i] \tag{2}$$

For any couple of vectors, CC_i and CC_j, a distance can be defined as:

$$\text{dist}(i, j) = \|CC_i - CC_j\| \tag{3}$$

where $\|\cdot\|$ represents a suitable vector norm (the Euclidean norm has been used in this work).

The weighted nearest neighbours (WNN) method first identifies the k nearest neighbours of vector CC_i, where k is a number to be determined and *neighbourhood* in this context is measured according to (3) as afore mentioned. This leads to the neighbour set, NS:

$$NS = \{\text{set of } k \text{ indexes, } q_1, \ldots, q_k, \text{ such that } CC_{q_j} \text{ closest to vector } CC_i\} \tag{4}$$

in which q_1 and q_k refer to the first and k-th neighbours respectively, in order of distance.

According to the WNN methodology, the h electricity consumptions are predicted by linearly combining the consumptions of the k vectors succeeding those in NS, that is,

$$C_i = \frac{1}{\sum_{j=1}^{k} \alpha_j} \cdot \sum_{j=1}^{k} \alpha_j C_{q_j} \tag{5}$$

where the weighting factors α_j are obtained from,

$$\alpha_j = \frac{1}{(\text{dist}(CC_{q_j}, CC_i))^2} \tag{6}$$

Obviously, α_j when $j = k$ (furthest neighbour) is lesser than α_j when $j = 1$ (nearest neighbour). Note also that, although the w consumptions contained in CC_i are used to determine the nearest neighbours, only the h consumptions of the vectors C_{q_j} are relevant in determining C_i.

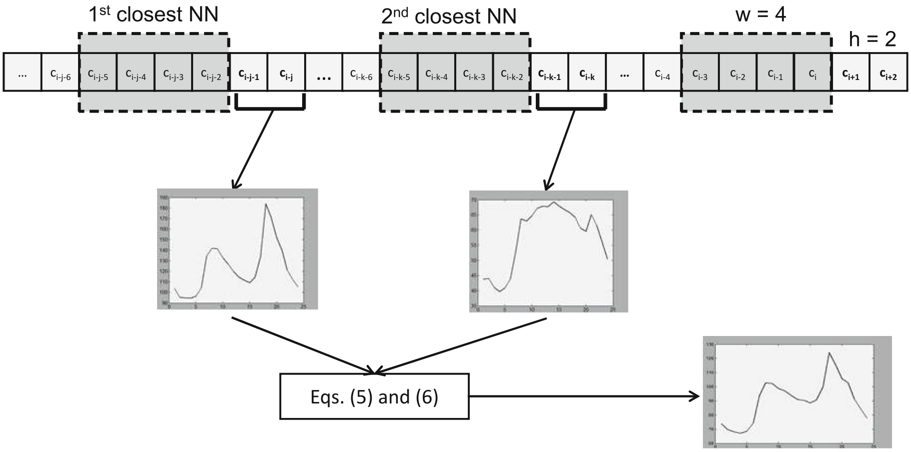


Fig. 1. Illustration of the WNN approach.

In order to find candidate neighbours, q_j, a window of w samples is simply slid along the entire dataset.

Figure 1 illustrates the basic idea behind the WNN algorithm, with $k = 2$ and $w = 4$. Values c_{i+1} and c_{i+2} ($h = 2$) are the target prediction. As $w = 4$, values $[c_{i-3}, c_{i-2}, c_{i-1}, c_i]$ are chosen as window. Later, minimal distances calculated according to Eq. (3) are searched for in the historical data. Sequences of values $s_1 = [c_{i-j-5}, ..., c_{i-j-2}]$ and $s_2 = [c_{i-k-5}, \ldots, c_{i-k-2}]$ are identified as the two nearest neighbours. In particular, s_2 is closer to w than s_1, and would therefore be denoted as q_2 and q_1, respectively. Finally, the forecast is performed by considering the h next samples to s_1 and s_2, according to Eqs. (5) and (6).

3.2 Algorithm Implementation for Apache Spark

The algorithm described in Sect. 3.1 has been implemented for Apache Spark, making the most of the RDD variables of Spark, in order to use it in a distributed way. This strategy makes the analysis of the datasets more efficient and faster. Therefore, every RDD created is split in blocks of the same size across the nodes that integrate the cluster, as it is shown in Fig. 2.

For a proper execution of the algorithm, several variables have to be defined from the beginning. These are:

1. The initial time series to be analysed.
2. The size of the window w whose values are taken as a pattern to look for the nearest neighbours.
3. The number of values h that needs to be predicted.
4. The number of nearest neighbours k that are going to be selected.

Since overwriting RDD variables cannot be done in Spark, a new RDD is created in each step of the algorithm. Hence, every time this section refers to a new transaction it means that a new RDD is being created.

Firstly, the data is loaded in Spark, split into different fields and finally just the energy consumption is selected, as shown in Fig. 3(a). An extra field with a numeric index is also added in this transaction. So the initial dataset in Spark is a RDD with just two fields, identification number with the position of the value of the time series, and the consumption itself (see Fig. 3(b)). Remark that, as

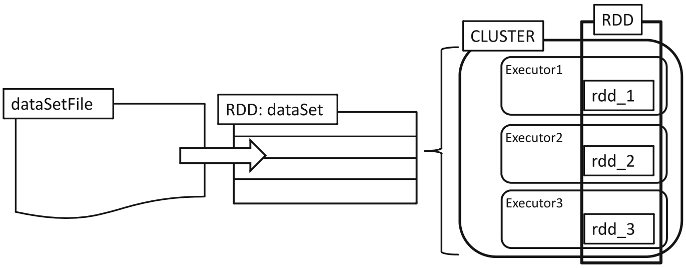
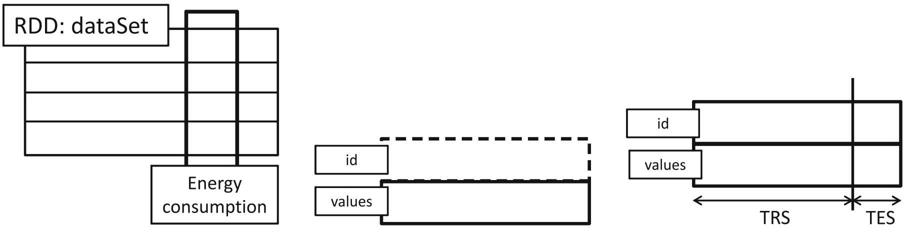


Fig. 2. Creation of a RDD variable in Spark and how it is managed in a cluster.

before mentioned, this data is split automatically across the nodes of the cluster. In a second transaction, the previous dataset is separated in two subsets, test set (TES) and training set (TRS), as Fig. 3(c) shows. TRS will be used to train the algorithm, whereas TES will be used to predict results and to check the accuracy of the prediction, comparing each predicted value to the actual one.



(a) Data selection. (b) Including an *id*. (c) TRS and TES.

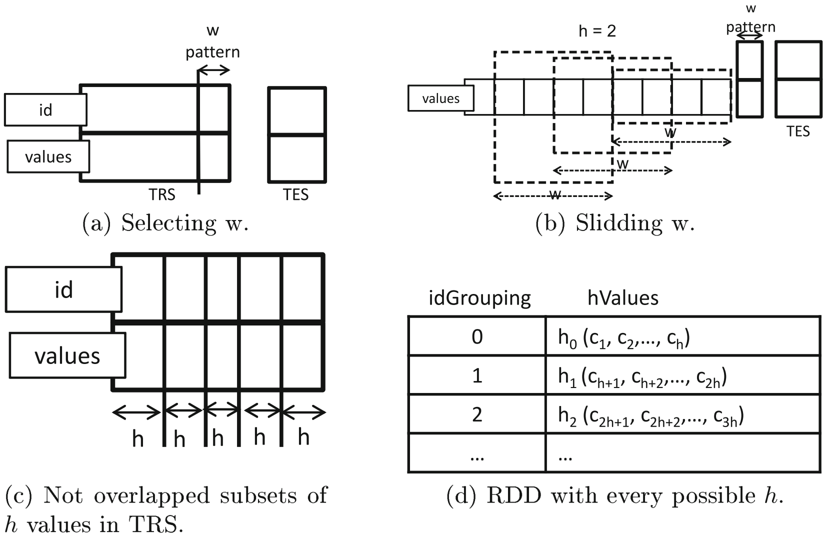
Fig. 3. Data preprocessing.

The next transaction only uses TRS for training purposes. Therefore, the w previous values to the h values to be predicted are selected as the pattern in this iteration, as depicted in Fig. 4(a). Now, the main goal is to store every possible subset of w values that can be formed of which their h future values are known. To achieve this, the windows w has to be shifted h values from the end of the time series, and from there, the w subsequent values are selected. Next iteration will repeat the same process as is illustrated in Fig. 4(b). For the following transaction, the TRS is divided in subsets of h values, as shown in Fig. 4(c). Thanks to map transformations that Spark provides to its RDD, this is done in one instruction all over the RDD located in the cluster. The key in this transaction is to group values just in one action without doing several iterations like it would have been done in other languages (Java 8's Stream is, perhaps, the sole exception). In this case, the RDD from the previous transaction is grouped by the rule id/h. As a result, the new RDD will contain a numeric *id* of the subsets following by their corresponding h values. This can be seen in Fig. 4(d), where *idGrouping* is the numeric *id* of each subset and h_i represents each subset of h values of the time series. In particular,

$$h_i = [c_{i \cdot h+1}, c_{i \cdot h+2}, ..., c_{(i+1) \cdot h}] \tag{7}$$

For instance, in the figure, h_0 is formed by the h values whose index *id* divided by h is 0, that is, $[c_1, c_2, ..., c_h]$.

As the formation of each subset of w values depends on the h_i previous subsets, a new RDD will be created with these subsets focusing on the RDD from the transaction before. So in this transaction, a dataset is also formed with the subsets of w values as well as the numeric *id* that matches with the RDD that contains the subsets h_i. This is represented in Fig. 5(a), where *idGrouping*



(a) Selecting w.

(b) Slidding w.

(c) Not overlapped subsets of h values in TRS.

idGrouping	hValues
0	h_0 (c_1, c_2,..., c_h)
1	h_1 (c_{h+1}, c_{h+2},..., c_{2h})
2	h_2 (c_{2h+1}, c_{2h+2},..., c_{3h})
...	...

(d) RDD with every possible h.

Fig. 4. kNN in Spark, phase 1.

is the numeric *id* of the subset and w_i represents each subset of w previous values to the h values of the subset h_i.

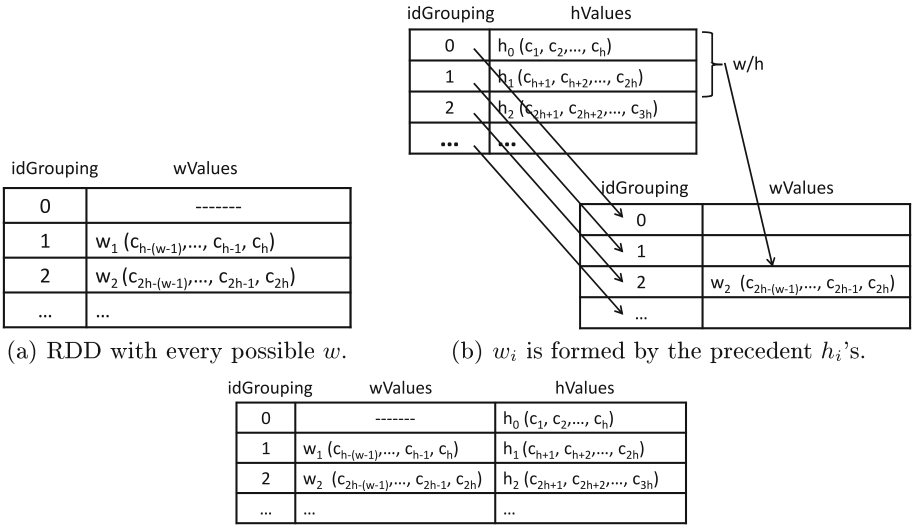
$$w_i = [c_{i \cdot h-(w-1)}, ..., c_{i \cdot h-1}, c_{i \cdot h}] \tag{8}$$

Due to the size of the windows w is usually greater than h, it can be observed that w_i cannot be defined when it does not exist w consecutive values previous to the values of h_i. For the same reason, it would be necessary not to look at just the values of the subset h_{i-1}, but in the w/h subsets before to build the subset w_i. This can be seen in Fig. 5(b), where the values for the w_2, for instance, are going to be formed by the values of the w/h previous h_i, in this case, h_0 and h_1.

In the next transformation, both RDDs need to be joined. Again, and thanks to the fact that both datasets share the same numeric *id*, Spark allows to do so by a simple action, obtaining a new RDD with the grouping id, the w values of the time series and the h values that follows it. This transaction can be seen in Fig. 5(c).

At this time the pattern w is compared to each w_i, obtaining the new field distance, which is calculated by the Euclidean distance and added to the previous RDD with just one action over Spark. Thus, the new dataset will contain the numeric id *idGrouping*, the w_i, the h_i and the distance d_i between the w pattern and w_i. This is shown in Fig. 6(a).

The next step of the algorithm sorts the previous RDD according to the distance, which Spark does rapidly just indicating the field for which the whole RDD is going to be sorted.



idGrouping	wValues
0	-------
1	w_1 ($c_{h-(w-1)}$,…, c_{h-1}, c_h)
2	w_2 ($c_{2h-(w-1)}$,…, c_{2h-1}, c_{2h})
…	…

(a) RDD with every possible w.

idGrouping	hValues
0	h_0 (c_1, c_2,…, c_h)
1	h_1 (c_{h+1}, c_{h+2},…, c_{2h})
2	h_2 (c_{2h+1}, c_{2h+2},…, c_{3h})
…	…

idGrouping	wValues
0	
1	
2	w_2 ($c_{2h-(w-1)}$,…, c_{2h-1}, c_{2h})
…	

(b) w_i is formed by the precedent h_i's.

idGrouping	wValues	hValues
0	-------	h_0 (c_1, c_2,…, c_h)
1	w_1 ($c_{h-(w-1)}$,…, c_{h-1}, c_h)	h_1 (c_{h+1}, c_{h+2},…, c_{2h})
2	w_2 ($c_{2h-(w-1)}$,…, c_{2h-1}, c_{2h})	h_2 (c_{2h+1}, c_{2h+2},…, c_{3h})
…	…	…

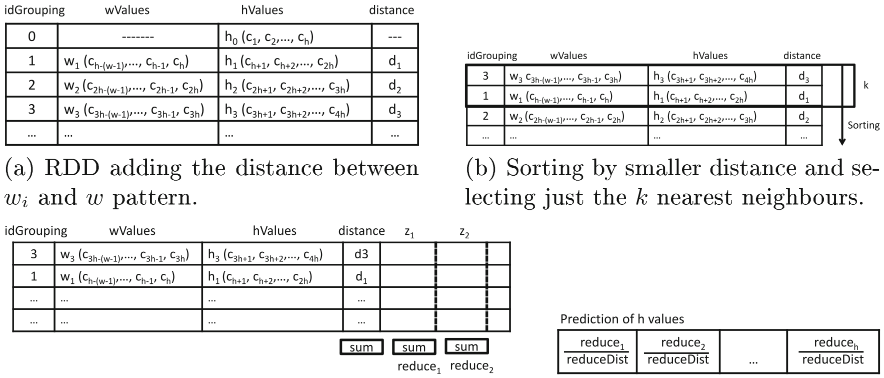
(c) RDD with joint w_i and h_i.

Fig. 5. kNN in Spark, phase 2.

After that, just the k nearest neighbours will be chosen. This is explained in Fig. 6(b), where k is the number of nearest neighbours to be selected.

The next transaction will calculate the prediction, applying the formulas (5) and (6). In Spark, before the value is predicted, it is necessary to do intermediate calculations. First, each value of each h_i is divided by the square distance. This is done in Spark with a new map transformation, adding a new field with the values of $h_i/(d_i)^2$. And finally, each of these fields needs to be summed, which in Spark is done with a reduce action over the field obtaining a number. This is illustrated in detail in Fig. 6(c), where the new columns z_j represents the division of the j-th value of each h_i between its square distance $(d_i)^2$, *sum* represents the sum of each column and $reduce_j$ is the name of the variable that gather that number. So first column z_1 will be the division of each $c_{i \cdot h+1}$ of each h_i between the distance $(d_i)^2$, then it will be summed and saved in the variable $reduce_1$. Then, it is just necessary to divide each sum of each field with the sum of the inverse of the square distance (reduceDist variable), obtaining the h values predicted as shown in Fig. 6(d).

Once all the predictions for the h values are made, the process begins again to obtain the following h forecasts, but this time updating the TRS, as shown in Fig. 7(a), where TRS *old* represents the initial TRS; and TRS *new* the new one including the previous TRS and the h real values previously predicted. The algorithm will then stop when the total predictions have the same size as the TES. This can be seen in Fig. 7(b). The final step lies in comparing the prediction with the real values in TES applying the formula of the mean relative error, defined in Eq. (9), as shown in Fig. 7(c).



idGrouping	wValues	hValues	distance
0	-------	h_0 (c_1, c_2,..., c_h)	---
1	w_1 ($c_{h-(w-1)}$,..., c_{h-1}, c_h)	h_1 (c_{h+1}, c_{h+2},..., c_{2h})	d_1
2	w_2 ($c_{2h-(w-1)}$,..., c_{2h-1}, c_{2h})	h_2 (c_{2h+1}, c_{2h+2},..., c_{3h})	d_2
3	w_3 ($c_{3h-(w-1)}$,..., c_{3h-1}, c_{3h})	h_3 (c_{3h+1}, c_{3h+2},..., c_{4h})	d_3
...	...	...	...

(a) RDD adding the distance between w_i and w pattern.

idGrouping	wValues	hValues	distance
3	w_3 $c_{3h-(w-1)}$,..., c_{3h-1}, c_{3h})	h_3 (c_{3h+1}, c_{3h+2},..., c_{4h})	d_3
1	w_1 ($c_{h-(w-1)}$,..., c_{h-1}, c_h)	h_1 (c_{h+1}, c_{h+2},..., c_{2h})	d_1
2	w_2 ($c_{2h-(w-1)}$,..., c_{2h-1}, c_{2h})	h_2 (c_{2h+1}, c_{2h+2},..., c_{3h})	d_2
...	...	...	...

(b) Sorting by smaller distance and selecting just the k nearest neighbours.

idGrouping	wValues	hValues	distance	z_1	z_2	
3	w_3 ($c_{3h-(w-1)}$,..., c_{3h-1}, c_{3h})	h_3 (c_{3h+1}, c_{3h+2},..., c_{4h})	d3			
1	w_1 ($c_{h-(w-1)}$,..., c_{h-1}, c_h)	h_1 (c_{h+1}, c_{h+2},..., c_{2h})	d_1			
...	...	...	...			
...	...	...	...			

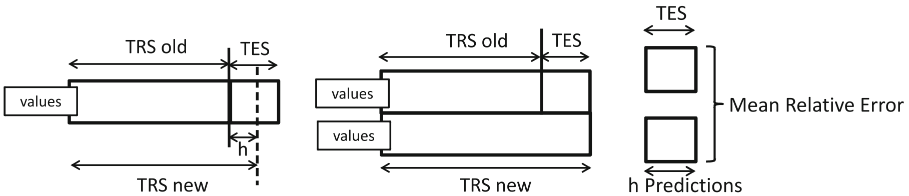
(c) RDD adding suboperations as columns before doing reduce sums actions.

Prediction of h values

$\frac{reduce_1}{reduceDist}$	$\frac{reduce_2}{reduceDist}$	...	$\frac{reduce_h}{reduceDist}$

(d) Computation of the prediction.

Fig. 6. kNN in Spark, phase 3.



(a) TRS updating. (b) Algorithm finishing. (c) MRE calculation.

Fig. 7. kNN in Spark, phase 4.

4 Results

This section presents the results obtained from the application of the proposed methodology to electricity consumption big data time series. Hence, Sect. 4.1 describes the used datasets. The experimental setup carried out is detailed in Sect. 4.2. Finally, the results are discussed in Sect. 4.3.

4.1 Datasets Description

The datasets used are related to the electrical energy consumption in two buildings located at a public university for years 2011, 2012 and 2013. The consumption is measured every fifteen minutes during this period. This makes a total of 35040 instances for years 2011 and 2013, and 35136 for the year 2012. TES consists of the the last three months of 2013 for both datasets.

Note that there were several missing values ($< 3\,\%$). In particular, some values are equal to 0. However, subsequent time stamps store the accumulated consumption for such instances. Therefore, the cleaning process consisted in searching for such 0 values and assuming that consumption had been constant during these periods of time. That is, the stored value after zeros is divided by the number of consecutive registered zeros and assigned to each one.

4.2 Design of Experiments

The proposed algorithm requires several variables to be executed. Since this is a preliminary study, an exhaustive analysis of best values has not been carried out and the following considerations have been taken into account:

1. The size of the window (w) has been set to 96, which represents all the values for a whole day.
2. As for the forecast horizon (h), it was firstly set to $h = 48$ (12 h) and secondly to $h = 96$ (24 h).
3. The number of nearest neighbours (k) varies from one to five.

The algorithm has been executed using each dataset described in Sect. 4.1 varying the aforementioned parameter settings (w, h and k). In short, each dataset has been executed 10 times.

To evaluate the runtime costs of the algorithm, each complete experimentation for each dataset has been executed using two processors, summing 20 executions in total.

The experimentation has been launched on a cluster, which consists of 2 Intel Xeon E7-4820 processors at 2 GHz, 18 MB cache, 8 cores per processor and 64 GB of main memory working under Linux Ubuntu. The cluster works with Apache Spark 1.4.1 and Hadoop 2.3.

Finally, in order to assess the performance of the algorithm, the well-known mean relative error (MRE) measure has been selected. Its formula is:

$$MRE = \frac{1}{N} \sum_{i=1}^{N} \frac{|v_{pred} - v_{actual}|}{v_{actual}} \tag{9}$$

where v_{pred} stands for the predicted values and v_{acutal} for the actual consumption values.

4.3 Electricity Consumption Big Data Time Series Forecasting

This section shows the results of applying the methodology proposed in Sect. 3.1 to the datasets described in Sect. 4.1 over the cluster described in Sect. 4.2. The algorithm has been tested on the last three months in the year 2013, for both buildings, resulting in 8832 forecasted instances.

Table 1 summarizes the results obtained for the first building. Analogously, Table 2 shows the results obtained for the second building. Note that the column *Duration* collects execution times in minutes. The values for the rightmost columns show the MRE associated with each k.

It can be noticed that to facilitate future comparative analysis, only two processors have been used. Additionally, horizons of prediction has been set to

Table 1. Electricity consumption forecasting for the first building.

w	h	Duration	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
96	48	143.99	0.3184	0.2902	0.3025	0.3132	0.3090
96	96	53.00	0.4653	0.4243	0.4530	0.4773	0.4708

Table 2. Electricity consumption forecasting for the second building.

w	h	Duration	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
96	48	143.79	0.3052	0.2749	0.2870	0.2912	0.3030
96	96	53.13	0.4807	0.4159	0.4407	0.4452	0.4686

48 (12 h) and 96 samples (24 h), thus representing usual short-term forecasting horizons in electricity consumptions.

It can be seen in both tables that the execution time varies along with the size of the h, up to 66 %. This is because the higher the value of h is, the lesser number of distances has to be computed. It can also be noticed that small values of k do not make significant difference to the accuracy of the predictions. In fact it just changes from one k to another by 5 %. Farthest studies need to be carried out to select the optimal k.

It must be admitted that the execution time is expected to be improved in future versions. This is partly due to the fact that the calculation of every w_i is the only part of the algorithm which is not made in a parallelized way, but in an iterative way. Since previous h_i must be checked to compute w_i, in some cases, two subsets w_i and w_j may have values from the same h_i. This means that for every w_i, all the previous h_i's need to be individually checked, and just the ones after it are discarded. In short, a formula that creates every w_i from the original time series following a MapReduce schema have not been found so far. Obviously, future research will address this issue.

5 Conclusions

An algorithm to forecast big data time series has been proposed. In particular, the algorithm is based on the weighted nearest neighbours paradigm. This work describes how to design it in Spark. It also provides results for real-world time series, e.g. electricity consumption for several buildings at a public university. The implementation has been launched on a 2-processor cluster generating satisfactory results in terms of both MRE and execution time. Future work is directed in integrating the code in the Spark MLlib as well as in reducing its computational cost.

Acknowledgments. The authors would like to thank the Spanish Ministry of Economy and Competitiveness, Junta de Andalucía, Fundación Pública Andaluza Centro de Estudios Andaluces and Universidad Pablo de Olavide for the support under projects TIN2014-55894-C2-R, P12-TIC-1728, PRY153/14 and APPB813097, respectively.

References

1. Box, G., Jenkins, G.: Time Series Analysis: Forecasting and Control. John Wiley and Sons, Hoboken (2008)
2. Canuto, S., Gonçalves, M., Santos, W., Rosa, T., Martins, W.: An efficient and scalable metafeature-based document classification approach based on massively parallel computing. In: Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 333–342 (2015)
3. Cover, T.M., Hart, P.E.: Nearest neighbor pattern classification. IEEE Trans. Inf. Theor. **13**(1), 21–27 (1967)
4. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
5. Machine Learning Library (MLlib) for Spark (2015). http://spark.apache.org/docs/latest/mllib-guide.html
6. Hamstra, M., Karau, H., Zaharia, M., Knwinski, A., Wendell, P.: Learning Spark: Lightning-Fast Big Analytics. O' Really Media, Sebastopol (2015)
7. Martínez-Álvarez, F., Troncoso, A., Riquelme, J.C., Aguilar, J.S.: Discovery of motifs to forecast outlier occurrence in time series. Pattern Recogn. Lett. **32**, 1652–1665 (2011)
8. Martínez-Álvarez, F., Troncoso, A., Riquelme, J.C., Aguilar, J.S.: Energy time series forecasting based on pattern sequence similarity. IEEE Trans. Knowl. Data Eng. **23**, 1230–1243 (2011)
9. Martínez-Álvarez, F., Troncoso, A., Asencio-Cortés, G., Riquelme, J.: A survey on data mining techniques applied to electricity-related time series forecasting. Energies **8**(11), 12361 (2015)
10. Minelli, M., Chambers, M., Dhiraj, A.: Big Data, Big Analytics: Emerging Business Intelligence and Analytics Trends for Today's Businesses. John Wiley and Sons, Hoboken (2013)
11. Muja, M., Lowe, D.G.: Scalable nearest neighbor algorithms for high dimensional data. IEEE Trans. Pattern Anal. Mach. Intell. **36**(11), 2227–2240 (2014)
12. Reyes-Ortiz, J.L., Oneto, L., Anguita, D.: Big data analytics in the cloud: spark on hadoop vs MPI/OpenMP on beowulf. Procedia Comput. Sci. **53**, 121–130 (2015)
13. Triguero, I., Peralta, D., Bacardit, J., García, S., Herrera, F.: MRPR: a mapreduce solution for prototype reduction in big data classification. Neurocomputing **150**, 331–345 (2015)
14. Troncoso, A., Riquelme, J.C., Riquelme, J.M., Martínez, J.L., Gómez, A.: Electricity market price forecasting based on weighted nearest neighbours techniques. IEEE Trans. Power Syst. **22**(3), 1294–1301 (2007)
15. White, T.: Hadoop, The Definitive Guide. O' Really Media, Sebastopol (2012)
16. Yang, M., Zheng, L., Lu, Y., Guo, M., Li, J.: Cloud-assisted spatio-textual k nearest neighbor joins in sensor networks. In: Proceedings of the Industrial Networks and Intelligent Systems, pp. 12–17 (2015)
17. Zhang, C., Li, F., Jestes, J.: Efficient parallel kNN joins for large data in mapreduce. In: Proceedings of the International Conference on Extending Database Technology, pp. 38–49 (2012)