Chatzigeorgakidis *et al. J Big Data (2018) 5:4*
https://doi.org/10.1186/s40537-018-0115-x

Journal of Big Data

RESEARCH Open Access

FML-kNN: scalable machine learning on Big Data using *k*-nearest neighbor joins

Georgios Chatzigeorgakidis[1,2*], Sophia Karagiorgou[3], Spiros Athanasiou[2] and Spiros Skiadopoulos[1]

*Correspondence:
chgeorgakidis@uop.gr
[1] Department of Informatics and Telecommunications, University of Peloponnese, Karaiskaki 70, 22100 Tripolis, Greece
Full list of author information is available at the end of the article

Abstract

Efficient management and analysis of large volumes of data is a demanding task of increasing scientific and industrial importance, as the ubiquitous generation of information governs more and more aspects of human life. In this article, we introduce FML-kNN, a novel distributed processing framework for Big Data that performs probabilistic classification and regression, implemented in Apache Flink. The framework's core is consisted of a *k*-nearest neighbor joins algorithm which, contrary to similar approaches, is executed in a single distributed session and is able to operate on very large volumes of data of variable granularity and dimensionality. We assess FML-*k*NN's performance and scalability in a detailed experimental evaluation, in which it is compared to similar methods implemented in Apache Hadoop, Spark, and Flink distributed processing engines. The results indicate an overall superiority of our framework in all the performed comparisons. Further, we apply FML-*k*NN in two motivating uses cases for water demand management, against real-world domestic water consumption data. In particular, we focus on forecasting water consumption using 1-h smart meter data, and extracting consumer characteristics from water use data in the shower. We further discuss on the obtained results, demonstrating the framework's potential in useful knowledge extraction.

Keywords: Big Data, Distributed processing, MapReduce, Data mining, Machine learning, Classification, Regression

Introduction

During the past few years, new database management and distributed computing technologies have emerged to satisfy the need for systems that can efficiently store and operate on massive volumes of data. *MapReduce* [1], a distributed programming model which maps operations on data elements (i.e., *mappers*) to several machines (i.e., *reducers*), set the foundation for this technology trend. This laid the groundwork for the development of open source distributed processing engines such as the Apache *Hadoop*, *Spark* and *Flink* [2], that efficiently implement and extend MapReduce. These engines offer a handful of tools that operate on Big Data stored in distributed storages, supported by distributed file systems such as the *Hadoop Distributed File System* (HDFS). Among them, Flink provides a mechanism for automatic procedure optimization and exhibits better performance on iterative distributed algorithms [3]. It also exhibits better overall performance, as it processes tasks in a pipelined fashion [4]. This allows the concurrent

execution of successive tasks, i.e., a reducer can start executing as soon as it receives it's input from a mapper, without requiring all the mappers to finish first.

Data analysis and knowledge extraction from Big Data collections is often performed by applying specific machine learning techniques. The simplicity along with the effectiveness of the *k-nearest neighbors* (*k*NN) algorithm, have motivated many research communities over the years with numerous applications and scientific approaches which exploit or improve its potential, especially in spatial databases and data mining. Of particular interest are the *k*NN *joins* methods [5], which retrieve the nearest neighbors of *every* element in a testing dataset (R) from a set of elements in a training dataset (S). Each data element consists of several *features*, which constitute the preliminary knowledge on which the neighbor retrieval is conducted. However, computing *k*NN joins on vast amounts of data can be very time consuming when conducted by a single CPU, as it requires computing *k*NN for each element in dataset R. Additionally, a possible extension of such methods to perform machine learning tasks such as classification or regression, magnifies the complexity. Various studies have been also carried out towards *approximate* solutions of *k*NN, where there is a trade-off between the algorithm's precision and complexity.

In this work we introduce a framework of methods for scalable data analysis and mining on Big Data collections. We present the *Flink Machine Learning kNN* (FML-*k*NN for short) framework which implements a probabilistic classifier and a regressor. It's core algorithm is an extension of F-*zk*NN [6], which is built upon an optimized version of the H-*zk*NNJ [7] distributed approximate *k*NN joins algorithm. In particular, the overall contributions of our work are as follows:

- We propose a novel, easily extendible distributed processing framework that performs probabilistic classification and regression using *k*NN joins.
- The framework operates in a single distributed session, saving I/O and communication resources. Similar approaches require three distributed sessions.
- We optimize the execution of the first processing stage using Flink's *broadcast sets*, contrary to F-*zk*NN, which performs expensive dataset propagation.
- We present a detailed experimental and comparative evaluation with similar approaches and exhibit our framework's efficiency in terms of scalability and wall-clock completion time.
- We conduct experiments on two real-world cases using water consumption related datasets and extract useful knowledge and insights towards the induction of more efficient water use.

The remainder of this work is organized as follows. "Related work" section reviews related work on similar approaches. "Preliminaries" section presents some preliminaries and essential basic concepts. "Methods" section presents FML-*k*NN. In "Results and discussion" section, we evaluate the framework's methods in terms of wall-clock completion time and scalability. We also present and discuss two case studies on knowledge extraction tasks over large amounts of water consumption data. Finally, "Conclusions" section concludes the paper and outlines our future research directions.

Related work

Various approaches leverage the potential of machine learning methods for knowledge extraction. Our work resides in the field of distributed computation of the *k*-nearest neighbors joins over Big Data and its extension to perform several machine learning tasks. In the following, we present a review of the relevant literature.

Several works in the literature have reported the superiority of *k*-nearest neighbors on machine learning tasks over similar approaches, both in terms of processing time, as well as in terms of accuracy [8–10]. There are numerous approaches that exploit its potential. Zhang et al. [11] introduced a multi-label classification method based on *k*NN which uses a *Maximum a Posteriori* (MAP) principle to predict the class of an new element. Oswald et al. [12] used an approximate *k*NN-based regression approach in order to forecast traffic flow. Wei and Keogh [13] presented that an 1NN classifier outperforms other similar methods in terms of error rate, when applied on time-series data. Xu [14] introduced a multi-label weighted *k*NN classifier, where the weights for each class are computed via mathematical optimization, using least squared errors (LSE). Gou et al. [15] presented a weighted voting scheme for such classifiers, where the distance between an element and its nearest neighbors determines the weight of each neighbor's vote. The query element is classified to the most weighted class. In our approach, we extend this functionality by also calculating the probability of each element belonging to each class. Also, by employing a *k*-nearest neighbors joins approach, we allow for simultaneous classification or regression on datasets of really high volume, addressing the challenges that arise when processing Big Data in a distributed environment.

Similarly to many data analysis and management tasks, *k*NN joins suffer from the *curse of dimensionality* [16]. Liao et al. [17] stated that as the number of dimensions increases, such techniques need an exponentially larger amount of CPU time. Consequently, executing a *k*NN joins method on Big Data requires prohibitively long-lasting operations. To overcome this issue, *dimensionality reduction* can be applied on the datasets by indexing their elements via a *Space Filling Curve* (SFC) [18]. This approach reduces data dimensionality to one dimension, which allows for a significantly faster execution of an approximate nearest neighbor search, based on the indexed elements. The most widely used SFCs are the *z-order*, *Gray-code* and *Hilbert*, among which Mokbel et al. [19] concluded that Hilbert is the most "fair", due to the fact that two consecutive points in the curve are always nearest neighbors. Yao et al. [20] used the *z*-order curve to significantly boost the query performance over huge amounts of data. Lawder et al. [21] efficiently executed range queries in data indexed by the Hilbert curve, while Faloutsos [22] presented a mathematical model for indexing the multi-attribute records of a data collection, using Gray-codes instead of binary values. Similarly, Chatzigeorgakidis et al. [6] and Zhang et al. [7] exploit the *z*-order curve in order to perform *k*NN joins on a distributed environment. To support selection variety, the proposed framework can operate by tuning and using the most preferable among these SFC methods, as space traversal quality and computation performance may differ according to the data context. Figure 1 shows an example of the recursive way the three SFCs scan the elements in a two-dimensional space.

The increasing scientific interest in the Big Data area has introduced modern distributed implementations of famous algorithms. More particularly, several approaches of

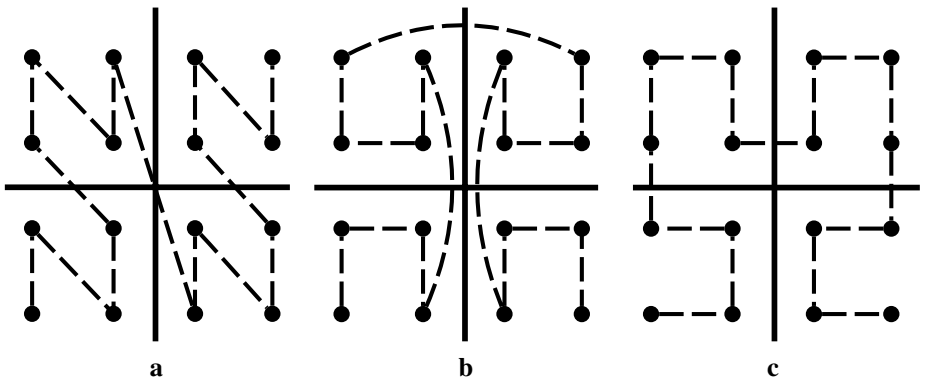

Fig. 1 The three Space Filling Curves. More particularly, **a** shows the z-order, **b** shows the Gray-code and **c** shows the Hilbert curve

MapReduce-based *k*NN joins algorithms have been proposed. Song et al. [23] present a review of the most efficient among them, denoting that all share the same three processing stages, i.e., (i) *data pre-processing*, (ii) *data partitioning and organization*, and (iii) *kNN computation* stage. They conclude that the SFC-based H-*zk*NNJ [7] algorithm, outperforms in terms of completion time other similar methods like *RankReduce* [24], which uses *Locality Sensitive Hashing* (LSH) [25]. Chatzigeorgakidis et al. [6] extended the functionality of H-*zk*NNJ [7], by delivering the F-*zk*NN probabilistic classifier, which performs classification on the results of a *k*NN joins query. The F-*zk*NN probabilistic classifier is based on the MapReduce programming model and executed in a single distributed session. However, the datasets need to be propagated between the first two execution stages using local caches, which results in increased communication costs. Our approach optimizes F-*zk*NN by avoiding this costly propagation using Flink's broadcast sets and augments its functionality by incorporating an additional regression analysis operation. Both the probabilistic classifier and the regressor are included in a wider distributed processing framework and are experimentally applied on water consumption related Big Data analysis tasks.

Similar approaches which apply machine learning methods on resource consumption data (e.g., energy, water) but still not from a Big Data perspective, include the work of Chen et al. [26] who used a *k*NN classification method and labeled water data to identify water usage. Naphade et al. [27] and Silipo and Winters [28] focused on identifying water and energy consumption patterns, providing analytics and predicting future consumptions but in high granularity levels and in small scale. Schwarz et al. [29] used *k*NN for classification and short-term prediction in energy consumption by using smart meter data, however, they focused on in-memory and non-distributed approaches, thus, limiting the applicability on larger datasets. Our approach partially relates to the work of Kermany et al. [30], where the *k*NN algorithm was applied for classification on water consumption data, in order to detect irregular consumer behavior. We take a step forward by applying our algorithm on two real-world case studies, delivering predictive analytics for water consumption in a Big Data scale.

Preliminaries

In the following, we present basic concepts regarding classification and regression based on *k*NN joins, as well as methods for dimensionality reduction, essential for the implementation of FML-*k*NN. We also briefly describe Apache Flink, the distributed processing engine that we used.

Classification

A *k*NN joins classifier algorithm categorizes new elements in a *testing* dataset (R). It detects the nearest neighbors of each elements in a *training* dataset (S) via a similarity measure, expressed by a distance function (i.e., Euclidean, Manhattan, Minkowski). In FML-*k*NN we used the Euclidean distance, through which, for each query element in the testing dataset R we obtain the dominant class (i.e., class membership) among its *k*NNs' classes. *k*NN classification in most cases is performed by a voting scheme, according to which, the class that appears more times among the nearest neighbors will be the resulting class. The voting scheme can be *weighted* when someone takes into account the distances between the nearest neighbors. Then each nearest neighbor has a weight according to its distance to the query element.

Let us consider the set of the k-nearest neighbors as $X = \{x_1^{NN}, x_2^{NN}, \ldots, x_k^{NN}\}$ and the class of each one as a set $C = \{c_1^{NN}, c_2^{NN}, \ldots, c_k^{NN}\}$. The weight of each nearest neighbor, indicating its impact on the final result, is calculated as follows:

$$w_i = \begin{cases} \frac{d_k^{NN} - d_i^{NN}}{d_k^{NN} - d_1^{NN}} & : d_k^{NN} \neq d_1^{NN} \\ 1 & : d_k^{NN} = d_1^{NN} \end{cases}, \quad i = 1, \ldots, k \tag{1}$$

where d_1^{NN} is the closest neighbor and d_k^{NN} the furthest one. By this calculation, the closest neighbors will be assigned a greater weight. We extend the approach to perform probabilistic classification (more details in "Methods" section).

Regression

Regression is a statistical process, used to estimate the relationship between one dependent variable and one or more independent variables. In the machine learning domain, regression is a supervised method, which outputs continuous values (instead of discrete values such as classes, categories, labels, etc.). These values represent an estimation of the target (dependent) variable for the new observations. A common use of the regression analysis is the prediction of a variable's values (e.g., future water/energy consumption, product prices, web pages visibility/prediction of potential visitors), based on existing/historical data. There are numerous statistical processes that perform regression analysis, however, in the case of *k*NN, regression can be performed by averaging the numerical target variables of the *k*NN as follows.

Considering the same set of *k*NNs $\left(X = \{x_1^{NN}, x_2^{NN}, \ldots, x_k^{NN}\}\right)$ and the target variable of each one as $V = \{v_1^{NN}, v_2^{NN}, \ldots, v_k^{NN}\}$, the value of the new observation will be calculated as:

$$v_r = \frac{\sum_{i=1}^{k} v_i^{NN}}{k}, \quad i = 1, \ldots, k \tag{2}$$

FML-*k*NN regressor implements the above procedure. At this point we should note that *k*-nearest neighbors performs non-linear classification and regression, as it does not seek a decision hyperplane to separate the data or a straight line to fit them. Instead, it seeks the closest elements in the neighborhood of the query element based on the Euclidean distance, which results to a non-linear traversal of the space.

Dimensionality reduction

In order to avoid expensive distance computations caused by high dimensional input data, we reduce their dimensionality to one. This is accomplished by three different SFC implementations, namely the *z*-order, the Gray-code and the Hilbert curve, all supported by FML-*k*NN in order to provide the flexibility of tuning according to specific needs, w.r.t. time performance or accuracy. Each curve scans the *n*-dimensional space in a dissimilar manner and exhibits different characteristics in terms of scanning "fairness" and computation complexity [19].

z-order curve The *z*-order curve (Fig. 1a) is computed by interleaving the binary codes of an element's features. This procedure takes place starting from the most significant bit (MSB) towards the least significant (LSB). For example, the *z* value of a 3-dimensional element with feature values 3 (011_2), 4 (100_2) and 5 (110_2), can be formed by first interleaving the MSB of each number (0, 1 and 1) going towards the LSB, thus, forming a final value of 011101100_2. This is a fast procedure, not requiring any costly CPU execution.

Gray-code curve The Gray-code curve (Fig. 1b) mapping computation is very similar to the *z*-order curve as it requires only an extra step. After obtaining the *z* value as described above, it is transformed to Gray-code by performing exclusive-or operations to successive bits. For example, the Gray-code value of 0100_2 would be calculated as follows. Initially, the MSB is left the same. Then, the second bit would be an exclusive-or of the first and second ($0_2 \oplus 1_2 = 1_2$), the third and exclusive-or of the second and third ($1_2 \oplus 0_2 = 1_2$) and the fourth an exclusive-or of the third and fourth ($0_2 \oplus 0_2 = 0_2$). Thus, the final Grey-code value would be 0110_2.

Hilbert curve Finally, the Hilbert curve (Fig. 1c) requires more complex computations in order to be calculated. The intuition behind Hilbert curve is that two consecutive points in the sequence are nearest neighbors, thus, avoiding "jumping" to farther elements, as in the *z*-order and Gray-code curves. The curve is generated recursively by rotating the two bottom quadrants at each recursive step. There are several algorithms that map coordinates to Hilbert coding. In this work, we employ the methods described in [31], offering both forward and inverse Hilbert mapping.

Apache Flink

Our framework was implemented using the Apache Flink distributed processing engine. Flink offers a variety of *transformations* on datasets and is more flexible than similar engines (i.e., Apache Hadoop and Apache Spark), due to the fact that it executes its jobs in a pipelined manner, thus, gaining in performance. Also, it efficiently supports iterative algorithms, which are extremely expensive in the standard MapReduce framework. The parallel tasks are executed by task managers, each one usually denoting a single machine with a number of further parallel processing slots, usually set to be the same as the number of available CPUs.

Flink is more appropriate for demanding computations performance-wise, as it does not require key-value pairs during the transitions that take place between the transformations. Instead, Java plain objects or just primitive types are used, optionally grouped in *tuples*. The grouping (partitioning) and sorting can be applied directly according to specific tuple elements or object variables, thus, avoiding the need of generating key-value pairs, which are required i.e., by Hadoop in order to properly partition and sort the elements during the shuffle and sort phase. Furthermore, Flink is equipped with built-in automatic job optimization, which achieves better performance compared to other engines.

Methods

This section outlines the design and implementation of FML-*k*NN. One of the main contributions of FML-*k*NN is the unification of the three different processing stages into a single Flink session. Multi-session implementations, regardless of the distributed platform on which they are developed and operated, are significantly inefficient due to the following reasons:

- Multiple initializations of the distributed environment: They increase the total wall-clock time needed by an application in order to produce the final results.
- Extra I/O operations during each stage: They introduce latency due to I/O operations and occupy extra HDFS space.

We avoid the above issues by unifying the distributed sessions into a single one. Figure 2 illustrates the unified session. The stages are executed sequentially and I/O operations on HDFS take place only during the start and end of the execution.

Dimensionality reduction and shifting

The input dataset is consisted of points in a d-dimensional space. To perform dimensionality reduction, we transform each point into SFC values via either z-order, Gray-code, or Hilbert curve. By taking a closer look on the way the SFCs fill a two-dimensional space from the smallest value to the largest, one could easily notice that some elements are falsely calculated being closer than others, as the curve scans them first. This can have a negative impact on the result's accuracy.

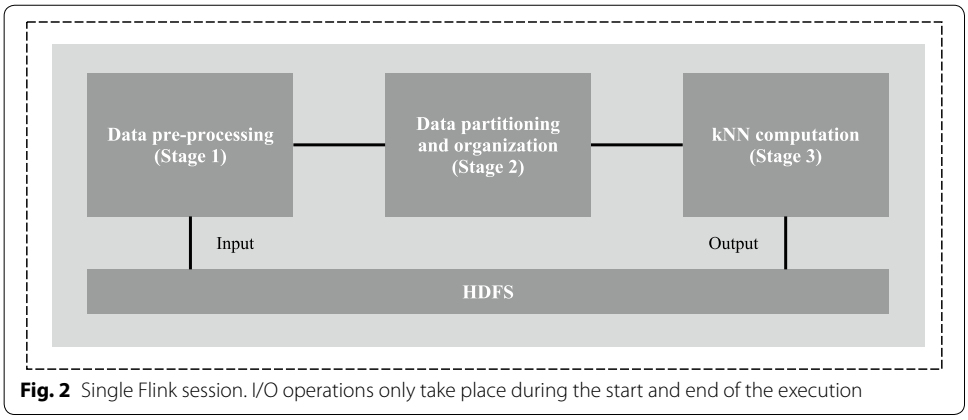

Fig. 2 Single Flink session. I/O operations only take place during the start and end of the execution

To diminish the negative effect of such false approximations, we generate a pre-determined number of alternate versions of the dataset, where each element is randomly shifted by a pre-calculated random vector. As an example, let us suppose that we have a dataset whose elements consists of three features and a random vector $v = \{3, 5, 2\}$. Then, all the elements of the dataset are shifted using this vector, e.g., an element $el = \{2, 6, 9\}$ will be altered to $el' = \{2 + 3, 5 + 6, 2 + 9\} = \{5, 11, 11\}$. This is demonstrated for z-order curve in Fig. 3, where the four bottom-left elements are shifted twice in the x-axis and once in the y-axis, altering the sequence in which they are scanned by the curve. This way, neighboring points that are distant on the curve will possibly be closer in the shifted dataset. This procedure compensates the lost accuracy, as it enables scanning the space in an altered sequence. During execution, the shifted dataset is concatenated with the original one and the algorithm is executed, producing *multiple* groups of kNNs, from which the final kNNs are determined. The limitation of this approach is the fact that the size of the input dataset is increased according to the number of shifts $i \in [1, \alpha]$, where α is the total number of shifts. Finally, we should mention that the above process is similar for all SFCs.

Partitioning

A crucial part of developing a MapReduce application is the way input data are partitioned in order to be delivered to the required reducers. Similar baseline distributed approaches of kNN joins problem perform partitioning on both R and S datasets in n blocks each and cross-check for nearest neighbors among all possible pairs, thus requiring n^2 reducers. We avoid this issue by computing n overlapping partitioning ranges for both R and S, using each element's SFC values. This way, we make sure that the nearest neighbors of each R partition's elements will be detected in the corresponding S partition. We calculate these ranges after properly sampling both R and S, due to the fact that this process requires costly sorting of the datasets.

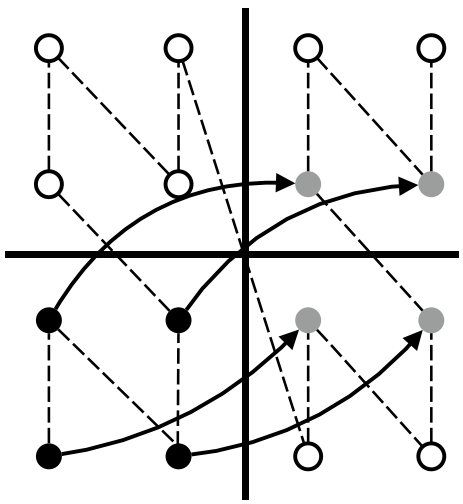

Fig. 3 Data shifting. The gray dots represent the shifted values

The FML-*k*NN distributed processing framework

FML-kNN workflow

FML-*k*NN has the same workflow as other similar approaches [23], and consists of three processing stages. The workflow of the algorithm is depicted in Fig. 4. The operations that each stage of FML-*k*NN performs are enumerated below (the Flink operation/transformation is in parentheses):

- Data pre-processing (stage 1)
  - Performs dimensionality reduction via SFCs on both R and S datasets (Flat Map R/S).
  - Shifts the datasets (Flat Map R/S).
  - Unifies the datasets and forwards to the next stage (Union).
  - Samples the unified dataset (Flat Map Sampling).
  - Calculates the partitioning ranges and broadcasts them to the next stage (Group Reduce).
- Data partitioning and organization (stage 2)
  - Partitions the unified dataset into n partitions, using the received partitioning ranges (Flat Map).
  - For each partition and each shifted dataset, the *k*NNs of each element in dataset R are calculated (Group Reduce).
- k NN computation (stage 3)
  - The final *k*NNs for each element in dataset R are calculated and classification or regression is performed (Group Reduce).

During data pre-processing, the sampling process is performed by a separate `flatMap` operation, which in return feeds the reducers with a smaller, sampled dataset used to calculate the partitioning ranges. The unified transformed datasets are directly passed to the mappers of stage 2, which also receive the partitioning ranges as a broadcast dataset via the `withBroadCastSet` operation. The data partitioning and organization

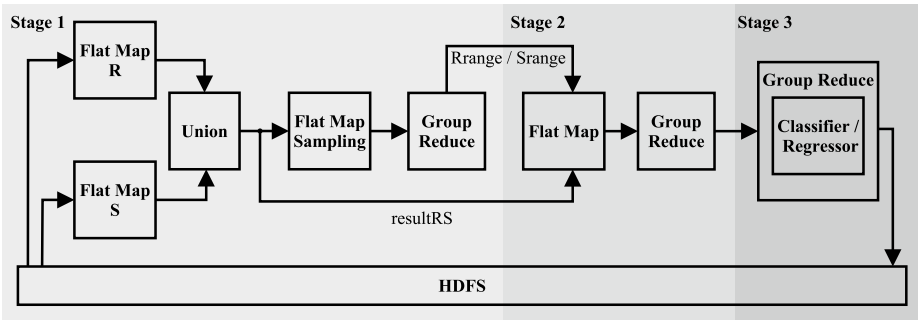

Fig. 4 Single-session FML-kNN. This image represents the unified process. The partitioning ranges are passed to the stage 2 as broadcast variables

stage directly feeds the input of the *k*NN computation. Flink's agility allows for the entire removal of the mapping procedure of stage 3, as it only propagates the results from the stage 2 to the reducers of stage 3. This increases the efficiency as it reduces the algorithm's resource requirements. In the following, we present each stage in more details.

Data pre-processing (stage 1)

The R and S datasets are read as plain text from HDFS and delivered to two separate concurrent `flatMap` transformations, identifiable by the input source file. During this stage, the SFC values (z values for z-order, g values for Grey-code curve and h values for Hilbert) and possible shifts, are calculated and passed to a `union` transformation, which creates a union of the two datasets. The unified and transformed datasets are then forwarded to the next stage (stage 2) and to a sampling process, which is performed by a separate `flatMap` transformation. The sampled dataset is then passed on a `groupReduce` transformation, grouped by a shift number. This way, α (number of shifts) reducers will be assigned with the task of calculating the partitioning ranges for R and S datasets, which are then broadcast to the next stage (data partitioning and organization).

Broadcasting the partitioning ranges significantly reduces the computational resources required by the reducers compared to F-z*k*NN, as it avoids the race condition between stages 1 and 2 (Fig. 5a). During the race condition, the transformed dataset is forwarded to the second stage before the partitioning ranges are calculated by the reducers, causing its mapping operation to be initiated, which would result in an error, as the partitioning ranges are required. To avoid this race condition, F-z*k*NN's reducers have to locally cache the transformed dataset while it is being sampled for the calculation of ranges, as depicted in Fig. 5b. FML-*k*NN overcomes this issue, by broadcasting the partitioning ranges, as this procedure blocks the execution of the second stage until the transformed dataset is ready. The left part of Fig. 4 depicts the whole process.

Algorithm 1 presents the pseudocode of stage 1. The random vectors are initially generated and cached on HDFS in order to be accessible by all the nodes which take part in the execution. They are then given as input to the algorithm, along with datasets R and S. Notice that $\mathbf{v}_1 = \vec{0}$ indicating that the datasets are not shifted during this iteration. This process takes place α times, where α is the number of shifts (Line 5). After shifting the datasets, during the first mapping phase (Lines 5–9), the elements' SFC values are calculated and collected to $\hat{R}_i^T$ and $\hat{S}_i^T$, where $i = 1, \ldots, \alpha$. Then, the sampling is performed

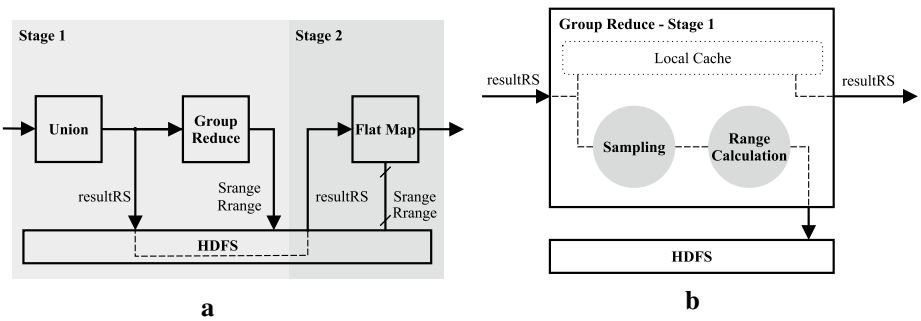

Fig. 5 F-zkNN race condition and solution. The reducers of the first stage have to locally cache the transformed dataset (**b**) to avoid the race condition (**a**)

by the second mapping phase (Lines 10–17). During the reduce phase (Lines 18–22), the partition ranges ($Rrange_i$ and $Srange_i$) for each shift are calculated using the sampled datasets and broadcast to stage 2 (Lines 18–20). The output consists of the unified transformed datasets, which finally feed the data partitioning and organization stage (Line 22).

Algorithm 1: FML-kNN (stage 1).

```
1  ▷ The pre-processing stage's input
   Input: Datasets R, S, random vectors V = {v_1, ..., v_α}, v_1 = 0⃗ and sampling threshold MinThreshold
2  ▷ The output, that will be emitted to the next stage
   Output: Transformed datasets R_i^T and S_i^T, i = 1, ..., α
3  begin
4      ▷ The same procedure is repeated for each shift
5      for i = 1, ..., α do
6          R_i = R + v_i
7          S_i = S + v_i
8          R_i^T ← calcSFC(R_i)
9          S_i^T ← calcSFC(S_i)
10         foreach x ∈ R_i^T ∪ S_i^T do
11             ▷ Sampling
12             r ← Random(0, 1)
13             if r < MinThreshold then
14                 if x ∈ R_i then
15                     InsertSample(s, R̂_i^T)
16                 else if x ∈ S_i then
17                     InsertSample(s, Ŝ_i^T)
18         Rrange_i ← calcRange(R̂_i^T)
19         Srange_i ← calcRange(Ŝ_i^T)
20         Broadcast(Rrange_i, Srange_i)
21         ▷ Emit to the partitioning and organization stage
22         return R_i^T ∪ S_i^T
```

Data partitioning and organization (stage 2)

The transformed datasets of stage 1 are partitioned to $n \times \alpha$ blocks via a custom partitioner, after fetching the previously broadcast partitioning ranges. Each block is then delivered to a different reducer through a `groupBy` operation. Finally, the nearest neighbors for each query element are calculated via proper range search operations and passed on the next stage (kNN computation). The middle part of Fig. 4 depicts this process.

Algorithm 2 presents the pseudocode of stage 2. During the map phase (Lines 7–18), after having read each shift's broadcast partition ranges (Line 6), the received transformed datasets are partitioned into $n \times \alpha$ buckets ($R^{g \times i}$ and $S^{g \times i}$, $i = 1, \ldots, \alpha$, $g = 1, \ldots, n$, Lines 13 & 18), α being the number of shifts and n the number of partitions. The partitions $S^{g \times i}$ are then sorted and emitted to the reducers (Lines 23–30) along with the corresponding partitions $R^{g \times i}$. There, for each $x \in R$ element, a range search is performed on the proper sorted partition in order for its kNN to be determined (Line 23) and its initial coordinates are calculated (Line 24). The initial coordinates of all neighboring elements' coordinates are then calculated (Line 26) and their distance to the $x \in R$ element is computed (Line 27). Finally, all nearest neighbors are integrated into the proper dataset ($R_{k \times \alpha}$, Line 28) along with the calculated distance and feed the stage 3, grouped by $x \in R$ elements.

Algorithm 2: FML-kNN (stage 2).

```
1  ▷ This stage's input are the datasets emitted during the partitioning and organization stage
   Input: Datasets R_i^T, S_i^T, i = 1, ..., α
2  ▷ The output, that will be emitted to the next stage
   Output: Dataset R_{k×α}
3  begin
4      ▷ Initialization of the dataset to be emitted
5      R_{k×α} = ∅
6      receiveBroadcast(Rrange_i, Srange_i)
7      ▷ Again, repeat for each shift
8      for i = 1, ..., α do
9          ▷ Partition the R elements
10         foreach x ∈ R_i^T do
11             for g = 1, ..., n do
12                 if zval(s) ∈ Rrange_i(g) then
13                     addIntoPartition(s, R^{g×i})
14         ▷ Partition the S elements
15         foreach x ∈ S_i^T do
16             for g = 1, ..., n do
17                 if zval(s) ∈ Srange_i(g) then
18                     addIntoPartition(s, S^{g×i})
19         for g = 1, ..., n do
20             ▷ Sorting is needed to properly perform range search
21             sort(S^{g×i})
22             foreach x ∈ R^{g×i} do
23                 RES ← rangeSearch(x, k, S^{g×i})
24                 CC_x ← calcCoords(x)
25                 foreach neighbor ∈ RES do
26                     CC_neighbor ← calcCoords(neighbor)
27                     CD ← calcDist(CC_x, CC_neighbor)
28                     R_{k×α} ← add(x, neighbor, CD)
29     ▷ Emit to the final stage, grouped by element
30     return R_{k×α}
```

kNN computation (stage 3)

The calculated $\alpha \times k$-nearest neighbors of each R element, are fetched from HDFS and mapped to $|R|$ reduce tasks. The final k-nearest neighbors are determined and passed to the classifier or regressor, depending on user preference. The latter calculate either the probability for each class (classifier) or the final value (regressor) for each element in R.

Classification In the case of classification, we extend the voting scheme, to perform probabilistic classification. We consider the set $P = \{p_j\}_{j=1}^{l}$, containing the probability that the query element will belong to each class, where l is the number of classes. The final probability for each class will be derived as follows:

$$p_j = \frac{\sum_{i=1}^{k} w_i \cdot I(c_j = c_i^{NN})}{\sum_{i=1}^{k} w_i}, \quad j = 1, \ldots, l \tag{3}$$

where $I(c_j = c_i^{NN})$ is a function which takes the value 1 if the class of the neighbor x_i^{NN} is equal to c_j.

Finally, the element will be classified as:

$$c_r = \arg\max_{c_j} P, \quad j = 1, \ldots, l \tag{4}$$

which is the class with the highest probability. The final result for each element is appended along with the calculated probabilities for each class in a result entry. Listing 1 showcases a four class classification where the element XYZ has been assigned to class *C*.

Listing 1 The element XYZ is classified to class C, which has the highestprobability. $XYZ|Result : C|A : 0.06|B : 0.03|C : 0.71|D : 0.2$

Regression For the case of regression, the final result for each element is calculated as described in "Regression" section and contains its predicted value and has the following format (Listing 2):

Listing 2 The value estimation of element XYZ. $XYZ|Result : 19.244469$

In both cases, the results for each query element are stored on HDFS in plain text.

Algorithm 3 presents the pseudocode of stage 3. During this stage, which consists of only a reduce operation, *k*NNs of each *R* element are fetched from the grouped set of $R_{k\times\alpha}$. Finally, for each query element either classification (Line 9) or regression (Line 11) is performed, after determining its final nearest neighbors (Line 7). The results are added to the resulting dataset (Line 9), which is then stored on HDFS (Line 12) in the proper format.

Algorithm 3: FML-*k*NN (stage 3).

```
1  ▷ The input is the grouped by element dataset emitted during the previous stage
   Input: Datasets R, R_{k×α}
2  ▷ The algorithm's results
   Output: Dataset R_f
3  begin
4      ▷ Initialization of the final dataset
5      R_f = ∅
6      foreach x ∈ R do
7          RES ← kNN(x, R_{k×i})
8          if classification then
9              FIN ← CLASSIFY(RES)
10         else if regression then
11             FIN ← REGRESS(RES)
12         R_f ← ADD(s, FIN)
13     ▷ Store the final results on HDFS
14     return R_f
```

Spark implementation

We implemented a three-sessions and a single-session Spark version of the probabilistic classifier, named *S-kNN*, in order to conduct a comparative evaluation among the different distributed processing engines. The architecture of the implementation is the same as described above. The main difference lies to the transformations and actions that were used in order to achieve similar functionality to the Flink implementations. The code for both implementations is open source and it can be found online [32].

Cost analysis

Zhang et al. [7] showed that the overall communication cost of H-z*k*NN is $O\left(\alpha\left(1/\epsilon^2 + |S| + |R| + k|R| + nk\right)\right)$, where α is the number of shifts, ϵ the sampling rate, n the number of partitions and k is the number of required nearest neighbors.

FML-*k*NN introduces some further communication within the stage 1 and between the stages 1 and 2. More specifically, $|R|$ and $|S|$ elements have to be communicated from the initial `flatMap` of stage 1 to the sampling `flatMap`. Similarly, the same number of elements have to be sent to the mappers of the stage 2. There is no extra communication from the unification of stage 3, due to the removal of its mappers. Thus, the rest of the communication cost remains unchangeable. Consequently, the overall communication cost of our method is $O\left(\alpha\left(1/\varepsilon^2 + 3|S| + 3|R| + k|R| + nk\right)\right)$.

As far as the CPU cost is concerned, Zhang et al. [7] estimate it to be $O\left(1/\epsilon^2 log\left(1/\epsilon^2\right) + nlog\left(1/\epsilon^2\right) + (|R| + |S|)log|S|\right)$ for H-z*k*NN. The FML-*k*NN introduces extra calculations in the stage 3, for classification and regression, which are $O(3k|R| + 2c|R|)$ and $O(2k|R|)$ respectively, where c is the number of classes for classification. However, since both are significantly less than $(|R| + |S|)log|S|$, the total CPU cost is finally equal to $O\left(1/\epsilon^2 log\left(1/\epsilon^2\right) + nlog\left(1/\epsilon^2\right) + (|R| + |S|)log|S|\right)$.

Results and discussion

FML-*k*NN was assessed in terms of wall-clock completion time and scalability, by conducting a comparative benchmarking among similar implementations, executed over different distributed processing engines. The latter are either executed in one (where possible), or three Spark or Hadoop (i.e., an extension of the original H-z*k*NNJ algorithm) sessions and are compared with the corresponding versions of the probabilistic classifier. Similar results are expected for the regressor which we have omitted to avoid repetition.

We also present two case studies which exhibit the framework's efficiency over useful insights extraction from water consumption events on a city scale level. Throughout our experiments, we used one synthetic and two real-world water consumption time-series datasets.

Experimental setup

In the following, we present the environmental setting of the experiments and the qualitative and quantitative metrics that we used to assess the performance of the classification and regression processes. We also present the parameters that were used and how they were determined in the context of the experimentation process.

System The algorithms were assessed on a pseudo-distributed environment. The setup includes a system with 4 CPUs, each containing 8 cores clocked at 2.13 GHz, 256 GB RAM and a total storage space of 900 GB. The CPUs support hyper-threading technology, running 2 separate threads per core. The total parallel capability of the system reaches the 64 threads ($4 \cdot 8 \cdot 2$). All (i.e., Flink-, Spark- and Hadoop-based) implementations were evaluated on the same HDFS, over a local *Yet Another Resource Negotiator* (YARN) resource manager. This way, each Flink task manager, Spark executor or Hadoop daemon runs on a different YARN container, represented by a separate Java process able to run one or more threads, simulating the distributed cluster.

Despite the significant differences in the distributed processing engines' configuration settings, they were all configured in order to use the same amount of system resources. The level of parallelism of all tasks for each engine was set to 16, in order to exploit the fact that, in an optimally determined setting and during the experiments, the stage 2 partitions the dataset into 8 subsets and the total number of shifts is 2. Thus, a maximum of 16 simultaneous tasks are executed in all cases. For Flink and Spark, a total of 4 task managers (one per CPU) and executors respectively were assigned 32768 MB of Java Virtual Machine (JVM) heap memory. Each task manager and executor was given a total of 4 processing slots (Flink) or executor cores (Spark). For Hadoop, the parallelism ability was set to 16 by assigning the mappers and the reducers to the same number of YARN containers, with 8192 MB of JVM each. Thus, the total amount of JVM heap memory assigned to each session is always 131072 MB (either $4 \cdot 32768$ MB, or $16 \cdot 8192$ MB).

Despite our attempt to assign similar amount of system resources to each distributed processing engine and due to the fact that each one offers a handful of configuration settings regarding execution, memory allocation, and job scheduling behavior, the performance of the different implementations may differ from the optimal one. However, after performing numerous benchmarking sessions, we believe that for the current system setting, the presented configuration is fair, achieving the highest possible performance for all three engines, while maintaining the level of parallelism at 16. The configuration was performed by taking into consideration the corresponding guide of each engine.

Parameters FML-*k*NN uses a variety of input parameters required by the underlying distributed *k*NN algorithm, in order to support the classification and regression processes. Regarding the value of the k parameter that was used throughout the experiments and due to the fact that the optimal value is problem specific, we performed a separate *cross-validation*-based evaluation for each of the case studies (see "Case studies" section). The best k parameter choice was performed in a way that maintains the best balance between completion time and accuracy. Most parameters were similarly chosen after performing appropriate cross-validation-based benchmarking.

Among the rest of the parameters, FML-*k*NN utilizes a vector of size equal to the input dataset's dimensions, indicating the weight of each feature according to its significance to the problem. Each feature is multiplied with its corresponding weight before the execution of the algorithm in order to perform the required scaling, according to the feature's importance. To automatically determine an optimal feature weighting vector, we provide the option of executing a genetic algorithm, which uses a specific metric, described in the next paragraph, as cost function. The parameters of the genetic algorithm, such as the size of the *population*, the probability to perform *mutation* or *crossover*, the *elite* percentage of the population and the number of the iterations can be directly determined by the user. This approach was applied to produce the optimal feature weighting vector for each of the cases that we studied.

Metrics Four different well known performance measures were used to evaluate the quality of the results obtained by the classifier and the regressor. These performance measures are included in the framework in order to offer the ability to assess the various data analysis tasks. *Accuracy* and *F-Measure* are implemented for classification, while *root mean square error* (RMSE) and *Coefficient of determination* (R^2) are used to evaluate

the quality of regression. A short description of what each of these metrics represents in our experimentation is listed below:

- Accuracy: The percentage of correct classifications (values from 0 to 100). It indicates the classifier's ability to correctly guess the proper class for each element.
- F-Measure: The weighted average of *precision* and *recall* of classifications (values from 0 to 1). Using this metric, we ensure good balance between precision and recall, thus, avoiding misinterpretation of the accuracy.
- Root mean square error (RMSE): Standard deviation between the real and predicted values via regression. This metric has the same unit as the target variable. It provides us with the insight of how close the guessed values are to the real ones.
- Coefficient of determination (R^2): Indicates the quality of the way the model fits the input data. It takes values from 0 to 1, with 1 being the perfect fit. A good fit means that the regressor is able to properly identify the variations of the training data.

The completion time comparison of the distributed processing engines is performed after simply obtaining the system time elapsed before and after each execution.

Datasets

For the experimental evaluation of FML-*k*NN we used two real-world water consumption related datasets coming from Switzerland and Spain. We also generated a synthetic dataset, based on an extended version of the Spain dataset, with a much larger amount of entries. Since the framework's algorithm is *k*NN-based, we normalized all the datasets' features from values ranging from 0 to 1, in order to avoid broader ranged features heavily affecting the result.

SWM dataset The first dataset contains hourly time-series of 1000 Spanish households' water consumption, measured with smart water meters. It covers a time interval of a whole year, i.e., from the 1st of July 2013 to the 30th of June 2014. The records include a household identifier, a timestamp and a meter measurement in liters. This dataset has a total number of 8.7 M records.

Shower dataset The second dataset includes shower events from 77 Swiss households collected with shower water meters. Each record contains a household and shower identifier, a meter measurement in liters, the number of times that the faucet was turned off during the shower and the corresponding duration, the total duration of each shower as a whole, as well as the average water temperature and flow rate. It also contains demographic information related to the age, income, number of males or females and total number of household members. This dataset counts 5795 records. While this dataset is not on a Big Data scale, we perform a case study based on it, in order to assess the framework's data analysis capability on water consumption data regarding a single water fixture and consumer activity.

Synthetic dataset In order to evaluate completion time performance on Big Data, we created a synthetic dataset via a Big Data generator. The latter is a part of the *BigDataBench* benchmark suite [33] and operates via .xml files, in which the user can determine the number of records and their features. Based on an extended version of the SWM dataset produced after proper feature extraction (see "SWM dataset" section), the

synthetic dataset's entries consist of an id, 10 features and two target variables of continuous (floating point with 10 decimals) and binary data representation, respectively. Each feature is ranged from 1 to 99 and the id is alphanumeric. The data representation, number and range of features was selected in order to maintain a relatively high number of dimensions, while being able to create a large number of records and at the same time avoiding exceeding the memory limits of the setup (approx. 8192 MB per mapper or reducer for the Hadoop case). Thus, the total size of the synthetic dataset has 100 M records (approx. 4.1 GB).

Benchmarking

In the following, we perform a comparative benchmarking of FML-*k*NN, in terms of scalability and wall-clock completion time, using the synthetic dataset and the probabilistic classifier. Similar results are expected for the regressor. The comparison involved:

- FML-*k* NN (single session): The proposed implementation, presented in the previous section.
- FML-*k* NN (three sessions): A three-sessions version of FML-*k*NN, where each stage is executed by a different Flink process.
- S-*k* NN (single session): An Apache Spark version with the same architecture as FML-*k*NN.
- S-*k* NN (three sessions): A three-sessions version of S-*k*NN, where each stage is executed by a different Spark process.
- F-z *k* NN: The single-session algorithm on which the core algorithm of FML-*k*NN was based.
- H-z *k* NNJ: An extended version of the algorithm to perform probabilistic classification executed in three separate sessions. This is the baseline method.

It is important to note that a single-session version of the H-z*k*NNJ algorithm is not possible, as a Hadoop session can only execute one map, followed by one reduce procedure. A single session requires the three stages to be executed in a sequential manner, which is not possible in Hadoop, as it would require mapping to be performed after reducing procedures several times.

Wall-clock completion time

Table 1 shows the probabilistic classifier's wall-clock completion time of all Flink, Spark and Hadoop versions, run in either three, or one sessions (possible only for FML-*k*NN, S-*k*NN and F-z*k*NN). The three-session F-z*k*NN is identical to the three-session FML-*k*NN implementation and its measurements are omitted. We used the synthetic dataset and we followed a 10–90% testing–training split scheme, resulting in 10 M records being used as the testing set (R) and 90 M as the training set (S). It is apparent that the Flink-based implementations perform significantly better than all implementations, in both three and single-session versions. This improved performance is due to Flink's ability to process tasks in a pipelined manner, which allows the concurrent execution of successive tasks, thus, gaining in performance by compensating time spent in other operations (i.e., communication between the cluster's nodes).

Table 1 Wall-clock completion time

Version	3 sessions				1 session
	1st	2nd	3rd	Total	Total
FML-*k*NN	03 m 45 s	24 m 59 s	01 m 48 s	30 m 32 s	30 m 25 s
S-*k*NN	07 m 12 s	29 m 15 s	03 m 01 s	39 m 28 s	33 m 03 s
F-z*k*NN	N/A	N/A	N/A	N/A	46 m 09 s
H-z*k*NNJ	06 m 00 s	31 m 19 s	0 5 m 54 s	43 m 13 s	N/A

The unified Flink version outperforms the rest of the implementations

The unified version implemented with Spark is significantly faster than the total wall-clock time of the corresponding three-sessions setting. This is due to the reduction of the I/O operations on HDFS during the beginning and end of each session. A critical role is also played by the omission of the mappers of stage 3, which introduced the additional overhead of forwarding the results of the second session to the proper reducers. For FML-*k*NN, the total time of the three-sessions version is similar to the unified one, again due to the pipelined way of execution, which compensates the time lost during HDFS I/O operations. The wall-clock completion time of S-*k*NN is only slightly lower for each stage than the baseline H-z*k*NNJ implementation, except during the third session, where it executes almost twice as fast. This is caused by the fact that the stage 3 does not include costly operations such as sorting and transforming the dataset coming from stage 2. The latter is partitioned according to the id of each query element (i.e., elements in *R* dataset), thus, taking advantage of all system resources and possibly Spark's documented better performance over Hadoop. F-z*k*NN performs worse than the rest of the implementations due to its resource-hungry propagation of the transformed dataset during the first stage.

It should be noted that the results do not suggest that Flink is superior to the other distributed processing engines. The comparison is limited to the current FML-*k*NN's core algorithm and the implementation and experimental setting are as fair as possible, considering the engines' different operational support. It proves, however, that our algorithm performs significantly better with Flink and supports our choice to use it as an underlying execution environment.

Scalability

The various implementations were also evaluated in terms of scalability. Figure 6 shows the way each version scales in terms of completion time for subsets of the synthetic dataset of different size, using the same 10–90% testing–training split scheme. The subset sizes varied from 10 M (1 M testing–9 M training), to 100 M (10 M testing–90 M training). As illustrated in the figure, the FML-*k*NN implementations exhibit similar performance and scale better as the dataset's size increases. However, the unified version, i.e., the one used at FML-*k*NN framework, has the advantage of not requiring user input and session initialization between the algorithm's stages. Flink's pipelined execution advantages are once again apparent if we compare the scalability performance of all the three-sessions implementations: The I/O HDFS operations cause the Spark and Hadoop versions to scale significantly worse than Flink, which performs similarly to the unified

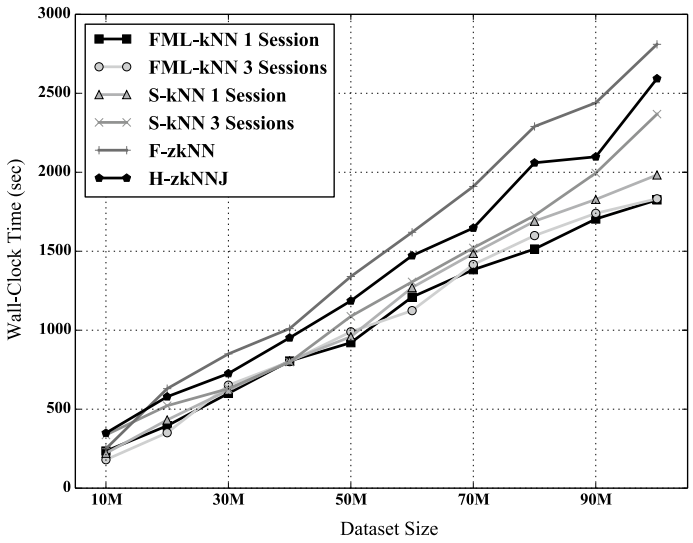

Fig. 6 Scalability comparison. Flink exhibits better scalability, especially after 90 M records of input dataset size

version. Finally, due to the aforementioned dataset propagation, F-z*k*NN scales worse than all the implementations.

We have executed the algorithm using the 100M synthetic dataset for different split schemes, more specifically for 10–90, 30–70, 50–50, 70–30 and 90–10% testing–training. The results are depicted in Fig. 7. FML-*k*NN performs better in all cases. Due to the fact that the CPU cost of all methods similarly depends on the size of R and S datasets (please refer to "Cost analysis" section), there are no large variations between the differences in the performance among all the implementations, and for all split cases.

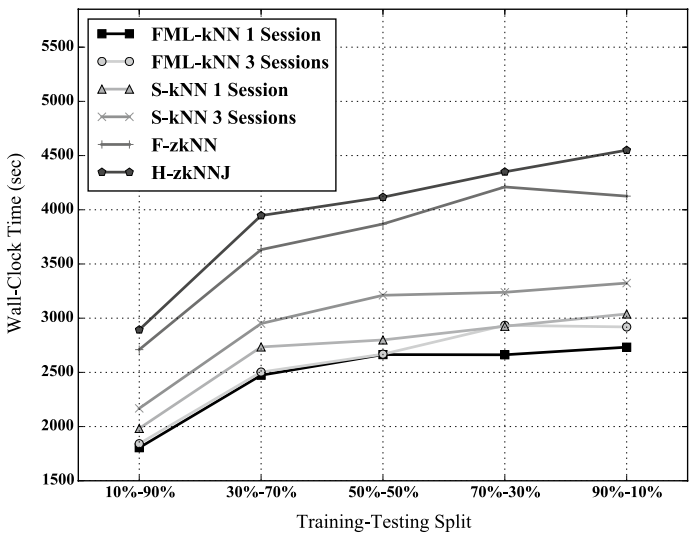

Fig. 7 Wall-clock completion time for different split sizes. FML-*k*NN performs better in all cases. Completion time is increased in a similar to logarithmic manner

More time is required to finish execution as the size of the *R* dataset is increased, due to the fact that the communication cost more heavily depends on the size of the *R* dataset. However, this is compensated in each split case by the reduced size of dataset *S*, causing the execution time to be increased in a similar to logarithmic manner.

Case studies

The framework's data analysis potential was evaluated using two real-world water related datasets in two independent case studies presented below. We focus on knowledge extraction from large volumes of historical water consumption data, towards the facilitation of useful insights acquisition for consumers and resource utilities in the water domain, which could induce lifestyle changes by promoting sustainability.

SWM dataset

The first case study showcases the potential of the framework's machine learning algorithms on a water consumption time-series forecasting task. Time-series data on water consumption pose several challenges on applying machine learning algorithms for forecasting purposes. Apart from the actual measurement values, one must take into account their correlation with previous values, their seasonality, the effect of contextual factors, etc. [34]. Thus, proper features that represent and correlate different aspects of the data need to be defined in order to take advantage of the time-series nature of such data.

Feature extraction The dataset consists of 8.7 M smart water meter hourly readings for 1000 households during a year, with each reading representing a record. The id of each element was set to be the smart meter id, together with the date and hour during which the consumption occurred. Initially, we removed any outlier records that could affect the final result. Such records belong to customers who do not exhibit normal water consumption behavior, i.e., they are frequently absent during the year or they continuously consume water (i.e. indication of possible leakage). We considered as frequently absent the users who did not consume any water for more than 40 days during a year, which indicates that they were away from their households. As a next step, we generated a total of nine different features for each data element.

Due to the qualitative nature of the extracted features, the Euclidean distance of FML-*k*NN cannot be applied directly on them. To overcome this, we assign each feature with a new value, which is derived during a pre-processing step, after we sort according to average water consumption. For example, for the feature indicating the season of the year, we obtain the average per season water consumption and we sort the seasons according to this value. Then, each season is assigned with a numerical value corresponding to its position in this sequence. This way, if supposedly winter was assigned with 1, summer with 2, autumn with 3 and spring with 4, we know that winter is closer to summer both in terms of consumption and Euclidean distance. The extracted features are presented in more details in the following.

- Hour: The hour during which the consumption occurred.
- Time Zone: We grouped the hours into four time zones of consumption: 1am–4am (sleeping), 5am–10am (morning activity), 11am–7pm (working hours) and 8pm–12am (evening activity).
- Day of week: The day of the week from Monday to Sunday.
- Month: The month of the year, from January to December.
- Season: The season of the year, from winter to autumn.
- Weekend: A binary feature indicating whether or not the consumption occurred during a weekend. We decided to include this feature after noticing differences between average hourly consumption during weekdays and weekends.
- Customer group: We run a *k-means* clustering algorithm, configured to run for time-series data, on the weekly average per-hour consumption time-series for each customer (for details, see below).
- Customer ranking: For each hour, we calculated the average hourly consumption of each customer and sorted according to it.
- Customer class: This feature represents a grouping of the customers to one of four classes according to their monthly average consumption, i.e. "environmentally friendly", "normal", "spendthrift", "significantly spendthrift".

Each record also contained two target variables, being the exact meter reading at each timestamp and a binary flag, indicating whether consumption occurred during that hour or not (i.e., if the meter reading was positive or zero).

The reason for choosing *k*-means for extracting the customer group feature was its significantly lower complexity compared to other clustering methods. As several evaluating works in the literature suggest [35–37], *k*-means achieves the best performance in terms of execution time and scalability. This is crucial in our case, as in a real world scenario the feature extraction procedure, as a pre-processing step is usually required to be as time efficient as possible. Additionally, *k*-means can produce accurate and tight clusters compared to similar methods [38], which renders it the best choice for this case study.

Since the input data to be clustered are time-series, we used *Dynamic Time Warping* (DTW) as a distance metric. DTW is a very commonly used metric for comparing time series. As opposed to other distance metrics (e.g., Euclidean distance), it can handle the case where two time series have similar form but are slightly shifted in the time axis. For more details and a formal definition of DTW, the reader can refer to [39]. Finally, we applied *k*-means on the average consumption time-series, with 10 as the number of clusters, which was determined as follows. Initially, we used a rule of thumb which provides a very fast estimation of the optimal number of clusters, described by Kodinariya et al. [40], which calculates *k* as follows:

$$k = \sqrt{\frac{n}{2}} \tag{5}$$

where n is the number of elements to be clustered. Considering the number of the customers (1,000), the above equation yielded $k = 22$. Starting from this value, we run the *k*-means algorithm several times, using lower (higher) number of clusters than 22. We decreased (increased) the number of clusters by 1 and repeated the clustering and

measured the *Davies-Bouldin index* [41] score of each execution. Finally, we chose the local best score, which in this case study was 10 clusters.

Procedure We first execute the probabilistic classifier, with the testing set (R) comprising of the last 2 weeks of water consumption for every user (approximately 336,000 records), in order to obtain the possibility of whether consumption will occur or not, during each hour. The rest of the dataset (approximately 8.36 M records) is used as the training set (S). We perform binary classification, obtaining an intermediate dataset indicating whether or not consumption will occur for each training record. Using the hours during which we predicted that water will be consumed, we ran the regressor, obtaining a full predicted time-series result of water consumption for each user. Before each algorithm's execution, we determined the optimal scale vector using the genetic approach.

In order to choose the optimal k parameter for both algorithms, we employed a tenfold *cross-validation* approach. We iteratively split the entire dataset into ten equal parts and executed each algorithm the same number of times, using a different subset as training set (S) while the rest of the sets, unified, comprised the testing set (R). As a metric, we used accuracy for classification and RMSE for regression. The k value that achieved the best balance between completion time and result quality was 15, for both the classifier and regressor.

Space Filling Curves accuracy evaluation FML-kNN supports three SFC-based solutions for reducing the dimensionality of the input data to just one dimension, namely the z-order, Grey code and Hilbert curves. We evaluated the completion time and approximation quality of each SFC, in order to choose the one that achieves the best balance between timing performance and approximation accuracy, regarding the water consumption dataset. Table 2 presents the metric and time performance related results of the probabilistic classifier and regressor for each SFC, which we obtained from running the algorithms using the tenfold cross-validation.

All three SFCs demonstrate similar performance in all aspects. Hilbert curve scores higher in all metrics as expected, however only slightly. Consequently, we chose the z-order curve for this case study, as it exhibits better time performance due to its decreased calculation complexity.

Results The classifier correctly predicted the 74.54% (F-Measure: 0.81) of the testing set's target variables, i.e., the hours during which some consumption occurred for the 2-week period. For these specific hours the regressor achieved a RMSE score of 19.5 and a coefficient of determination score of 0.69. The results were combined into a single file, forming the complete time-series of the dataset's last 2 weeks' (June 16–30 2014) water consumption prediction for all the users.

Table 2 Space Filling Curves' performance.

Curve	Classification			Regression		
	Accuracy (%)	F-Measure	Wall-clock time	RMSE	R^2	Wall-clock time
z-order curve	70.24	0.775	1 m 20 s	18.86	0.64	0 m 59 s
Hilbert curve	70.54	0.78	1 m 32 s	18.69	0.66	1 m 15 s
Gray-code curve	70.4	0.777	1 m 25 s	18.81	0.64	1 m 5 s

The Hilbert-order curve performs slightly better in all metrics, but is slower

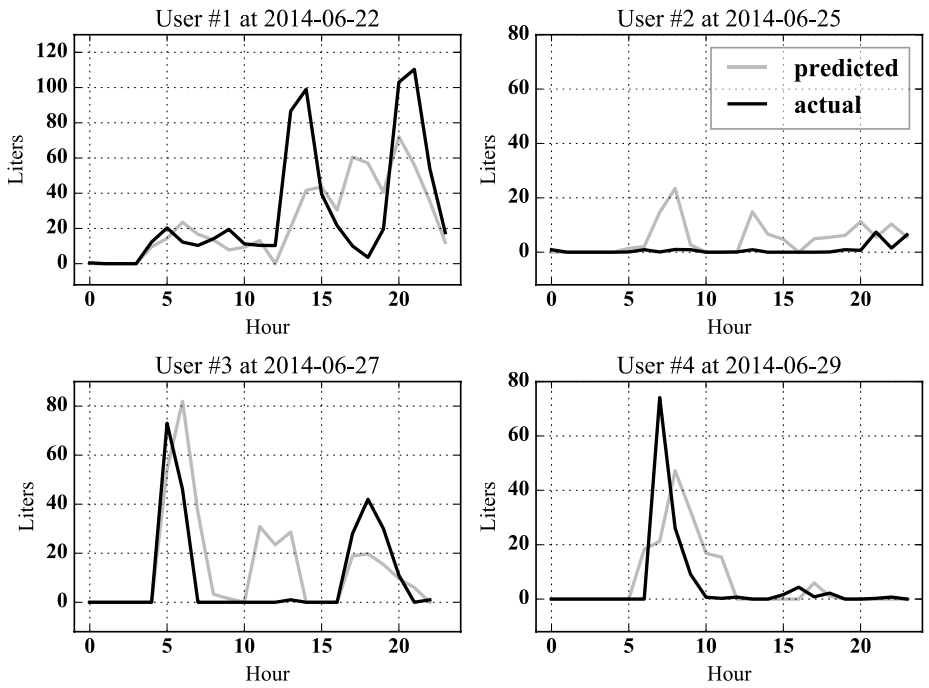

Fig. 8 Personalised hourly water consumption prediction. The predicted values follow the trend of the real ones, but miss sudden events. This is apparent during 16 and 20 h for #user1 and during 10 and 15 h for #user3. #user2 had very little consumption during that day and our framework predicted that no consumption will occur

Figure 8 shows four users' consumption prediction versus the actual one, during four different days. The prediction for user #4 was close to reality. The results seem to follow the real values, but are not able to properly follow the observed ones. This indicates that it is rather difficult to accurately predict a single user's future water consumption, due to possible unforeseen or random events during a day, a fact which justifies the rather large RMSE score. For example, consumptions higher than 20 liters during an hour (e.g., user #3 around 6:00) could indicate a shower event, while larger consumptions (>50 l) over more than 1 h could suggest usage of the washing machine or dish washer (e.g., user #3 from 16:00 to 20:00), along with other activities. In order to assess the generalization of the results, we calculated the average RMSE of the hour and volume of the peak consumption during each day, as well as the average RMSE of the total water use per day, for all the predictions. The rather high errors, (8.89 h, 28.9 and 132.23 l respectively), confirm the previous observations regarding the difficulty in predicting random daily events. However, despite all the above, our algorithms' predictions are able to mostly follow the overall behavior during most days (e.g., users #3 and #1).

The results are particularly interesting if we aggregate the total predicted consumption of all users during each hour. The two upper diagrams in Fig. 9 illustrate this case for two different days. It is apparent that our algorithms' predictions are able to properly follow the real aggregated consumption during each hour. This indicates that the negative effect of the sudden/unusual/unforeseen events is lost when we attempt to predict the total hourly water consumption of a larger number of users. The two lower diagrams present the same prediction performed by feeding our algorithms with already aggregated hourly

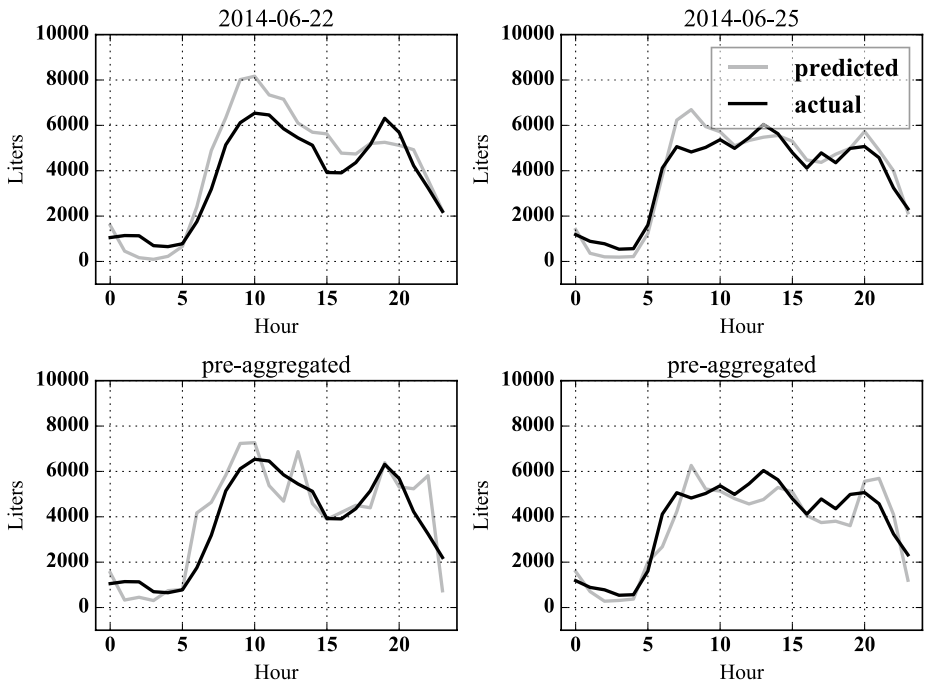

Fig. 9 Aggregated hourly water consumption prediction. The predicted values seem to correctly follow the general trend of consumption.

consumption data. While the pre-aggregated dataset's predictions appear to less diverge from the actual values in certain points, the non-aggregated results are smoother and better assemble the actual overall behavior. This is due to the fact that the non-aggregated dataset also contains user-specific features and is thus able to perform predictions based on user similarity, thus, yielding more accurate predictions.

Shower dataset

This case study exhibits the framework's potential in extracting useful knowledge from shower water consumption data, using the shower dataset. More specifically, the probabilistic classifier is used (the z-order curve is selected, similarly to the previous case study) in order to predict specific characteristics of a user, from shower water consumption events.

Feature extraction As mentioned, the dataset, apart from shower water consumption related data, also included demographic information. In this case study we focused on predicting the sex, age and income of the person that generates a shower event, using classification. For the case of sex, we used the shower events for which we knew whether the person was male or female (binary classification), i.e., households with only one, or only same sex inhabitants. Consequently, each record consisted of all the smart meter measurements (shower stops, break time, shower time, average temperature, average flow rate) as features and the sex as the target variable. Due to the dataset's rather small size, the age and income prediction was also performed by binary classification, i.e., determine whether the person that generates a shower event is of age less than 35 years

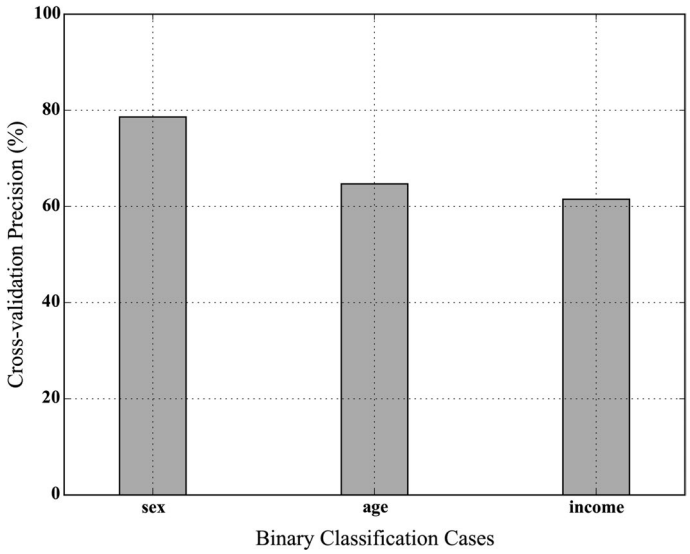

Fig. 10 Cross-validation accuracy for sex, age and income prediction. Sex of the shower user is easier to predict than the rest of the demographics

or not, or has income less than 3000 CHF or not. The features of each record were the same in all three cases.

Procedure We ran the probabilistic classifier in order to perform binary classification of the shower events for each of the target variables. Tenfold *cross-validation* was also used in this case. The latter, similarly to the previous case, helped us determine the optimal value of k parameter, which was set to 10.

Results The results obtained by the classification are illustrated in Fig. 10. The classifier achieved cross-validation accuracy of 78.6, 64.7 and 61.5% for sex, age and income prediction respectively. Despite the fact that a larger dataset would generate more confident results, it is safe to conclude to that predicting the age and income of a shower-taker based solely on her showers is a non-trivial task. On the other hand, sex is easier to predict as it directly affects the actions of a person during a shower.

Conclusions

In this article, we have presented a novel distributed processing framework, which supports a probabilistic classification and a regression approach. Its core algorithm is an extension of a distributed approximate kNN implementation, optimized to operate efficiently in a single distributed session. It was implemented with the Apache Flink scalable data processing engine.

We have conducted a detailed experimental evaluation in order to assess the framework in terms of wall-clock completion time and scalability, comparing it with similar approaches based on Apache Spark and Apache Hadoop. Our framework outperformed all competing implementations, managing to execute significantly faster on the same workloads and scale better for larger datasets, as a result of our optimizations and its ability to execute in a single distributed session. Furthermore, we performed two water

consumption related real-world case studies, demonstrating our framework's potential in knowledge extraction tasks on data of very high volume.

With real-world data collection continuously evolving in all fields, knowledge extraction from daily activities becomes very challenging. The directions for future work are as follows. We will perform extended case studies on more datasets from various sources, in order to establish our framework's ability in performing ad-hoc data mining tasks. Furthermore, we will explore its applicability on data stream mining applications, where the input is a continuous flow of data records. Finally, we will enhance our framework's knowledge discovery capacity, by extending it with more distributed machine learning approaches, in an attempt to raise its potential on the continuously growing field of Big Data analytics.

Abbreviations
FML: Flink Machine Learning; kNN: k-nearest neighbors; SFC: Space Filling Curves; LSE: least square error; RMSE: root mean squared error; SWM: smart water meter; YARN: Yet Another Resource Negotiator; HDFS: Hadoop Distributed File System.

Authors' contributions
GC's contribution was researching and developing the presented algorithms, performing the experiments and writing the manuscript. SK provided scientific guidance and support for all of the work presented in this paper and contributed in writing the manuscript. SA and SS contributed in supervising this work, as well as in evaluating the manuscript. All authors read and approved the final manuscript.

Author details
[1] Department of Informatics and Telecommunications, University of Peloponnese, Karaiskaki 70, 22100 Tripolis, Greece. [2] Institute for the Management of Information Systems, ATHENA R.C., Artemidos 6 & Epidavrou, 15125 Maroussi, Athens, Greece. [3] Harokopio University, Eleftheriou Venizelou 70, 17676 Kallithea, Athens, Greece.

Acknowledgements
The infrastructure for experimentation was provided by the University of Peloponnese (https://www.uop.gr).

Competing interests
The authors declare that they have no competing interests.

Ethics approval and consent to participate
Not applicable.

Availability of data and materials
The code of single and three session versions of FML-kNN and S-kNN is available at https://github.com/DAIAD/FML-kNN_2017_paper. The smart water meter and shower datasets are available at https://github.com/DAIAD/data.

Funding
This work was partially funded by the General Secretariat for Research and Technology (GSRT), the Hellenic Foundation for Research and Innovation (HFRI) and the EU project "DAIAD - Open Water Management—from droplets of participation to streams of knowledge" under Grant agreement No. 619186.

Publisher's Note

Received: 19 November 2017 Accepted: 22 January 2018
Published online: 06 February 2018

References
1. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. Commun ACM. 2008;51(1):107–13.
2. Alexandrov A, Bergmann R, Ewen S, Freytag JC, Hueske F, Heise Arvid, Kao O, Leich M, Leser U, Markl V, et al. The stratosphere platform for big data analytics. VLDB J. 2014;23(6):939–64.
3. Galilee J, Zhou Y. A study on implementing iterative algorithms using big data frameworks. In: School of IT Research Conversazione Posters; 2014.
4. Flink jobs and scheduling. https://ci.apache.org/projects/flink/flink-docs-master/internals/job_scheduling.html. Accessed 21 Dec 2017.
5. Böhm C, Krebs F. The k-nearest neighbour join: turbo charging the KDD process. Knowl Inf Syst. 2004;6(6):728–49.

6. Chatzigeorgakidis G, Karagiorgou S, Athanasiou S, Skiadopoulos S. A MapReduce based k-NN joins probabilistic classifier. In: Proceedings of IEEE BigData; 2015. P. 952–7.
7. Zhang , Li F, Jestes J. Efficient parallel KNN joins for large data in MapReduce. In: Proceedings of EDBT; 2012. P. 38–49.
8. Amancio DR, Comin CH, Casanova D, Travieso G, Bruno OM, Rodrigues FA, da Fontoura Costa L. A systematic comparison of supervised classifiers. PloS ONE. 2014;9(4):e94137.
9. Yang Y. An evaluation of statistical approaches to text categorization. Inf Retr. 1999;1(1):69–90.
10. Colas FPR, et al. Data mining scenarios for the discovery of subtypes and the comparison of algorithms. Leiden: Leiden Institute of Advanced Computer Science (LIACS), Faculty of Science, Leiden University; 2009.
11. Zhang ML, Zhou ZH. A k-nearest neighbor based algorithm for multi-label classification. In: 2005 IEEE International Conference on Granular Computing, Vol. 2; 2005. P. 718–21.
12. Oswald RK, Scherer WT, Smith BL. Traffic flow forecasting using approximate nearest neighbor nonparametric regression. Center for Transportation Studies, University of Virginia; 2001.
13. Wei L, Keogh E. Semi-supervised time series classification. In: Proceedings of ACM SIGKDD; 2006. P. 748–53.
14. Xu J. Multi-label weighted k-nearest neighbor classifier with adaptive weight estimation. In: International conference on Neural Information Processing; 2011. P. 79–88.
15. Gou J, Xiong T, Kuang Y. A novel weighted voting for k-nearest neighbor rule. JCP. 2011;6(5):833–40.
16. Berchtold S, Keim DA. Indexing high-dimensional space: database support for next decades's applications. In: Proceedings of ACM SIGMOD; 1998.
17. Liao S, Lopez MA, Leutenegger ST. High dimensional similarity search with space filling curves. In: Proceedings of IEEE ICDE; 2001. P. 615–22.
18. Sagan H. Space-filling curves. New York: Springer; 2012.
19. Mokbel MF, Aref WG, Kamel I. Performance of multi-dimensional space-filling curves. In: Proceedings of ACM SIGSPATIAL; 2002. P. 149–54.
20. Yao B, Li F, Kumar P. K nearest neighbor queries and knn-joins in large relational databases (almost) for free. In: Proceedings of IEEE ICDE; 2010. P. 4–15.
21. Lawder JK, King PJH. Querying multi-dimensional data indexed using the hilbert space-filling curve. ACM SIGMOD Rec. 2001;30(1):19–24.
22. Faloutsos C. Multiattribute hashing using gray codes. ACM SIGMOD Rec. 1986;15:227–38.
23. Song G, Rochas J, Huet F, Magoulès F. Solutions for processing k nearest neighbor joins for massive data on MapReduce. In: Proceedings of PDP; 2015. P. 279–87.
24. Stupar A, Michel S, Schenkel R. Rankreduce: processing k-nearest neighbor queries on top of mapreduce. In: Proceedings of LSDS-IR; 2010. P. 13–8.
25. Indyk P, Motwani R. Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of ACM STOC; 1998. P. 604–13.
26. Chen F, Dai J, Wang B, Sahu S, Naphade M, Lu C. Activity analysis based on low sample rate smart meters. In: Proceedings of ACM SIGKDD. 2011. P. 240–8.
27. Naphade M, Lyons D, Kohlmann CA, Steinhauser C. Smart water pilot study report. https://www.cityofdubuque.org/1348/Smarter-Water. Accessed 31 Jan 2018.
28. Silipo R, Winters P. Big data, smart energy, and predictive analytics. Time Ser Predict Smart Energy Data. 2013;1:37.
29. Schwarz C, Leupold F, Schubotz T, Januschowski T, Plattner H. SAP Innovation Center. Rapid energy consumption pattern detection with in-memory technology. Int J Adv Intell Syst. 2012;5(3&4):415–26.
30. Kermany E, Mazzawi H, Baras D, Naveh Y, Michaelis H. Analysis of advanced meter infrastructure data of water consumption in apartment buildings. In: Proceedings of ACM SIGKDD; 2013. P. 1159–67.
31. Lawder JK. Calculation of mappings between one and n-dimensional values using the hilbert space-filling curve. Technical report, Birkbeck College, University of London; 2000.
32. Daiad repository. https://github.com/daiad/utility-flink-fml-knn. Accessed 15 Nov 2017.
33. Big data benchmark. http://prof.ict.ac.cn/bigdatabench/. Accessed 15 Nov 2017.
34. House-Peters LA, Chang H. Urban water demand modeling: review of concepts, methods, and organizing principles. Water Resour Res. 2011. https://doi.org/10.1029/2010WR009624.
35. Singh N, Singh D. Performance evaluation of k-means and heirarichal clustering in terms of accuracy and running time. Int J Comput Sci Inf Technol. 2012;3(3):4119–21.
36. Panda S, Sahu S, Jena P, Chattopadhyay S. Comparing fuzzy-C means and k-means clustering techniques: a comprehensive study. In: Advances in Computer Science, Engineering & Applications; 2012. P. 451–60.
37. Sehga G, Sehga DK. Comparison of various clustering algorithms. Int J Comput Sci Inf Technol. 2014;5(3):3074–6.
38. JUng YG, Kang MS, Heo J. Clustering performance comparison using k-means and expectation maximization algorithms. Biotechnol Biotechnol Equip. 2014;28(sup1):S44–8.
39. Yi BK, Jagadish HV, Faloutsos C. Efficient retrieval of similar time sequences under time warping. In: Data Engineering, 1998. Proceedings., 14th International Conference on, IEEE; 1998. P. 201–8.
40. Kodinariya T, Makwana PR. Review on determining of cluster in k-means clustering. Int J. 2013;1:90–5.
41. Davies DL, Bouldin DW. A cluster separation measure. IEEE Trans Pattern Anal Mach Intell. 1979;1(2):224–7.