Knowledge-Based Systems 206 (2020) 106374

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

Fast anytime retrieval with confidence in large-scale temporal case bases

Mehmet Oğuz Mülâyim *, Josep Lluís Arcos

Artificial Intelligence Research Institute, IIIA-CSIC, Spanish National Research Council, Campus UAB, 08193, Barcelona, Spain

ARTICLE INFO

Article history:
Received 30 January 2020
Received in revised form 15 May 2020
Accepted 4 August 2020
Available online 12 August 2020

Keywords:
Large-scale case-based reasoning
Exact and approximate k-nearest neighbor search
Anytime algorithms

ABSTRACT

This work is about speeding up retrieval in Case-Based Reasoning (CBR) for large-scale case bases (CBs) comprised of temporally related cases in metric spaces. A typical example is a CB of electronic health records where consecutive sessions of a patient forms a sequence of related cases. k-Nearest Neighbors (kNN) search is a widely used algorithm in CBR retrieval. However, brute-force kNN is impossible for large CBs. As a contribution to efforts for speeding up kNN search, we introduce an *anytime* kNN search methodology and algorithm. *Anytime Lazy kNN* finds exact kNNs when allowed to run to completion with remarkable gain in execution time by avoiding unnecessary neighbor assessments. For applications where the gain in exact kNN search may not suffice, it can be interrupted earlier and it returns best-so-far kNNs together with a confidence value attached to each neighbor. We describe the algorithm and methodology to construct a probabilistic model that we use both to estimate confidence upon interruption and to automatize the interruption at desired confidence thresholds. We present the results of experiments conducted with publicly available datasets. The results show superior gains compared to brute-force search. We reach to an average gain of 87.18% with 0.98 confidence and to 96.84% with 0.70 confidence.

1. Introduction

Industrial scale machine learning (ML) systems have to deal with larger amounts of digital data everyday due to the exponential growth of both its generation and availability [1]. Being a member of the instance-based learning subdivision of the larger ML family, many Case-Based Reasoning (CBR) systems are not exempt from this laborious opportunity either. Reminiscent of human thinking, CBR is based on two assumptions observed in real world that similar problems have similar solutions and that problems are likely to recur [2]. Hence, it stores past problem solving experiences as cases in its case base (CB) and when a new query is made to the system, it retrieves similar past problems in its CB and reuses their solutions by adapting them to the query [3,4]. This type of reasoning is known as lazy learning in ML literature since a CBR system does not build a model prior to a query – as opposed to eager learning methods which do so – and generalizes its cases every time a query is posed to the system. This behavior is an advantage of CBR for continuously changing large CBs since it discards the need to re-train learned models with the updated data. However, due to its lazy nature, the efficiency of its retrieval phase affects overall system performance. And in practice, a growing CB eventually causes the so-called swamping utility problem [5,6] which emerges when adding new cases to a CB degrades the system efficiency instead of improving it.

Being simple and effective, k-Nearest Neighbor (kNN) search is a widely used algorithm in CBR retrieval in particular and in instance-based learning in general. The naïve approach to find the kNNs of a query is to perform a brute-force search in the CB by evaluating the similarity of each case to the given query and return the k most similar cases. The runtime complexity[1] of this method may be acceptable for small sized CBs, but it implies an excessive execution time for large-scale CBs due to expensive similarity calculations and is likely to evolve into above mentioned utility problem. There has been significant research to speed up nearest neighbor search (NNS) some of which we will review shortly in the next section. For occasions where the speed-up to find exact neighbors is still not computationally feasible, some efforts resorted to approximation methods to find approximate enough neighbors instead.

As a contribution to these efforts, to address both exact and approximate NNS, this article introduces an *anytime* kNN search

* Corresponding author.
E-mail addresses: oguz@iiia.csic.es (M.O. Mülâyim), arcos@iiia.csic.es (J.L. Arcos).

[1] Runtime complexity of brute-force kNN is $\mathcal{O}(ndk) + \mathcal{O}(nk \log k)$ where n is the number of instances the query is made against, d the dimension of each instance, k the number of nearest neighbors searched for; the first part of the complexity is for distance calculations and the second part is for sorting the neighbors.

https://doi.org/10.1016/j.knosys.2020.106374
0950-7051/

algorithm, *Anytime Lazy kNN* (*ALK*). We base our algorithm on a fast exact kNN algorithm *Lazy kNN* [7], and we extend it to a fully-fledged anytime algorithm. *ALK* finds exact kNNs when allowed to run to completion and otherwise, if interrupted, it returns best-so-far kNNs together with a *confidence* value attached to each neighbor. Confidence values reflect the expected qualities of approximate kNNs in terms of their similarities to the query compared to those of the exact kNNs. The proposed algorithm is also resumable and confidence values for approximate results increase over allocated time. Furthermore, we provide a means of confidence prediction to automatize the interruption of the algorithm by trading time with confidence in output.

ALK is equally efficient in exact kNN search as the original Lazy kNN in terms of avoiding unnecessary distance calculations that would be carried out by a standard brute-force search and it can save up significant execution time. Additionally, and as the main contribution of this work, we show that it reaches superior gains even when it is interrupted at very high confidence thresholds implying that we are very close to the exact kNNs. So, *ALK* gives the expert both the option to wait for the completion of the algorithm to obtain exact kNNs and the option to interrupt the search – manually and/or automatically – any time when a prompter response is needed and get best-so-far kNNs instead. In the latter case, he or she may also opt to resume the algorithm to get an output with a better confidence for the approximate results if so desires.

Anytime Lazy kNN, like its predecessor, excels specifically at domains where the CB can be organized as sequences of temporally related cases and the similarity metric takes into account the evolution of a sequence instead of treating each case individually. A good example to such domain can be found in *healthcare* where the electronic health record of a patient represents the sequence of his/her consecutive sessions and each new session is typically an update to this sequence. A search for similar patients regarding their medical histories should consider their session sequences. And, depending on whether the whole medical history or a part of it is queried, the similarity metric would use a time window encompassing the complete sequence or a subsequence of the health record respectively. Another natural example can be a *time series* (TS) dataset, where each instance is a sequence of temporally observed data. Here, each data point is essentially an update to the sequence and to assess the similarity between a query and a TS sequence in the dataset, the time window can cover the sequence fully or partially.

This article is organized as follows. Section 2 gives a background for our proposal. In Section 2.1, first we briefly review the main approach in CBR community to overcome the utility problem, then we mention various methods developed so far to speed up NNS in instance-based learning in general. Section 2.2 clarifies the concepts we use throughout this paper describing the organization of a CB for our domain of interest. In Section 2.3 we present information on the components and desired characteristics of anytime algorithms. We present the details of our proposed anytime kNN search methodology and our *Anytime Lazy kNN* algorithm itself in Section 3. Section 4 describes how we evaluated our algorithm and Section 5 gives highly encouraging results of experiments we conducted with real-world small to large time series datasets. Finally, we discuss the outcomes and future work in Section 6.

2. Background

2.1. Related work

Beside the obligation to deal with large scale data that comes with ever-growing CBs, current availability of the tools to interpret big data is also encouraging CBR researchers to work on systems that could benefit from hundreds of millions of cases (e.g. [8,9]). Working with CBs of this scale could not be imagined until recently. Quite the contrary, till today the main approach in CBR community to tackle this problem has been to control the CB growth via case base maintenance (CBM) techniques while preserving the competence of the overall CB (for competence definition see [10], for CBM examples see [11,12], for CBM dimensions see [13]). Of course, big data tools do not eliminate the need for CBM altogether, maintaining the correctness of cases and an adequate index are always useful for an efficient CBR. However, these tools may indeed outdate CB compression strategies as discussed in [8].

One of the efforts to overcome the computational overhead of kNN search has been to use parallel architectures (e.g. [14–16]). Other notable effort is using search trees such as k-d trees and its variants to partition the multidimensional search space and conduct NNS by pruning parts of the tree that cannot include the nearest neighbor (e.g. [17,18], see also [19] for a comparison of search tree based NNS algorithms). However, beside the cost of their construction and maintenance, k-d trees are prone to the curse of dimensionality [20] phenomenon and although improvement suggestions exist (e.g. [21]), they are usually not recommended for high dimensional spaces.[2]

Another common proposal to tackle the run-time bottleneck of NNS is using approximation in applications where approximate neighbors may be acceptable. Approximate methods retrieve a similar enough neighbor instead of the exact one in return for the improvement in retrieval speed. Typical techniques for approximate matching are using search tree adaptations (e.g. [23]), hashing (e.g. [24]) or proximity graphs (e.g. [25]). In CBR context, an approximate retrieval would imply approximate solutions. An example to an adaptation method of approximate solutions that could compensate for the retrieval accuracy in large case bases is suggested in [26].

When the retrieval time is a dire constraint, thanks to their ability to provide always a solution upon interruption, anytime algorithms [27,28] have also been incorporated in NNS, such as in data stream mining (e.g. [29,30]), databases (e.g. [31]) and in CBR retrieval [32] among other applications. If interrupted before completion, an anytime CBR retrieval would return the k best cases found so far with respect to their similarities to the query.

Ueno et al. [30] proposes an anytime algorithm for nearest neighbor classification where they use a presorted (worst first) index which is created by assigning ranks to all instances based on their contribution in classification on the training set. This index is used as heuristics in NNS.

In Schaaf's "Fish and Sink" (FaS) [32], the CB also holds "aspect" distances of the cases among each other which are weighted according to the "view" of the user asking the query. The NNS in CB starts with a predefined order of cases and directly tested (DT) cases "sink" with regards to their distances to the query, dragging down their view neighbors with them and labeling those neighbors as indirectly tested (IDT) cases. The time of interruption is important and only after all cases are labeled either as DT or IDT, FaS can show best k cases found so far regarding the relative depths of the DT and IDT cases. A prior interruption yields unconfident results. If FaS is not interrupted, it tests and sorts all cases.

Ricci and Avesani [33] use an anytime algorithm in learning a local similarity metric to be used in the CBR retrieval where the distance around a trial case is measured using the metric attached to that case. Their anytime algorithm updates the distance between an input case c and its neighbors depending on the role

[2] A good rule of thumb is to use k-d trees only if $N \gg 2^d$, where N is the number of data points and d is the number of dimensions [22].

of the neighbors in solving c by incorporating a reinforcement learning procedure that adjusts the local weights.

On the other hand, with regard to the temporal dimension in CBR, Montani and Portinale [34] emphasize the importance of temporality especially in healthcare domain and propose a framework for the representation and retrieval of cases that are in the form of time series data in medical applications of CBR. The retrieval is addressed both at case and history levels, the latter taking into account the evolution of the temporally related cases.

As a more recent work, *Lazy kNN* [7] which our algorithm is based on is a novel NNS which significantly reduces the number of evaluated cases in kNN search for large-scale CBs composed of temporally related cases. Lazy kNN leverages the triangle inequality in metric spaces by using it as a cutoff in NNS and evaluates only the "true kNN candidate" cases for a given query. A further advantage of the algorithm is that it does not need an extra data structure for an overall partitioning of the problem space, instead, it keeps track of the evaluated cases per sequence throughout the sequence's history of updates.

2.2. Domain of interest

In CBR, a case typically consists of three parts: the *problem* representing the query, the *solution* suggested by the CBR system for the query and the *outcome* (feedback) after applying the solution. In this work, we only take into account problem parts of cases for CBR retrieval and use them as queries in kNN search. And we are particularly interested in domains where the similarity measure takes into account the history of updates to a sequence of temporally related cases instead of treating each update as an individual case.

For clarification purposes, the concepts *sequence*, *update*, *time window* and *query* that are used in this paper regarding our domain of interest are depicted in Fig. 1.

A problem sequence is created for each particular initial problem (e.g. a sequence is created for each patient in their first session) to represent the history of related problems to follow. We regard each consecutive problem related to this sequence as an update to the sequence. Consecutive queries for a problem sequence to be used in kNN search for CBR retrieval are formed as follows: The initial problem (which can be seen as the 0th update) for every sequence is the first query. Each following update invokes the generation of a new query by applying a *time window* onto the sequence. In this work, we use *expanding* and *fixed-width* time window approaches. For each consecutive update, the former approach expands the time window encompassing whole problem sequence and the latter approach slides the time window of width w encompassing the last w updates to form the new query. Note that both approaches envelop the same updates for the first w problem updates, and for a given sequence, they generate the same number of queries (e.g. 10 queries in total in Fig. 1).

In CBR context, after CBR's problem solving episode for a query, the query is saved as the problem part of a new case. For our domain, all consecutive episodes for the queries of the same problem sequence thus form a sequence of *temporally related cases*.

In implementation, case generation by time window approach can be conceptual or literal. In other words, there may be only a single data structure representing a problem sequence, and the case for a particular update can be formed on the fly when needed. Or, there may be l data structures for the cases of a sequence of length l.

The problem updates encompassed by the time window comprise the problem part of the case which is used by the similarity measure in kNN search of the CBR retrieval conducted for the query. Both approaches generate cases that encompass different number of updates. The CBR system designer should decide whether cases of different lengths should be compared in kNN search and if this is deemed necessary, the similarity metric – which has to satisfy the *triangle inequality* for our algorithm – should be implemented accordingly.

In the following subsection we highlight the important characteristics and concepts of anytime algorithms as a background for the proposed algorithm that excels in domains we have just described.

2.3. Anytime algorithms

An anytime algorithm (AA) is a computer algorithm designed in such a way that given an input problem, it can provide a best-so-far solution at any time it is interrupted[3] and the solution is accompanied by a quality value reflecting how close the interruption output is to the exact solution. In the core of a typical AA lies a function which incrementally improves the solution. The *quality measure*[4] can be based on any characteristic of the output which is deemed important. It is preferable that the output quality monotonically increases over computation time and the improvement in quality is greater at the early stages of execution and it diminishes over time.

After being interrupted, an AA can resume its execution if it is allocated more computation time. AAs also bear statistical information about their output quality over execution time for the input data they received. This information makes them predictable and it can be used for meta-reasoning about allocated computation time [35] (see [28] for a list of desired properties of AAs and quality metrics). All these characteristics make AAs useful in application domains where complete computation to solve the problem at hand is ineffective or impossible in real-time.

Performance Profile (PP) of an AA is used in estimating the quality of the output of the algorithm as a function of the amount of execution time [36]. An AA can have several PPs to track different result attributes [37]. After devising a quality measure, the PP of an anytime algorithm is typically constructed by using a simulation method. The algorithm is run with numerous input instances which are provided within a training dataset that can be randomly generated using the domain knowledge and the quality of the results over execution time are recorded. This statistical data composed of *(execution time, output quality)* pairs forms the *Quality Map* of the algorithm. Once the quality map is obtained, the corresponding PP of the algorithm can be derived from it as a formula by means of curve fitting methods or it can be represented as a table which reflects the discrete probability distribution of quality for discrete time allocations. The latter representation is called the *Performance Distribution Profile* (PDP) of the anytime algorithm and helps us give more accurate decisions compared to a single value of a fitted function. In our work, we opted for PDPs (see [36,38] for further discussion on possible representations of PPs).

In accordance with the above description of a preferable AA, and regarding the focus of this work, an anytime kNN search algorithm is expected to monotonically improve on its k-nearest neighbors and provide a quality value attached to the best-so-far neighbors upon interruption. Also the improvement in output quality is preferred to be diminishing over time which means that the nearest neighbors found in early stages of search are almost as close to the query as the exact neighbors. Furthermore, the algorithm should be able to resume its execution without a major

[3] Some anytime algorithms may need a short initialization time before they can be interrupted, e.g. [30].

[4] We use the terminology for anytime algorithm components given in [28].

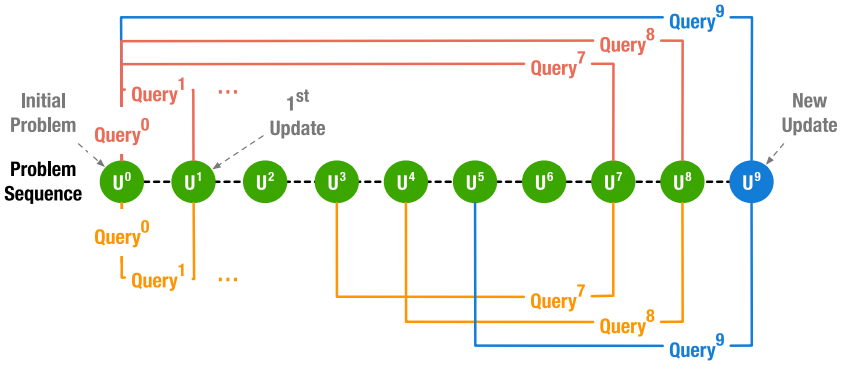

Fig. 1. Expanding (top) *vs* Fixed-width (below) Time Window approaches to form queries. A problem *sequence* is created for each particular initial problem. Each consecutive problem related to this sequence is an *update*. A *query* is generated for each update, including the initial problem (i.e. 0th update). To form the query for the uth problem update; the former approach *expands* the *time window* encompassing all existing $u+1$ updates; while the latter approach, for a *fixed-width* w, slides the window on the sequence covering the last w updates only (above w=5).

overhead if it is allocated extra computation time to improve on its results.

In the following section, after discussing the challenges in building such a well-tempered anytime kNN search algorithm, we point out why we deem Lazy kNN a very good candidate to be converted into an AA which overcomes these difficulties. Then, we provide the steps of this conversion as we detail the implementations of the above mentioned AA concepts at each step until we ultimately achieve our fully-fledged *Anytime Lazy kNN* search algorithm.

3. Anytime Lazy kNN

The main difficulty of converting an exact kNN search to an anytime algorithm lies in the quality assessment of the best-so-far neighbors. Having the search interrupted, we would like to compare the similarities of approximate and exact kNNs to the query. However, it is impossible to build an accurate quality measure for such an assessment. The reason is obvious, exact kNNs remain unknown till the end of search and even though we might have already found them earlier, we cannot be aware of this until we evaluate all candidates in the search space. Consequently, for an algorithm which searches kNNs all at once (e.g. brute-force kNN search), exact kNNs are available only as a whole and after the completion of the search.

As an enhancement to partially ease the constraint of having to wait till completion, if we conduct kNN search in an incremental fashion finding k-nearest neighbors one by one in k iterations, we would at least guarantee the exactness of the top i-1 NNs when interrupted at the ith iteration ($i \leq k$). Of course, this method would make sense only if extra iterations do not imply an additional cost of redundant similarity calculations. And clearly, although incremental search bears the possibility of providing some of the exact kNNs upon interruption, it does not eliminate the need to assess the quality of the remaining kNN list members that are yet-to-be ascertained for exactness.

In this section, we first deal with the design of an anytime kNN algorithm that exhibits the desired performance profile outlined in Section 2.3. For the core of the AA, we propose to extend an existing algorithm, Lazy kNN [7]. Specifically, although Lazy kNN was designed to provide exact kNNs as a whole list after running to completion, we show how it can be converted to an incremental kNN search without any redundant computation in Section 3.1. Then, we explain how Incremental Lazy kNN can be used in the core of an AA and present the pseudo-code of *Anytime Lazy kNN* (henceforth *ALK*) in Section 3.2.

Once we have the AA algorithm, we introduce the steps required to define the appropriate mechanism to assess the quality of best-so-far kNNs at any given moment. In AA literature, when accuracy is not an option as a metric to build a deterministic quality measure, it is common to resort to a certainty metric to reflect a degree of correctness of intermediate results, e.g. by using the probability distribution of output quality over time [28].

Fig. 2 illustrates the steps of the generation and use of the probabilistic quality measure that we implemented for our algorithm. Top two steps are to build the PDP of *ALK* for an application domain. In the first step, *ALK* runs kNN search simulations and gathers statistical data to generate the Quality Map of the algorithm. We detail this step in Section 3.3. In the second step which is described in Section 3.4, PDP is built out of the quality map. PDP endows our algorithm with the ability to predict the *expected quality* of best-so-far kNNs upon interruption. Henceforth, in concordance with CBR literature [39], we refer to expected output quality as *confidence* and define our confidence estimator based on PDP in Section 3.5. We also show how PDP helps us to automatize the interruption upon reaching a given confidence threshold. In the final step of Fig. 2, *ALK* is now ready to serve as an anytime kNN search algorithm. It accepts a query and an optional interruption point, and returns the exact/best-so-far kNNs together with a confidence value for each neighbor.

3.1. Incremental Lazy kNN

Lazy kNN's strength in speeding up exact kNN search comes from the evaluation of only the true kNN candidate cases in the CB for a query. During the execution of Lazy kNN, best-so-far kNN list members and their positions in the list are subject to change until the last candidate case in the CB is evaluated. And, this is only when the exact kNN list is provided as output. This is because, with respect to the triangular inequality used in the candidacy assessment, even the last candidate can potentially surpass the nearest neighbor found so far. Due to this nature of having to run to completion, Lazy kNN cannot say how confident it is of its best-so-far kNNs when interrupted.

To gradually improve on each of the kNNs and to be able to provide at least some of the exact NNs upon interruption, we developed an incremental version of Lazy kNN where we basically invoke the original algorithm k times iteratively. And, at each iteration i we find the ith exact NN. Thus, if the algorithm is interrupted during the ith iteration, it ensures that the top i-1 NNs are the exact NNs of the query.

The beauty of this conversion from standard to incremental kNN search is that, despite reiterations, Incremental Lazy kNN does not carry out any redundant similarity calculation compared to the original version of the algorithm. Though surprising it might be at first glance, this behavior is due to the fact that we

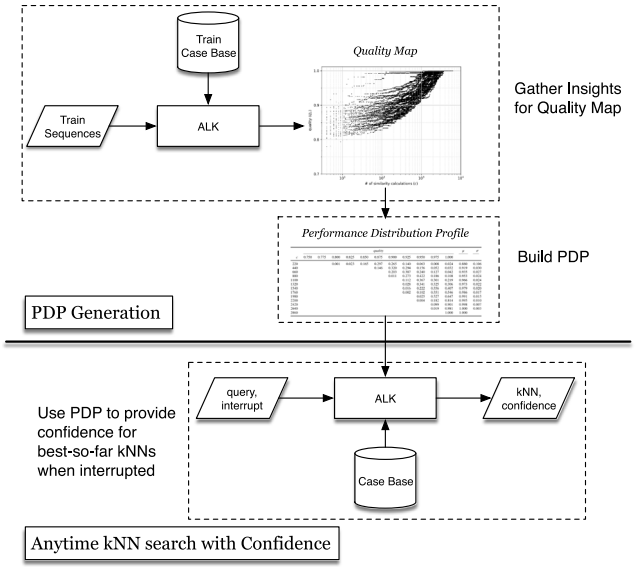

Fig. 2. Generation and use of confidence. First, the *Quality Map* of the algorithm for the application domain is created out of interruption simulations with training data. Then, *Performance Distribution Profile* is generated out of Quality Map. PDP endows *ALK* with the ability to attach *confidence* values to its best-so-far kNNs when interrupted. PDP also provides a means to automatize interruption at given confidence thresholds.

Algorithm 1: Anytime Lazy kNN

```
1  Class AnytimeLazyKNN:
       Attributes: _k: k of kNN
                   _current_iter: Iteration index to start/resume the NNS from
                   _current_index: Index to the candidate assessment in search
                   space to start/resume the NNS from
                   _query: The query formed for the problem update in the
                   sequence for which the NNS is to be started/resumed
       Methods : _Construct(k):
                 IncrementalLazyKNN(query, interrupt)
2  Function _Construct(k):
3      this._k ← k
4      this._current_iter ← 1
5      this._current_index ← null
6      this._query ← null
7      return this
8  Function IncrementalLazyKNN(this, query, interrupt):
9      if query <> this._query then                          // Update ?
10         this._query ← query
11         this._current_iter ← 1
12         this._current_index ← null
13     for iter ← this._current_iter to this._k do           // Iterate × k
14         kNN, this._current_index ← LazyKNN*(iter, this._query, interrupt,
           this._current_index)
15         if this._current_index <> null then               // Interrupt ?
16             this._current_iter ← iter
17             break
18     return kNN, confidence(kNN, interrupt)
```

always evaluate the minimum number of candidate cases at each iteration, and after k iterations, the total number of assessed candidates equals the number of assessments made by the original Lazy kNN.

With respect to the desired monotonicity property of an AA, we can argue that exact kNN search in general exhibits monotonicity since any kNN search algorithm could maintain best-so-far neighbors even if it cannot improve any of them after a new neighbor candidate evaluation, and provide these when interrupted. However, for all-at-once algorithms this gradual improvement is for the kNN list as a whole (i.e. any member can be improved during search any time). On the other hand, Incremental Lazy kNN possesses a monotonicity at individual nearest neighbor level that serves better for AA purposes. Given more time, current exact NNs will not change but the approximate ones are likely to be replaced by nearer neighbors, eventually all kNNs becoming the exact ones.

We note that the incremental nature of our algorithm is analogous to Broder's incremental NNS [40] which also finds kNNs iteratively starting from the first, ending with the kth; but it differs from the incremental retrieval concept of Cunningham et al. [41] and Jurisica et al. [42] because there, the iteration takes place in conversational CBR systems where the retrieval is incrementally refined via iterative user interactions.

The following subsection gives our *Anytime Lazy kNN* search algorithm with Incremental Lazy kNN at its core.

3.2. Anytime Lazy kNN algorithm

Besides monotonicity, another desired property of AAs is preemptability [28], that is, the capability to resume their execution after being interrupted. Incremental Lazy kNN can easily be made resumable by introducing two attributes to the algorithm to preserve the point in NNS where the interruption has occurred: (1) current iteration and, (2) the index of the next candidate to be assessed.

The simplified pseudo-code of *ALK* is given in Algorithm 1. For every particular problem sequence, a unique instance of the `AnytimeLazyKNN` class is instantiated (e.g. one instance for each patient). Then, each update (including the initial problem) to the sequence is passed as a *query* to the `IncrementalLazyKNN` method together with an optional interruption point (*interrupt*). *_current_iter* and *_current_index* are the two above-mentioned private instance attributes respectively to be used for resumability.

`IncrementalLazyKNN` uses a slightly extended version (`LazyKNN*`) of the original Lazy kNN which accepts two additional optional arguments: *interrupt* and *_current_index* and returns new *_current_index* beside kNNs. `LazyKNN*` is iteratively called k times, each time passing the iteration index *iter* as the k argument. The return value for *_current_index* is *null* if no interruption occurs, otherwise it points to the first candidate to be assessed if the algorithm is resumed.

Upon exit, `IncrementalLazyKNN` method returns the *kNN* list and the confidence of the algorithm for each member of the list. If the algorithm is run to completion, the kNNs will be exact and their confidence values will be 1. If the algorithm is interrupted, best-so-far kNN list is returned together with the expected quality values for each member of the list provided by the `confidence` system function which is based on the PDP of the algorithm that we will explain in Section 3.4.

But, as explained in 2.3, before building the PDP of *ALK*, we need to generate its quality map. And in the following subsection we show how we do it after defining our quality measure.

The complete code of *ALK* including all its functionality that will be covered throughout the rest of this article is publicly available at: https://github.com/IIIA-ML/alk.

3.3. Quality measure and map

Quality measure of an AA is usually a function of execution time. However, in order to have a measure independent of the computer platform that the algorithm runs on, we opted to implement a measure which is a function of the number of similarity calculations carried out so far in kNN search. So, given a *query* and

a number of calculations c as the interruption point, we define the quality measure for *ALK* as follows:

$$\mathcal{Q}_c = \frac{sim(NN_c^k, query)}{sim(NN_E^k, query)} \tag{1}$$

where *sim* is the similarity metric, NN_c^k and NN_E^k are the best-so-far and exact kth NN respectively, $\mathcal{Q}_c$ gives the output quality. If need be, c can be translated into actual execution time for the platform used.

Having the quality measure, the quality map can be generated as follows. After each similarity calculation c made for an input *query*, the similarity of the best-so-far kth neighbor (i.e. the dividend in Eq. (1)) is recorded. When the simulation ends for that query, the similarity of the exact kth NN (i.e. the divisor in Eq. (1)) is obtained. Then, by backtracking the simulation, the quality $\mathcal{Q}_c$ that we would get after each similarity calculation c is calculated using the recorded *sim* data. Eventually, all $(c, \mathcal{Q}_c)$ pairs collected during simulations provide us with the *QualityMap* of our algorithm.

However, this quality map does not reflect neither the temporal relations between cases nor the incremental nature of *ALK*. If we want a finer-grained quality measure incorporating these characteristics as well, we may add two more dimensions to the map. Since *ALK* finds the kNNs in an incremental fashion, the first extra dimension would be the index i of a nearest neighbor in the kNN list. This would allow us to assign a quality value per neighbor.

As for the temporal dimension, we could use the index u of the update on the problem sequence for which the query is generated for. This dimension provides even a finer-grained quality map, because, the more updates are covered by the time window, the less difference will have been introduced by the new update. Therefore, the more similar will be two successive queries and intuitively, the more similar will be their neighbors. Consequently, for a new query covering multiple updates, although *ALK* needs to evaluate all candidates within the neighbors of prior queries for the sake of mathematical exactness, it is very likely that the exact kNNs are found within the neighbors of the recent queries of the same sequence. As a result, this likelihood leads to higher quality values after fewer calculations. This phenomenon also helps *ALK* to fulfill the desired AA property of diminishing output quality values. In other words, the increase in the quality of best-so-far kNNs are likely to be higher in the early similarity assessments during the kNN search compared to the later assessments.

Hence, we define the finer-grained quality as follows:

$$\mathcal{Q}_c^{u,i} = \frac{sim(NN_c^{u,i}, query^u)}{sim(NN_E^{u,i}, query^u)} \tag{2}$$

where $NN_c^{u,i}$ is the ith NN returned by the algorithm when it is interrupted after c similarity calculations during the NNS for the uth query of a problem sequence, and $NN_E^{u,i}$ is the exact ith NN for the same query.

Fig. 3 shows an example to the quality map of *ALK* generated by using the quality measure in Eq. (2). It is generated throughout simulations with input sequences taken from the *SwedishLeaf* dataset (see Section 4.2). The figure is a 2D excerpt of the 4D map plotted for the third nearest neighbor of the queries generated for the tenth updates (i.e. $\mathcal{Q}_c^{10,3}$) of input sequences.

3.4. Performance distribution profile

As the final step to have *Anytime Lazy kNN*, we use the quality map to build the PDP of our algorithm which is essentially the discrete probability distribution of output quality over computation time.

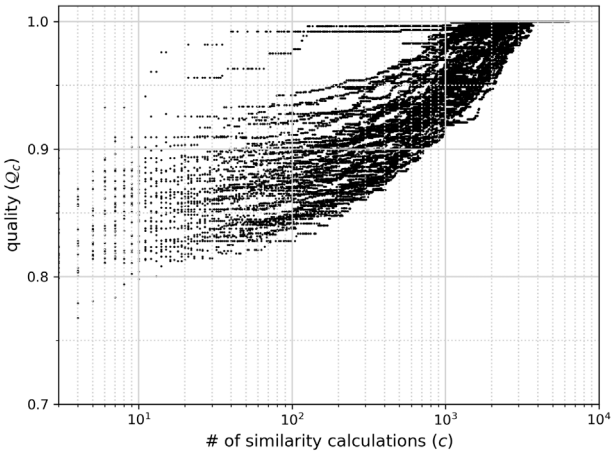

Fig. 3. Quality map of *Anytime Lazy kNN* for the *SwedishLeaf* dataset. Here the map is shown for the 10th update and the 3rd nearest neighbor. Every point on the map is an instance of output quality observed throughout simulations.

For this purpose, we first discretize the calculation range into m discrete calculation values $c_1, \ldots, c_m$ of equal intervals of δ_c, where c_m is the maximum number of similarity calculations performed for a test input during simulations. Then we discretize the quality range $[0, 1]$ into n discrete quality values $q_1, \ldots, q_n$ of equal intervals of δ_q.

In coherence with the finer-grained *QualityMap* presented above, we create a four-dimensional array $\mathcal{PDP}$ to hold the discrete probability distribution of quality. The dimensions of $\mathcal{PDP}$ are the maximum update number for an input sequence during simulations, k of kNN, the number of calculation intervals m and the number of quality intervals n respectively.

Thus, an entry in $\mathcal{PDP}[u, i, r, v]$ represents the discrete probability that the output quality of the best-so-far ith NN is in $(q_v - \delta_q, q_v]$ after a number of similarity calculations in $(c_r - \delta_c, c_r]$ made during the kNN search for the uth problem update.

The values of system parameters δ_c and δ_q, hence the size of the $\mathcal{PDP}$ can be adjusted with respect to the desired accuracy of the performance information. The smaller δ_c and δ_q are, the more accurate will be the expected quality predicted by the $\mathcal{PDP}$. The corresponding *Performance Distribution Profile* of the quality map in Fig. 3 is given in Table 1.

3.5. Confidence

Equipped with PDP, our algorithm becomes ready to predict the output quality when the kNN search for an unseen query is interrupted. We refer to the expected quality as the *confidence* of our algorithm in conformity with the CBR literature [39]. Specifically, we define our confidence measure as the weighted mean of the probability distribution of quality for a calculation interval in $\mathcal{PDP}$:

$$\mu_c^{u,i} = \sum_v q_v \, \mathcal{P}_c^{u,i}(q_v) \tag{3}$$

where $\mathcal{P}_c^{u,i}(q_v)$ is a shorthand for $\mathcal{PDP}[u, i, r_c, v]$ and r_c is the interval in the calculation range where c falls into. This equation gives us the confidence $\in [0, 1]$ of the best-so-far ith NN of the uth update when *ALK* is interrupted after c number of similarity assessments during kNN search. Since confidence gives us a mean value μ, we also provide the (weighted) standard deviation σ of the quality distribution to be used together with it:

$$\sigma_c^{u,i} = \sqrt{\sum_v \left(q_v - \mu_c^{u,i}\right)^2 \mathcal{P}_c^{u,i}(q_v)} \tag{4}$$

Table 1
Performance Distribution Profile with Confidence.

c	*Quality*											μ	σ
	0.750	0.775	0.800	0.825	0.850	0.875	0.900	0.925	0.950	0.975	1.000		
220			0.001	0.023	0.165	0.297	0.265	0.140	0.063	0.008	0.024	0.880	0.106
440						0.146	0.320	0.294	0.156	0.052	0.032	0.919	0.030
660							0.203	0.387	0.240	0.127	0.042	0.935	0.027
880							0.011	0.273	0.422	0.186	0.108	0.953	0.024
1100								0.112	0.367	0.301	0.219	0.966	0.024
1320								0.028	0.341	0.325	0.306	0.973	0.022
1540								0.016	0.222	0.356	0.407	0.979	0.020
1760								0.002	0.102	0.351	0.546	0.986	0.017
1980									0.025	0.327	0.647	0.991	0.013
2200									0.004	0.182	0.814	0.995	0.010
2420										0.099	0.901	0.998	0.007
2640										0.019	0.981	1.000	0.003
2860											1.000	1.000	

The $\mathcal{PDP}$ is essentially a 4-dimensional array; here the table for update=10 and the 3rd nearest neighbor in kNN entry corresponding to the quality map in Fig. 3 is shown. *confidence* for a calculation range is the expected quality which is defined as the weighted mean of the probability distribution of quality for that range. Note that $\delta_c = 220$ and $\delta_q = 0.025$ settings are used in the generation of this $\mathcal{PDP}$.

The confidence μ and its deviation σ for the quality map example in Fig. 3 are given in Table 1. In this $\mathcal{PDP}$ excerpt, we can see that σ is higher for early calculations when fewer candidates are assessed and it decreases as we assess more candidates. When c does not coincide exactly with the $\mathcal{PDP}$ calculation intervals, we use linear *interpolation* for both μ and σ.

Beside giving us a confidence value to reason with the best-so-far kNNs, the PDP of our algorithm also provides us with a means to automatize the interruption itself. This may be achieved in two ways: either by specifying a time of execution or reaching a desired confidence threshold for the output.

In the former case, the execution time can be translated into a number of similarity calculations depending on the computer system the algorithm is running on. In the latter case, the PDP gives us the estimated number of similarity calculations to reach a desired confidence. In either case, obtained number of calculations is used for interruption.

While selecting interruption points, the standard deviation of the probability distribution of quality can be taken into account. Thus, if we define the *confidence threshold* for interruption as $\mu+z\sigma$, the choosing of the z offset parameter, that is the number of σ's subtracted from or added to μ, would depend on how precautious or optimistic we want to be with the confidence provided by the $\mathcal{PDP}$ respectively.

The trustworthiness of a probabilistic model depends on how much of the plausible problem space is represented by the model. In other words, the more representative the input queries used in simulations are for our domain, the more we can trust our model in predicting the confidence of our algorithm for future queries. Regarding CBR's fundamental representativeness assumption, we can safely assume that the CB that we will train our model with is representative of our problem space. Nevertheless, as we will describe in Section 4.2, in our experiments we chose our training and test datasets in a way to enable a rigorous testing of the representativeness of the PDP as well.

4. Evaluation methodology

There have been three main goals for the development of *ALK*: (1) To be able to interrupt kNN search and get best-so-far kNNs when exact kNN search is not feasible; (2) to attach confidence values to best-so-far kNNs that indicate how much we can trust each one of them in the reasoning process; and, (3) to be able to automatize interruption upon reaching given confidence thresholds. The previous section detailed the steps of how we developed such an anytime algorithm and a *confidence* measure that gives us the estimated output *quality*.

Confidence plays a key role in *ALK* both to assess the quality of the best-so-far output and to determine thresholds to automatize interruption. Therefore, to evaluate whether we met our above-mentioned goals, first we need to have a notion of efficiency for our confidence estimation. In other words, we would like to know how much we can trust the confidence measure itself. Accordingly, we define our efficiency measure in Section 4.1 and explain how to interpret it. Later, in Section 4.2, we describe the real world time series datasets that we used for our experiments to build CBs each of which was organized as sequences of temporally related cases. In Section 4.3, we give the settings and insights of the experiments that we conducted both to demonstrate empirically that our confidence measure is efficient enough, and to provide evidence regarding the speed-up achieved by *ALK* when we interrupt the algorithm upon reaching given confidence thresholds.

4.1. Efficiency of confidence

Confidence μ is an estimator of how close the best-so-far kNNs are to the query compared with the exact kNNs. In other words, it is an estimator of the expected quality of the best-so-far kNNs. Therefore, to measure the efficiency of a confidence estimation, we utilize the ratio of the confidence μ and the observed actual quality $\mathcal{Q}$ and we formally define *efficiency* as follows:

$$\eta_c^{u,i} = \frac{\mu_c^{u,i} + z\sigma_c^{u,i}}{\mathcal{Q}_c^{u,i}} \tag{5}$$

We use a $z\sigma$ offset value to be able to incorporate the deviation of the confidence provided by the $\mathcal{PDP}$ along with the μ itself upon interruption. The factor z can be chosen regarding how prudent we want to be with raw confidence values. $z=-1$ would be a more cautious choice than the neutral $z=0$, and $z=1$ would be a more optimistic one. An efficiency value $\eta \gg 1$ would signal an overconfident confidence measure that can possibly mislead reasoning with best-so-far kNNs; whereas $\eta \ll 1$ would imply an overcautious measure suggesting longer times of execution till reaching an acceptable approximation. On the other hand, while interpreting efficiency, we should bear in mind that due to its discrete nature, the precision of $\mathcal{PDP}$ (i.e. the choosing of δ_c and δ_q) affects the accuracy of quality estimation as well.

An example plot for the efficiency of confidence is shown in Fig. 4. Interruption tests for this plot were conducted on four different CBs generated out of the *SwedishLeaf* dataset by four different w and *step* settings. The confidence thresholds for interruption were selected with $z = -1$ setting. For all CBs, we

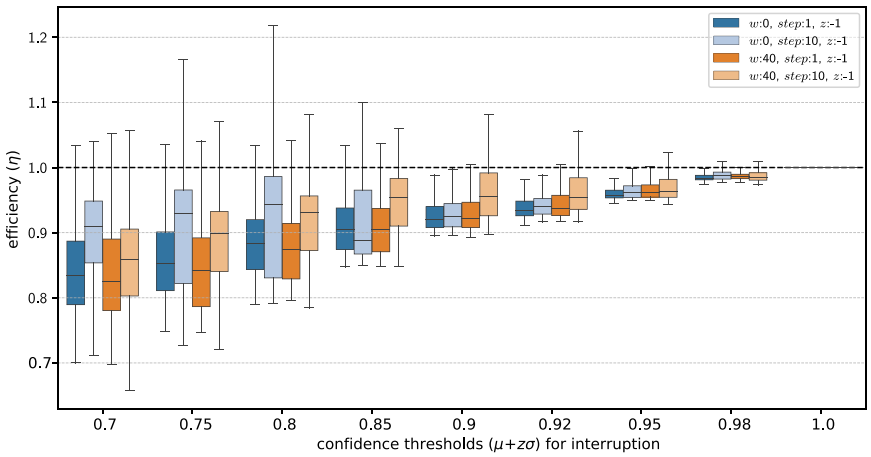

Fig. 4. Efficiency of confidence. $\frac{\mu+z\sigma}{Q_*}$ ratio for interruption tests on the *SwedishLeaf* dataset, with z=−1 and different time window width w and *step* settings and using the PDP for which an excerpt is given in Table 1.

see that the dispersion in efficiency is larger for lower confidence thresholds and it diminishes for higher thresholds. But, the efficiency almost consistently remains below 1.0, converging to 1.0 as the confidence threshold also gets closer to 1.0, to which we ultimately reach when we have the exact kNNs. This is a behavior that we desired by setting the z parameter to -1 preferring to be precautious with the confidence provided by the PDP. And indeed, Fig. 4 shows that, when the algorithm was interrupted at μ-$\sigma \in \{0.7, \ldots, 0.98\}$, the actual quality of the best-so-far kNNs were higher than these confidence threshold values.

4.2. Datasets

We used eleven univariate time series datasets available at the UEA & UCR Time Series Classification Repository [43] as real world data in our experiments. Every dataset in the repository is available as a two-pack of train and test sub-datasets. Some test sub-datasets are larger than their train reciprocals. Since, for the scope of this article, we are only interested in the retrieval of the problem parts of cases and not in classification with their solutions, we took the liberty to use the larger one to generate the CB_TRAIN to be used in building the PDP for a dataset.

On the other hand, as in any probabilistic model, the representativeness of the PDP built for a specific training CB is crucial for the efficiency of the confidence estimation when the same PDP is used for future queries. Therefore, to be able to test the representativeness of the PDP as well, we carried out interruption tests on the CB_TEST generated out of the corresponding smaller sub-dataset. In other words, in interruption tests both the CB and the test sequences were unseen.

To build a CB out of a given univariate TS dataset, as described in Section 2.2, we treated each instance in the dataset as a *sequence* for our CB and each data point of the TS instance as an *update* to the sequence. Then we applied *time window* on every instance to create individual *cases* of the sequence. Each window represented the problem part of a case and we regarded each data point enveloped by the window as a *feature* of that case. We adopted two approaches to implement time windows. The first approach used a sliding window of *fixed-width* and the second one an *expanding* window. For a TS instance of length l (i.e. having l data observations), both approaches generated l cases.

To generate even more diverse CBs out of a given TS dataset, we also incorporated time window *step* concept. With a $step > 1$ setting, we moved the time window in steps over the sequence and we generated cases for every *step* number of updates instead of doing it for each update in a sequence. Having $step \geq 1$ as an optional parameter, for a problem sequence of length l, both expanding and fixed-window approaches generated $\lceil l/step \rceil$ number of cases. Table 2 summarizes the datasets and their corresponding CBs used in our experiments to build PDPs.

4.3. Experiment settings

With respect to distance measures, *ALK* is based on Lazy KNN algorithm that relies on distance measures that are true metrics as mentioned in sub Section 2.1. Therefore, we used normalized *euclidean distance* in all experiments.[5] For every CB, normalization was done by taking into account the *min* and *max* values of the related univariate TS dataset. And the similarity *sim* between a *query* and a *case* in CB was defined as: $sim(query, case) = 1 - distance(query, case)$.

As mentioned in Section 2.2, regardless of the time window approach used, a decision has to be made with respect to how to measure the distance between two cases of different number of features. A straightforward decision could be not measuring this distance at all and returning 0. However in order to test our algorithm with larger CBs, in our experiments, we opted to extend the shorter case to the length of the longer one by filling in missing features with values that maximized the distance.

After deciding the similarity assessment method, for each TS dataset given in Section 4.2, we launched four experiments for combinations of two time window approaches (*Expanding* and fixed-width time window of width w=40) and two window step settings ($step \in [1, 10]$). Having k set to 9, each experiment for a configuration 3-tuple of $(dataset, w, step)$ was conducted in three stages:

In the first stage, using the larger TS sub-dataset, we generated a set of sequences of temporally related cases for the experiment configuration. Then, we split this set into two parts; where one part served as the CB_TRAIN and the other part as input test sequences for our algorithm. For each input sequence, we generated input queries along its updates starting from the initial problem. By feeding the algorithm with these queries over CB_TRAIN, we generated the Quality Map for the dataset as described in Section 3.3.

Consecutively, in the second stage, we built its PDP as described in Section 3.4. While building PDP for each dataset, we

[5] Interested reader can refer to [44,45] for empirical comparison of similarity measures for time series data in general. Mueen et al.'s work [46] may be of special interest for time series where they also benefit from the triangle inequality for early abandoning of the costly distance measuring to find exact motifs.

Table 2
Time series datasets used to generate CBs of temporally related cases in experiments for building PDPs, and their corresponding CB sizes generated with two different time window *step* settings.

#	Dataset	Type	Sequences	Updates	CB Size	
					$step = 10$	$step = 1$
1	*PowerCons*	Device	180	144	2700	25,920
2	*SwedishLeaf*	Image	625	128	8125	80,000
3	*Strawberry*	Spectrograph	613	235	14,712	144,055
4	*EOGHorizontalSignal*	Medical	362	1250	45,250	452,500
5	*InsectWingbeatSound*	Sensor	1980	256	51,480	506,880
6	*ECG5000*	Medical	4500	140	63,000	630,000
7	*UWaveGestureLibraryX*	Motion	3582	315	114,624	1,128,330
8	*Yoga*	Image	3000	426	129,000	1,278,000
9	*Phoneme*	Sound	1896	1024	195,288	1,941,504
10	*Mallat*	Simulated	2345	1024	241,535	2,401,280
11	*MixedShapesRegularTrain*	Image	2425	1024	249,775	2,483,200

discretized the range of similarity calculations and the quality range with $\delta_c = \lceil c_m/400 \rceil$ and δ_q=0.05 interval settings respectively, where c_m was the highest number of similarity calculations reached for that experiment.

In the third stage, out of the smaller TS sub-dataset, we generated the CB_TEST and the set of unseen input queries for interruption tests, using the same method and time window configuration in the first stage. Then, we fed the algorithm with these queries over CB_TEST and for each query, we interrupted the algorithm using a set of confidence thresholds as interruption points.

In coherence with the dividend of the efficiency Eq. (5), confidence thresholds for interruptions were determined with a $z\sigma$ offset. We chose to be slightly cautious with PDP's confidence estimation and set z to -1. The thresholds were chosen for the 9th NN and taking into account the problem update index of the query. So, for example, given the uth query for a test sequence and a threshold of 0.95, the algorithm was automatically interrupted after the number of similarity calculations needed for the μ-σ provided by PDP for the 9th NN of a uth query to reach 0.95.

5. Results

In this section, we provide the results of average gains in similarity assessments upon interruption and average efficiency of confidence estimation along experiments to assess if we met our design goals for *ALK*. Precisely, at each interruption, similarities of the best-so-far kNNs to the query, and the confidence μ together with its deviation σ for each member of the kNNs were recorded. Finally, we let the algorithm finish and we obtained the similarities of the exact kNNs to the query. And later, by backtracking, we calculated the actual qualities $\mathcal{Q}$. This allowed us to calculate and record the efficiency η of the confidence for each NN using the Eq. (5). At interruptions, we also recorded the gain in similarity calculations for the query, i.e. the percentage of the avoided calculations upon interruption with respect to the total number of calculations that would have been carried out by a brute-force search.

Finally, recorded efficiency and gain values throughout experiments gave us the answers we were looking for. We show the average gain of the algorithm for the set of confidence thresholds used as interruption points in Table 3. In the same table, we also provide the average efficiency of confidence for each experiment together with its average deviation.

The average gain at an interruption threshold was calculated out of the gains of all test queries at that threshold. While interpreting the average gains, we note that, especially for the expanding window setting, the gain for later updates will be greater than earlier updates of a sequence. As mentioned in Section 3.1, gains of uninterrupted *Anytime Lazy kNN* are precisely the gains that would be achieved by the original Lazy kNN. In Table 3, average gains for uninterrupted runs are in the range of $[28.07\%, 96.35\%]$. While the upper limit of this range can correspond to quite acceptable execution times for some CBs to wait for the exact kNNs, the lower limit may not be tolerable for very large CBs. And in the latter case, we would have to trade time for approximate results and this is when we benefit from the true merit of *ALK*.

In the table, for many configurations, we observe a notable leap between the gain for an uninterrupted run and the corresponding gain upon an interruption at a confidence threshold as high as 0.98. And for some experiment configurations, like the *Phoneme* with w=40 and *step*=10, this difference is tremendous. In this example, although the algorithm reaches a confidence threshold of 0.98 with an average 94.06% gain, it ends up doing many more calculations to ascertain the exact kNNs which ultimately reduces the gain down to 28.07% level. The confidence being quite efficient $(\overline{\eta} = 0.94, \overline{\sigma_\eta} = 0.05)$ for this experiment, this phenomenon occurs due to the fact that, in this particular CB, there are many similar kNN candidates which need to be assessed, but, in the end they cannot win over best-so-far kNNs. Another common observation is that we reach higher gains for *step*=1 compared to *step*=10, because with the former configuration, less change is introduced per update and the kNNs of the new query are more similar to the kNNs of the previous one. Thus, less calculations are needed to obtain the exact kNNs and, in the case of interruption, to reach the desired confidence threshold.

We also see that for $z = -1$ setting, the quality estimation of PDP (i.e. the confidence) was quite efficient with a relatively minor deviation throughout our experiments $(\overline{\eta} \in [0.89,\ 0.98], \overline{\sigma_\eta} \in [0.02,\ 0.09])$. The average efficiency was slightly below 1, which means that the actual quality was a bit above the confidence threshold. This was a desired behavior of efficiency, since we wanted to be precautious by lowering the confidence μ by $-\sigma$ for interruptions. In other words, we preferred to have slightly lower gain in execution time to giving slightly overconfident results.

The *gain* concept that we have been using throughout the article is based on the percentage of the avoided similarity calculations compared to the number of similarity calculations that would have been carried out by brute-force search. This definition let us establish a platform-independent measure for the speed-up in kNN search by *ALK*. To give a more thorough view of this speed-up, we also provide Table 4 that translates the gain to real execution time for a test platform that we used. The table gives the average execution times for kNN search per sequence update conducted by brute-force search and *ALK*. The former evaluates all the cases in the case base while our algorithm assesses only the true kNN candidates. The table shows that even when *ALK* is not interrupted and run to completion to find exact kNNs, it is faster than brute-force search by orders of magnitude. However, the real contribution of our algorithm is for the occasions when

Table 3
Average Gain % upon Interruptions at Confidence Thresholds.

Dataset	Time window		Unint.	Interruption at μ-σ =								Efficiency	
	w	*step*		0.98	0.95	0.92	0.90	0.85	0.80	0.75	0.70	$\overline{\eta}$	$\overline{\sigma_\eta}$
PowerCons	Expanding	1	75.65	97.78	98.63	98.96	99.01	99.17	99.30	99.38	99.45	0.92	0.07
		10	47.67	75.47	84.37	87.62	88.06	89.46	91.10	92.27	93.47	0.95	0.07
	40	1	69.62	97.44	98.49	98.91	98.96	99.16	99.27	99.39	99.46	0.92	0.08
		10	35.12	74.78	81.52	84.14	84.32	85.33	88.91	91.26	96.04	0.97	0.05
SwedishLeaf	Expanding	1	75.96	98.79	99.14	99.25	99.31	99.43	99.52	99.61	99.66	0.92	0.07
		10	47.55	76.04	79.95	81.38	82.10	87.00	91.68	93.24	96.36	0.94	0.07
	40	1	79.11	98.97	99.14	99.25	99.32	99.44	99.54	99.62	99.66	0.92	0.07
		10	38.01	63.69	75.34	84.17	89.44	95.32	96.29	96.34	96.37	0.94	0.07
Strawberry	Expanding	1	81.59	98.74	99.10	99.20	99.26	99.39	99.50	99.55	99.57	0.92	0.06
		10	56.39	74.92	80.33	83.01	83.51	87.07	96.16	96.75	97.07	0.94	0.06
	40	1	89.08	98.95	99.15	99.25	99.30	99.43	99.52	99.55	99.57	0.93	0.06
		10	57.79	79.98	83.81	85.27	85.91	92.04	95.29	98.03	98.34	0.94	0.07
EOGHorizontalSignal	Expanding	1	91.12	99.51	99.60	99.68	99.72	99.73	99.73	99.73	99.73	0.95	0.03
		10	76.32	98.84	98.96	99.02	99.20	99.41	99.50	99.57	99.63	0.94	0.07
	40	1	96.35	99.39	99.47	99.63	99.73	99.73	99.73	99.73	99.73	0.95	0.03
		10	87.04	99.08	99.21	99.24	99.25	99.29	99.33	99.63	99.68	0.91	0.08
InsectWingbeatSound	Expanding	1	81.50	95.22	96.09	96.90	97.18	97.34	97.43	97.48	97.50	0.94	0.05
		10	55.96	58.04	62.29	68.29	71.22	78.89	87.11	91.28	93.59	0.89	0.09
	40	1	84.97	95.40	96.25	97.03	97.25	97.41	97.50	97.56	97.60	0.95	0.05
		10	42.27	44.81	48.76	61.17	70.70	84.19	88.62	90.03	94.11	0.91	0.08
ECG5000	Expanding	1	76.71	92.23	94.56	95.89	96.42	97.09	97.32	97.41	97.46	0.93	0.05
		10	48.09	48.09	48.09	48.09	48.09	65.35	86.86	87.28	95.64	0.94	0.07
	40	1	79.90	93.51	95.34	96.37	96.61	97.07	97.31	97.40	97.45	0.94	0.05
		10	48.79	73.24	81.79	85.49	85.63	86.00	86.37	86.74	89.11	0.92	0.08
UWaveGestureLibraryX	Expanding	1	84.20	98.15	98.66	98.84	98.87	98.91	98.93	98.95	98.95	0.96	0.03
		10	61.93	84.27	87.09	89.42	90.60	93.00	95.62	97.76	98.06	0.91	0.07
	40	1	91.78	98.11	98.75	98.89	98.91	98.95	98.97	98.99	98.99	0.96	0.03
		10	61.62	90.84	93.73	93.90	94.00	94.26	97.69	97.76	98.13	0.93	0.06
Yoga	Expanding	1	86.70	95.22	96.65	97.04	97.14	97.22	97.24	97.26	97.28	0.96	0.03
		10	67.32	79.12	85.34	87.85	88.74	90.99	94.15	95.65	96.38	0.89	0.08
	40	1	92.59	95.42	96.66	96.89	96.93	97.20	97.23	97.26	97.28	0.96	0.03
		10	63.99	70.68	89.03	91.44	91.83	92.84	95.20	95.70	95.79	0.93	0.06
Phoneme	Expanding	1	88.95	96.27	97.54	97.56	97.56	97.56	97.57	97.57	97.57	0.97	0.02
		10	71.60	90.53	93.00	94.48	95.25	96.42	97.06	97.37	97.52	0.92	0.06
	40	1	66.97	96.29	97.54	97.56	97.57	97.57	97.57	97.57	97.57	0.97	0.02
		10	28.07	94.06	94.63	95.13	95.46	96.25	97.27	97.31	97.50	0.94	0.05
Mallat	Expanding	1	89.66	90.29	90.66	90.75	90.78	90.82	90.84	90.87	90.87	0.97	0.02
		10	72.83	73.29	74.24	76.14	77.86	83.29	86.10	86.84	86.96	0.91	0.06
	40	1	93.07	93.48	93.98	94.07	94.13	94.21	94.25	94.28	94.30	0.97	0.02
		10	67.74	69.40	71.43	75.86	77.85	83.05	85.49	86.00	86.97	0.93	0.05
MixedShapesRegularTrain	Expanding	1	90.52	97.92	98.60	98.63	98.65	98.66	98.67	98.67	98.67	0.97	0.02
		10	74.30	96.23	96.94	97.32	97.55	98.06	98.38	98.51	98.59	0.92	0.06
	40	1	95.68	97.95	98.56	98.57	98.62	98.67	98.68	98.68	98.68	0.98	0.02
		10	76.13	95.36	95.70	97.11	97.32	97.62	97.63	98.43	98.56	0.95	0.04

The algorithm is interrupted at the number of calculations when the μ-σ ($z = -1$) value provided by $\mathcal{PDP}$ reaches to the given confidence thresholds. For e.g., a threshold of 0.95 means "stop when μ-σ reaches 0.95". The table summarizes average *gains* in terms of % of avoided similarity assessments during tests compared to a brute-force search. The gains upon interruption can be compared to the gain at the 'Unint.' column which is achieved when the algorithm is run to completion uninterrupted for exact kNNs. We also provide the mean of efficiency η together with the mean of its standard deviation σ_η to reflect the *efficiency* of the quality estimation (i.e. *confidence*) for each experiment.

this speed-up is still not feasible and we have to resort to approximate kNNs. In this case, even when we interrupt *ALK* at a high confidence threshold like 0.98, the speed-up drastically increases. For e.g., for the *EOGHorizontalSignal* experiment with w=40 and *step*=1 setting, *ALK* delivers best-so-far kNNs of the query for a sequence update with 0.98 confidence 163.93 times faster than the brute-force search on average, reducing the execution time from 8.057 s down to 0.049 s. For the same experiment, the speed-up factor increases to 370.37 if we settle with a confidence of 0.85. Table 4 also reveals that the speed-up is more dramatic for the larger case base of the same dataset. The CB generated for a dataset with *step*=1 is 10 times larger than the CB of a *step*=10 as explained in Section 4.2. And this observation underpins the very purpose of the *ALK*, larger the CB is, higher becomes the speed-up. We also note that the average execution time for an experiment is inversely proportionate to the corresponding average gain given in Table 3.

For each application, the definition of acceptable approximate results will be different. For some critical decisions, we may need approximate results with very high confidence whereas for less critical situations we may conform with less confidence. The gain that we will achieve for interrupting the algorithm with higher or lower thresholds will change according to the nature of the CB and used time window configurations. But in any case, if we opt for the approximate results instead of the exact ones, *ALK* will boost the speed-up in kNN search higher than Lazy kNN.

Table 4
Average Execution Times and Speed-up Factors per Update.

Dataset	*step*	Uninterrupted			*ALK* interrupted at μ-σ =							
		Brute	*ALK*		0.98		0.95		0.85		0.70	
		ms	ms	Speed	ms	Speed	ms	Speed	ms	Speed	ms	Speed
PowerCons	1	425.86	129.37	×3.29	10.90	×39.06	6.43	×66.23	3.58	×119.05	2.30	×185.19
	10	24.10	15.64	×1.54	6.08	×3.97	4.45	×5.41	3.54	×6.82	0.95	×25.25
SwedishLeaf	1	1295.09	270.54	×4.79	13.34	×97.09	11.14	×116.28	7.25	×178.57	4.40	×294.12
	10	69.93	43.28	×1.62	25.39	×2.75	17.25	×4.06	3.27	×21.37	2.54	×27.55
Strawberry	1	1437.21	156.94	×9.16	15.09	×95.24	12.22	×117.65	8.19	×175.44	6.18	×232.56
	10	98.52	41.59	×2.37	19.72	×5.0	15.95	×6.18	7.84	×12.56	1.64	×60.24
EOGHorizontalSignal	1	8057.42	294.10	×27.4	49.15	×163.93	42.70	×188.68	21.76	×370.37	21.76	×370.37
	10	710.63	92.10	×7.72	6.54	×108.7	5.61	×126.58	5.05	×140.85	2.27	×312.5
InsectWingbeatSound	1	958.27	144.03	×6.65	44.08	×21.74	35.94	×26.67	24.82	×38.61	23.00	×41.67
	10	54.77	31.62	×1.73	30.22	×1.81	28.06	×1.95	8.66	×6.33	3.23	×16.98
ECG5000	1	1370.28	275.43	×4.98	88.93	×15.41	63.86	×21.46	40.15	×34.13	34.94	×39.22
	10	69.98	35.84	×1.95	18.73	×3.74	12.74	×5.49	9.80	×7.14	7.62	×9.18
UWaveGestureLibraryX	1	6408.30	526.76	×12.17	121.12	×52.91	80.10	×80.0	66.01	×97.09	64.72	×99.01
	10	354.34	136.00	×2.61	32.46	×10.92	22.22	×15.95	8.19	×43.29	6.63	×53.48
Yoga	1	2329.18	172.59	×13.5	106.68	×21.83	77.79	×29.94	64.52	×36.1	63.35	×36.76
	10	138.37	49.83	×2.78	40.57	×3.41	15.18	×9.12	6.64	×20.83	5.83	×23.75
Phoneme	1	4383.57	1447.89	×3.03	162.63	×26.95	107.84	×40.65	106.52	×41.15	106.52	×41.15
	10	435.54	313.28	×1.39	25.87	×16.84	23.39	×18.62	11.89	×36.63	10.89	×40.0
Mallat	1	867.53	60.12	×14.43	56.56	×15.34	52.23	×16.61	49.88	×17.39	49.45	×17.54
	10	62.13	20.04	×3.1	19.01	×3.27	17.75	×3.5	9.02	×6.89	8.10	×7.67
MixedShapesRegularTrain	1	9590.46	414.31	×23.15	196.60	×48.78	138.10	×69.44	126.59	×75.76	126.59	×75.76
	10	1076.32	256.92	×4.19	49.94	×21.55	46.28	×23.26	25.51	×42.19	15.50	×69.44

Average execution times (in milliseconds) of kNN search per sequence update are given for select experiments with time window width w=40 setting. Speed-up factors are given with respect to the execution time of a brute-force search. Note that, average execution time and speed-up factor for an experiment are proportionate to its average gain in Table 3.
Test environment specs: CPU: 12 × Intel(R) Core(TM) i7-8700K @ 3.70 GHz; Memory: 31GiB; OS: Ubuntu 18.04; Python: 3.7.3.

6. Discussion and conclusions

In this work we introduced *Anytime Lazy kNN (ALK)*, an anytime exact and approximate kNN algorithm to boost CBR retrieval in large-scale CBs of temporally related cases, e.g. a CB of electronic health records of patients. Our algorithm is based on Lazy kNN [7] which is an effective exact kNN algorithm in CBR literature for such domains. However, for some applications, the notable speed-up provided by this algorithm may not suffice and the execution time for exact kNN may still be intolerable. Therefore, to fit the CBR system with an approximate retrieval option for time-critical applications, we described our methodology to transform Lazy kNN to *ALK*. Specifically, after presenting our domain of interest and the concepts we used throughout this paper, we detailed how we constructed a probabilistic model of adjustable accuracy to estimate the quality of best-so-far kNNs upon interruption. We referred to the expected quality as the *confidence* of the algorithm for its output in accordance with the CBR literature. Later, we showed how we can implement confidence thresholds to automatize the interruption with options to be precautious, neutral or optimistic about the confidence provided by our probabilistic model. Before experimentation, we explained how we can treat a time series as a sequence of temporally related cases and showed different ways of generating CBs out of a TS dataset by different time window settings. Furthermore, we devised a means to measure the *efficiency* of confidence estimation to be used throughout experiments. Finally, we presented the results of numerous experiments conducted on publicly available TS datasets of diverse domains and characteristics.

The results show us that we can reach superior speed-up in approximate kNN search even when we interrupt the algorithm at very high confidence thresholds which means that best-so-far kNNs are almost as near to the query as the exact kNNs. So, with *ALK*, the expert can opt to wait for the completion of the algorithm to obtain exact kNNs or, he or she can interrupt the search manually/automatically any time when a prompter response is needed and get best-so-far kNNs together with a confidence value for each NN to reason with.

The *Performance Distribution Profile* used by our algorithm as the basis of its probabilistic model discretizes the calculation range linearly. A way to increase both the accuracy and efficiency of confidence can be to discretize this range logarithmically, attributing more importance to the beginning of the calculation range given the fact that NNs are actually found in the early stages of *ALK* execution for the domains of our interest.

We mainly focused on its use in a CBR context, but *ALK* can be applied to any kNN search domain that exhibits temporal relation between examples in a metric search space. For example, we tested *ALK* with univariate TS datasets, but in a like manner, given an adequate metric distance for a multivariate TS (see [47] for examples), our algorithm would work just the same. One way to build a CB from a multivariate TS dataset can be to treat each instance as a sequence and each time-dependent variable of the instance as a multi-valued case feature. The number of feature values would be determined by the applied time window.

On the other hand, we only used the problem space for retrieval. However, a case also bears a solution part and as future work we want to enhance retrieval by incorporating the solution space as well. Especially, low confidence zones in a CB's solution space (e.g. categorized as dubiosity patterns in [48]) can help *ALK* further tune its retrieval performance. For example, if the solutions of best-so-far kNNs exhibit a "border" pattern where two competing solutions (say classes) exist; then the algorithm may use this information to encourage resuming the search expecting a possible change in kNNs favoring one of the competing solutions.

The complete code of *ALK* algorithm including all its functionality covered throughout this article and simple instructions to reproduce the experiments are publicly available at the online repository: https://github.com/IIIA-ML/alk.

CRediT authorship contribution statement

Mehmet Oğuz Mülâyim: Development and design of the methodology, Implementation of the proposed algorithm, writing and revising the manuscript. **Josep Lluís Arcos:** Development and design of the methodology, Implementation of the proposed algorithm, writing and revising the manuscript.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We thank to all contributors and maintainers of the UEA & UCR Time Series Classification Repository. This work has been funded by the project Playing and Singing for the Recovering Brain: Efficacy of Enriched Social-Motivational Musical Interventions in Stroke Rehabilitation (Play&Sing), Spain, 201729.31, Fundació La Marató de TV3, Spain; and, by the project Innobrain, Spain, COMRDI-151-0017 (RIS3CAT comunitats), and Feder, Spain funds. Mehmet Oğuz Mülâyim is a Ph.D. Student of the doctoral program in Computer Science at the Universitat Autònoma de Barcelona.

References

[1] D. Reinsel, J. Gantz, J. Rydning, The digitization of the world - from edge to core, IDC White Paper (November) (2018) https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf. (Accessed 20 January 2020).

[2] D.B. Leake, CBR in Context: The Present and Future, in: D.B. Leake (Ed.), Case-Based Reasoning: Experiences, Lessons & Future Directions, The AAAI Press/The MIT Press, Menlo Park, 1996, pp. 3–30, chapter 1.

[3] A. Agnar, E. Plaza, Case-based reasoning: Foundational issues, methodological variations, and system approaches, AI Commun. 7 (1) (1994) 39–59, http://dx.doi.org/10.3233/AIC-1994-7104.

[4] J. Kolodner, Case-Based Reasoning, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[5] A.G. Francis, A. Ram, The utility problem in case-based reasoning, in: AAAI Case-Based Reasoning Workshop, AAAI Press, Washington DC, 1993, p. 160.

[6] B. Smyth, P. Cunningham, The utility problem analysed: A case-based reasoning perspective, in: European Workshop on Advances in Case-Based Reasoning, Springer, 1996, pp. 392–399, http://dx.doi.org/10.1007/BFb0020625.

[7] M.O. Mülâyim, J.L. Arcos, Perks of being lazy: Boosting retrieval performance, in: International Conference on Case-Based Reasoning, in: LNAI, 11156, Springer Verlag, 2018, pp. 309–322, http://dx.doi.org/10.1007/978-3-030-01081-2_21.

[8] V. Jalali, D.B. Leake, Harnessing hundreds of millions of cases: Case-based prediction at industrial scale, in: International Conference on Case-Based Reasoning, in: Lecture Notes in Computer Science, Springer Verlag, 2018, pp. 153–169, http://dx.doi.org/10.1007/978-3-030-01081-2_11.

[9] J. Woodbridge, B. Mortazavi, A.A. Bui, M. Sarrafzadeh, Improving biomedical signal search results in big data case-based reasoning environments, Pervasive Mob. Comput. 28 (2016) 69–80, http://dx.doi.org/10.1016/j.pmcj.2015.09.006.

[10] B. Smyth, M.T. Keane, Remembering to forget: A competence-preserving case deletion policy for case-based reasoning systems, in: International Joint Conference on Artificial Intelligence, vol. 1, Citeseer, 1995, pp. 377–382.

[11] D.B. Leake, B. Smyth, D.C. Wilson, Q. Yang, Introduction to the special issue on maintaining case-based reasoning systems, Comput. Intell. 17 (2) (2001) 193—-195, http://dx.doi.org/10.1111/0824-7935.00139.

[12] J.M. Juarez, S. Craw, J.R. Lopez-Delgado, M. Campos, Maintenance of case bases: Current algorithms after fifty years, in: IJCAI International Joint Conference on Artificial Intelligence, 2018, pp. 5457–5463, http://dx.doi.org/10.24963/ijcai.2018/770.

[13] D.C. Wilson, D.B. Leake, Maintaining case-based reasoners: Dimensions and directions, Comput. Intell. 17 (2) (2001) 196–213, http://dx.doi.org/10.1111/0824-7935.00140.

[14] A.S. Arefin, C. Riveros, R. Berretta, P. Moscato, GPU-FS-kNN: A software tool for fast and scalable kNN computation using GPUs, PLoS ONE 7 (8) (2012) http://dx.doi.org/10.1371/journal.pone.0044000.

[15] V. Garcia, E. Debreuve, M. Barlaud, Fast k nearest neighbor search using GPU, in: IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops, (2) IEEE, 2008, pp. 1–6, http://dx.doi.org/10.1109/CVPRW.2008.4563100.

[16] J. Kolodner, Retrieving events from a case memory: A parallel implementation, in: Proc. of 1988 Case-Based Reasoning, 1988, pp. 233–249.

[17] S. Wess, K.-D. Althoff, G. Derwand, Using k-d trees to improve the retrieval step in case-based reasoning, in: European Workshop on Case-Based Reasoning, Springer, 1993, pp. 167–181, http://dx.doi.org/10.1007/3-540-58330-0_85.

[18] P.N. Yianilos, Data structures and algorithms for nearest neighbor search in general metric spaces, in: SODA, ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1993, pp. 311–321.

[19] A.M. Kibriya, E. Frank, An empirical comparison of exact nearest neighbour algorithms, in: Knowledge Discovery in Databases: PKDD 2007, Springer Berlin Heidelberg, 2007, pp. 140–151, http://dx.doi.org/10.1007/978-3-540-74976-9_16.

[20] R. Bellman, Dynamic Programming, first ed., Princeton University Press, Princeton, NJ, USA, 1957.

[21] C.M. Eastman, S.F. Weiss, Tree structures for high dimensionality nearest neighbor searching, Inf. Syst. 7 (2) (1982) 115–122, http://dx.doi.org/10.1016/0306-4379(82)90023-0.

[22] J.L. Bentley, Multidimensional binary search trees used for associative searching, Commun. ACM 18 (9) (1975) 509–517, http://dx.doi.org/10.1145/361002.361007.

[23] M. Muja, D.G. Lowe, Scalable nearest neighbor algorithms for high dimensional data, IEEE Trans. Pattern Anal. Mach. Intell. 36 (11) (2014) 2227–2240, http://dx.doi.org/10.1109/TPAMI.2014.2321376.

[24] A. Andoni, P. Indyk, Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions, in: IEEE Symposium on Foundations of Computer Science, IEEE, 2006, pp. 459–468, http://dx.doi.org/10.1109/FOCS.2006.49.

[25] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, H. Zhang, Fast approximate nearest-neighbor search with K-nearest neighbor graph, in: International Joint Conference on Artificial Intelligence, AAAI Press, 2011, pp. 1312–1317, http://dx.doi.org/10.5591/978-1-57735-516-8/IJCAI11-222.

[26] V. Jalali, D.B. Leake, Scaling up ensemble of adaptations for classification by approximate nearest neighbor retrieval, in: International Conference on Case-Based Reasoning, in: Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2017, pp. 154–169, http://dx.doi.org/10.1007/978-3-319-61030-6_11.

[27] T. Dean, M. Boddy, An analysis of time-dependent planning, in: Seventh AAAI National Conference on Artificial Intelligence, vol. 88, AAAI, 1988, pp. 49–54.

[28] S. Zilberstein, Using anytime algorithms in intelligent systems, AI Mag. 17 (3) (1996) 73–83, http://dx.doi.org/10.1609/aimag.v17i3.1232.

[29] P. Kranen, T. Seidl, Harnessing the strengths of anytime algorithms for constant data streams, Data Min. Knowl. Discov. 19 (2) (2009) 245–260, http://dx.doi.org/10.1007/s10618-009-0139-0.

[30] K. Ueno, X. Xi, E. Keogh, D.-J. Lee, Anytime classification using the nearest neighbor algorithm with applications to stream mining, in: International Conference on Data Mining, ICDM, IEEE, 2006, pp. 623–632, http://dx.doi.org/10.1109/ICDM.2006.21.

[31] W. Xu, D. Miranker, R. Mao, S. Ramakrishnan, Anytime K-nearest neighbor search for database applications, in: International Conference on Data Engineering Workshop, IEEE, 2008, pp. 426–435, http://dx.doi.org/10.1109/ICDEW.2008.4498354.

[32] J.W. Schaaf, "Fish and Sink" An anytime-algorithm to retrieve adequate cases, in: International Conference on Case-Based Reasoning, Springer, 1995, pp. 538–547, http://dx.doi.org/10.1007/3-540-60598-3_50.

[33] F. Ricci, P. Avesani, Learning a local similarity metric for case-based reasoning, in: International Conference on Case-Based Reasoning, vol. 1010, Springer Berlin Heidelberg, 1995, pp. 301–312, http://dx.doi.org/10.1007/3-540-60598-3_27.

[34] S. Montani, L. Portinale, Accounting for the temporal dimension in case-based retrieval: A framework for medical applications, Comput. Intell. 22 (3–4) (2006) 208–223, http://dx.doi.org/10.1111/j.1467-8640.2006.00284.x.

[35] J. Grass, S. Zilberstein, Anytime algorithm development tools, ACM SIGART Bull. 7 (2) (1996) 20–27, http://dx.doi.org/10.1145/242587.242592.

[36] S. Zilberstein, Operational Rationality Through Compilation of Anytime Algorithms (PhD thesis), University of California at Berkeley, 1993.

[37] M. Boddy, T. Dean, Solving time-dependent planning problems, in: Eleventh International Joint Conference on Artificial Intelligence, 1989, pp. 979–984.

[38] J. Grass, Reasoning about computational resource allocation, XRDS: Crossroads, ACM Mag. Stud. 3 (1) (1996) 16–20, http://dx.doi.org/10.1145/332148.332154.

[39] W. Cheetham, Case-based reasoning with confidence, in: European Workshop on Advances in Case-Based Reasoning, (No. 1898) Springer, 2000, pp. 15–25, http://dx.doi.org/10.1007/3-540-44527-7_3.
[40] A.J. Broder, Strategies for efficient incremental nearest neighbor search, Pattern Recognit. 23 (1–2) (1990) 171–178, http://dx.doi.org/10.1016/0031-3203(90)90057-R.
[41] P. Cunningham, B. Smyth, A. Bonzano, An incremental retrieval mechanism for case-based electronic fault diagnosis, Knowl.-Based Syst. 11 (3–4) (1998) 239–248, http://dx.doi.org/10.1016/S0950-7051(97)00049-X.
[42] I. Jurisica, J. Glasgow, J. Mylopoulos, Incremental iterative retrieval and browsing for efficient conversational CBR systems, Appl. Intell. 12 (3) (2000) 251–268, http://dx.doi.org/10.1023/A:1008375309626.
[43] A. Bagnall, J. Lines, W. Vickers, E. Keogh, The UEA & UCR time series classification repository, 2018, http://www.timeseriesclassification.com. (Accessed 20 January 2020).
[44] J. Serrà, J.L. Arcos, An empirical evaluation of similarity measures for time series classification, Knowl.-Based Syst. 67 (2014) 305–314, http://dx.doi.org/10.1016/j.knosys.2014.04.035.
[45] X. Wang, A. Mueen, H. Ding, G. Trajcevski, P. Scheuermann, E. Keogh, Experimental comparison of representation methods and distance measures for time series data, Data Min. Knowl. Discov. 26 (2) (2013) 275–309, http://dx.doi.org/10.1007/s10618-012-0250-5.
[46] A. Mueen, E. Keogh, Q. Zhu, S. Cash, B. Westover, Exact discovery of time series motifs, in: SIAM International Conference on Data Mining, Society for Industrial and Applied Mathematics, 2009, pp. 473–484, http://dx.doi.org/10.1137/1.9781611972795.41.
[47] D. Kotsakos, G. Trajcevski, D. Gunopulos, C.C. Aggarwal, Time-series data clustering, in: C.C.. Aggarwal, C.K. Reddy (Eds.), Data Clustering, Chapman and Hall/CRC, 2013, pp. 357–380, http://dx.doi.org/10.1201/9781315373515-15, chapter 15.
[48] M.O. Mülâyim, J.L. Arcos, Understanding dubious future problems, in: European Conference on Case-Based Reasoning, in: LNAI, 5239, Springer, Berlin, Heidelberg, 2008, pp. 385–399, http://dx.doi.org/10.1007/978-3-540-85502-6_26.