UAB

Universitat Autònoma de Barcelona

PhD in Computer Science

Research line: Artificial Intelligence

Anytime Case-Based Reasoning in Large-Scale Temporal Case Bases

PhD Student: Mehmet Oğuz Mülâyim
PhD Advisor: Josep Lluís Arcos Rosell
PhD Tutor: Josep Lluís Arcos Rosell
PhD Admission Date: 15/11/2017
Contact mail: oguz@iiia.csic.es

Bellaterra (Cerdanyola del Vallès), September 24, 2020

Abstract

Case-Based Reasoning (CBR) methodology's approach to problem-solving that "similar problems have similar solutions" has proved quite favorable for many industrial artificial intelligence applications. However, CBR's very advantages hinder its performance as case bases (CBs) grow larger than moderate sizes. Searching similar cases is expensive. This handicap often makes CBR less appealing for today's ubiquitous data environments while, actually, there is ever more reason to benefit from this effective methodology. Accordingly, CBR community's traditional approach of controlling CB growth to maintain performance is shifting towards finding new ways to deal with abundant data.

As a contribution to these efforts, this thesis aims to speed up CBR by leveraging both problem and solution spaces in large-scale CBs that are composed of temporally related cases, as in the example of electronic health records. For the occasions when the speed-up we achieve for exact results may still not be feasible, we endow the CBR system with anytime algorithm capabilities to provide approximate results with confidence upon interruption. Exploiting the temporality of cases allows us to reach superior gains in execution time for CBs of millions of cases. Experiments with publicly available real-world datasets encourage the continued use of CBR in domains where it historically excels like healthcare; and this time, not suffering from, but enjoying big data.

Contents

List of Definitions

List of Figures

List of Tables

List of Algorithms

Acknowledgements

Thank you Josep Lluís, for counting me in all those years after we said one day we would resume from where we left. And thank you for your innate optimism and insightful tutorship during my unconventional PhD journey. I consider myself privileged to have worked with and learned from you. You are definitely one of my heroes!

Gracias Ana (Beltran) por tu apoyo más amable con todo el papeleo tedioso que necesitaba hacer antes de y durante mi estancia. Ya sé que hubo muchos pasos de los cuales ni me enteré entonces gracias a ti.

Thank you Jordi (Sabater) for helping me out at the very right moment with my accommodation. I could not wish for more peace. I am truly indebted for your trust.

Thank you Josep (Puyol). Your kind guidance when I first contacted IIIA years back paved all my way till here. And it has always been a pleasure to make music next to you, Jordi, Eva, Marco and Angel at our concerts.

Thanks once more to my master's degree teachers Carles, Enric, Eva, Francesc, Pere, Pedro, Josep, Marco and JAR; I appreciate all the wisdom you transmitted.

Thanks to my fellow PhD friends Toni, David, Ramon, Adán, Borja, Ewa, Núria, Dimitra, Athina, Marc, Thiago and Albert for the countless laughters and thought provoking conversations that we shared.

IIIA is a great place to be and, like everyone else who has involved in, I feel in a big family surrounded by bright people. Thanks to our directors during my stay Francesc, Ramon and Carles, and to the management, Montse's, Ana and Eva, and to all IIIA members for creating this cosy and inspiring home.

This work has been funded in part by the project *Play&Sing* - Playing and Singing for the Recovering Brain: Efficacy of Enriched Social-Motivational Musical Interventions in Stroke Rehabilitation, 201729.31, Fundació La Marató de TV3; and, by the project *Innobrain* - New Technologies for the Innovation in Cognitive Stimulation and Rehabilitation, COMRDI-15-1-0017 (RIS3CAT comunitats); and the project *SGR*, 2017-SGR-172 from Generalitat de Catalunya; and FEDER funds.

I dedicate this work to my beloved family...

Chapter 1

Introduction

Inspired by Schank's dynamic memory theory for reminding and learning in cognitive science (Schank, 1982), Case-Based Reasoning's (CBR) approach to problem-solving is very reminiscent to everyday human reasoning. CBR suggests a solution to a present problem by remembering and reusing past similar problem-solving experiences that it keeps in its case base (CB). Subsequently, CBR learns from each new experience, both from success and failure in the suggested solution, by saving the new experience in its CB for future use.

CBR has been one of the most successful approaches to build intelligent systems that are able to learn from experience (e.g. Goel and Díaz-Agudo, 2017). Especially, it had a significant impact on domains where reasoning is inherently based on experience. For example, a doctor tries to diagnose a patient based on his/her past experience with patients showing similar symptoms. A help-desk operator tries to solve an issue based on the solution to the most similar past problem, if available. A music recommender is expected to recommend a song that is popular among listeners of similar preferences and listening history. An attorney justifies his/her arguments using similar past cases as precedents, etc.

Therefore, CBR's reasoning is memory-based. It typically starts with an initial CB that is expected to be enough representative of future problems in the domain. Consequently, every problem-solving experience that is worth to remember (usually only those which cannot be readily inferred from existing cases) is saved as a new case. As the CB grows, its coverage of the problem and solution spaces of the domain also expands and the CBR system becomes likely to suggest more accurate solutions to future problems. Nevertheless, a growing CB also causes the slow-down of the search for similar past problems because now there are more cases to evaluate and similarity assessments are computationally expensive. And eventually, the size of the CB reaches a point that the time spent on this search makes a CBR system practically unusable. For example, *k-Nearest Neighbors* (kNN) (Cover and Hart, 1967) is a widely used algorithm[1] in CBR for the retrieval of similar past problems. However, its linear runtime complexity slows down the retrieval considerably as the CB grows. The phenomenon of the decrease in overall CBR system performance due to CB growth is known as the *utility problem* (e.g. Francis and Ram, 1993) and has been a major focus in CBR research until today. The main approach to tackle this problem has been controlling the CB growth either by deletion of cases (e.g. Smyth and Keane, 1995) or avoidance of their retention into CB (e.g. Muñoz-Avila, 1999). Both approaches trade performance for solution

[1]We will be using the *kNN* acronym both for the algorithm itself and the k nearest neighbors of a query point.

accuracy, however they inevitably imply a loss of information in the CB as a side effect too. You might never know if a case that is considered redundant and left out of the CB today would not prove critical in the future. For example, a feature of cases can later be deemed (or learned to be) more crucial in similarity comparisons, and this would have ranked a left-out case more important.

On the other hand, a case base "doesn't have to be very large to be very useful" (Kolodner, 1996, p. 358). If solutions to future problems can successfully be suggested by adapting the solutions to existing cases, the CB can be kept minimum in size. However, where accurate answers could be vital like healthcare domain, relying only on adaptation and representativeness of a small case base may not suffice. You may want to have more past evidence backing your present diagnosis. Also, choosing representative cases can be a problem on its own. We have to make sure that the selected cases are and will continue to be representative as the CB evolves.

Besides, increasingly abundant digital data makes controlling CB growth less practical. While data is flowing into intelligent systems as never before, it may not even be possible to decide on time which data goes and which stays. Naturally, CBR community is adapting its approach and exploring ways on how to benefit from the abundance of data instead of suppressing its growth. For example, a recent research is leveraging big-data tools to find *approximately* similar past problems to a present problem among millions of cases (Jalali and Leake, 2015).

When it is inevitable to resort to approximate results for the sake of performance, *anytime algorithms* (T. Dean and Boddy, 1988) offer a very appealing choice for algorithm design. An anytime algorithm provides the best-so-far result when interrupted. The algorithm quickly finds a good-enough approximate result and, if allocated more execution time, improves on it ultimately yielding the exact result. Riesbeck (1996, pp. 384-5) argues that "CBR is a select and adapt algorithm" and "the selection process should be an 'anytime' algorithm" to serve as an "intelligent component" of a system. He adds that "...true CBR systems, typically have an 'anytime' capability". He attributes this capability to finding "some answer almost immediately" and then improve the answer as more similar cases are reminded by retrieval if allocated more time. However, there is more to an anytime algorithm than just to improve its solution over execution time. It is desired to attach a quality value to its approximate results. Also, it is desired that the improvement in the quality is more pronounced in the early stages of computation compared to the final stages. For example, if an acceptable quality can only be reached towards the end of execution, it could make sense to wait just a bit more to have the exact solution. Hence, although anytime algorithms are very tempting to implement, not all algorithms can be easily transformed into their 'anytime' versions to comply with desired properties (for these properties see Zilberstein, 1996, p.74).

Against the challenges that CBR is facing with increasingly available large-scale case bases, we saw a promising opportunity to tackle this challenge for a particular type of domain where the case bases are comprised of *temporally related cases* as in the examples of electronic health records and meteorological data. In such a *temporal case base* the search for similar past cases has to take into account the evolution of cases instead of treating each case individually. The temporal dimension regarding the evolution of cases has long started to gain the attention of CBR researchers (e.g. Montani and Portinale, 2006).

Under the light of above-mentioned challenges and opportunities, in the following section we explicitly state our motivations for this thesis. In section 1.2, we headline our contributions to CBR research. Section 1.3 outlines the organization of the dissertation.

1.1 Motivations

The motivation of our research presented in this dissertation stems from the answers to the implied questions in the following subsections' titles:

1.1.1 Why CBR still matters

The advantages that CBR historically brought to developing artificial intelligence (AI) applications are still worthy today. These advantages (Leake, 1996b, pp. 3-4) (Kolodner and Leake, 1996, p. 33) can be summarized as below:

- Domains like healthcare, fault diagnosis, help desk systems where each problem solving is committed to corporate memory as an experience are very "natural domains" for CBR (Leake, 1996b, pp. 4) ;
- CBR allows a *quicker start* to building applications in domains that are not easy/possible to formalize completely. Instead of extracting rules from training data or hand-crafting them, CBR can start with a smaller initial CB of past experiences. The initial CB grows incrementally as CBR attains new cases after every new problem solving experience;
- In poorly understood domains, generalization of rules may be imperfect or impossible. CBR's reasoning based on past episodes makes the *rule generalization unnecessary*;
- Some *eager learning* methods like artificial neural networks typically produce black boxes and thus, it is harder or impossible to give explanations to their reasoning for the results. CBR's results are *easier to explain* since they are based on similar past evidence and the chosen adaptation method is explicitly known. Especially in domains where decisions may be life-critical or where there is a high cost of failure, experts may need a convincing justification to apply the suggested solution;
- CBR's incremental learning with new cases improves the solution accuracy without having to retrain a model with the updated data that is needed by eager methods.

Noted as a CBR research challenge by Goel and Díaz-Agudo (2017), CBR can bring a "cognitive approach to big data". As mentioned above, there are domains which demand explanation for the solutions/predictions of an AI application. Unless an eager learning method produces a black-box model of a domain that always gives correct results and the experts trust it without questioning, justification of the results of an AI application will always support decision making. And, CBR can offer explainability as one of its innate strengths to these domains.

1.1.2 Why 'temporal case bases' matter

Health sciences have been one of the major application domains for AI in general (Jiang et al., 2017), and for CBR in particular (Bichindaritz and Marling, 2010; Begum et al., 2011). The knowledge acquisition, reasoning and learning practices intrinsic to this domain make it a perfect match with CBR. The characteristics of healthcare case bases have been one of the driving inspirations for our research. In particular, a medical CB bears a temporal dimension that has to be dealt with specifically. The cases in this CB are not only snapshots of information in a point of time, but they are *temporally related* as in the health record of a patient where each of his/her sessions is a case. And consequently, a search for similar patients has to take into account the medical histories in the form of temporally related cases instead of assessing individual cases. Although time dimension has been a part of many CBs, the evolution of temporally related cases are addressed by CBR research rather recently (e.g. Sánchez-Marré et al., 2005; Montani and Portinale, 2006).

On the other hand, time series data is becoming increasingly available in healthcare as biomedical signals. Time series is a sequence of data observations for a particular phenomenon. Furthermore, time series data generation is accelerating everyday due to the increase in integrated and wearable devices in many other fields as well. Time series analysis has grown to be an investigation field on its own and there is ever-growing research for their analysis and interpretation. Time series data has long been integrated into case bases (e.g. Nakhaeizadeh, 1994; Funk and Xiong, 2006). And, similarity assessment within a CB of biomedical signals inevitably has to consider the evolution of the phenomenon represented by the cases (e.g. Montani and Portinale, 2006).

1.1.3 Why 'anytime CBR' matters

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 (Reinsel, Gantz, and Rydning, 2018). Being a member of the instance-based learning (Aha, Kibler, and Albert, 1991) subdivision of the larger ML family, many CBR systems are not exempt from this laborious opportunity either. However, if CBR is to benefit from ubiquitous digital data, instead of controlling the CB growth, there will have to be found new ways to tackle case bases of unprecedented scales with millions of cases. From our perspective, the above-mentioned "anytime capability" of CBR systems becomes more crucial for their efficiency and usability with such large-scale case bases. In particular, since reasoning in CBR starts with retrieving similar past cases to the present problem, an anytime CBR retrieval will have to yield 'good-enough' approximate neighbors of a target query in a short time and improve on them given more execution time, eventually finding the exact neighbors if allowed to run to termination.

1.1.4 Why 'anytime kNN' matters

k-Nearest Neighbors algorithm seamlessly fits CBR's approach to problem solving. It is a non-parametric learning method, i.e. there are no parameters of a model to learn from training data. All training data is kept in memory and generalized for every unseen problem. Hence, kNN is an instance-based learning method. kNN is commonly used for classification and regression tasks in many fields (see Chávez et al., 2001). Given a distance (or similarity) measure and k value, the output for a target query is decided/calculated by finding and using the labels of its k nearest neighbors. So in a way, just like CBR, kNN also assumes that the solution for a query is similar to the solutions of its neighbors.

Using kNN in CBR retrieval brings several advantages of *lazy learning* (Aha, 1997). Primarily, all available data is used as a model itself. Contrary to eager learning models, kNN does not need any re-training of a previously built model for updated and/or continuously growing data. Second, the approximation of a suggested solution is carried out locally, and this gives CBR the ability to deal with specific cases (e.g. cases that form separate small groups in the problem space) more accurately. Besides, since case bases incrementally grow after each problem solving experience and they are not just randomly collected data, the susceptibility of kNN to *noise* in data is less likely to occur in CBR.

Given a dataset of examples (or CB), the naïve approach to find the kNN of a query is to exhaustively calculate the distances between the query and the instances in the dataset and return the nearest k instances. Unless the algorithm is massively parallelized, the *linear time complexity* of this method prohibits its usage with very large data.

To speed up the kNN, many techniques (some of which we will be briefly reviewing in section 2.1) opt for *approximate* neighbors. Besides, for some applications we may do just fine with similar enough neighbors and there may not be a critical need for the exact ones. When approximation is a choice on the table, a nice approach is to use anytime algorithms as they offer both approximate and exact result options. When interrupted, they yield approximate results of improving quality over execution time, and if they are allowed to finish, they return exact results. Therefore, a 'well behaving' Anytime kNN search can be the foundation to an Anytime CBR. Nevertheless, as we will discuss in Chapter 4, implementing an anytime kNN algorithm is not trivial. However, leveraging the temporality of cases in a *temporal case base* gives us the means to achieve it.

1.2 Contributions

Pursuing our motivations reflected above, we present our following contributions to the CBR research:

1.2.1 Speed-up in exact kNN search

We developed an algorithm, *Lazy kNN* (Mülâyim and Arcos, 2018), that achieves significant speed-up in kNN search in *temporal case bases*. The gain in speed is thanks to the evaluation of only the true *kNN candidates* of a query in a CB. The candidacy assessment leverages the triangle inequality property of *metric* spaces and the temporal relation between the cases. The algorithm is tested with moderate-to-large sized case bases generated out of publicly available time series datasets.

1.2.2 Further speed-up by anytime kNN search

To deal with the occasions where the speed-up in exact search provided by *Lazy kNN* may still be insufficient for time-critical decisions, we extended our algorithm to *Anytime Lazy kNN* (*ALK*) by endowing it with desired anytime algorithm capabilities (Mülâyim and Arcos, 2020). *Anytime Lazy kNN* finds exact kNN when allowed to run to completion with remarkable gain in execution time by avoiding unnecessary neighbor assessments. And, for applications where the gain in exact kNN search may not suffice, it can be interrupted earlier and it returns best-so-far kNN together with a *confidence* value attached to each neighbor. The confidence for an approximate neighbor is based on a probabilistic model. *ALK* can also be interrupted *automatically* after reaching a desired *confidence threshold*. Furthermore, we devised a means to measure the *efficiency* of confidence estimation. *Anytime Lazy kNN* is evaluated on the same datasets as *Lazy kNN* to indicate the further speed-up achieved by the interruption of the algorithm at given confidence thresholds.

1.2.3 Accounting for the solution space

The ultimate goal of most CBR systems is to suggest solutions. Both *Lazy kNN* and *Anytime Lazy kNN* attain the speed-up by leveraging the properties of the metric *problem space*. Accordingly, to account for the *solution space* as well, we extended *ALK* for its use in *classification*, a common task for CBR systems in our target domains. *Anytime Lazy kNN Classifier* can suggest a solution for both approximate and exact kNN of a target problem. *ALK Classifier* also gives an option to be interrupted automatically upon guaranteeing the exact solution. Thus, when used for exact classification, it exceeds the speed-up of its predecessor *Lazy kNN*.

A desired property of CBR systems is to attach a confidence for their solutions. However, to the best of our knowledge, all existing *solution confidence* measures in CBR literature use exact neighbors of a query. To provide a solution confidence when *ALK Classifier* suggests a solution with best-so-far kNN as well, we formally transformed some of the confidence measures in literature to be used in approximate classification.

1.2.4 More than algorithms, a methodology

We see our work more than the introduction of efficient algorithms. We believe that the candidacy assessment, proposed data structures to track kNN search histories and the methodology that we used in the construction of the anytime kNN algorithm and confidence estimations can be applied to a multitude of application domains that operate with temporal case bases in metric spaces. It will be seen that these mentioned mechanisms are parametric and/or extendable in our algorithms. For example, our algorithms can work with any other case representation and its accompanying similarity metric. Besides, even the search for kNN candidates can be carried out in a different way that is deemed to fit better for a particular application domain. We give examples to candidate search alternatives.

1.2.5 Open source code

All algorithms given as pseudocodes in this dissertation are publicly available as open source code at the online repository https://github.com/IIIA-ML/alk. Moreover, at the same address, simple instructions are provided to easily repeat the experiments and reproduce the results that we shared in the thesis.

1.3 Dissertation organization

In the following chapter, we give further background information and a brief overview of the related work for case-based reasoning, temporal case bases and anytime algorithms. We also give explicit definitions of the concepts in our domain of interest that are used throughout the dissertation.

Chapter 3 introduces *Lazy kNN* as our first contribution and details the theory and the mechanism behind its speed-up in exact kNN search that will form the foundation for the following contributions. Then, we describe how we generate the CBs for experiments that will be used to test all algorithms. Finally, we share the experiment results for *Lazy kNN*.

Chapter 4 presents *Anytime Lazy kNN* (*ALK*) as the second contribution of this thesis. We describe the steps to transform *Lazy kNN* to *ALK*. We detail the probabilistic model of adjustable accuracy used in the estimation of the confidence for best-so-far kNN. Then, we show how we can automatize interruption by reaching a given confidence threshold. Moreover, we introduce a means to measure the efficiency of confidence estimations. We share the experiment results which empirically demonstrate that we can reach superior speed-up even when the algorithm is interrupted at very high confidence thresholds. And we shed more light to the insights of this superior gain. We conclude this chapter by providing alternative ways to search kNN candidates.

Chapter 5 explores the solution space and presents *ALK Classifier*, an extension to *ALK* for classification tasks. We describe how and when *ALK Classifier* can guarantee the exact solution with best-so-far kNN. The chapter also introduces the extensions to a set of solution confidence measures in literature to be used with approximate kNN instead of exact ones.

In Chapter 6, we discuss the outcomes of this dissertation, provide recommendations for the application of proposed algorithms, and suggest future work to further enhance our contributions.

Chapter 2

Background

This chapter presents background information within the scope of this dissertation. In Section 2.1 we briefly summarize the *Case-Based Reasoning* paradigm and highlight the challenges for large-scale case bases of past and today. In Section 2.2, we introduce our domain of interest, namely *temporal case bases*, and related terminology that we will be using throughout the thesis. Section 2.3 gives a brief overview of *anytime algorithms* that forms the foundation of our contributions.

2.1 Case-Based Reasoning

With its roots in the cognitive theory of the role of memory for understanding (Schank, 1982), CBR methodology, just like us, assumes that "*similar problems have similar solutions*" (e.g. Leake, 1996b, p.1). Thus, CBR solves a present problem by *reusing* or adapting the solutions to similar previous problems *retrieved* from its *case base*. A *case* typically consists of three parts: the *problem* representing the query, the *solution* suggested by the CBR system and the *outcome* (feedback) after applying the solution (Kolodner and Leake, 1996, p.39). Most simplistic—yet very common—problem description of a case is a vector of *surface features* where each feature is represented by an *attribute-value* pair. Depending on the problem being solved, the solution can be a label for classification, or a real value for regression, an action schema for planning, etc. Figure 2.1 depicts the retrieval of similar problems to the target query from the *problem space* and the suggestion of a solution by adaptation in the *solution space*.

Each suggested solution may further be *revised* by the feedback from an expert or from the outcome of its application to the real world. Finally, the problem together with its accepted solution are *retained* as a new case in the CB for future use. By the retention of a solved case, CBR system essentially *learns* a new problem solving experience and this completes the typical CBR cycle given in Figure 2.2. For the foundations of these steps see (Agnar and Plaza, 1994), for a thorough review of them and, particularly, for alternative approaches to case representation and retrieval see (López De Mántaras et al., 2005).

CBR's method of learning is known as *lazy learning* (Aha, 1997) in Machine Learning literature since a CBR system does not build a model prior to a query—as opposed to *eager learning* methods (e.g. artificial neural networks) which do so, and generalizes its cases every time a query is posed to the system. This behaviour is an advantage of CBR for continuously changing large CBs since it discards the need to re-train learned models with the updated data.

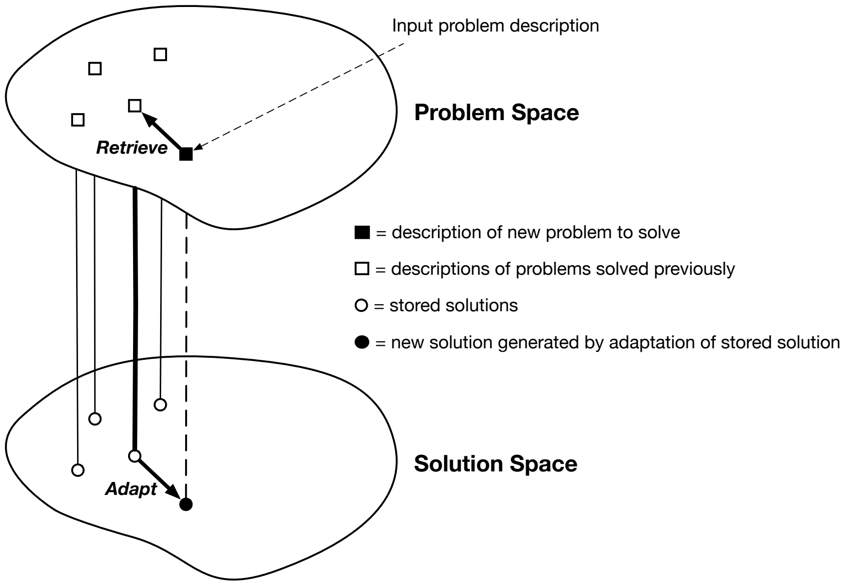

Input problem description
Problem Space
Retrieve
■ = description of new problem to solve
□ = descriptions of problems solved previously
○ = stored solutions
● = new solution generated by adaptation of stored solution
Solution Space
Adapt

Figure 2.1: **Problem & solution spaces in CBR** and the generation of a new solution. Adapted from (Leake, 1996b).

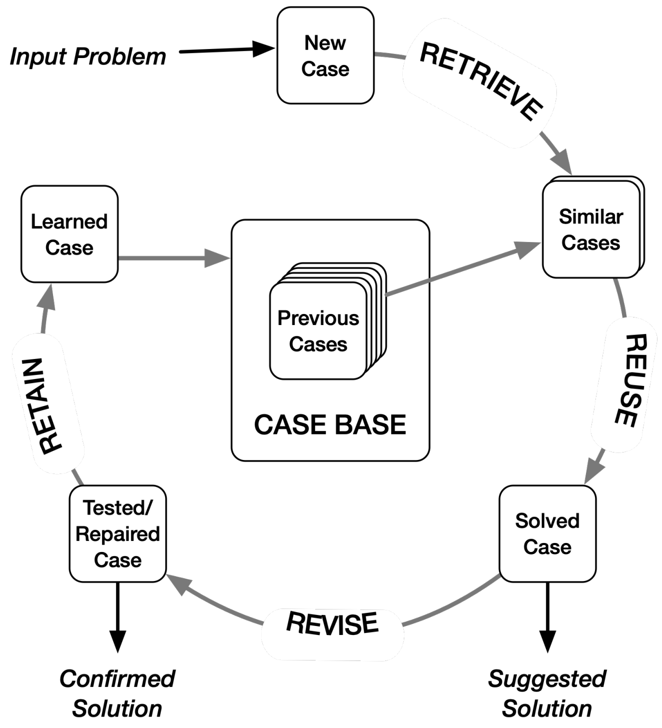


Figure 2.2: **The CBR cycle & its '4 REs'.** Adapted from the emblematic depiction by Agnar and Plaza (1994), still serving as the 'coat of arms' in the field.

CBR has been successfully applied to a variety of domains for diverse tasks such as classification, design, planning, justification and explanation. For example in *help desk systems*, Compaq's SMART (Acorn and Walden, 1992) stores prior problem-solving knowledge and increases the effectiveness of the customer support staff. It also handles previously unseen cases by revision. The product is awarded by the Association for the Advancement of Artificial Intelligence (AAAI) as an 'Innovative Application' in AI. In *industrial aiding systems*, Lockheed's CLAVIER (Hinkle and Toomey, 1994) suggests most appropriate layouts of composite material to be used in autoclave curing ovens. CBR reduced the need for the availability of experts during the process and eliminated the costs due to incompatible loads. Before the application of CBR, other reasoning methods methods like rule-based systems were used and found to be impractical due to high complexity of the processes (Mark, Simoudis, and Hinkle, 1996, p. 274). *Law*, as a natural domain where knowledge is in the form of 'cases', has been an area of particular interest for CBR (for a survey see Rissland, Ashley, and Branting, 2005). CBR has also been helpful for *educational* purposes where students can learn from concrete past problems and their solutions (for a review see Kolodner, Cox, and González-Calero, 2005). Case-based *recommender systems* is also a very active and prolific research area in CBR. Example applications vary from e-commerce product recommendation (e.g. Burke, 1999) to music recommenders (Baccigalupo and Plaza, 2007) and travel advisors (e.g. Ricci, Arslan, et al., 2002). One of the interesting aspects of this domain is that, in order to avoid mundane recommendations, CBR is usually expected to have diversity within retrieved similar content for a query (e.g. McSherry, 2002). For a review and a framework describing case-based recommenders, see (e.g. Bridge et al., 2005). *Health sciences* have always been another major application domain of CBR for a variety of purposes including diagnosis (e.g. López and Plaza, 1993), prognosis (e.g. Armengol, Palaudàries, and Plaza, 2001), therapy planning (e.g. Marling, Shubrook, and Schwartz, 2008), personalized medicine recommendation (e.g. Torrent-Fontbona and López, 2019) and tutoring of medicine students (e.g. Kwiatkowska and Atkins, 2004). For a survey of medical CBR applications see (Begum et al., 2011).

Besides, there has been significant research demonstrating that CBR is not only capable of suggesting similar solutions but, with flexible retrieval and adaptation mechanisms, it can also introduce novelty in solutions that leads to *creativity* (Kolodner, 1994). With a case base of buildings of different styles, Schmitt (1993) proposes case adaptation and case combination as a means to suggest innovative architectural designs. Given a musical score and an inexpressive interpretation of it, Arcos, López De Mántaras, and Serra (1998)'s SaxEx generates an expressive musical performance of the score. Beside musical background knowledge, SaxEx uses a CB of sample human performances and associated high-level expressiveness parameters which are automatically extracted from these recordings. Using an ontology for story generation and a case base of tales, Gervás et al. (2005) generate a story in the form of natural language as a response to a user query. The query determines the components of the desired story like characters, roles, places and actions. For more examples of CBR applications see (Leake, 1996a; Cheetham and Watson, 2005; Goel and Díaz-Agudo, 2017). The success of CBR led to development of generic *CBR frameworks* as well, e.g. Noos (Arcos and Plaza, 1997), jCOLIBRI (Díaz-Agudo et al., 2007), myCBR (Stahl and Roth-Berghofer, 2008) and eXiT*CBR (López, Pous, et al., 2011)—a framework specifically developed for medical applications.

Success stories for CBR exhibit common application domain characteristics. In these domains, first, the above-mentioned basic assumption of CBR holds, i.e. similar problems have reasonably similar solutions. And to this end, there can be built a notion of similarity so that the CBR system can reliably assess the similarity between a present problem and a past experience. Second, a complete formalization of domain knowledge is not practical or not possible, hence, training a domain model or generalizing rules for the domain is also impractical. However, it is still possible to construct an initial case base that is representative enough of the domain. Here, representativeness is two-fold: (1) case representation comprises necessary information of the problem and the solution to be effectively used in retrieval and reuse steps, and, (2) the initial CB plausibly covers the problem and the solution spaces. See (Mark, Simoudis, and Hinkle, 1996, p.292) for an interesting discussion on the laborious knowledge engineering effort behind case representation and domain coverage for the commercially successful CLAVIER system. In the same work, their claim that "adaptation is for humans" (ibid., p.293) is also of interest, especially when the cost of failure is high, as it is so in their domain.

2.1.1 Utility problem in CBR

The coverage of a CB widens as new cases from the uncovered regions of the problem and solution spaces are revised and retained. By the retention of new cases the CB becomes richer in experience and CBR system is likely to provide more precise solutions in the future. Nevertheless, the richness in the number of cases comes with a cost. Due to its lazy nature, the efficiency of CBR's retrieval phase affects overall system performance. And in practice, despite the seeming advantage for better problem solving, a growing CB eventually causes the so-called *swamping utility problem* (Francis and Ram, 1993; Smyth and Cunningham, 1996). This problem emerges when adding new cases to a CB degrades the system efficiency instead of improving it. Specifically, degradation happens due to computationally expensive search for similar past problems. For example, being simple and effective, *k-Nearest Neighbors* (kNN) (Cover and Hart, 1967) search is a widely used algorithm in CBR retrieval in particular and in *instance-based learning* (Aha, Kibler, and Albert, 1991) in general. The naïve approach to find the kNN of a query is to perform a *brute-force* search (a.k.a. linear search) in the CB by evaluating the similarity of each case to the given query and return the k most similar cases. The linear 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 costly similarity calculations, and is likely to evolve into the utility problem.

Among considerable efforts in early CBR research to improve retrieval performance are using parallel architectures for brute-force search (Kolodner, 1988) and especially, defining *index vocabularies* to guide retrieval (e.g. Schank et al., 1990). However, as CBR came into more use and CBs started to grow in size, the main approach in CBR community to tackle the utility problem has shifted towards controlling the CB growth via *case base maintenance* (CBM) techniques. CB size is kept under control primarily via deletion of select cases while preserving the competence of the overall CB (e.g. Smyth and Keane, 1995) or by avoiding their retention (e.g. Muñoz-Avila, 1999). The reduction in CB size, in one way or another, inevitably causes a loss in information that was rightfully learned by the CBR system. However, this trade-off has been justified by the increase in retrieval efficiency. For competence definition see (Smyth and Keane, 1995), for CBM examples see (Leake et al., 2001; Juarez et al., 2018), and for more dimensions of CBM see (Wilson and Leake, 2001).

On the other hand, despite the present ubiquity of digital data in domains such as healthcare (W. Raghupathi and V. Raghupathi, 2014) where CBR historically excelled at, the loss of information

[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.

itself may be costly in terms of a decrease in accuracy of life-critical solutions. If effective strategies to overcome the utility problem cannot be developed, once-very-effective CBR is bound to be less appealing for these domains, and at best it can play a less important role downgrading its proven potential. CBR community has an open eye for this challenge and 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 millions of cases—a scale far beyond previous CBR applications. Goel and Díaz-Agudo (2017) refer to *big-data* as one of the "hot" topics in the field.

For example, to tackle the computational cost of retrieval in large-scale case-based regression, Jalali and Leake (2015) use *Locality Sensitive Hashing* (LSH) (Indyk and Motwani, 1998) for approximate nearest neighbors search on top of a *MapReduce* framework (J. Dean and Ghemawat, 2004) that allows parallel processing of multiple queries. LSH technique partitions similar cases into the same 'buckets' and tries to maximize the probability of the collision of their 'hashes'. And, with an appropriate hash function, a target query is directed to the bucket where its nearest neighbors are expected to reside without actual similarity comparison of the query with the cases. Thus, LSH discards the need to reduce the CB size for speeding up retrieval. However, due to its approximate nature, LSH may imply a loss in solution accuracy as well. The authors show they can compensate this loss with their "ensemble" method for adaptation. Woodbridge et al. (2016) emphasize the challenge against the immensity of data produced by medical devices and they propose a *Monte Carlo* approximation method on an LSH scheme to improve the search speed and result quality in a medical CBR system with a case base of millions of biomedical signals. In another application for on-line smart grids optimization, Troiano, Vaccaro, and Vitelli (2016) also apply MapReduce to the retrieval of similar past 'power system states' from a large-scale smart grid database to suggest an 'optimal power flow' solution.

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. Besides, there is more to CBR maintenance than just refining the case base. Other knowledge containers like similarity and adaptation methods can require maintenance too (Wilson and Leake, 2001). That said, big-data tools may indeed outdate CB compression strategies as discussed in (Jalali and Leake, 2015, p.184).

Due to the commonness of the problem they address, nearest neighbor search (NNS) and its natural variant kNN have been extensively used in a plethora of fields apart from CBR. For application examples and general overview of NNS see (Chávez et al., 2001). Accordingly, there have been vast efforts to speed up NNS by researchers from this variety of fields. There are two ramifications in these contributions. First is speeding up the search to find *exact* neighbors of a target query. Second, when finding exact neighbors would not be computationally feasible no matter what, some efforts resorted to approximation methods to find *approximate* enough neighbors instead. And, some proposed methods can serve for both purposes—as our proposal in this dissertation.

A notable method to overcome the computational overhead of both exact and approximate kNN search is using *search trees* such as *k-d trees* and its variants. Search trees partition the multidimensional search space and NNS is conducted by pruning parts of the tree that cannot include the nearest neighbor(s) (e.g. Wess, Althoff, and Derwand, 1993; Yianilos, 1993). See also (Kibriya and Frank, 2007) for a comparison of search tree based exact NNS algorithms. However, beside the cost of their construction and maintenance, k-d trees are prone to the *curse of dimensionality* phenomenon (Bellman, 1957) and although improvement suggestions exist (e.g. Eastman and Weiss, 1982), they are usually not recommended for high dimensional spaces.[2] In fact, high-dimensionality may become a problem for NNS in a more general sense. This phenomenon occurs when/if the distance between nearest and farthest neighbors of a point in space becomes negligible as the dimensionality increases. Beyer et al. (1999) show the conditions when it is meaningful to use NNS, especially when approximate NNS is used as heuristics. Typical techniques for approximate matching in high dimensions are using search tree adaptations (e.g. Muja and Lowe, 2014), proximity graphs (e.g. Hajebi et al., 2011) and hashing (e.g. Andoni and Indyk, 2006). On the other hand, recent availability of GPUs for general purpose computation, and the parallelizable nature of brute-force NNS led some researchers also to harness GPUs to speed up exact kNN (e.g. Garcia, Debreuve, and Barlaud, 2008; Arefin et al., 2012).

When time is a dire constraint, thanks to their ability to provide always a solution upon interruption, *anytime algorithms* (T. Dean and Boddy, 1988; Zilberstein, 1996) have also been incorporated in NNS. For example, in data stream mining, Kranen and Seidl (2009) apply their own approaches to existing anytime versions of Support Vector Machines, NNS and Bayes classifiers. They implement a simple classification confidence measure of their own and evaluate their approaches on constant data streams and compare these three classifiers. In the same domain, Ueno et al. (2006) propose 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. For database applications, Xu et al. (2008) use an anytime search strategy for kNN searches with Multi-Vantage Point Trees (Bozkaya and Ozsoyoglu, 1997). We give examples to usages of anytime algorithms in CBR in section 2.3.

2.2 Temporal case base / Domain of interest

Temporal dimension has been taken into account since earlier CBR research for application domains like times series prediction (Nakhaeizadeh, 1994), eating disorders in healthcare (Bichindaritz and Conlon, 1996), robot navigation (Ram and Santamaría, 1997) and more recently, fault prediction in oil well drilling (Jære, Aamodt, and Skalle, 2002). Some of these early works (e.g. last three mentioned) propose frameworks for the representation of temporal cases and for the steps of the full CBR cycle in Figure 2.2. In these works, temporality has been represented either as a *point* in time or an *interval* of time as a *feature* of case problem description. However, the representation of time dimension 'per case' is not enough to meet the needs of domains where a set of cases represent an evolution of a phenomenon. Healthcare is such a domain that for diagnosis, prognosis and/or treatment of patients, medical experts usually need to take into account the evolution of a patient's health rather than considering only his/her current condition (e.g. Sox, Higgins, and Owens, 2013, p.8). Therefore, to compare the medical histories of patients, there has to be more to retrieval than just treating each session as an individual case. That is, medical histories of patients represented as sequences of related cases should be comparable against each other as well.

Accordingly, reasoning with *temporally related cases* is addressed by later work in CBR community. Sánchez-Marré et al. (2005) propose an "Episode-Based Reasoning" framework where a sequence of temporally related cases is referred to as an "episode". They use a method for

[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 (Bentley, 1975, p.516).

the abstraction of cases to form episodes and the similarity is calculated by "the aggregation of the similarity values among cases belonging to each episode". They evaluate their framework in wastewater treatment domain. Montani and Portinale (2006) 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. For a history level comparison, first they "summarize" the contents of cases in a history by means of *temporal abstraction* (Shahar, 1997) that allows qualitative abstraction of time series data. In a more recent work, Van den Branden et al. (2011) present a CBR system as an additional component to a proprietary software for clinical decision support. Although the temporal relation between cases is not defined explicitly, their cases can capture multiple instances of "special forms" belonging to different clinical stages where each form may hold multiple "instances" of clinical data for a patient. They use weighted kNN (Wettschereck, Aha, and Mohri, 1997) so that recent data instances contribute more to the similarity.

In this dissertation, we are particularly interested in domains where:

- the CB can be organized as *sequences of temporally related cases*;
- the *similarity* metric takes into account the *evolution* of a sequence (partially/fully) instead of treating each case individually (e.g. considering the partial or complete medical history of a patient instead of treating each of his/her sessions as a standalone case);
- the problem space is a *metric space*.

We give a brief summary of the properties of metric spaces in the following subsection. Then, in subsection 2.2.2, we define the concepts for the temporal case bases regarding our domain of interest.

2.2.1 Metric problem space

The set of problems comprised by cases in a case base and the distance measure used to calculate the distance (a.k.a. dissimilarity) between two problems define the *problem space* for a CBR system. A problem space X becomes a *metric space* if the distance measure $d : X \times X \to \mathbf{R}$ is a true *metric* and thus satisfies the following four axioms (M. M. Deza and E. Deza, 2009, p. 4): $\forall x, y, z \in X$,

1. $d(x, y) \geq 0$ (*non-negativity*)
2. $d(x, y) = 0 \iff x = y$ (*identity of indiscernibles*)
3. $d(x, y) = d(y, x)$ (*symmetry*)
4. $d(x, y) \leq d(x, z) + d(z, y)$ (***triangle inequality***)

Euclidean space is arguably the most popular metric space used in CBR systems. Where each case is represented by a *feature vector* of attribute-value pairs, the simplest *euclidean distance* in an n-dimensional space between a case $\mathbf{x}$ and a target problem $\mathbf{y}$ is given by the formula:

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2} = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{2.1}$$

where x_i and y_i are the values of the i^{th} attributes of $\mathbf{x}$ and $\mathbf{y}$ respectively.

To avoid an unintended bias for an attribute that has a wider numeric range, it is also common practice to *normalize* attributes' values and scale them into the same range before distance calculation. Furthermore, since *similarity* plays a more central role in CBR, the distance measure d itself is normalized into the $[0, 1]$ range and it is converted to a similarity metric sim as:

$$sim(\mathbf{x}, \mathbf{y}) = 1 - d(\mathbf{x}, \mathbf{y}) \tag{2.2}$$

The *triangle inequality* property of metric spaces is of particular interest in kNN search as it provides the upper-bound of distance between two points x, y when the distances of these two points to a common third point z are known. And it has been widely exploited for speeding up NNS in many different domains such as time series (e.g. Mueen et al., 2009), databases (e.g. Xu et al., 2008) and CBR retrieval (e.g. Schaaf, 1996; Montani, Bellazzi, et al., 1998). We will also be leveraging this property of metric spaces to define the upper-bound of similarity (section 3.1) which helps us to identify kNN candidates of a query (section 3.2).

2.2.2 Definitions of domain concepts

For clarification purposes, we define the concepts for a temporal case base with respect to our domain of interest below:

Definition 2.1 (Problem sequence)**.** *A sequence of temporally related problems (detailed in Definition 2.4) that belong to a particular entity (e.g. to a patient, to a particular sensor that sends time series data).*

Definition 2.2 (Update)**.** *Any new information/data for a particular problem sequence that is related to problem description (e.g. a consecutive session for a particular patient, a new data observation from a particular sensor).*

Definition 2.3 (Time window)**.** *A conceptual time frame marking the start and end updates on a sequence, and which is used to derive a problem description. Specifically used for similarity assessments in CBR retrieval. Depending on the design choice for a CBR system, 'expanding' or 'fixed-width' time window approach is applied.*

Definition 2.3.1 (Expanding time window)**.** *Time window that expands for each consecutive update, encompassing whole problem sequence.*

Definition 2.3.2 (Fixed-width time window)**.** *Time window of fixed width w that slides for each consecutive update, encompassing the last w updates to form the problem description (a.k.a. sliding time window). Where u is the index of the end update for the time window, this approach encompasses updates between $[0, u]$ when $u < w$; whereas, for $u \geq w$, updates between $(u - w, u]$ are covered.*

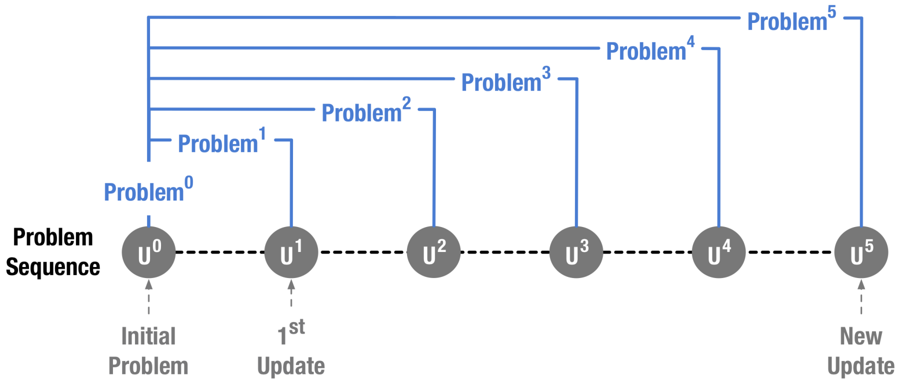
Definition 2.4 (Problem)**.** *A problem description generated for the updates encompassed by a time window. The 'initial problem' is derived from the initial update regardless of the time window approach (e.g. a sequence is created for each patient in their first session, for the first data received from a sensor)*

Definition 2.5 (Query)**.** *Invoked by a new update of a particular sequence, it is the target problem that is posed as a query to the CBR system.*

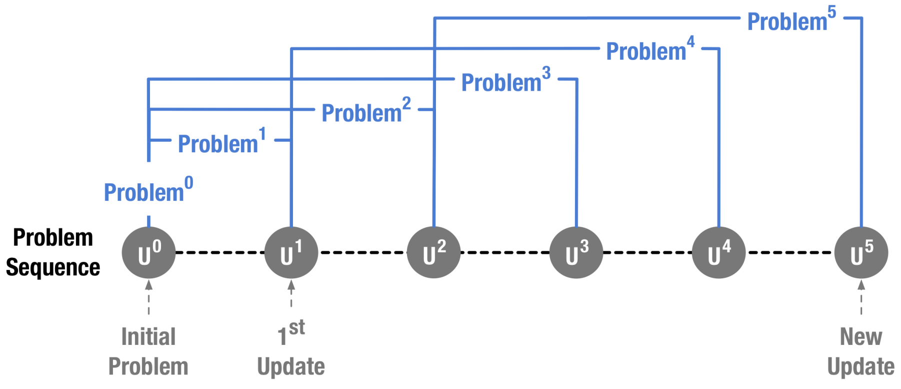
Definition 2.6 (Case)**.** *After CBR's problem solving episode, the problem posed as the query and its solution are saved as the problem and solution parts of a new case.*

Definition 2.7 (Temporally related cases)**.** *All consecutive problem solving episodes for the queries of the same problem sequence form a sequence of 'temporally related cases'.*

Definition 2.8 (Temporal case base)**.** *A case base comprising sequences of temporally related cases.*



(a) Time window $width$: *Expanding*



(b) Fixed time window $width$: 4

Figure 2.3: **Expanding** ***vs*** **fixed-width time window approaches to derive problem descriptions** out of the updates of a problem sequence.

Figure 2.3 illustrates the basic defined concepts. We interpret the figure following the examples in definitions for healthcare domain: A unique electronic health record (*problem sequence*) is created for the initial session (*initial problem*) of each particular patient. The initial problem (which can be seen as the 0^{th} *update* to the sequence) invokes a *query* for similar patients. When the problem solving episode ends by suggesting a solution, say a therapy plan, the problem-solution pair is saved as the initial *case* of the patient. Each following update invokes the generation of a new query by applying a *time window* onto the sequence. Depending on the design choice for whether the whole medical history or a part of it should be queried, the similarity metric would use an *expanding* (Figure 2.3a) or *fixed-width* (Figure 2.3b) time window approach respectively to derive the problem description for a 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. six queries in total in Figure 2.3).

After each CBR's problem solving episode, the problem that served as a query and its solution are saved as the problem and solution parts of a new case. All consecutive problem solving episodes 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.

Regardless of the implementation for case generation, we underline the important fact that consecutive problems of a sequence bear the information of their predecessors. For the expanding time window approach the information of predecessors is complete (see Figure 2.3a), and for the fixed-width approach it is partial (see Figure 2.3b).

We note that our definition of 'sequence of temporally related cases' (Def. 2.1 & 2.7) is conceptually akin to the "history" in (Montani and Portinale, 2006). We leave the generation of problem description to implementation. Their temporal abstraction method could be used in our proposal as well, as long as a similarity *metric* is used later with the abstracted data. Our sequence definition is also reminiscent of the "episode" in (Sánchez-Marré et al., 2005). An interesting aspect of this latter temporal framework is that it allows a case to belong to multiple episodes. For our domain of interest, a case can belong to a single problem sequence, e.g. as a session can belong only to a particular patient. Furthermore, in certain aspects, our 'problem' encompassing sequence 'updates' resembles the "instances" of clinical data captured by a "special form" in (Van den Branden et al., 2011). Their special forms are of more general purpose and can hold demographic information about patients as well. Whereas in our problem description, we are mainly interested in features that can be used in similarity assessment.

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

[3]Some anytime algorithms may need a short initialization time before they can be interrupted (e.g. Ueno et al., 2006).

in quality is greater at the early stages of execution and it diminishes over time.

After being interrupted, an AA should 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 (Grass and Zilberstein, 1996). See (Zilberstein, 1996) 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 (Zilberstein, 1993). An AA can have several PPs to track different result attributes (Boddy and T. Dean, 1989). 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 (Grass, 1996) and (Zilberstein, 1993, p.45) for further discussion on possible representations of PPs.

In accordance with the above description of a preferable AA, and regarding the focus of this dissertation, an anytime kNN search algorithm to be used in CBR retrieval 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 overhead if it is allocated extra computation time to improve on its results.

Although not a very common practice, anytime algorithms have been used in CBR as well. For example, in Schaaf's "Fish and Sink" (FaS) (Schaaf, 1995), the author deals with domains where "aspects" of similarity between two cases might change. 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 labelling those neighbors as "indirectly tested" (IDT) cases. The time of interruption is important and only after all cases are labelled 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. In this work, although the 'quality' of results are discussed (ibid., p. 545), there is no explicit definition of quality as such.

Ricci and Avesani (1995) use an anytime algorithm not to speed up CBR retrieval but in learning a local similarity metric to be used in 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

[4]In this dissertation we follow the terminology for anytime algorithm components given in (Zilberstein, 1996).

input case c and its neighbors depending on the role of the neighbors in solving c by incorporating a *reinforcement learning* procedure that adjusts the local weights.

In the following chapter, we introduce our base algorithm that significantly speeds up exact kNN search for CBR retrieval. Then, in Chapter 4, we convert it to an anytime algorithm exhibiting all desired properties mentioned in this section.

Chapter 3

Lazy kNN

For our domain of interest, the temporal nature of case bases offers opportunities to speed up the kNN search in CBR retrieval, in particular when similarity measures take into account the evolution of cases rather than treating them individually.

As described above, when a new update is appended to a problem sequence, the case formed for this update bears the information of previous updates. Consequently, as the number of updates in a problem sequence increases, the difference between the cases formed for the last and penultimate updates decreases due to shared long history between them. Therefore, when this difference becomes minimal, it is intuitive to think that the kNN of both cases will likely be similar, if not identical. Effectively, the fundamental assumption of CBR that "*similar problems have similar solutions*" also supports this hypothesis.

This intuition brings about a thought-provoking question on similarity estimation: How similar a neighbor of a case can get to the consecutive case of the same sequence? Following the previous health record example, if patient C was found similar for patient P's eighth therapy session P^8, at most how similar can C become for P's ninth session P^9? And, more importantly, could C even be a candidate for k most similar patients for P^9? The last question is of particular interest. Because, if somehow we can prove that C cannot be a kNN candidate for P^9, then there is no use of evaluating it in that kNN search. That is, there is no need to calculate its similarity to P^9 since it could never make it to the final kNN list. After all, kNN are what we are looking for within the problem space.

This chapter is organized as follows. In the following section, we define the upper-bound of similarity of a case to a consecutive query of a problem sequence in metric spaces. Later in section 3.2, we show how we can leverage this upper-bound to identify kNN candidates and more importantly, how we can rule out non-candidate cases. Section 3.3 presents the complete *Lazy kNN* algorithm that utilizes this candidacy assessment and remarkably speeds up kNN search by avoiding unnecessary similarity calculations. Then, we describe how we built small-to-large temporal CBs out real-world datasets for our experiments in section 3.4. Finally, we give the results of the experiments that empirically demonstrate the speed-up achieved by *Lazy kNN* in section 3.5.

3.1 Upper-bound of similarity in a temporal case base

In this section we define the *upper-bound of similarity* in a temporal case base by leveraging the triangle inequality detailed previously in subsection 2.2.1. Later in section 3.2, we show how this upper-bound serves us to identify the *true kNN candidates* in a CB for a target problem.

The *candidacy assessment* that we will describe forms the basis of our proposed algorithm in which a case in the CB is evaluated *only* when it is deemed a candidate for a query. The assessment of true candidates in such a *lazy* fashion helps us to avoid unnecessary distance calculations and, as we will show, results in remarkable speed-up in kNN search.

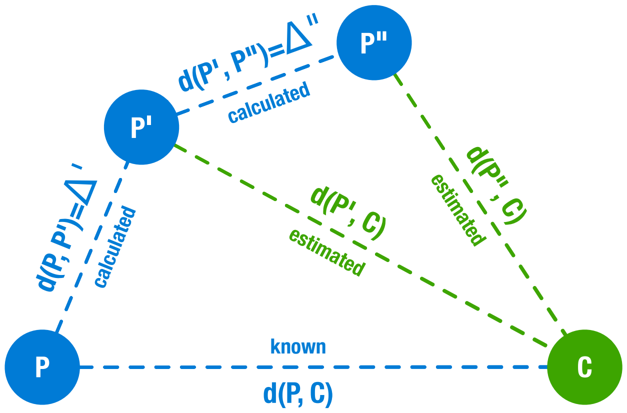


Figure 3.1: Using the **triangle inequality** to calculate the **upper-bound of similarity** of a case to two consecutive problems of the same sequence. Where d is the distance metric, P' and P'' are the problems for the first and second consecutive updates to the initial problem P in the same sequence respectively, and C is a case in the CB whose distance to P we had calculated before; we have $d(P,C) \leq d(P',C) + \Delta'$ and $d(P,C) \leq d(P'',C) + \Delta' + \Delta''$.

As illustrated in Figure 3.1, given a normalized metric d and four points in problem space

$$\langle P, P', P'', C \rangle$$

where P' and P'' are the problems for the first and second consecutive updates to initial problem P in the same sequence respectively, and C is a case in the CB whose distance to P we had calculated before; these four points satisfy following inequalities:

$$d(P,C) \leq d(P',C) + \overbrace{d(P,P')}^{\Delta'} \tag{3.1}$$

$$d(P',C) \leq d(P'',C) + \overbrace{d(P',P'')}^{\Delta''}$$

Then, using the latter inequality in the former, we get:

$$d(P,C) \leq d(P'',C) + \Delta' + \Delta'' \tag{3.2}$$

Now, where $sim(x,y) = 1 - d(x,y)$, we can transform the distance inequalities (3.1) and (3.2) into inequalities for similarities. Following the inequality (3.1):

$$\underbrace{1 - d(P,C)}_{sim(P,C)} \geq \underbrace{1 - d(P',C)}_{sim(P',C)} - \Delta'$$

$$sim(P,C) \geq sim(P',C) - \Delta'$$

which leads to:

$$sim(P',C) \leq sim(P,C) + \Delta' \tag{3.3}$$

And following the inequality (3.2):

$$\underbrace{1 - d(P,C)}_{sim(P,C)} \geq \underbrace{1 - d(P'',C)}_{sim(P'',C)} - \Delta' - \Delta''$$

$$sim(P,C) \geq sim(P'',C) - \Delta' - \Delta''$$

which leads to:

$$sim(P'',C) \leq sim(P,C) + \Delta' + \Delta'' \tag{3.4}$$

The inequality (3.3) states that, just by calculating Δ', we know that a new update P' can be more similar to *any* case C than its predecessor P is to C, at best by a degree of Δ'. We would like to reemphasize the word "*any*" here, because note that Δ' calculation does not involve any other case but only the two consecutive problems P and P'. Likewise, a following problem P'' can be more similar to any case C in the CB than P is to it, at best by a degree of $\Delta' + \Delta''$ as shown in the inequality (3.4). Figure 3.1 depicts above inequalities between three consecutive problems of a sequence and a case in a CB. And if we generalize the inequality (3.4), we have:

$$sim(P^u, C) \leq sim(P^j, C) + \sum_{s=j+1}^{u} \Delta^s \tag{3.5}$$

where $\Delta^s = d(P^{s-1}, P^s)$. Finally, the right-hand side of the inequality (3.5) gives us the following generic definition of the upper-bound of similarity:

Definition 3.1 (Upper-bound of similarity)**.** *Given a problem sequence P, a case C and a distance metric d; when we know the similarity of C to a prior problem P^j in the sequence—$sim(C, P^j)$, the upper-bound of similarity of C to any later coming problem P^u—$UB(C, P^u)$—is obtained by adding the sum of all inter-problem distances between P^j and P^u to the already known similarity:*

$$UB(C, P^u) = sim(C, P^j) + \sum_{s=j+1}^{u} \Delta^s, \qquad \textit{where } \Delta^s = d(P^{s-1}, P^s)$$

3.2 Lazy assessment of kNN candidates

We designed our proposed algorithm *Lazy kNN* in order to exploit the upper-bound of similarity defined above. *Lazy kNN* utilizes this upper-bound in identifying the true kNN candidate cases for a given query.

Definition 3.2 (kNN candidate)**.** *A case C is a kNN candidate for a consecutive problem P^u of the sequence P if its upper-bound of similarity $UB(C, P^u)$ is greater than the similarity of any of the best-so-far kNN to P^u.*

Accordingly, during the kNN search, a case in the CB is *evaluated*—i.e. its *actual similarity* to the query is calculated—only when it is deemed a candidate by *Lazy kNN*, hence the *lazy*. The calculated similarity, of course, may or may not meet the expectancy and the case may not make it to the kNN list. However, to find exact kNN of a query, it is necessary to calculate all candidates' similarities since they all have the potential to be a kNN member.

On the other hand, and most importantly, this candidacy assessment helps us to avoid unnecessary similarity calculations in kNN search. If a case is not deemed a kNN candidate for a query, there is no use in calculating its distance. Because, if the upper-bound of similarity of a case cannot beat the best-so-far kNN, there is no chance that its actual similarity would do so. *Lazy kNN* leverages this ability to ignore "non-candidate cases" in the CB and thus speeds up kNN search considerably. And it does so not by checking the candidacy of each individual case, but by using the candidacy assessment to detect *cut-off points* in search where the triangle inequality assures that from that point onwards there could be no case which can beat the kNN found so far. Thus, **the true strength of *Lazy kNN* comes from its ability to ignore 'bulks' of cases with a 'single' candidacy check**.

Definition 3.3 (Cut-off point in kNN search)**.** *When we know the neighbors of problem P^{u-1} sorted in a descending order regarding their similarities to P^{u-1}, during the kNN search for the next update P^u, the first neighbor C of P^{u-1} which proves not to be a kNN candidate marks the cut-off point in kNN search within neighbors of P^{u-1}.*

Specifically, in the definition above, the neighbors following C would have equal or lower similarities to P^{u-1} due to being sorted and this assures that their upper-bounds of similarity given by Definition (3.1) cannot beat best-so-far kNN either. And as a result, together with C, the rest of the neighbors of P^{u-1} can also be ignored without further candidacy checking among them.

Of course, a neighbor C of P^{u-1} which is ignored at the kNN search for P^u can prove to be a kNN candidate for a later problem P^{u+1}. Precisely, this happens when its upper-bound of similarity to P^{u+1} (i.e. $sim(C, P^{u-1}) + \Delta^u + \Delta^{u+1}$) beats the k^{th} NN of P^{u+1} found so far. Therefore, remembering the neighbors of previous problems in a sequence helps us to identify kNN candidates for later problems and discard non-candidate cases in their kNN searches. And for this purpose, *Lazy kNN* maintains a data structure that we call $RANK$ for every problem sequence posed as a query to the CBR system.

The $RANK$ instance created for a problem sequence P holds assessed cases during the kNN search for each problem in the sequence separately and in sorted order regarding their similarities to that particular problem. The distance between each problem and its predecessor in the sequence (i.e. Δ) is also stored in this structure for candidacy assessments. More formally, where:

P^j is the target problem for the j^{th} update of the problem sequence P;

NN is the sorted list of $(case, sim)$ 2-tuples formed for every $case$ to whose similarity sim to P^j is actually calculated; and NN is sorted in a descending order, the most similar case being at the top;

$Stage^j$ is the (NN, Δ^j) 2-tuple created for P^j and;

$NN(Stage^j)$ denotes the NN in $Stage^j$;

$RANK$ is the reversed list of $Stage$s that are populated for all past problems in the problem sequence P. When a new problem P^u arrives, the content of this list is as follows:

$$RANK = [Stage^{u-1}, Stage^{u-2}, \ldots, Stage^1, Stage^0]$$

Figure 3.2a gives an example of a $RANK$ instance maintained for a problem sequence P_7. After the kNN search for the 3rd consecutive problem (i.e. P_7^3), there are four stages including the one for the initial problem ($Stage^0$). When *Lazy kNN* terminates for a $k = 3$ setting, top three cases in $Stage^3$ are the kNN of P_7^3. Note that, the similarities of all cases in a j^{th} stage were evaluated for the corresponding update P_7^j. A case C_a^b denotes the b^{th} case in the a^{th} sequence of temporally related cases in the CB.

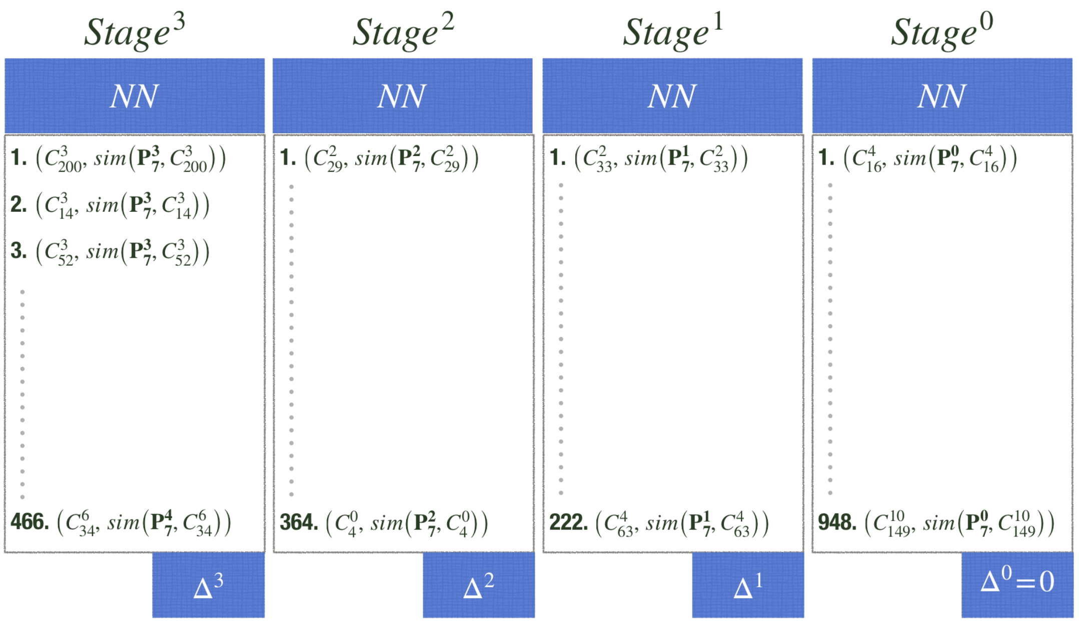
Lazy kNN algorithm works as follows. When the initial problem P^0 of a new sequence P is posed as a query, *Lazy kNN* creates an instance of $RANK$ dedicated to this particular sequence. The definition of the upper-bound of similarity in Definition (3.1) of a case C to a problem P^u requires the knowledge of C's similarity to a problem of any prior sequence update. Therefore, the candidacy assessment cannot be applied to the initial problem P^0. So, *Lazy kNN* delegates the kNN search for the initial problem P^0 to any search algorithm of choice that will return all cases in the CB sorted in a descending order regarding their similarities to P^0. Once we know all neighbors of the initial problem P^0, they are added into $Stage^0$ maintaining their order. And, starting from the first consecutive problem P^1, the algorithm uses the candidacy assessment we have just outlined to speed up kNN search.

To find the kNN of u^{th} problem P^u, *Lazy kNN* uses the $RANK$ instance maintained for the sequence P. First, it creates an empty $Stage^u$ for the new query and calculates the distance of the query with the predecessor problem (i.e. Δ^u). Later, *Lazy kNN* follows the above-mentioned (p. 3) intuition that the neighbors of P^u are likely to be within the neighbors of the previous problem. So, the kNN search starts from $Stage^{u-1}$ which holds the assessed cases for P^{u-1}, and continues backwards till and including $Stage^0$ in $RANK$.[1]

The search in $Stage^{u-1}$ starts by calculating the similarities of the top k cases in $Stage^{u-1}$. These cases are removed from $NN(Stage^{u-1})$ and placed into $NN(Stage^u)$ in a ranked order regarding their similarities to P^u. Thus, these k cases form the initial kNN. Throughout the rest of the kNN search, for a case to be a true kNN candidate, its upper-bound of similarity to the query has to beat (i.e. be greater than) the similarity of the current k^{th} member of $NN(Stage^u)$. Then, continuing from $Stage^{u-1}$, all $Stage$s in $RANK$ are iterated in the search for the candidate cases.

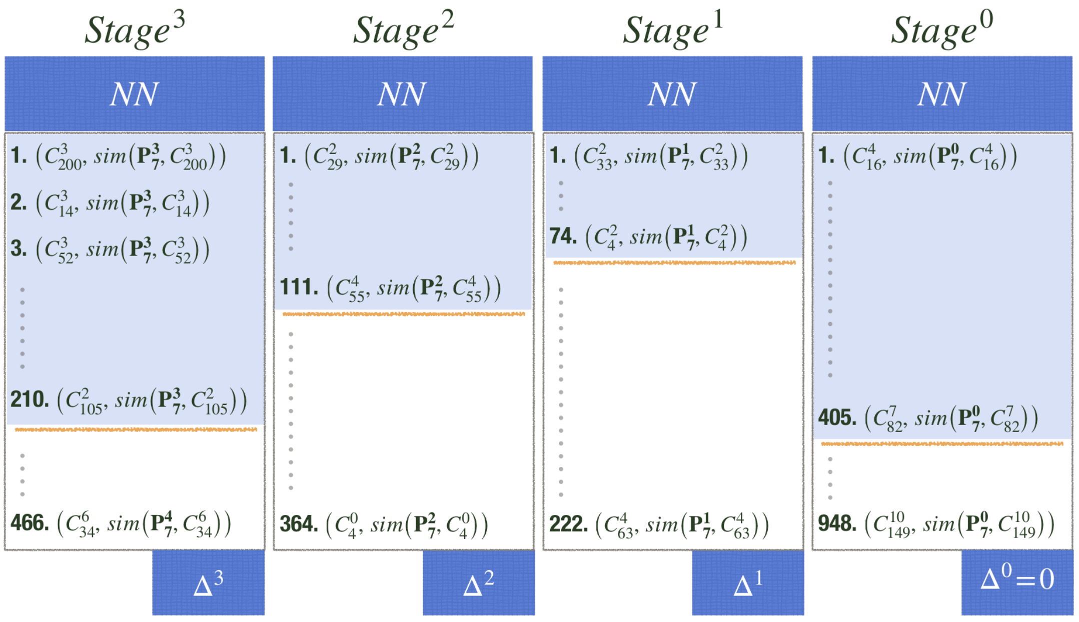
As a rule, if a case is deemed a kNN candidate for P^u, (i) its actual similarity to P^u is calculated, (ii) it is removed from the $Stage^j$ it was in and (iii) it is added to $NN(Stage^u)$ together with its calculated similarity. As we mentioned, the NN list of a $Stage$ is maintained sorted. In order to avoid excessive sorting during kNN search, if a candidate case's actual similarity surpasses the current k^{th} NN, it is inserted at the appropriate position within the top k members of the $NN(Stage^u)$ list. And if it does not beat, it is appended to the end of the list instead. And the whole $NN(Stage^u)$ list is sorted *only once* at the end of the kNN search. Insertion of the winning

[1]This is not the only way to iterate $RANK$, some alternatives will be presented in section 4.9.



$Stage^3$
$Stage^2$
$Stage^1$
$Stage^0$
NN
1. $(C^3_{200}, sim(\mathbf{P}^3_7, C^3_{200}))$
2. $(C^3_{14}, sim(\mathbf{P}^3_7, C^3_{14}))$
3. $(C^3_{52}, sim(\mathbf{P}^3_7, C^3_{52}))$
466. $(C^6_{34}, sim(\mathbf{P}^4_7, C^6_{34}))$
1. $(C^2_{29}, sim(\mathbf{P}^2_7, C^2_{29}))$
364. $(C^0_4, sim(\mathbf{P}^2_7, C^0_4))$
1. $(C^2_{33}, sim(\mathbf{P}^1_7, C^2_{33}))$
222. $(C^4_{63}, sim(\mathbf{P}^1_7, C^4_{63}))$
1. $(C^4_{16}, sim(\mathbf{P}^0_7, C^4_{16}))$
948. $(C^{10}_{149}, sim(\mathbf{P}^0_7, C^{10}_{149}))$
Δ^3
Δ^2
Δ^1
$\Delta^0 = 0$

(a) After kNN search for P_7^3



$Stage^3$
$Stage^2$
$Stage^1$
$Stage^0$
NN
1. $(C^3_{200}, sim(\mathbf{P}^3_7, C^3_{200}))$
2. $(C^3_{14}, sim(\mathbf{P}^3_7, C^3_{14}))$
3. $(C^3_{52}, sim(\mathbf{P}^3_7, C^3_{52}))$
210. $(C^2_{105}, sim(\mathbf{P}^3_7, C^2_{105}))$
466. $(C^6_{34}, sim(\mathbf{P}^4_7, C^6_{34}))$
1. $(C^2_{29}, sim(\mathbf{P}^2_7, C^2_{29}))$
111. $(C^4_{55}, sim(\mathbf{P}^2_7, C^4_{55}))$
364. $(C^0_4, sim(\mathbf{P}^2_7, C^0_4))$
1. $(C^2_{33}, sim(\mathbf{P}^1_7, C^2_{33}))$
74. $(C^2_4, sim(\mathbf{P}^1_7, C^2_4))$
222. $(C^4_{63}, sim(\mathbf{P}^1_7, C^4_{63}))$
1. $(C^4_{16}, sim(\mathbf{P}^0_7, C^4_{16}))$
405. $(C^7_{82}, sim(\mathbf{P}^0_7, C^7_{82}))$
948. $(C^{10}_{149}, sim(\mathbf{P}^0_7, C^{10}_{149}))$
Δ^3
Δ^2
Δ^1
$\Delta^0 = 0$

(b) Evaluated cases for P_7^4

Figure 3.2: $\boldsymbol{RANK}$ **during** the **kNN search** for the 4^{th} consecutive problem of the sequence P_7 (i.e. P_7^4). The orange colored horizontal lines are the *cut-off* points in the search within that particular stage. Evaluated candidates in the blue shaded areas will be removed from their current stages and placed into the new $Stage^4$ created for this problem. C_a^b denotes the b^{th} case in the a^{th} sequence of temporally related cases in the CB.

candidates adequately guarantees that we always have a ranked kNN list during the search. Thus, when a candidate really surpasses the k^{th} NN found so far, the kNN list and hence the threshold for candidacy for the remaining cases change. And this is enough for the algorithm to work.

During the kNN search, if a case in a $Stage^j$ proves not to be a candidate (i.e. its upper-bound of similarity remains equal to or below the current k^{th} NN's), this means that from that point onwards there will not be any candidates in $Stage^j$. So, the search continues from the next stage, $Stage^{j-1}$. The algorithm continues to iterate all stages in a likewise manner till after evaluating the last candidate case in the last non-empty stage in $RANK$. When all candidate cases in all $Stage$s in $RANK$ are evaluated, *Lazy kNN* guarantees that the top k cases in $Stage^u$ are the *exact kNN* of P^u.

Figure 3.2b depicts evaluated cases during the kNN search for P_7^4 in blue shaded areas of stages. The horizontal lines right after the last evaluated case marks the cut-off point in search in the stage. For example, the line in $Stage^3$ shows that the 211^{th} case was the first case in this stage which proved not to be kNN candidate. That is, its upper-bound of similarity could not beat the best k^{th} NN found so-far. Thus, the search continued from $Stage^2$ which yielded 111 candidates, and so on. All the cases in shaded areas are removed as they are evaluated during the search and placed into $Stage^4$.

If all of the cases in an intermediate $Stage^j$ are evaluated as candidates, $Stage^j$ becomes empty. But we still keep this empty intermediate stage in $RANK$ since we will need its attached Δ^j for the sequence updates to come. However, an empty stage at the end of the $RANK$ can be purged, as its attached Δ will never contribute to future kNN searches.

We also note that the initial $Stage^0$ had all cases in the CB and later during the evolution of the problem sequence, evaluated cases for consecutive problems are moved into the corresponding stages. Therefore the size of $RANK$ never changes and is equal to the CB size throughout the evolution of the sequence. In Figure 3.2a, the CB size is the sum of the size of all stages which is 2,000 cases. When the kNN search for P_7^4 finishes, the evaluated cases will be moved into $Stage^4$, but the $RANK$ size will still remain the same. In this illustrated example, we see that *Lazy kNN* evaluated only 800 out of 2,000 cases to find the exact kNN.

Below section presents the pseudo-code of *Lazy kNN*.

3.3 *Lazy kNN* algorithm

The detailed pseudo-code of the components and the core algorithm of *Lazy kNN* are given in Algorithms 3.1, 3.2 and 3.3. Algorithm 3.1 defines the `LazyKNN` class structure, namely, private class attributes, private and public methods; and implements the object constructor `_Construct`. An instance of this class is created for each particular problem sequence. Two other classes `Stage` and `Assessment` are for inner use to track evaluated cases throughout queries for the consecutive problems of the sequence in $RANK$. For each query, including the initial problem, a `Stage` instance is created and inserted into the top of the $RANK$ list. Each `Stage` instance holds the Δ distance to the previous query and is populated with the evaluated candidates during the kNN search in the form of `Assessment` instances. Each `Assessment` instance holds an evaluated case and its calculated similarity to the query for the related problem in the sequence. We note that we use 'zero-based' indexing for the lists in the pseudo-code.

Algorithm 3.2 implements the two public methods which are the means of interaction with the `LazyKNN` object for kNN search. `InitialSearch` is used only for the initial problem of the sequence for which the object was created. To find the kNN, this method delegates the search to the $fn\_initialNNS$ function passed as an argument to the constructor. This can be a linear or any

Algorithm 3.1: Lazy KNN Class

```
1  Class LazyKNN:
       Attributes : $\_CB$: Case Base
                    $\_k$: k of kNN
                    $\_fn\_initialNNS$: kNN search function to be used for the initial problem of the sequence. This
                    function has to return all cases in the $\_CB$ sorted in a descending order with respect to their—exact
                    and/or maximum expected—similarities to the problem
                    $\_fn\_dist$: Distance measure to be used both in the provided $\_fn\_initialNNS$ and consecutive
                    searches by _LazyKNN. It must be a metric
                    $\_RANK$: List of Stage items where the stage for the latest problem of the sequence is the first
                    item with index 0
                    $\_query$: The latest problem in the sequence for which the current kNN search is to be conducted
       Methods    : InitialSearch($query$) : see Algorithm 3.2
                    ConsecutiveSearch($query$) : see Algorithm 3.2
                    _Construct($CB, k, fn\_initialNNS, fn\_dist$) : see below
                    _LazyKNN() : see Algorithm 3.3
2  Class Stage:
       Attributes : $NN$: List of Assessment items, holds evaluated cases for a particular problem in the sequence
                    sorted in a descending order with respect to their similarities to the problem
                    $\Delta$: Distance between the related problem and its predecessor in the sequence; measured by the
                    $\_fn\_dist$
3  Class Assessment:
       Attributes : $case$: Case
                    $similarity$: Similarity of $case$ to the related problem in the sequence that the resided Stage was
                    created for
4  Function _Construct($CB, k, fn\_initialNNS, fn\_dist$):
       Input      : see related class instance attributes above
       Output     : A LazyKNN instance to be used for a particular problem sequence (e.g. history of treatment sessions
                    of a particular patient)
5      $this.\_CB \leftarrow CB$
6      $this.\_k \leftarrow k$
7      $this.\_fn\_initialNNS \leftarrow fn\_initialNNS$
8      $this.\_fn\_dist \leftarrow fn\_dist$
9      $this.\_RANK \leftarrow []$
10     $this.\_query \leftarrow null$
11     return $this$
```

other kNN search of choice. $fn\_initialNNS$ has to return all cases in the CB sorted in a descending order regarding their exact and/or maximum expected similarities to the problem. It does not need to be a complete exact search either. As long as it returns the exact kNN and the upper-bound of similarities of the not-evaluated cases, *Lazy kNN* can perfectly exploit this partial information as well.[2] For the queries of consecutive updates to the sequence, `ConsecutiveSearch` is called. `ConsecutiveSearch` calls the private `_LazyKNN` method which is the core *Lazy kNN* algorithm implemented in Algorithm 3.3. Both `InitialSearch` and `ConsecutiveSearch` return exact kNN of the $query$.

As it can be seen in the Algorithm 3.3, throughout the consecutive problems of the sequence, we delay the similarity assessment of a case in a *lazy* manner until we consider it as a true candidate for the kNN list (line 10 in Algorithm 3.3). Note that a case can become a kNN candidate for more than one problem in a sequence. We also note that *Lazy kNN* sorts the neighbors that could

[2]As an extreme example, if standard k-d tree search is used in $fn\_initialNNS$, discarded cases' similarities can all be assigned to the k^{th} neighbor's similarity. This would result making almost all cases as candidates for the next problem and *Lazy kNN* would perform almost like a linear-search for P^1, but the algorithm would work, and starting from P^2 it would speed up the search as usual.

Algorithm 3.2: Lazy KNN Class - Public Methods

```
1  Function InitialSearch($this$, $query$):
       Input   : Initial problem of a particular problem sequence
       Output  : Exact kNN list
2      $stage^0 \leftarrow$ new Stage
3      $stage^0.NN \leftarrow this.\_fn\_initialNNS(query, this.\_CB, this.\_fn\_dist)$
4      $stage^0.\Delta \leftarrow 0$
5      $this.\_RANK$.insert($stage^0$)
6      $this.\_query \leftarrow query$
7      return $stage^0.NN[: this.\_k]$

8  Function ConsecutiveSearch($this$, $query$):
       Input   : The _query for the new problem in the sequence for which 'this' object was created
       Output  : See output of the _LazyKNN method in Algorithm 3.3
9      $stage^u \leftarrow$ new Stage
10     $stage^u.NN \leftarrow []$
11     $stage^u.\Delta \leftarrow this.\_fn\_dist(query, this.\_query)$
12     $this.\_RANK$.insert($stage^u$)
13     $this.\_query \leftarrow query$
14     return $this.$_LazyKNN()
```

Algorithm 3.3: Lazy KNN - Core algorithm

```
1  Function _LazyKNN($this$):
       Input   : None. No need to pass the query, it is accessed via the instance attribute _query
       Output  : Exact kNN list
2      $stage^u \leftarrow this.\_RANK[0]$
3      $sum\Delta \leftarrow stage^u.\Delta$
4      $sort\_flag \leftarrow False$
5      for $j \leftarrow 1$ to $|this.\_RANK|-1$ do          // Iterate previous Stages in _RANK
6          $stage^j \leftarrow this.\_RANK[j]$
7          foreach $assess$ in $stage^j.NN$ do
8              $case \leftarrow assess.case$
9              $sim \leftarrow assess.similarity$
10             if $|stage^u.NN| < this.\_k$ or $(sim + sum\Delta) > stage^u.NN[this.\_k-1].similarity$ then
11                 $stage^j.NN$.remove($assess$)                // case is candidate
12                 $sim \leftarrow 1 - this.\_fn\_dist(this.\_query, case)$   // Calculate case's sim
13                 $new\_assess \leftarrow$ new Assessment($case, sim$)
14                 if $sim > stage^u.NN[this.\_k-1].similarity$ then
15                     $stage^u.NN$.insert($new\_assess, i$)      // Insert case to kNN, $i<k$
16                 else
17                     $stage^u.NN$.append($new\_assess$)
18                     $sort\_flag \leftarrow True$
19             else                                          // case is not candidate
20                 break                                     // Continue with the next Stage
21         $sum\Delta \leftarrow sum\Delta + stage^j.\Delta$                     // Accumulate $\Delta$s
22     if $sort\_flag$ then $stage^u.NN$.sort_descending()
23     return $stage^u.NN[: this.\_k]$
```

not make it to the kNN list only 'once' at most as mentioned previously (ibid. line 22). The computational cost for the positioning of a winning candidate in the kNN list (ibid. line 15) should be negligible for small k values.

In the formalization of the upper-bound of similarity and in the algorithm presented, for the sake of simplicity, we have assumed that the CB remains unchanged between the updates of a problem sequence P. To deal with the changes to the CB, the algorithm can be updated easily by adding

Table 3.1: **Time series datasets used to generate CBs of temporally related cases** in experiments and their corresponding CB sizes generated with two different time window $step$ settings. Data points are essentially updates to the time sequence. While the time window moves on a sequence, the data points encompassed by the window for each $step$ constitute a case and eventually the cases generated out of a sequence forms a *sequence* of *temporally related cases*.

#	Dataset	Type	Sequences	Length	CB Size	
					$step$=10	$step$=1
1	*PowerCons*	Device	180	144	2,700	25,920
2	*SwedishLeaf*	Image	625	128	8,125	80,000
3	*Strawberry*	Spectrograph	613	235	14,712	144,055
4	*EOGHorizontalSignal*	Medical	362	1,250	45,250	452,500
5	*InsectWingbeatSound*	Sensor	1,980	256	51,480	506,880
6	*ECG5000*	Medical	4,500	140	63,000	630,000
7	*UWaveGestureLibraryX*	Motion	3,582	315	114,624	1,128,330
8	*Yoga*	Image	3,000	426	129,000	1,278,000
9	*Phoneme*	Sound	1,896	1,024	195,288	1,941,504
10	*Mallat*	Simulated	2,345	1,024	241,535	2,401,280
11	*MixedShapesRegularTrain*	Image	2,425	1,024	249,775	2,483,200

following behaviors:

1. If new cases are incorporated into the CB after the kNN search for the last problem P^u, the similarities of these cases to the next problem P^{u+1} have to be calculated in the kNN search for P^{u+1};

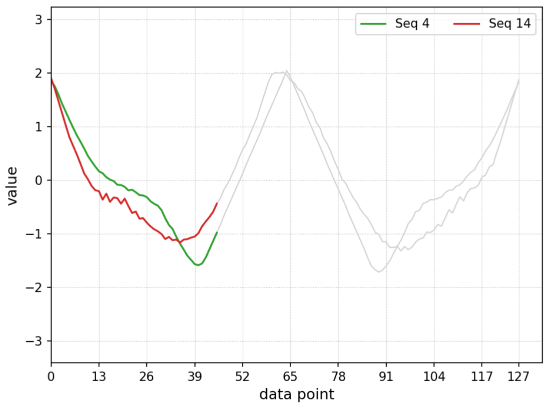
2. If old cases are modified after the kNN search for the last problem P^u and if these modifications affect similarity calculations, these cases should be removed from $RANK$ list and their similarities to the next problem P^{u+1} have to be calculated in the kNN search for P^{u+1};

3. If old cases are deleted, they have to be deleted from $RANK$ list as well.

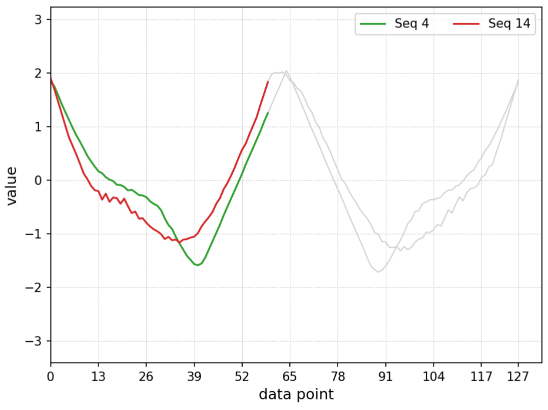
3.4 Experiment datasets

In order have real-world data from different application domains in our experiments, we used eleven univariate time series datasets publicly available at *The UEA & UCR Time Series Classification Repository* (Bagnall et al., 2018). 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. In order to build a larger case base with each dataset, we took the liberty to use the larger sub-dataset in *Lazy kNN* experiments.

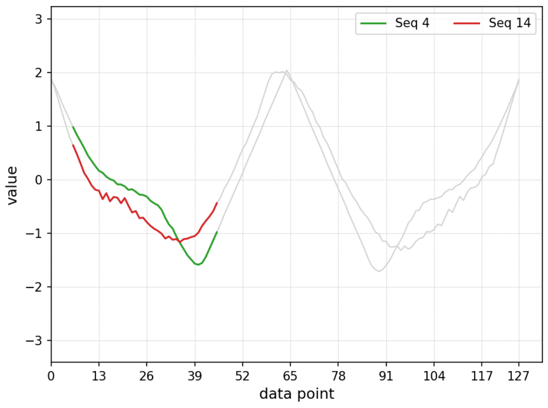
To build a CB out of a given univariate TS dataset, as described in section 2.2, we treated each instance in the dataset as a *sequence* for our CB and each data point of the TS instance as an *update* to the sequence. Then we applied *time window* on every instance to create individual *case*s of the 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



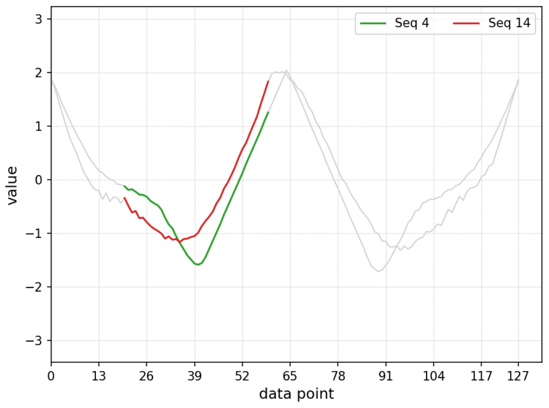
(a) Time window: *Expanding*, $step = 1$, Update: 45



(b) Time window: *Expanding*, $step = 10$, Update: 5



(c) Time window: $width = 40$, $step = 1$, Update: 45



(d) Time window: $width = 40$, $step = 10$, Update: 5

Figure 3.3: **Generating cases out of univariate time series.** The two example time series instances (light gray colored) are taken from the *SwedishLeaf* dataset. Each subsequence (green/red colored) on a time series forms the *problem* part of a unique *case*. Cases from the same time series instance form a unique *sequence* of *temporally related cases*.

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 $step \geq 1$ 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 3.1 summarizes the datasets and their corresponding CBs used in our experiments.

Figure 3.3 shows examples to the generation of cases out of a univariate time series dataset with different time window settings. Each subsequence on a time series forms the *problem* part of a unique *case*. A subsequence is defined by index of the sequence *update* and the time window $width$ and *step* settings. Cases from the same time series instance form a unique *sequence* of *temporally related cases*.

3.5 Evaluation of *Lazy kNN*

The experiments in this section aim to show the speed-up achieved by *Lazy kNN*. In order to have a measure independent of the computation platform that the algorithm works on, we defined our speed-up measure in terms of the number of similarity assessments made in kNN search as follows:

Definition 3.4 (Gain of *Lazy kNN*)**.** *Given P^u—the target problem for the u^{th} update of the sequence P, $NN(Stage^u)$—the list of cases in the CB whose similarities to P^u are calculated, and where $|X|$ denotes the cardinality of a set X, the gain of Lazy kNN at the kNN search conducted for P^u is the percentage of the number of* avoided *similarity calculations by the algorithm compared to a linear search for P^u:*

$$gain(P^u) = \frac{|CB| - |NN(Stage^u)|}{|CB|} \times 100\ \% \tag{3.6}$$

If need be, $gain$ can be translated into actual execution time by using the average duration of a similarity calculation on a particular platform.

In subsection 3.5.1 we describe the experiment settings and finally we present the results in section 3.6.

3.5.1 Experiment Settings

With respect to distance and similarity measures, since *Lazy kNN* relies on true metrics, we used normalized *euclidean distance* and the similarity functions given in Eq. (2.1) and (2.2) respectively in all experiments.[3]

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 straight-forward 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 3.4 we launched four experiments for combinations of two different time window width and step settings. We used time window width $w \in \{Expanding, 40\}$ and time window $step \in \{1, 10\}$. Having k set to 9, each experiment for a configuration 3-tuple of $\big(dataset,\ w,\ step\big)$ was conducted as follows:

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_TEST and the other part as input 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_TEST, we recorded the $gain$ of *Lazy kNN* at the kNN search for each query using the Eq. (3.6). The results are presented in the section below. Note that, the

[3]See (Serrà and Arcos, 2014; Wang et al., 2013) for empirical comparison of similarity measures for time series data in general. Mueen et al. (2009)'s work may be of special interest where they also benefit from the triangle inequality for early abandoning of the costly distance measuring to find exact motifs in time series.

difference between the size of a CB in Table 3.1 and its corresponding CB_TEST in Table 3.2 for a $\big(dataset,\ step\big)$ setting is the number of input queries used in that experiment that are extracted from the former CB.

3.6 Results

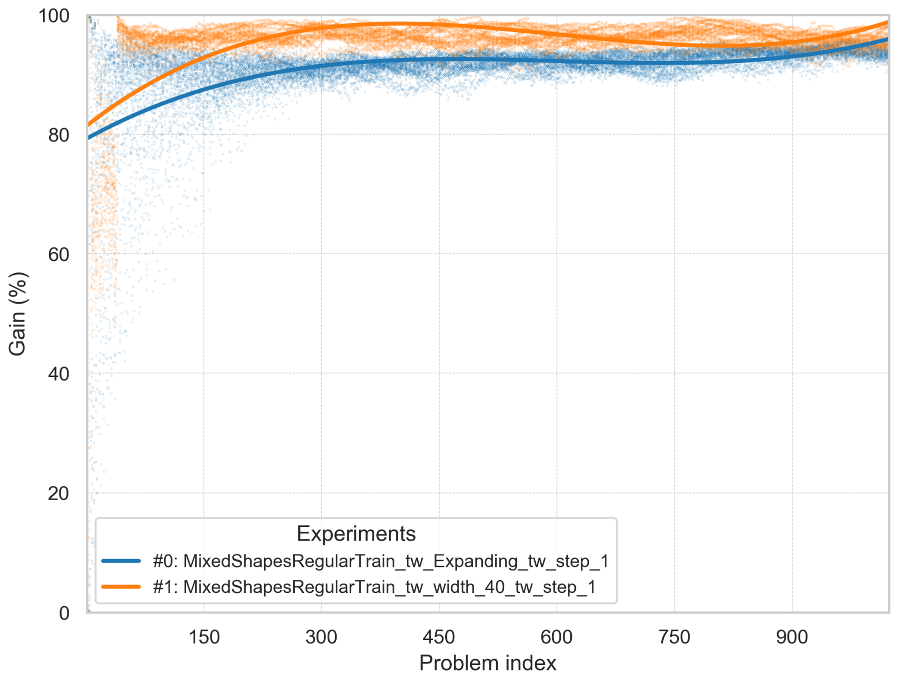
Table 3.2 lists the average gain of *Lazy kNN* for each experiment conducted with a specific $\big(dataset,\ w,\ step\big)$ setting. Average gain for an experiment is calculated throughout the gains in kNN searches for all input queries (i.e. for all problems of all input sequences). The results show that *Lazy kNN* achieves notable to outstanding speed-up. Average $gain$s lie in the range of $\big[33.63\%, 96.65\%\big]$ for exact kNN search in CBs of different characteristics and sizes from 2,430 to 2,457,600 cases. And more importantly, we see that *Lazy kNN* yields a higher gain when the generated CB is larger for each dataset. For example, for the $\big($*MixedShapesRegularTrain*, $w = 40$, $step = 1\big)$ setting, the average gain is 95.53%. This means that on average 2,347,745 cases out of a CB of size 2,457,600 were *ignored* by *Lazy kNN* per query since they proved not to be kNN candidates. Thanks to the effective *candidacy assessment* method of the algorithm on $RANK$ data structure, it is possible to discard higher proportions of cases in kNN search in large CBs where sequences of temporally cases are longer. And of course, this is a much desired result as the gain in a large CB is more significant regarding actual execution time.

Figures 3.4a and 3.4b give a more in-depth view of the gain of *Lazy kNN* for experiments with the *MixedShapesRegularTrain* dataset as an example. The plots reflect four different combinations of time window settings as described in subsection 3.5.1. The difference between two figures is the time window step setting used by the experiments which defines the number of problems in a sequence, and thus the CB size. Time window $step = 1$ setting in Figure 3.4a yields more problems per sequence, and hence, a larger CB compared to the smaller CB generated with $step = 10$ setting used in Figure 3.4b. See section 3.4 for case and CB generation details and Table 3.1 for CB sizes.

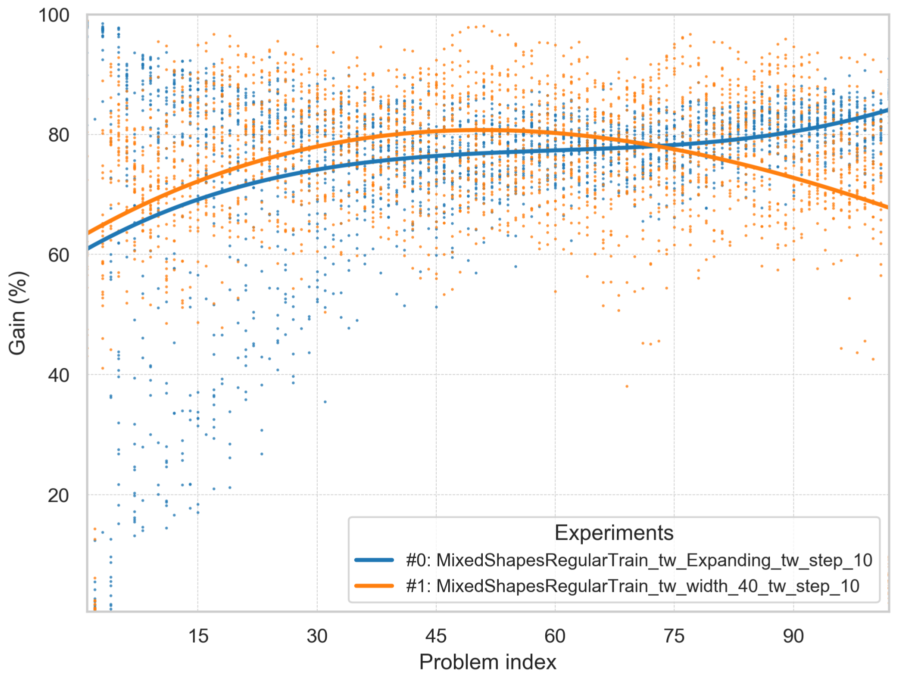
These two figures illustrate the phenomenon of "higher gain in larger CB" that we described above. *Lazy kNN*'s gain is much higher on average in the larger CBs of Figure 3.4a than the ≈ 10 times smaller CBs of the Figure 3.4b. Second observation is the higher variance of gain in Figure 3.4b compared to Figure 3.4a. In Figure 3.4b, $step = 10$ setting introduces more change between two consecutive problems of a sequence (i.e. Δs are greater). And depending on the input sequence, this may cause the gain to fluctuate in a wider range. The variance is more pronounced for the experiment with $(w = 40, step = 10)$ in the same figure where a consecutive problem P^u has 25% different feature values (i.e. 10 in 40) than P^{u-1}.

Table 3.2: **Average gain of *Lazy kNN***. The table summarizes average gains of the algorithm in terms of % of avoided similarity assessments compared to a brute-force search throughout all problems of input sequences.

	Time window		Problem Updates	|CB_TEST|	Gain (%)
Dataset	w	$step$			
PowerCons	Expanding	1	144	23,328	75.90
		10	15	2,430	48.39
	40	1	144	23,328	70.67
		10	15	2,430	34.64
SwedishLeaf	Expanding	1	128	71,936	76.05
		10	13	7,306	48.22
	40	1	128	71,936	80.62
		10	13	7,306	37.38
Strawberry	Expanding	1	235	129,485	81.69
		10	24	13,224	56.76
	40	1	235	129,485	89.21
		10	24	13,224	58.65
EOGHorizontalSignal	Expanding	1	1,250	440,000	91.18
		10	125	44,000	76.01
	40	1	1,250	440,000	96.65
		10	125	44,000	87.24
InsectWingbeatSound	Expanding	1	256	501,760	82.88
		10	26	50,960	59.27
	40	1	256	501,760	85.17
		10	26	50,960	41.51
ECG5000	Expanding	1	140	623,700	77.13
		10	14	62,370	50.00
	40	1	140	623,700	80.99
		10	14	62,370	50.80
UWaveGestureLibraryX	Expanding	1	315	1,116,990	84.31
		10	32	113,472	62.22
	40	1	315	1,116,990	91.66
		10	32	113,472	65.74
Yoga	Expanding	1	426	1,266,498	87.16
		10	43	127,839	68.12
	40	1	426	1,266,498	92.82
		10	43	127,839	64.22
Phoneme	Expanding	1	1,024	1,922,048	89.06
		10	103	193,331	71.96
	40	1	1,024	1,922,048	63.29
		10	103	193,331	33.63
Mallat	Expanding	1	1,024	2,390,016	90.43
		10	103	240,402	75.14
	40	1	1,024	2,390,016	94.01
		10	103	240,402	68.90
MixedShapesRegularTrain	Expanding	1	1,024	2,457,600	90.88
		10	103	247,200	75.42
	40	1	1,024	2,457,600	95.53
		10	103	247,200	76.72



(a) k=9, time window $width \in [Expanding, 40]$, $step$=1 where $|CB| = 2,483,200$.



(b) k=9, time window $width \in [Expanding, 40]$, $step$=10 where $|CB| = 249,775$.

Figure 3.4: **Gain of *Lazy kNN*** for experiments with the *MixedShapesRegularTrain* dataset

Both in Figure 3.4a and 3.4b, we also see that the gain of the algorithm is lower and it has more variance during the first problems in a sequence. And as the number of consecutive problems in a sequence increases, the gain converges into a narrower band around a higher average gain. This behaviour is due to the increase in shared history between consecutive problems in a sequence. The resemblance of histories is not only important for two successive problems, but it may also transcend further back than the previous problem. In other words, as the sequence grows longer with more problems, i.e. for $u \gg 0$, kNN of P^u are more likely to be found within the evaluated cases for P^{u-2} than within near neighbors of P^0.

Another interesting pattern is the spiking convergence of gain for the experiment with $w = 40$ setting (i.e. orange colored) in Figure 3.4a. This pattern occurs due to our similarity function described in subsection 3.5.1. Until the 39^{th} consecutive problem, the target problem has less than 40 features, and during similarity calculations, the missing features are filled with values so as to maximize the distance to the evaluated case. It is after 40^{th} problem that the target problem fully benefits from the shared history with its predecessors, and target problem's similarity is calculated "as is" to the candidate cases.

Finally, we can say that no matter what time window settings are, *Lazy kNN* provides a remarkable speed-up in kNN search.[4] The speed-up reaches to significant factors when the problem sequences have more updates and the CB's are larger. Thus, the results of these experiments endorse the use of *Lazy kNN* on large-scale temporal CBs, this being the very motivation of its design.

3.7 Summary

In this chapter, we introduced an *exact* kNN search algorithm *Lazy kNN* that excels in application domains with large-scale temporal case bases. We described how we can exploit the triangle inequality to avoid unnecessary similarity assessments in metric spaces which is the common problem space of most of CBR systems. Then we presented our proposed algorithm. Later in experiments, we showed how we generated small-to-large temporal CBs out of publicly available real-world time series datasets of different domains and characteristics. And we presented and commented on the experiment results with these CBs.

The results empirically demonstrated that the proposed algorithm reaches notable to remarkable speed-up in kNN search. The average speed-up was higher for the larger CB compared to the smaller CB, both generated out of the same dataset. These results add to the merit of using *Lazy kNN* in large-scale CBs which was the main focus of the algorithm's design.

Lazy kNN is an exact kNN search algorithm. And for certain domains and/or applications where response time is more critical, it is possible that even the speed-up provided by the algorithm may not meet time limitations. Or, for some domains where good enough neighbors would also suffice, we may even not be interested in finding exact neighbors anyways. On the other hand, there may be applications that we cannot trust the accuracy of our distance measure and/or the representation of data. Therefore, exact kNN would not make much sense. In either of these cases, we can conform with *approximate* kNN instead of exact ones as it is the common practice in many kNN search implementations described earlier in Chapter 2.

[4]As a theoretical experiment, *Lazy kNN* would work as a linear search with no speed-up only if $w = step$. This would mean that a problem sequence has no shared history with its predecessor and each problem is totally a different case. But then, this would not be a temporal case base for which *Lazy kNN* is designed.

We deal with both exact and approximate kNN search by introducing an *anytime* kNN algorithm in the next chapter. After discussing the challenges in building a well-tempered anytime kNN search algorithm, we point out why we deem *Lazy kNN* a very good candidate to be converted into an anytime algorithm which overcomes these difficulties. Then, we explain how we endow *Lazy kNN* with the capability to provide *best-so-far kNN* with *confidence* any time it is interrupted, and ultimately how we achieve our fully-fledged *Anytime Lazy kNN* search algorithm.

Chapter 4

Anytime Lazy kNN

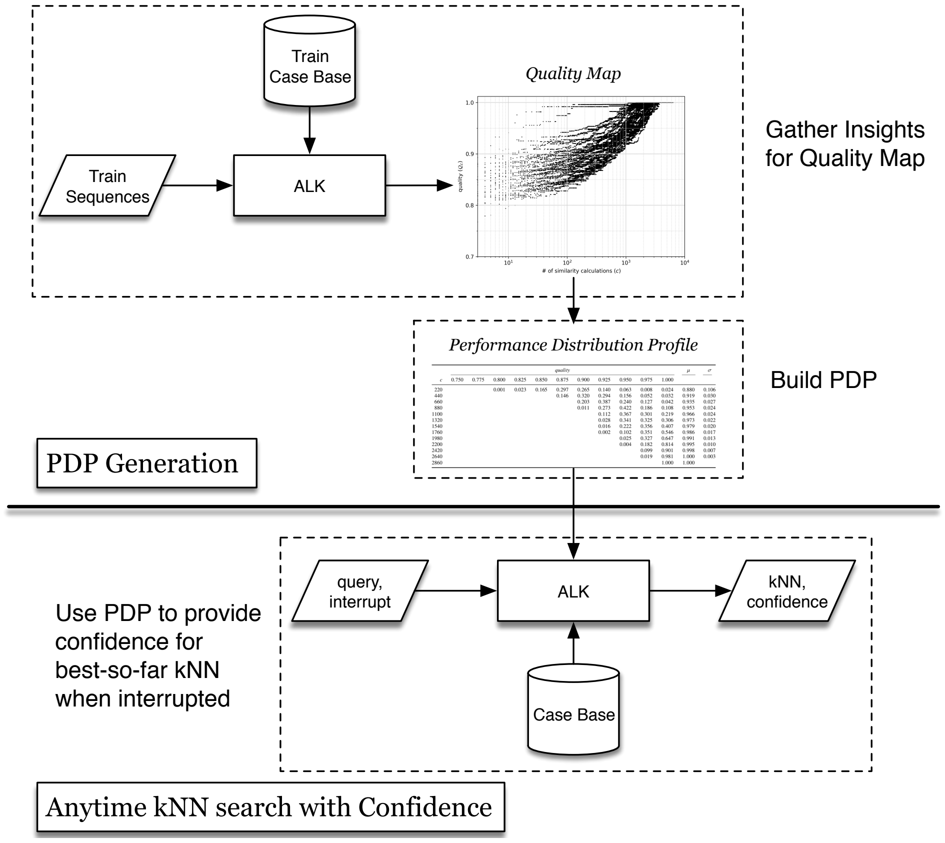
In accordance with the description of a preferable anytime algorithm given in section 2.3, an anytime kNN search algorithm is expected to monotonically improve on its k nearest neighbors and provide a quality value attached to the best-so-far neighbors upon interruption. Also the improvement in output 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 overhead if it is allocated extra computation time to improve on its results. But, it is not straightforward to build such a well-tempered anytime kNN search algorithm.

The main difficulty of converting an *exact* kNN search to an anytime algorithm lies in the quality assessment of the *best-so-far* neighbors. The quality in AAs is a notion of closeness to exact solutions. Therefore, having the search interrupted, we would like to compare the similarities of approximate and exact kNN to the query. However, it is impossible to build an accurate quality measure for such an assessment. The reason is obvious, exact kNN 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 kNN all at once (e.g. brute-force kNN search), exact kNN 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 $i-1$ nearest neighbors (NNs) when interrupted at the i^{th} 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 kNN 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 chapter, we first deal with the design of an anytime kNN algorithm that exhibits the desired performance profile outlined in section 2.3. And, we show how we refactor *Lazy kNN* to serve as the core of the proposed AA. Specifically, although *Lazy kNN* was designed to provide exact kNN as a whole list after running to completion, we show how it can be converted to an incremental kNN search without any redundant similarity assessments in section 4.1. Then, we explain how *Incremental Lazy kNN* can be used in the core of an AA and present the *Anytime Lazy kNN* (henceforth *ALK*) algorithm in section 4.2.

Once we have the AA algorithm, we introduce the steps required to define the appropriate mechanism to assess the quality of best-so-far kNN 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 (Zilberstein, 1996). As a road map, Figure 4.1 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 *Performance Distribution Profile* (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 4.3. In the second step which is described in section 4.4, PDP is built out of the quality map. PDP endows our algorithm with the ability to predict the *expected quality* of best-so-far kNN upon interruption. In concordance with CBR literature (e.g. Cheetham, 2000), we refer to expected output quality as *confidence* and define our confidence estimator based on PDP in section 4.5. We also show how PDP helps us to automatize the interruption upon reaching a given *confidence threshold*. In the final step of Figure 4.1, *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 kNN together with a confidence value for each neighbor.



Train Case Base
Quality Map
Train Sequences
ALK
quality (q_c)
1.0
0.9
0.8
0.7
10^1
10^2
10^3
10^4
# of similarity calculations (c)
Gather Insights for Quality Map
Performance Distribution Profile
Build PDP
PDP Generation
query, interrupt
ALK
kNN, confidence
Case Base
Use PDP to provide confidence for best-so-far kNN when interrupted
Anytime kNN search with Confidence

	quality											*μ*	*σ*
c	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	

Figure 4.1: **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 kNN when interrupted. PDP also provides a means to automatize interruption at given confidence thresholds.

Section 4.6 describes our methodology to evaluate *ALK* and the experiment settings. Section 4.7 gives highly encouraging results of experiments we conducted with the same real-world time series datasets used in *Lazy kNN* experiments in the previous chapter. We see that the speed-up can be dramatically increased by *ALK* even for interruptions at high confidence thresholds. Section 4.8 presents the insights of this superior gain in execution time achieved by *ALK*. Finally in section 4.9, we present alternative ways to search for kNN candidates which may further boost *ALK*'s performance.

4.1 *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, the exact kNN list is provided as output only after the evaluation of the last candidate. 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 kNN when interrupted.

To gradually improve on each of the kNN and to be able to provide at least some of the exact NNs upon interruption, we developed an incremental version of *Lazy kNN*. The new version can basically be regarded as invoking the original algorithm k times iteratively, while at each iteration i, we find the i^{th} exact NN. Thus, if the algorithm is interrupted during the i^{th} iteration, it ensures that the top i-1 NNs are the exact NNs of the query.

The beauty of the conversion of the algorithm 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 behaviour is due to the fact that we 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 *Lazy kNN*. On the other hand, if *Lazy kNN* is invoked k times 'as is' without any alteration in the algorithm, this would cause k times sorting of the assessed cases and would become an extra overload prolonging the execution time. But, in the following section we will show that sorting can be reduced to only once (at most) easily by a little tweak in the handling of the evaluated candidates.

With respect to the desired *monotonicity* property of an AA mentioned in section 2.3, we can argue that exact kNN search in general exhibits monotonicity. Because, 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. That is, any kNN 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 kNN becoming the exact ones.

We note that the incremental nature of our algorithm is analogous to Broder (1990)'s incremental nearest neighbor search which also finds kNN iteratively starting from the first, ending with the

k^{th}; but it differs from the incremental retrieval concept of Cunningham, Smyth, and Bonzano (1998) and Jurisica, Glasgow, and Mylopoulos (2000) because there, the iteration takes place in conversational CBR systems where the retrieval is incrementally refined via iterative user interactions.

The section below gives our *Anytime Lazy kNN* search algorithm with *Incremental Lazy kNN* at its core.

4.2 *ALK* algorithm

Besides monotonicity, another desired property of AAs is *preemptability* (Zilberstein, 1996), 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 kNN search where the interruption has occurred: (1) index of the current iteration and, (2) index of the next candidate in $RANK$ to be assessed.

The detailed pseudo-code of the components of *ALK* are given in Algorithms 4.1, 4.2 and 4.3. As can be seen, these algorithms are extensions to the Algorithms 3.1, 3.2 and 3.3 presented for *Lazy kNN* in section 3.3 respectively. We give all shared components (e.g. `Stage` and `Assessment` classes) once more here for completeness' sake.

Algorithm 4.1 defines `AnytimeLazyKNN` class structure, namely, private class attributes, private and public methods; and implements the object constructor `_Construct`. An instance of this class has to be created for each particular problem sequence. Two other classes `Stage` and `Assessment` are for inner use to track evaluated cases throughout the queries for consecutive problems of the sequence in $RANK$.

Algorithm 4.2 implements the three public methods which are the means of interaction with the object. `InitialSearch` is used only for the initial problem of the sequence for which the object was created. To find the kNN, this method utilizes the $fn\_initialNNS$ function passed as an argument to the constructor. For the queries of consecutive updates to the sequence, `ConsecutiveSearch` is called with an optional $interrupt$ argument that is the number of similarity assessments after which we want to interrupt the algorithm. If the algorithm is wished to be resumed, `ResumeLastSearch` is called, again with an optional $interrupt$ argument. Both `ConsecutiveSearch` and `ResumeLastSearch` call the private `_IncrementalLazyKNN` method and return its output.

`_IncrementalLazyKNN`, the core of *ALK*, is implemented in Algorithm 4.3. It is the incremental, interruptible and resumable implementation of *Lazy kNN*. Compared to the core Algorithm 3.3 of *Lazy kNN* it can be seen that $RANK$ is iterated k times in Algorithm 4.3. And, at an $iter^{th}$ iteration , the upper-bound of similarity of a case in $RANK$ has to beat the $iter^{th}$ NN found so far to be a candidate (line 13) whereas the candidacy assessment in *Lazy kNN* was for the k^{th} NN (line 10 in Algorithm 3.3). We remind that 'zero-based' indexing is used for the lists in the pseudo-code. If a $case$ is deemed a candidate, its actual similarity to the target problem is calculated. After each similarity calculation, the counter $calc$ is incremented and is checked against the optional interruption point passed with the $interrupt$ (line 23).

If the calculated similarity of a candidate actually wins over the $iter^{th}$ NN, the winner replaces the current NN. As mentioned in the above section, to avoid the extra overload of k times sorting

Algorithm 4.1: Anytime Lazy KNN Class

```
1  Class AnytimeLazyKNN:
       Attributes : $\_CB$: Case Base
                    $\_k$: k of kNN
                    $\_fn\_initialNNS$: kNN search function to be used for the initial problem of the sequence. This
                    function has to return all cases in the $\_CB$ sorted in a descending order with respect to their—exact
                    and/or maximum expected—similarities to the problem
                    $\_fn\_dist$: Distance measure to be used both in the provided $\_fn\_initialNNS$ and consecutive
                    searches by _IncrementalLazyKNN. It must be a metric
                    $\_RANK$: List of Stage items where the stage for the latest problem of the sequence is the first
                    item with index 0
                    $\_current\_iter$: Iteration index to start/resume the _IncrementalLazyKNN from
                    $\_current\_index$: Index to the candidate assessment within $\_RANK$ to start/resume the
                    _IncrementalLazyKNN from
                    $\_query$: The latest problem in the sequence for which the current kNN search is to be conducted.
                    This holds either a new problem or the latest one when an interruption occurred in its kNN search
       Methods    : InitialSearch(query) : see Algorithm 4.2
                    ConsecutiveSearch(query, interrupt) : see Algorithm 4.2
                    ResumeLastSearch(interrupt) : see Algorithm 4.2
                    _Construct(CB, k, fn_initialNNS, fn_dist) : see below
                    _IncrementalLazyKNN(interrupt) : see Algorithm 4.3
2  Class Stage:
       Attributes : $NN$: List of Assessment items, holds evaluated cases for a particular problem in the sequence
                    sorted in a descending order with respect to their similarities to the problem
                    $\Delta$: Distance between the related problem and its predecessor in the sequence; measured by the
                    $fn\_dist$
3  Class Assessment:
       Attributes : case: Case
                    similarity: Similarity of case to the related problem in the sequence that the resided Stage was
                    created for
4  Function _Construct(CB, k, fn_initialNNS, fn_dist):
       Input      : see related class instance attributes above
       Output     : An AnytimeLazyKNN instance to be used for a particular problem sequence (e.g. history of
                    treatment sessions of a particular patient)
5      $this.\_CB \leftarrow CB$
6      $this.\_k \leftarrow k$
7      $this.\_fn\_initialNNS \leftarrow fn\_initialNNS$
8      $this.\_fn\_dist \leftarrow fn\_dist$
9      $this.\_RANK \leftarrow []$
10     $this.\_current\_iter \leftarrow 1$
11     $this.\_current\_index \leftarrow 1$
12     $this.\_query \leftarrow null$
13     return $this$
```

of all assessed cases in k iterations, we always maintain the best-so-far kNN in order, as we did so in *Lazy kNN*. And in order to keep kNN ordered at all times, when a candidate's actual similarity cannot beat the $iter^{th}$ NN, we check if it beats any of the remaining members of the kNN. And if it beats any, we insert the case at its rank in kNN. On the other hand, if the candidate cannot beat any of the kNN, it is simply appended to Stage.NN. Therefore, just like in *Lazy kNN*, throughout kNN search, the sorting of all assessed candidates that could not make it to the kNN list is carried out 'once' at most (line 32). The computational cost for the positioning of a winning candidate in the kNN list (line 19) should be negligible for small k values.

_IncrementalLazyKNN returns the kNN list and the confidence of the algorithm for each member of the list. If the algorithm is run to completion, the kNN will be exact and their confidence values will be 1. If the algorithm is interrupted, best-so-far kNN list is returned together with

Algorithm 4.2: Anytime Lazy KNN Class - Public Methods

```
1  Function InitialSearch(this, query):
       Input  : Initial problem of a particular problem sequence
       Output : Exact kNN list and their associated confidence values (i.e. 1)
2      $stage^0 \leftarrow$ new Stage
3      $stage^0.NN \leftarrow this.\_fn\_initialNNS(query, this.\_CB, this.\_fn\_dist)$
4      $stage^0.\Delta \leftarrow 0$
5      $this.\_RANK$.insert($stage^0$)
6      $this.\_query \leftarrow query$
7      return $stage^0.NN[: this.\_k]$, $[1] * this.\_k$
8  Function ConsecutiveSearch(this, query, interrupt =null):
       Input  : The _query for the new problem in the sequence for which 'this' object was created and an optional
                interruption point for similarity calculations
       Output : See output of the _IncrementalLazyKNN method in Algorithm 4.3
9      $stage^u \leftarrow$ new Stage
10     $stage^u.NN \leftarrow []$
11     $stage^u.\Delta \leftarrow this.\_fn\_dist(query, this.\_query)$
12     $this.\_RANK$.insert($stage^u$)
13     $this.\_current\_iter \leftarrow 1$
14     $this.\_current\_index \leftarrow 1$
15     $this.\_query \leftarrow query$
16     return $this.$_IncrementalLazyKNN(interrupt)
17 Function ResumeLastSearch(this, interrupt =null):
       Input  : Optional new interruption point for similarity calculations
       Output : See output of the _IncrementalLazyKNN method in Algorithm 4.3
18     return $this.$_IncrementalLazyKNN(interrupt)
```

the expected quality values for each member of the list provided by the confidenceKNN system function. confidenceKNN is based on the PDP of the algorithm generated for the application domain and time window settings.

Generation of the Quality Map, PDP and confidence will be described in the following sections 4.3, 4.4 and 4.5 respectively. The complete code of *ALK* including all its functionality that will be covered throughout the rest of the dissertation is publicly available at the online repository: https://github.com/IIIA-ML/alk

4.3 Quality measure and quality 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 could define the quality measure for *ALK* as follows:

$$\mathcal{Q}_c = \frac{sim(NN^k_c, query)}{sim(NN^k_E, query)} \tag{4.1}$$

where sim is the similarity metric, NN^k_c and NN^k_E are the best-so-far and exact k^{th} NN respectively, $\mathcal{Q}_c$ gives the output quality as the ratio of their similarities to the $query$ after c number

Algorithm 4.3: Anytime Lazy KNN - Core algorithm

```
Function _IncrementalLazyKNN(this, interrupt =null):
    w
Input  : Optional interruption point for similarity calculations. No need to pass the query, it is accessed via the
         instance attribute $\_query$
Output : kNN list and the list of associated confidence for each neighbor. If the algorithm is run to completion,
         kNN are exact and their confidence values are 1; otherwise, at least some of the kNN are approximate and
         the confidence values are given by the system-provided confidenceKNN function $\in [0, 1]$
$stage^u \leftarrow this.\_RANK[0]$
$sort\_flag \leftarrow False$
$calc \leftarrow 0$
for $iter \leftarrow this.\_current\_iter$ to $this.\_k$ do                      // Iterate $\_RANK$ $\_k$ times
    $sum\Delta \leftarrow \sum_{j=0}^{this.\_current\_index-1} this.\_RANK[j].\Delta$
    for $j \leftarrow this.\_current\_index$ to $|this.\_RANK|-1$ do                 // Iterate Stages in $\_RANK$
        $stage^j \leftarrow this.\_RANK[j]$
        foreach $assess$ in $stage^j.NN$ do
            $case \leftarrow assess.case$
            $sim \leftarrow assess.similarity$
            if $|stage^u.NN| < iter$ or $(sim + sum\Delta) > stage^u.NN[iter-1].similarity$ then
                $stage^j.NN$.remove($assess$)                                  // $case$ is candidate
                $sim \leftarrow 1 - this.\_fn\_dist(this.\_query, case)$                // Calc $case$'s sim
                $calc \leftarrow calc + 1$
                $new\_assess \leftarrow$ new Assessment($case, sim$)
                if $sim > stage^u.NN[this.\_k-1].similarity$ then
                    $stage^u.NN$.insert($new\_assess$, $i$)     // Insert $case$ to kNN, $iter-1 \leq i < k$
                else
                    $stage^u.NN$.append($new\_assess$)
                    $sort\_flag \leftarrow True$
                if $calc = interrupt$ then                                        // Interrupt?
                    $this.\_current\_iter \leftarrow iter$
                    $this.\_current\_index \leftarrow j$
                    if $sort\_flag$ then $stage^u.NN[iter+1:]$.sort_descending()
                    return $stage^u.NN[:this.\_k]$, confidenceKNN($u, this.\_k, calc$)
            else                                                            // $case$ is not candidate
                break                                          // Continue with the next Stage
        $sum\Delta \leftarrow sum\Delta + \Delta^j$                                       // Accumulate $\Delta$s
    $this.\_current\_index \leftarrow 1$
if $sort\_flag$ then $stage^u.NN[this.\_k:]$.sort_descending()
return $stage^u.NN[:this.\_k]$, $[1] * this.\_k$
```

of similarity assessments. If need be, c can be translated into actual execution time by using the average duration of a similarity calculation on the platform *ALK* is running on.[1]

However, a quality map generated with this measure would not reflect neither the incremental nature of *ALK* nor the temporal relations between cases. 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 kNN 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 problem in the sequence for which the query is generated for. This dimension provides even a finer-grained quality map, because, the more sequence updates are covered by the time window, the less difference will have been introduced by the new problem. Therefore, the more similar will be two successive queries and intuitively, the more similar will be their neighbors. Consequently, for a new query covering

[1]Just like the $gain$ of the algorithm can be translated to execution time as described in definition 3.4.

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 kNN 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 kNN 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:

Definition 4.1 (Quality of a best-so-far NN)**.** *When ALK is interrupted, the output quality of the best-so-far i^{th} NN is the ratio of the similarities of that NN and of the i^{th} exact NN to the query. Formally, where P^u is the u^{th} query of a problem sequence P, $NN_c^{u,i}$ is the i^{th} NN returned by the algorithm when it is interrupted after c similarity calculations during the kNN search for P^u, and $NN_E^{u,i}$ is the exact i^{th} NN for the same query, the quality of $NN_c^{u,i}$ is:*

$$\mathcal{Q}_c^{u,i} = \frac{sim(NN_c^{u,i}, P^u)}{sim(NN_E^{u,i}, P^u)} \tag{4.2}$$

Now after having the finer-grained quality measure, we can formally define the quality map of *ALK* as follows:

Definition 4.2 (Quality map of *ALK*)**.** *The quality map of ALK on a particular case base is the set of $(u, i, c, \mathcal{Q}_c^{u,i})$ 4-tuples generated out of all problems of input sequences used in kNN search simulations on that case base; where u is the problem index in a sequence, i is the kNN member index, c is the number of similarity assessments made for a query.*

Given representative input sequences for a CB, the quality map of *ALK* for that particular CB can be generated as follows. When an input query for a problem in a sequence is passed to the *ALK*'s core algorithm (Algorithm 4.3), the number of the $Stage$s in $RANK$ created for that sequence gives us the problem index u of the query. After each similarity assessment (Algorithm 4.3, line 16), we record the $(u, i, calc, sim(NN_c^{u,i}, P^u))$ 4-tuple for each of the best-so-far kNN members. Note that we record this data after every calculation regardless of whether or not the evaluated candidate alters the best-so-far kNN list. When the simulation ends for that query, the exact kNN and their similarities (i.e. the divisor in Eq. (4.2)) are obtained. Then, by backtracking the simulation, the quality $\mathcal{Q}_c^{u,i}$ values for each 4-tuple are calculated using the Eq. (4.2). Eventually, all $(u, i, c, \mathcal{Q}_c^{u,i})$ 4-tuples collected during simulations provide us with the $QualityMap$ of our algorithm for this particular CB and time window settings.

Figure 4.2 shows an example to the quality map of *ALK* generated by using the quality measure in Eq. (4.2). It is generated throughout simulations with input sequences taken from the *SwedishLeaf* dataset (see subsection 4.6.2). The figure is a 2D excerpt of the 4D map plotted for the third nearest neighbors of the queries generated for the tenth updates (i.e. $\mathcal{Q}_c^{10,3}$) of input sequences.

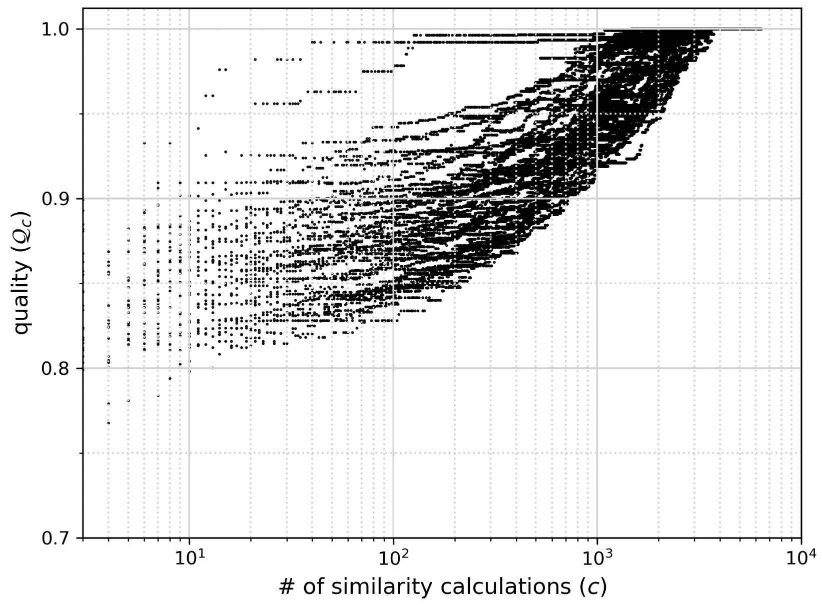


Figure 4.2: **Quality map** of *ALK* for the *SwedishLeaf* dataset. Here the map is shown for the 10^{th} consecutive problem and the 3^{rd} nearest neighbor. Every point on the map is an instance of output quality observed throughout simulations.

4.4 Performance distribution profile

As the final step to have *Anytime Lazy kNN* fully functional as an anytime algorithm, we define its performance distribution profile:

Definition 4.3 (Performance distribution profile of *ALK*)**.** *The performance distribution profile (PDP) of ALK on a particular case base is the discrete probability distribution of quality of best-so-far kNN over number of similarity assessments. PDP is essentially a four-dimensional array and is built out of the quality map of ALK for that particular case base. The dimensions are problem and kNN member indices, the numbers of calculation and quality intervals. Each entry is the probability value of having an output quality within the given quality range.*

In order to build the PDP, 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 to generate the $QualityMap$. Then we discretize the quality range $\left[0, 1\right]$ into n discrete quality values $q_1, \ldots, q_n$ of equal intervals of δ_q.

In coherence with the $QualityMap$ defined above, we create a four-dimensional array $\mathcal{PDP}$ to hold the discreet probability distribution of quality. The dimensions of $\mathcal{PDP}$ are the maximum problem index in input sequences during simulations, k of kNN, the number of calculation intervals m, and the number of quality intervals n respectively.

And we populate $\mathcal{PDP}$ out of *Quality Map* in such a way that the entry $\mathcal{PDP}\left[u, i, r, v\right]$ represents the discrete probability of the output quality of the best-so-far i^{th} NN to be $\in \left(q_v - \delta_q, q_v\right]$ after a number of similarity calculations $\in \left(c_r - \delta_c, c_r\right]$ made during the kNN search for the u^{th} problem in a sequence. The values of system parameters δ_c and δ_q, hence the size of the $\mathcal{PDP}$ can be

Table 4.1: **Performance distribution profile with confidence**. The $\mathcal{PDP}$ that corresponds to the Quality Map in Fig. 4.2 is given. The $\mathcal{PDP}$ is essentially a 4-dimensional array; here the table for 10^{th} consecutive problem and the 3^{rd} nearest neighbor 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 $\mathcal{PDP}$.

	quality											μ	σ
c	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	

adjusted with respect to the desired accuracy of the quality estimation. 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 Figure 4.2 is given in Table 4.1. For example, the entry at $c = 800$, $quality = 0.925$ says that after $\big(660, 800\big]$ number of similarity assessments made for the 10^{th} consecutive problem in a sequence, the probability that the quality of the 3^{rd} NN lies in $\big(0.9, 0.925\big]$ is 0.273.

4.5 Confidence

Equipped with PDP, our algorithm becomes ready to predict the output quality of best-so-far kNN when the search for an unseen query is interrupted. Henceforth, we refer to the expected output quality as the *confidence* of our algorithm in conformity with the CBR literature (e.g. Cheetham, 2000). And we define it as follows:

Definition 4.4 (Confidence for a best-so-far NN)**.** *The confidence of ALK for a member of the best-so-far kNN is the expected quality for that NN. When ALK is interrupted after c number of similarity assessments, confidence (μ) is the weighted mean of the probability distribution of quality for the related calculation interval in $\mathcal{PDP}$ for that NN. When c does not coincide exactly with the calculation intervals, linear interpolation is used.*

Formally, the confidence $\overset{u,i}{\underset{c}{\mu}} \in \big[0, 1\big]$ of the best-so-far i^{th} NN of the u^{th} consecutive problem in a sequence when *ALK* is interrupted after c number of similarity assessments during kNN search is:

$$\overset{u,i}{\underset{c}{\mu}} = \sum_v q_v \, \overset{u,i}{\underset{c}{\mathcal{P}}}(q_v) \tag{4.3}$$

where $\overset{u,i}{\underset{c}{\mathcal{P}}}(q_v)$ is a shorthand for $\mathcal{PDP}\big[u, i, r_c, v\big]$ and r_c is the interval in the calculation range where c falls into.

Since confidence gives us a mean value μ, we also provide its standard deviation σ to be used together with it:

Definition 4.5 (Standard deviation of confidence for a best-so-far NN)**.** *When ALK is interrupted after c number of similarity assessments, standard deviation of confidence (σ) for a member of the best-so-far kNN is the weighted standard deviation of the probability distribution of quality for the related calculation interval in $\mathcal{PDP}$ for that NN. When c does not coincide exactly with the calculation intervals, linear interpolation is used.*

Formally, the standard deviation of confidence $\overset{u,i}{\underset{c}{\sigma}}$ of the best-so-far i^{th} NN of the u^{th} consecutive problem in a sequence when *ALK* is interrupted after c number of similarity assessments during kNN search is:

$$\overset{u,i}{\underset{c}{\sigma}} = \sqrt{\sum_v \Big(q_v - \overset{u,i}{\underset{c}{\mu}}\Big)^2 \overset{u,i}{\underset{c}{\mathcal{P}}}(q_v)} \tag{4.4}$$

The confidence μ and its deviation σ for the quality map example in Figure 4.2 are given in Table 4.1. 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 kNN, 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 by the average calculation time on the computer system that 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.

Interruption point for a confidence threshold can be selected by also taking into account the standard deviation of confidence provided by PDP along with the confidence itself. Accordingly, the function defined below can be used to determine the number of similarity assessments needed to reach a threshold.

Definition 4.6 (Interruption point for a confidence threshold)**.** *Given the problem index u of a query, a confidence threshold τ for interruption, the index i of the kNN member to check against the threshold, and a z offset parameter for the standard deviation of confidence; the $Interrupt$ function defined below gives the number of similarity assessments needed for the i^{th} NN of the u^{th} problem in a sequence to reach the desired threshold τ:*

$$Interrupt(u, i, \tau, z) = \big\{c \colon \overset{u,i}{\underset{c}{\mu}} + z\overset{u,i}{\underset{c}{\sigma}} = \tau\big\}$$

Passing the return value of $Interrupt$ function as the $interrupt$ parameter to *ALK*'s core algorithm in Algorithm 4.3 would automatize the interruption of the algorithm upon reaching the confidence threshold τ. The choosing of the z offset parameter, i.e. 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 $\mathcal{PDP}$ respectively. $z = -1$ would be a more cautious choice than the neutral $z = 0$, and we would be making more assessments than we would with the latter choice. Whereas with $z = 1$, we would be more optimistic and interrupt the algorithm after fewer similarity assessments suggested by the raw confidence value for $z = 0$.

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 (e.g. Smyth, 1998), 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 subsection 4.6.2, 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.

4.6 Evaluation of *ALK*

There have been three main goals for the development of *ALK*: (1) To be able to interrupt kNN search and get best-so-far kNN when exact kNN search is not feasible; (2) to attach confidence values to best-so-far kNN 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 sections detailed the steps of how we developed such an anytime algorithm and a *confidence* measure that gives us the expected 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 subsection 4.6.1 and explain how to interpret it. Later, in subsection 4.6.2, we describe how we built temporal CBs out of time series datasets to be used in the experiments for *ALK*. In subsection 4.6.3, we give the settings and insights of the experiments that we carried out. The results both demonstrate empirically that our confidence measure is efficient enough, and they provide evidence regarding the speed-up achieved by *ALK* when we interrupt the algorithm upon reaching given confidence thresholds.

4.6.1 Efficiency of confidence

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

Definition 4.7 (Efficiency of confidence)**.** *When ALK is interrupted after c number of similarity assessments during the kNN search for the u^{th} problem of a sequence, the efficiency of confidence*

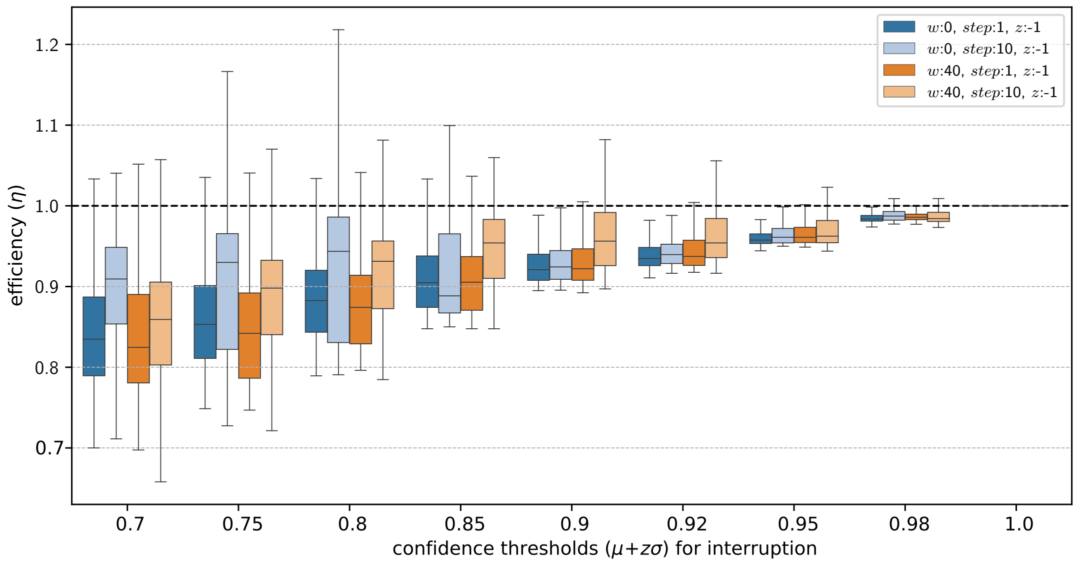


Figure 4.3: **Efficiency of confidence.** $\frac{\mu+z\sigma}{Q}$ ratio for interruption tests on the *SwedishLeaf* dataset, with $z=-1$ and different time window width w and $step$ settings and using the PDP for which an excerpt is given in Table 4.1.

for the i^{th} *NN,* $\overset{u,i}{\eta_c}$*, is:*

$$\overset{u,i}{\eta_c} = \frac{\overset{u,i}{\mu_c} + z\overset{u,i}{\sigma_c}}{\overset{u,i}{Q_c}} \tag{4.5}$$

Just like in confidence thresholds, we use the $z\sigma$ offset parameter to be able to incorporate the deviation of the confidence into efficiency measure, if wanted so. When interrupted with a confidence threshold τ, it makes sense to measure the efficiency with the same z value used in the $Interrupt$ function to determine the interruption point (Def. 4.6). An efficiency value $\eta \gg 1$ would signal an overconfident confidence measure that can possibly mislead reasoning with best-so-far kNN; 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 Figure 4.3. Interruption tests for this plot were conducted on four different CBs generated out of the *SwedishLeaf* dataset by four different time window width w and $step$ settings. The confidence thresholds for interruption were selected with $z = -1$ setting. For all CBs, we 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. And the efficiency converges to 1.0 as the confidence threshold also gets closer to 1.0, to which we ultimately reach when we have the exact kNN. 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, Figure 4.3 shows that, when the algorithm was interrupted at $\mu-\sigma \in \{0.7, \ldots, 0.98\}$, the actual quality of the best-so-far kNN were higher than these confidence threshold values since the efficiency stays below 1.0.

4.6.2 Datasets

In the experiments for *ALK* we used the same eleven univariate time series datasets described in section 3.4. For every dataset, we chose the larger of the 'train' and 'test' sub-datasets to generate the CB_TRAIN and input sequences to be used in building the PDP for that particular 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 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 input sequences were unseen.

The same method described in section 3.4 was used in the application of *time window* on TS sequences in the generation of both CB_TRAIN and CB_TEST. Both *expanding* and *fixed-width* time window approaches were combined with the time window *step* concept. See Table 3.1 for the datasets and their corresponding CBs used in our experiments to build PDPs.

4.6.3 Experiment settings

In all experiments we used normalized *euclidean distance* and the similarity functions given in Equations (2.1) and (2.2) respectively. To measure the distance between two cases of different number of features, again 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 as described in subsection 3.5.1.

After deciding the similarity assessment method, for each TS dataset given in subsection 4.6.2, we launched four experiments for combinations of two different time window width and step settings. We used time window width $w \in \{\textit{Expanding}, 40\}$ and time window $step \in \{1, 10\}$. Having k set to 9, each experiment for a configuration 3-tuple of $\big(\textit{dataset}, w, step\big)$ 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 4.3.

Consecutively, in the second stage, we built its PDP as described in section 4.4. While building PDP for each dataset, we discretized the range of similarity calculations and the quality range with δ_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 definition in Eq. (4.5), 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 9^{th} NN and taking into account the problem index of the query. And the set of confidence thresholds for interruption was

$\tau = \mu\text{-}\sigma \in \{0.70, 0.75, 0.80, 0.85, 0.90, 0.92, 0.95, 0.98\}$. So, for example, given the u^{th} query for a test sequence and a threshold $\tau = 0.95$, the algorithm was automatically interrupted after the number of similarity calculations needed for the $\mu\text{-}\sigma$ provided by PDP for the 9^{th} NN of a u^{th} query of an input sequence to reach 0.95.

4.7 Results

In order to assess if we met our design goals for *ALK*, in this section we provide the results of average gains upon interruptions, and average efficiency of confidence estimation along experiments. We calculated these values precisely as follows. At each interruption, similarities of the best-so-far kNN to the query, and the confidence μ together with its deviation σ for each member of the kNN were recorded. Finally, we let the algorithm finish and we obtained the similarities of the exact kNN to the query. And later, by backtracking, we calculated the actual qualities Q. This allowed us to calculate and record the efficiency η of the confidence for each NN using the Eq. (4.5). At interruptions, we also recorded the $gain$ in similarity calculations for the query given in the Eq. (3.6).

Finally, the 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 4.2. 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. As mentioned in section 4.1, gains of uninterrupted *ALK* are precisely the gains that would be achieved by the original *Lazy kNN*. 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 discussed in section 3.6. In Table 4.2, we see that 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 kNN, 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 Table 4.2, 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 particular 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 kNN, and this 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 kNN. The following section will shed more light into the underlying reasons for these leaps. 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 a consecutive problem, and thus the kNN of a problem are more similar to the kNN of its predecessor in the problem sequence. Thus, less calculations are needed to obtain the exact kNN 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 $\big(\overline{\eta} \in \big[0.89,\ 0.98\big], \overline{\sigma_\eta} \in \big[0.02,\ 0.09\big]\big)$. The average efficiency was slightly below 1.0, which means that the actual quality was a bit above the confidence threshold. This was a desired behaviour 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.

Table 4.2: **Average gain upon interruptions at confidence thresholds**. 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 kNN. 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.

	Time window		Unint.	Interruption at μ-$\sigma =$								Efficiency	
Dataset	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 *gain* concept that we have been using throughout the dissertation 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 lets 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 4.3 that translates the gain into real execution time for a test platform that we used. The table gives the average execution times for kNN search per query 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 for a query. The table shows that even when *ALK* is not interrupted and run to completion to find exact kNN, it is faster than brute-force search by orders of magnitude. However, the real contribution of our algorithm is for the occasions when this speed-up is still not feasible and we have to resort to approximate kNN. In this case, even when we interrupt *ALK* at a high confidence threshold like 0.98, the speed-up drastically increases. For example, in the *EOGHorizontalSignal* experiment with $w = 40$ and $step = 1$ setting, *ALK* delivers best-so-far kNN 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 seconds down to 0.049 seconds. For the same experiment, the speed-up factor increases to 370.37 if we settle with a confidence of 0.85. Table 4.3 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 $step = 10$ setting as explained in section 3.4. 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 4.2.

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 4.3: **Average execution time and speed-up factor per query**. Average execution time of kNN search per query is given in milliseconds. Speed-up factor is given with respect to the execution time of a brute-force search. Note that, average execution time for an experiment is inversely proportionate to its average gain in Table 4.2.

			Uninterrupted			*ALK* interrupted at μ-$\sigma=$							
	Time window		Brute	*ALK*		0.98		0.95		0.85		0.70	
Dataset	w	$step$	ms	ms	Speed	ms	Speed	ms	Speed	ms	Speed	ms	Speed
PowerCons	Expanding	1	421.93	102.74	4.11x	9.37	45.05x	5.78	72.99x	3.50	120.48x	2.32	181.82x
		10	28.85	15.10	1.91x	7.08	4.08x	4.51	6.4x	3.04	9.49x	1.88	15.31x
	40	1	425.86	129.37	3.29x	10.90	39.06x	6.43	66.23x	3.58	119.05x	2.30	185.19x
		10	24.10	15.64	1.54x	6.08	3.97x	4.45	5.41x	3.54	6.82x	0.95	25.25x
SwedishLeaf	Expanding	1	1826.04	438.98	4.16x	22.10	82.64x	15.70	116.28x	10.41	175.44x	6.21	294.12x
		10	86.03	45.12	1.91x	20.61	4.17x	17.25	4.99x	11.18	7.69x	3.13	27.47x
	40	1	1295.09	270.54	4.79x	13.34	97.09x	11.14	116.28x	7.25	178.57x	4.40	294.12x
		10	69.82	43.28	1.61x	25.35	2.75x	17.22	4.06x	3.27	21.37x	2.53	27.55x
Strawberry	Expanding	1	2118.25	389.97	5.43x	26.69	79.37x	19.06	111.11x	12.92	163.93x	9.11	232.56x
		10	119.03	51.91	2.29x	29.85	3.99x	23.41	5.08x	15.39	7.73x	3.49	34.13x
	40	1	1437.21	156.94	9.16x	15.09	95.24x	12.22	117.65x	8.19	175.44x	6.18	232.56x
		10	98.52	41.59	2.37x	19.72	5.0x	15.95	6.18x	7.84	12.56x	1.64	60.24x
EOGHorizontalSignal	Expanding	1	10206.39	906.33	11.26x	50.01	204.08x	40.83	250.0x	27.56	370.37x	27.56	370.37x
		10	1485.92	351.87	4.22x	17.24	86.21x	15.45	96.15x	8.77	169.49x	5.50	270.27x
	40	1	8057.42	294.10	27.4x	49.15	163.93x	42.70	188.68x	21.76	370.37x	21.76	370.37x
		10	710.63	92.10	7.72x	6.54	108.7x	5.61	126.58x	5.05	140.85x	2.27	312.5x
InsectWingbeatSound	Expanding	1	1656.07	306.37	5.41x	79.16	20.92x	64.75	25.58x	44.05	37.59x	41.40	40.0x
		10	80.18	35.31	2.27x	33.65	2.38x	30.24	2.65x	16.93	4.74x	5.14	15.6x
	40	1	958.27	144.03	6.65x	44.08	21.74x	35.94	26.67x	24.82	38.61x	23.00	41.67x
		10	54.77	31.62	1.73x	30.22	1.81x	28.06	1.95x	8.66	6.33x	3.23	16.98x
ECG5000	Expanding	1	1736.38	404.40	4.29x	134.92	12.87x	94.46	18.38x	50.53	34.36x	44.10	39.37x
		10	107.11	55.60	1.93x	55.60	1.93x	55.60	1.93x	37.12	2.89x	4.67	22.94x
	40	1	1370.28	275.43	4.98x	88.93	15.41x	63.86	21.46x	40.15	34.13x	34.94	39.22x
		10	69.98	35.84	1.95x	18.73	3.74x	12.74	5.49x	9.80	7.14x	7.62	9.18x
UWaveGestureLibraryX	Expanding	1	6669.74	1053.82	6.33x	123.39	54.05x	89.37	74.63x	72.70	91.74x	70.03	95.24x
		10	808.32	307.73	2.63x	127.15	6.36x	104.35	7.75x	56.58	14.29x	15.68	51.55x
	40	1	6408.30	526.76	12.17x	121.12	52.91x	80.10	80.0x	67.29	95.24x	64.72	99.01x
		10	354.34	136.00	2.61x	32.46	10.92x	22.22	15.95x	20.34	17.42x	6.63	53.48x
Yoga	Expanding	1	3189.04	424.14	7.52x	152.44	20.92x	106.83	29.85x	88.66	35.97x	86.74	36.76x
		10	309.08	101.01	3.06x	64.54	4.79x	45.31	6.82x	27.85	11.1x	11.19	27.62x
	40	1	2329.18	172.59	13.5x	106.68	21.83x	77.79	29.94x	65.22	35.71x	63.35	36.76x
		10	138.37	49.83	2.78x	40.57	3.41x	15.18	9.12x	9.91	13.97x	5.83	23.75x
Phoneme	Expanding	1	5085.99	562.00	9.05x	189.71	26.81x	125.12	40.65x	124.10	40.98x	123.59	41.15x
		10	610.76	173.46	3.52x	57.84	10.56x	42.75	14.29x	21.87	27.93x	15.15	40.32x
	40	1	4383.57	1447.89	3.03x	162.63	26.95x	107.84	40.65x	106.52	41.15x	106.52	41.15x
		10	435.54	313.28	1.39x	25.87	16.84x	23.39	18.62x	16.33	26.67x	10.89	40.0x
Mallat	Expanding	1	1329.64	137.49	9.67x	129.11	10.3x	124.19	10.71x	122.06	10.89x	121.40	10.95x
		10	189.64	51.52	3.68x	50.65	3.74x	48.85	3.88x	31.69	5.98x	24.73	7.67x
	40	1	867.53	60.12	14.43x	56.56	15.34x	52.23	16.61x	50.23	17.27x	49.45	17.54x
		10	62.13	20.04	3.1x	19.01	3.27x	17.75	3.5x	10.53	5.9x	8.10	7.67x
MixedShapesRegularTrain	Expanding	1	11534.43	1093.46	10.55x	239.92	48.08x	161.48	71.43x	154.56	74.63x	153.41	75.19x
		10	1380.13	354.69	3.89x	52.03	26.53x	42.23	32.68x	26.77	51.55x	19.46	70.92x
	40	1	9590.46	414.31	23.15x	196.60	48.78x	138.10	69.44x	127.55	75.19x	126.59	75.76x
		10	1076.32	256.92	4.19x	49.94	21.55x	46.28	23.26x	25.62	42.02x	15.50	69.44x

Test environment specs: CPU: 12 $\times$ Intel(R) Core(TM) i7-8700K @ 3.70GHz; Memory: 31GiB; OS: Ubuntu 18.04; Python: 3.7.3.

4.8 Insights on *ALK*'s gain

This section aims to scrutinize the superior gain provided by *ALK*, and in particular, intents to answer how the algorithm can remarkably exceed the gain of *Lazy kNN* even for interruptions at confidence thresholds as high as 0.98. Indeed, this was one of the striking observations for many experiments in the previous results section (cf. columns 'Unint.' and '0.98' in Table 4.2). The confidence thresholds in experiments were selected for the k^{th} NN (for details see subsection 4.6.3). So, as given by the threshold definition 4.6, interrupting the algorithm at a threshold of 0.98 for $z = -1$ setting, we expect the similarity of the best-so-far k^{th} NN to the query to be at most 2% below the similarity of the exact k^{th} NN. So, the algorithm must have evaluated enough number of "good" candidates to reach this very close approximation. Therefore, the observed leap in gain for high thresholds indicates that we obtain fairly near—if not exact—kNN after much fewer similarity assessments compared to the total number of candidates.

On the other hand, this observed phenomenon corroborates our intuition as one of the motivations in the design of our proposed algorithm. In chapter 3 (p. 20), we discussed that as the shared history between two consecutive queries of the same problem sequence grows, it is likely to find the kNN within the neighbors of recent queries.

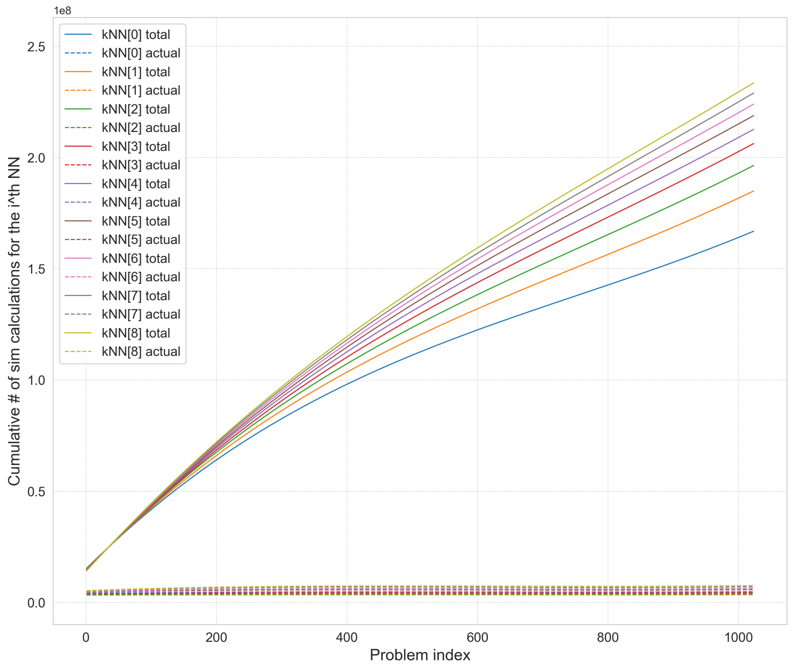
In order to thoroughly clarify the interpretation of the results given above, we wanted to see when exact kNN are actually found even if the search continues among other candidates in the guidance of their upper-bounds of similarity. To have this information, for each kNN member, we tracked the *actual* number of similarity assessments made till that NN was found and the *total* number of evaluated candidates to ascertain its exactness. More specifically, we kept a counter of performed similarity assessments during kNN search. And whenever a nearer neighbor replaced an existing kNN member (shifting it down) after c^{th} assessment, we set the *actual* value of all affected kNN members to c as their ranks are decided at the same time. The counter value at the end of i^{th} iteration gave the $total$ value for i^{th} kNN member. And in the end of kNN search, we had the *actual* values for all kNN members.

Figure 4.4 shows the actual and total number of similarity calculations recorded in two experiments with *MixedShapesRegularTrain* dataset.[2] To better mark the difference between these two values throughout updates, the figure plots the cumulative sum of assessments made for each kNN member until that problem index. More formally, where $actual(u, i)$ and $total(u, i)$ are the actual and total values recorded for the i^{th} NN during the kNN search for the u^{th} problem of an input sequence respectively, cumulative sums of these values are calculated as follows:

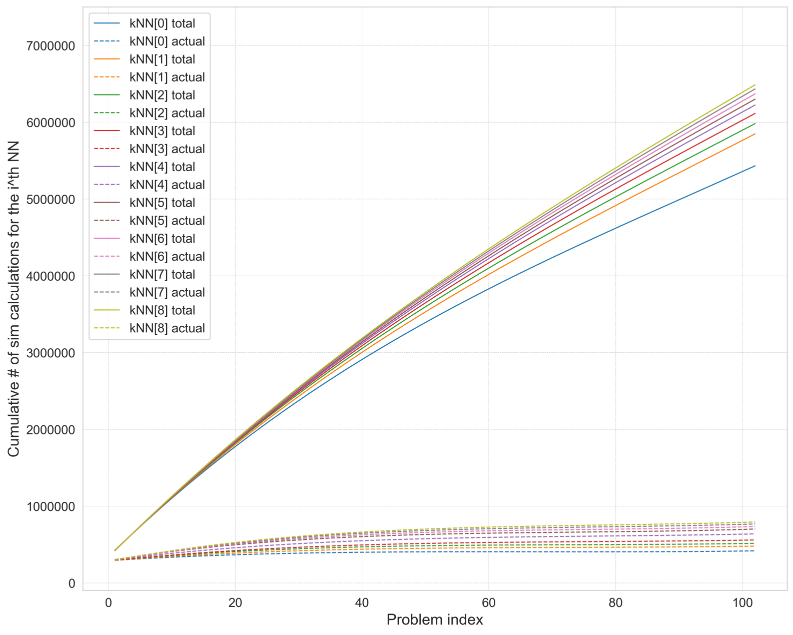
$$cumsum\_actual(u,i) = \sum_{x=1}^{u}\sum_{y=1}^{i} actual(x,y)$$
$$cumsum\_total(u,i) = \sum_{x=1}^{u}\sum_{y=1}^{i} total(x,y)$$

Later while plotting in Figure 4.4, curve fitting is applied to the cumulative sum data points obtained from all input queries. In the figure, we can clearly see that the *actual* number of similarity

[2]The actual and total values in Figure 4.4 were gathered throughout the same corresponding experiments used in Figures 3.4a and 3.4b.



(a) Time window: $width = Expanding$, $step = 1$



(b) Time window: $width = Expanding$, $step = 10$

Figure 4.4: ***Actual* vs *total* number of similarity assessments to find exact kNN** in experiments with the *MixedShapesRegularTrain* dataset with $k = 9$ setting. We see that *ALK actually* finds exact kNN after very few similarity calculations compared to the *total* number of evaluated candidates to mathematically guarantee that they are the exact kNN. The numbers are cumulative for each NN and for each problem index of the test sequence.

calculations needed to find each NN is quite below than the *total* number of evaluated all candidates for that NN. In other words, although we have to evaluate all candidates during the search for exact kNN, the figure shows that exact kNN are indeed found among the early candidates which are essentially within neighbors of recent predecessor problems. Consequently, the *actual* values reflect to the *quality* of best-so-far kNN. The sooner we approach to exact kNN, the higher quality we will get for fewer similarity calculations. And thus, the PDP will estimate higher confidence for smaller number of calculations. So, when we interrupt the algorithm at these estimate confidence thresholds, we will have higher *gain*. And this explains the reason behind the leap in gain between interruption at high confidence thresholds and uninterrupted run in Table 4.2.

Of course, as seen in Table 4.2, the increase in gain can be very drastic such as $28.07\% \rightarrow 94.06\%$ for threshold 0.98 in $(\textit{Phoneme}, 40, 10)$ experiment, or more modest like $55.96\% \rightarrow 58.04\%$ for threshold 0.98 in $(\textit{InsectWingbeatSound}, \textit{Expanding}, 10)$ experiment. The leap in gain is defined by how good the neighbors found in the early stages of kNN search are compared to the exact kNN. A big difference between the ratios of *actual* and *total* values to the CB size is a good indicator that we would have this leap upon interruption. But, it is not the only indicator. If there are many good candidates for a query, *actual* values would be closer to *total*. However, we could still be finding very approximate NNs from the beginning of search. In this case, the leap should be expected as well since the quality would be high starting from the early stages of search. So, we can say that the selection of interruption thresholds and the expected gain should depend on the CB and domain characteristics.

Finalizing this section, we would like to note another aspect of the kNN search conducted by *ALK*. Figure 4.4 also shows that most of the similarity assessments made by *ALK* is by far for the nearest neighbor (i.e. $kNN[0]$ in the figure). Later, remaining $k-1$ NNs are found relatively very fast. This observation may not be very surprising. The first iteration of the incremental core of *ALK* evaluates all candidates for 'the' nearest neighbor. And every time a candidate beats a best-so-far kNN member, the winner shifts the loser down (see Algorithm 4.3, line 19) and sets a new candidacy threshold for the rest of the cases in $RANK$. Eventually, when the first iteration ends and the second starts, there is already a very high threshold for the candidates of the second nearest neighbor (i.e. $kNN[1]$). Precisely, it was the penultimate best-so-far nearest neighbor before losing against the exact NN. So, high candidacy threshold results in fewer candidates. And this goes on likewise at each iteration for the rest of kNN.

4.9 Alternative RANK iterations

Until now both *Lazy kNN* and *ALK* searched kNN candidates in $RANK$ in a **Top-down** fashion starting from the most recent $Stage$ towards the initial one. This section aims to highlight that this is not necessarily the only way for iteration over $RANK$. As long as the upper-bound of similarity of a $case$ is calculated accordingly taking into account the $Stage$ that $case$ resides in, the algorithms are capable of telling if the $case$ proves to be a candidate or not. Thus, the mathematical certainty in candidacy assessment thanks to the triangle inequality gives us a freedom of iterating over the $RANK$ in any manner that suits us best. Below we give four examples to alternative iterations:

Bottom-up

As the extreme alternative to top-down style, this iteration searches kNN candidates in $RANK$ just in the opposite direction. The kNN search for a consecutive problem P^u starts from the top case in $Stage^0$ created for the initial problem P^0. Although it is more likely that the true kNN are to be found in nearer predecessor problems as discussed in the introduction to Chapter 3, it is mathematically possible that the accumulated Δs until u^{th} stage may yield kNN candidates of P^u within $Stage^0$ even for a $u \gg 0$ (see Definition 3.1).

Interleaved

This alternative is a hybrid of top-down and bottom-up iterations. For a target problem P^u, the candidacy assessments are made in both directions alternately. One case is assessed in top-down direction (starting from $Stage^{u-1}$) and the next one in bottom-up direction (starting from $Stage^0$).

Jumping

After evaluating every n^{th} candidate in a stage, this iteration makes a momentary jump to the next stage for candidacy assessment. Precisely, after every n^{th} evaluation within the *current* stage $Stage^j$, if the top case in $Stage^{j-1}$ is a candidate, we calculate its similarity and continue back with the $Stage^j$. This iteration expects this momentary randomness in candidate selection to improve the gain by reducing the number of candidates.

Exploit Approaching Candidates

During the kNN search for P^u, if a candidate proves nearer to P^u than it was to a previous problem P^j ($j < u$), this iteration exploits the predecessor and successor cases of that candidate to check if they get nearer to P^u as well. More formally, where seq^x is the x^{th} case in a sequence seq and seq^x resides in $Stage^j$:

If $candidate(seq^x, P^u) \wedge sim(seq^x, P^u) > sim(seq^x, P^j)$, then we give priority to previous and later candidate problems in seq whose similarities to P^u are $\geq sim(seq^x, P^u)$. The set of these consecutive problems in seq can be formalized as follows:

$\{seq^y \mid candidate(seq^y, P^u) \wedge sim(seq^y, P^u) \geq sim(seq^x, P^u), \forall y \in [a, b]\}$ where a, b are the indices of the first predecessor and successor cases of seq^x, respectively, that give lower similarity than $sim(seq^x, P^u)$.

Exploit Approaching Candidates iteration expects that if a case seq^x got nearer to P^u, then its adjacent cases in the same sequence starting from seq^{x+1} and seq^{x-1} can prove to be even nearer to P^u. However, in the implementation of this iteration, accessing adjacent cases in $RANK$ is not straightforward as we don't have the knowledge of the $Stage$ they reside in. $RANK$ does not hold the indices of cases per se. So, an efficient implementation of this iteration requires an additional hash table $RANK\_HASH$ for fast and arbitrary access to the cases in $RANK$. Moreover, $RANK\_HASH$ should be maintained as candidate cases are moved between $Stage$s during kNN search. Access to the hash table does not introduce a significant time complexity given that its maintenance is also handled cautiously. However, this alternative iteration does have an additional space cost. Here we present this example just to show the possibility of implementing more complex iterations.

In our experiments, Bottom-up and Interleaved alternatives did not surpass the gain of Top-down iteration as expected. In particular, candidates were found within the stages at the far end of $RANK$ thanks to their upper-bound of similarities. However, the actual similarity of these candidates usually could not win over best-so-far kNN. And when they did win, they were soon replaced by candidates from more recent stages. And finally, the number of more similarity assessments made by these two iteration methods lowered the gain of *Lazy kNN* compared to top-down iteration. Although there is no need to present their experiment results, we still wanted to mention these alternatives as examples to extreme and hybrid iterations.

On the other hand, *Jumping* iteration did improve the gain slightly. Table 4.4 shows that this alternative has equal or slightly better gain on average compared to Top-down for the CBs of these select datasets. And interestingly, the gain is higher when the jumps are more frequent. It seems that sometimes we find better kNN candidates for P^u in the penultimate stage $Stage^{u-2}$ than the candidates in the last stage $Stage^{u-1}$. This is most probably due to the fact that although upper-bound of similarity provides us with a *radius* to identify our candidates in the problem space, it does not indicate a *direction*. And the sporadic jumps are sometimes made towards the right direction and this increases the gain of the algorithm.

Table 4.5 shows the gains of *Lazy kNN* with the *Exploit Approaching Candidates* iteration, compared to Top-down. These results show that with this alternative we may have a slight increase in gain as well. Nevertheless, due to the considerations regarding its additional space complexity and maintenance effort for the hash table that we discussed above, this iteration method requires careful studying of the domain and CB characteristics beforehand to make sure that it is worth using it.

The alternative iterations in this section are discussed on their own with comparison to Top-Down. However, *Lazy kNN* and *ALK* can benefit from alternative approaches using them in a hybrid way too. For example, for the first problems corresponding to the early sequence updates, the algorithm cannot make the most of a long shared history between these consecutive problems. Then, it may resort to the Jumping iteration hoping that random search of candidates may result in higher gain compared to Top-Down, as it is the case for some experiments in Table 4.4. And when the number of consecutive problems in a sequence increases, the algorithm can switch to Top-Down as it is more likely that the kNN are within the neighbors of recent predecessor problems. Or, we can switch back and forth between these two type of iterations when one of them does not seem to yield good enough candidates.

The complete code for Jumping and Exploit Approaching Candidates iterations together with simple instructions for reproducing the experiments given in this section are publicly available at the online repository: https://github.com/IIIA-ML/alk

Table 4.4: **Average gain (%) with Jumping vs Top-down iterations**

	Time window		TopDown	Jump after every # of candidates =				
				50	10	5	2	1
Dataset	w	$step$						
PowerCons	40	1	70.81	70.81	70.81	70.82	70.83	70.83
		10	35.81	35.82	35.82	35.85	35.98	36.17
	Expanding	1	75.76	75.76	75.76	75.76	75.77	75.78
		10	48.22	48.22	48.26	48.33	48.43	48.43
SwedishLeaf	40	1	80.09	80.09	80.10	80.10	80.10	80.11
		10	36.26	36.32	36.52	36.70	37.18	37.56
	Expanding	1	76.16	76.16	76.16	76.16	76.17	76.18
		10	47.70	47.72	47.88	48.02	48.24	48.33
Strawberry	40	1	89.24	89.25	89.25	89.25	89.25	89.25
		10	58.70	58.71	58.75	58.80	58.89	59.00
	Expanding	1	81.65	81.65	81.65	81.65	81.66	81.67
		10	56.87	56.98	57.53	57.74	58.03	58.20
EOGHorizontalSignal	40	1	96.34	96.34	96.35	96.35	96.35	96.35
		10	87.74	87.75	87.77	87.78	87.79	87.79
	Expanding	1	91.27	91.27	91.27	91.27	91.27	91.28
		10	75.95	75.96	75.95	75.95	76.00	76.03
InsectWingbeatSound	40	1	85.81	85.81	85.81	85.81	85.82	85.82
		10	43.91	43.93	43.96	43.97	43.97	43.97
	Expanding	1	82.67	82.67	82.67	82.67	82.67	82.67
		10	59.37	59.39	59.49	59.56	59.72	59.79
ECG5000	40	1	80.56	80.56	80.56	80.57	80.58	80.60
		10	50.42	50.46	50.58	50.71	50.87	50.89
	Expanding	1	77.07	77.07	77.08	77.08	77.09	77.10
		10	49.85	49.89	49.90	49.98	50.17	50.48
UWaveGestureLibraryX	40	1	91.80	91.80	91.80	91.80	91.80	91.80
		10	56.12	56.13	56.13	56.16	56.23	56.29
	Expanding	1	84.21	84.21	84.21	84.21	84.21	84.22
		10	61.85	61.86	61.95	61.96	62.01	62.08

Table 4.5: **Average gain (%) with Exploit Approaching Candidates vs Top-down iterations**

	Time window		TopDown	ExploitCandidates
Dataset	w	$step$		
PowerCons	40	1	72.96	72.97
		10	30.69	30.71
	Expanding	1	75.12	75.14
		10	48.01	48.05
SwedishLeaf	40	1	80.21	80.21
		10	37.86	37.89
	Expanding	1	76.17	76.17
		10	48.27	48.34
Strawberry	40	1	88.83	88.83
		10	58.74	58.84
	Expanding	1	81.71	81.72
		10	56.12	56.83
EOGHorizontalSignal	40	10	84.32	84.33
	Expanding	10	77.14	77.18
ECG5000	40	10	46.01	46.08
	Expanding	10	50.04	50.29

4.10 Summary

The exact kNN search algorithm *Lazy kNN* introduced previously in Chapter 3 reaches remarkable speed-ups in CBR retrieval in domains with large-scale CBs of temporally related cases. However, for some applications and/or domains the speed-up provided by this algorithm may not suffice and the execution time for finding exact kNN may still not be feasible. Therefore, to fit the CBR system with an approximate retrieval option for time-critical applications, in this chapter we described our methodology to transform *Lazy kNN* to *Anytime Lazy kNN* (*ALK*).

Specifically, we detailed how we constructed a probabilistic model of adjustable accuracy to estimate the quality of best-so-far kNN upon interruption. We referred to the expected quality as the *confidence* of the algorithm for its output in accordance with 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 estimation provided by our probabilistic model. Before experimentation, we explained how we generated CBs out of time series datasets for training our probabilistic model and for interruption tests. 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 kNN are almost as near to the query as the exact kNN. So, with *ALK*, the expert can opt to wait for the completion of the algorithm to obtain exact kNN or, he/she can interrupt the search manually/automatically any time when a prompter response is needed and get best-so-far kNN together with a confidence value for each NN to reason with.

We also presented alternative ways of iterating over $RANK$ for kNN candidates. And with some of them we were able to achieve higher average gains compared to the conventional Top-down iteration style. These iteration examples indicate that we can even consider implementing domain-specific $RANK$ iterations when domain knowledge could help us find better candidates in shorter times.

Until now, we managed the speed-ups in kNN search by exploiting the *problem space* only. Precisely, we achieved this performance by leveraging the similarity between consecutive problems of a sequence in temporal case bases. However, in CBR, there is more to the case base than the problem space. In the next chapter, we show how we can further increase the speed-up by exploiting the *solution space*, in particular for classification purposes.

Chapter 5

Anytime Lazy kNN Classifier

A kNN classification algorithm that depends on a kNN search which ascertains exact kNN all-at-once has to wait for the completion of the search to provide the exact solution. However, a kNN classifier utilizing *Anytime Lazy kNN* (*ALK*) could yield the exact solution class before the termination of the kNN search by leveraging the incremental essence of the algorithm. Specifically, depending on the classification method, there may be occasions when the already-found exact NNs suffice to decide the class of the query without the need to find the remaining exact members of the kNN list.

In this chapter, we propose extensions to *ALK* for its use as a kNN classifier. And doing so, we empirically demonstrate how we further increase *ALK*'s gain in execution time by exploiting the solution space for classification. The chapter is organized as follows. In section 5.1, we describe the conditions where the algorithm can be terminated earlier with an exact solution for three major kNN classification methods. In section 5.2, we introduce the extensions to *ALK* for classification and give the pseudo-code of the new algorithm *ALK Classifier*. Later we present the results which show the increase in gain by *ALK Classifier* compared to uninterrupted *ALK*. In the same section, we also explore the *accuracy* in solutions that we would suggest at interruptions upon reaching confidence thresholds for kNN.

Furthermore, having a measure of confidence for suggested solutions is a desired feature of CBR systems (e.g. Cheetham, 2000). Therefore, beside the confidence for approximate kNN that we defined in section 4.5, we also present extensions to several solution confidence measures in CBR literature for classification with those approximate kNN in section 5.5. In particular, regardless of the exactness of solution, extended versions of these measures take into account best-so-far NNs instead of all-exact NNs used by their original versions.

5.1 Interrupting *ALK Classifier* with exact solution

In the following two subsections, we detail how we can pinpoint the moments of early termination by guaranteeing exact solution for widely-used kNN classification methods—namely, majority, plurality and distance-weighted voting. We cover the first two together as they are closely related. And in subsection 5.1.3, we give a generic method for the decision of exact solution with the exact NNs found so far.

5.1.1 Majority and plurality votes

In *majority* vote (a.k.a simple majority), all of the k nearest neighbors participate in classification with equal votes regardless of their proximities to the query. The majority class wins. If there is no simple majority (i.e. a class represented by more than half of the kNN), the decision can be taken by *plurality* vote (a.k.a. relative majority). In plurality vote, the winner class has the number of votes that is greater than any other class. k is usually chosen an odd number to avoid ties. However, a tie can occur even in this case and it has to be dealt with one way or another; e.g. by random choice between classes of equal votes, increasing the k etc.

ALK Classifier can be interrupted automatically and guarantee the exact solution class if the following conditions are met for majority and plurality votes, respectively:

1. When the majority of the kNN are exact and there is a majority class among them, there is no need to find the rest of the exact kNN members. For example, with $k = 9$, if five of the seven exact NNs are of the same class, e.g. (A, A, A, B, A, B, A), we do not need to continue the kNN search for the eighth and ninth NNs.

2. When the majority of the kNN are exact but there is no majority class, however, some of the exact NNs form a plurality which cannot be overcome by the remaining exact kNN members, then there is no need to continue either. For example, with $k = 9$, if the classes of the already-found seven exact NNs are (B, A, A, D, A, C, A), then there is no need to continue the search for the eighth and ninth NNs. “A” does not have the majority (five) votes, but it would win by plurality vote (four) because no other class can win even if the eighth and ninth NNs are in its favor.

5.1.2 Distance-weighted vote

Majority and plurality votes have a drawback when the distances of kNN to the query vary widely. For example, even though there is a very near neighbor, the voting ends in favor of the very distant neighbors of another class just because they form the majority. And in this example we might have preferred to label the query with the nearest neighbor’s class instead. To tackle this issue, *distance-weighted* vote gives more importance to nearer neighbors than further ones where each neighbor contributes to voting inversely proportional to its distance to the query. In other words, the more similar a NN is to the query, the higher is its contribution.

For this voting, *ALK Classifier* ensures the exactness of the solution when either of the following conditions is met:

1. Just like the Majority Vote in subsection 5.1.1, when the majority of the kNN are exact and there is a majority class among them, then there is no need to continue the kNN search. This is because, due to the incremental nature of *ALK Classifier*, the remaining exact kNN members cannot be nearer to the query than the already-found furthest exact neighbor. Therefore, the total contribution of the remaining exact NNs cannot exceed the contribution of the already-found exact NNs of the majority class;

2. When the majority of the kNN are not exact but there are at least two exact NNs among them, it is still possible to determine if the exact solution can be guaranteed as follows. At

any given moment in kNN search, the furthest exact NN sets the lower-bound of distance for candidacy (d_{LB}) for the remaining kNN candidates. As mentioned above, *ALK Classifier* guarantees that no kNN candidates surpass the furthest exact NN, although they can equal it. So, if there is a chance that the runner-up class (i.e. the class that currently has the second highest vote) could win—'assuming' that the remaining exact kNN members are all of this class and their distances to the query equal d_{LB}, we should continue the search. Otherwise, we can stop since the class of the solution is guaranteed to not change.

For example in Figure 5.1, while *ALK Classifier* is searching for the 4^{th} NN where $k = 5$, a weighted vote among green exact NNs and red approximate NNs yields a 'circle' class for the target query t. However, if the search continued, two *imaginary* candidates shown as hollow triangles would change the solution class to a 'triangle'. So, to obtain the exact solution in *ALK Classifier* classification, the search should continue until no such supposed candidates could change the voting result. And, d_{LB} can be used to determine the cut-off point for *ALK Classifier* classification where no *imaginary* candidates can change the current solution.

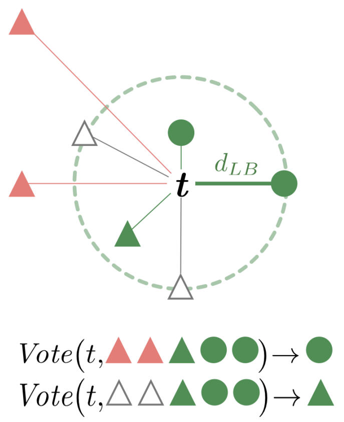

t
d_{LB}
$Vote(t,$▲▲▲●●$) \rightarrow$ ●
$Vote(t,$△△▲●●$) \rightarrow$ ▲

Figure 5.1: **Distance-weighted classification with *ALK Classifier*.** Green filled NNs are exact, red filled NNs are approximate, and hollow NNs are imaginary best rival NNs of the target query t.

5.1.3 Guaranteeing exact solution

For each of the majority, plurality and distance-weighted voting, we observe a common pattern in the decision of whether or not we can guarantee the exact solution with best-so-far kNN. While deciding, what we essentially do is checking the solution that is suggested by the current exact neighbors against the solution with the full kNN list where the missing members are filled with imaginary neighbors. In majority and plurality votes, these imaginary neighbors are of the runner-up class. For $k = 9$, in the majority vote example (A, A, A, B, A, B, A) in subsection 5.1.1, imaginary neighbors would be (B, B). And in the plurality example (B, A, A, D, A, C, A) in the same place, they would be any of the (B, B), (C, C), (D, D) pairs.

In distance-weighted vote, imaginary neighbors are also of the runner-up class that are supposedly at the same distance to the target query as the furthest exact NN. In Figure 5.1 these are the hollow triangles at the distance d_{LB} set by the furthest exact NN, i.e. the further green circle.

On the other hand, for any of these three types of voting, there is also an implicit prerequisite of having a single leading solution among the competing classes of exact NNs. Otherwise, if there is more than one leading class with the same vote, the exact solution cannot be guaranteed since we cannot know which one of the competing classes will come forward during the rest of the kNN search.

So, for these three votings, the decision of whether or not we can guarantee the exact solution at any moment in kNN search consists of three steps: (1) make sure there is a single leading class among the competing exact NNs, and if so; (2) create an imaginary best rival and fill the missing exact kNN members with this rival; (3) check the solution with the current exact NNs against the solution with the supposed kNN list. If the solutions of both are the same, it means that no matter how good the future rivals are, they cannot change the current solution. Thus, we can guarantee the exact solution with the exact NNs found so far. However, if the solution with the supposed kNN is different, it means there is still chance for the other classes to win. Therefore, the exact solution cannot be guaranteed for the moment. In the *ALK Classifier* algorithm that we introduce in the following section, we will use this generic method of decision for the exact solution.

5.2 *ALK Classifier* algorithm

In this section, we fully detail *ALK Classifier*—as an extension to *ALK*—in four sub-algorithms covering its class structure, implementations of the public and auxiliary class methods and the core algorithm respectively.

Algorithm 5.1 defines the `AnytimeLazyKNNClassifier` class structure that is a sub-class of `AnytimeLazyKNN` introduced in Algorithm 4.1. It extends the parent class with an additional private attribute $\_fn\_reuse$ and a public method `SuggestSolution`. And, it overrides parent's constructor. `_Construct` calls parent constructor and sets the $\_fn\_reuse$ with the CBR reuse function passed as an argument that will serve as the classification method (e.g. plurality vote). Just like its parent, an instance of this class has to be created for each particular problem sequence.

Algorithm 5.2 implements the new public method `SuggestSolution`. This method can be called any time we want to suggest a solution for the current query by reusing the solutions of best-so-far/exact kNN. `SuggestSolution` returns the solution suggested by the $\_fn\_reuse$ function. Together with the solution itself, the method also returns the confidence for the solution with the system-provided `confidenceSOLN` function.

In Algorithm 5.4, `AnytimeLazyKNNClassifier` overrides parent's core method. The new implementation of `_IncrementalLazyKNN` provides an extra option to interrupt the algorithm when an exact solution is guaranteed. So, `_IncrementalLazyKNN` can be interrupted in two ways: (i) after a given number of similarity calculations (line 22); or (ii) when the exact NNs within the best-so-far kNN guarantee an exact solution (line 31). For the latter interruption choice, the *interrupt* argument of the `ConsecutiveSearch` and `ResumeLastSearch` methods should be -1 (see Algorithm 4.2). If $interrupt = -1$, the auxiliary method `_IsExactSolution` (Algorithm 5.3) is called after each iteration in `_IncrementalLazyKNN` to check if the exact solution is guaranteed with current exact NNs (Algorithm 5.4 line 31). And if this is the case, the algorithm is interrupted. `_IncrementalLazyKNN` returns the kNN list and the confidence of the algorithm for each member of the list. If the algorithm is run to completion, the kNN will be exact and their confidence values will be 1. If the algorithm is interrupted, best-so-far kNN list

Algorithm 5.1: Anytime Lazy KNN Classifier Class

```
1 Class AnytimeLazyKNNClassifier(base AnytimeLazyKNN):
      Attributes : $\_fn\_reuse$: Function to suggest the solution out of kNN (e.g. plurality vote for classification), for
                   parent attributes see Algorithm 4.1
      Methods    : SuggestSolution(): see Algorithm 5.3, for parent methods see Algorithm 4.1

2 Function _Construct($CB$, $k$, $fn\_initialNNS$, $fn\_dist$, $fn\_reuse$):
      Input      : see related class instance attributes above
      Output     : An AnytimeLazyKNNClassifier object to be used for a particular problem sequence (e.g.
                   history of treatment sessions of a particular patient)
3     $this \leftarrow$ AnytimeLazyKNN._Construct($CB$, $k$, $fn\_initialNNS$, $fn\_dist$, $fn\_reuse$) // see Alg 4.1
4     $this.\_fn\_reuse \leftarrow fn\_reuse$
5     return $this$
```

Algorithm 5.2: Anytime Lazy KNN Classifier Class - New Public Method

```
1 Function SuggestSolution($this$):
      Output     : The suggested solution for the best-so-far/exact kNN and the confidence for this solution. The
                   solution confidence is calculated by the system-provided confidenceSOLN measure
2     $stage^u \leftarrow this.\_RANK[0]$
3     return $this.\_fn\_reuse(stage^u.NN[:this.\_k])$, confidenceSOLN($stage^u.NN, this.\_k$)
```

Algorithm 5.3: Anytime Lazy KNN Classifier Class - Auxiliary Method

```
1 Function _IsExactSolution($this$, $iter$):
      Input      : The # of completed iterations, i.e the # of exact NNs
      Output     : True if the solution with current exact NNs ($NN_E$) is equal to the solution with the supposed kNN
                   ($kNN_S$); False otherwise. $kNN_S$ consists of $NN_E$ and imaginary exact NNs. The latter are the
                   clones of an imaginary neighbor ($N_S$) that is of the runner-up class and has the similarity of the
                   furthest exact NN to the query
2 $stage^u \leftarrow this.\_RANK[0]$
3 $kNN_A \leftarrow stage^u.NN[:this.\_k]$                          // Best-so-far kNN
4 $NN_E \leftarrow kNN_A[:iter]$                                    // Exact NNs
5 if not $SingleLeadingSolution(NN_E)$ then return $False$
6 $N_S \leftarrow ImaginaryBestRival(NN_E)$                        // Imaginary best rival neighbor
7 $kNN_S \leftarrow NN_E + [N_S] * (this.\_k - iter)$               // Supposed kNN
8 return $this.\_fn\_reuse(NN_E) \overset{?}{=} this.\_fn\_reuse(kNN_S)$
```

is returned together with the expected quality values for each member of the list provided by the confidenceKNN system function. confidenceKNN is based on the PDP of the algorithm generated for the application domain and time window settings as detailed in section 4.4.

Algorithm 5.3 implements the above-mentioned _IsExactSolution following the generic method detailed in subsection 5.1.3. In this method, we first ensure that there is only a *single leading solution* among current exact NNs (NN_E). Otherwise, since we cannot know which of the equally voted leading classes will win, we cannot guarantee the exact solution. So, the method quits returning $False$. If there is a single leading solution however, we create a supposed kNN list (kNN_S) to check if there is any chance that the *runner-up* class in NN_E (i.e. the second highest voted in competing classes) could win if the kNN search continued. kNN_S consists of current exact NNs NN_E and *imaginary* future exact NNs that are *clones* of an imaginary best rival neighbor N_S. N_S is a fictive case that is of the runner-up class and has the similarity of the furthest exact NN to the query. This is the maximum similarity that we can assign to N_S, because, due to the

Algorithm 5.4: Anytime Lazy KNN Classifier - Core algorithm

```
Function _IncrementalLazyKNN(this, interrupt =null):
    Input   : Optional interruption point for similarity calculations—null for no interruption, $> 0$ for interruption
              after a number of similarity calculations, $-1$ for interruption when an exact solution is guaranteed.
              No need to pass the query, it is accessed via the instance attribute $\_query$
    Output  : kNN list and the list of associated confidence for each neighbor. If the algorithm is run to
              completion, kNN are exact and their confidence values are 1; otherwise, at least some of the kNN
              are approximate and the confidence values are given by the system-provided confidenceKNN
              function $\in [0, 1]$
    $stage^u \leftarrow this.\_RANK[0]$
    $sort\_flag \leftarrow False$
    $calc \leftarrow 0$
    for $iter \leftarrow this.\_current\_iter$ to $this.\_k$ do                          // Iterate $\_RANK$ $\_k$ times
        $sum\Delta \leftarrow \sum_{j=0}^{this.\_current\_index-1} this.\_RANK[j].\Delta$
        for $j \leftarrow this.\_current\_index$ to $|this.\_RANK|-1$ do              // Iterate Stages in $\_RANK$
            $stage^j \leftarrow this.\_RANK[j]$
            foreach $assess$ in $stage^j.NN$ do
                $case \leftarrow assess.case$
                $sim \leftarrow assess.similarity$
                if $|stage^u.NN| < iter$ or $(sim + sum\Delta) > stage^u.NN[iter-1].similarity$ then
                    $stage^j.NN$.remove($assess$)                                 // $case$ is candidate
                    $sim \leftarrow 1 - this.\_fn\_dist(this.\_query, case)$                // Calc $case'$s sim
                    $calc \leftarrow calc + 1$
                    $new\_assess \leftarrow$ new Assessment$(case, sim)$
                    if $sim > stage^u.NN[this.\_k-1].similarity$ then
                        $stage^u.NN$.insert($new\_assess$, $i$) // Insert $case$ to kNN, $iter-1 \le i < k$
                    else
                        $stage^u.NN$.append($new\_assess$)
                        $sort\_flag \leftarrow True$
                    if $calc = interrupt$ then                                    // Interrupt w/ $calc$
                        $this.\_current\_iter \leftarrow iter$
                        $this.\_current\_index \leftarrow j$
                        if $sort\_flag$ then $stage^u.NN[iter+1:]$.sort_descending()
                        return $stage^u.NN[: this.\_k]$, confidenceKNN($u, this.\_k, calc$)
                else                                                              // $case$ is not candidate
                    break                                                 // Continue with the next Stage
            $sum\Delta \leftarrow sum\Delta + \Delta^j$                                       // Accumulate $\Delta$s
        $this.\_current\_index \leftarrow 1$
        if $interrupt$=-1 and this._IsExactSolution($iter$) then                 // Interrupt w/ soln
            $this.\_current\_iter \leftarrow iter$
            return $stage^u.NN[: this.\_k]$, confidenceKNN($u, this.\_k, calc$)
    if $sort\_flag$ then $stage^u.NN[this.\_k:]$.sort_descending()
    return $stage^u.NN[: this.\_k]$, $[1] * this.\_k$
```

incremental nature of our algorithm, we know that none of the remaining exact NNs can surpass the current furthest exact NN. If there is no runner-up class in NN_E, this class is chosen from existing classes in the $\_CB$. Finally, we compare the suggested solutions with NN_E and kNN_S. If the two solutions are different, it means that the imaginary neighbors which we created to back up the runner-up class were able to change the current leading solution. Then, it is possible that the solution changes as new *actual* exact NNs are found. So in this case, we cannot guarantee the exact solution for the moment and `_IsExactSolution` returns $False$. On the other hand, if the solutions with NN_E and kNN_S are the same, it means that even if the remaining exact kNN are of the runner-up class, it is impossible to change the current solution. Therefore, we can guarantee the exact solution with the current exact NNs that we found and `_IsExactSolution` returns $True$. After an interruption with exact solution, calling `SuggestSolution` would yield the exact solution together with a confidence value for the suggested solution provided by `confidenceSOLN`.

The complete code of *ALK Classifier* including simple instructions for reproducing the experiments that will be covered in the following section is publicly available at the online repository: https://github.com/IIIA-ML/alk

5.3 Evaluating *ALK Classifier*

In section 3.5, we evaluated the gain of our first proposed algorithm *Lazy kNN* in terms of avoided similarity assessments. *Lazy kNN* is an exact kNN search algorithm that provides the kNN all-at-once after termination and consisted the foundation for two more algorithms that followed. Later in section 4.6, we evaluated the gain of the second proposed algorithm *ALK* at interruptions upon reaching given confidence thresholds for kNN. And we compared the gain of the algorithm at interruptions against the gain at uninterrupted runs.[1] And, in this section our goal is to evaluate the gain of our third proposed algorithm *ALK Classifier* when it is interrupted automatically upon guaranteeing the exact solution during kNN search.

For this purpose, we launched experiments with *ALK Classifier* using CBs that were generated out of time series datasets introduced in section 3.4. CB generation settings were the same as the ones described in section 4.6.3. For each $\big(\textit{dataset},\, w,\, step\big)$ setting, we launched three parallel experiments that used the same input sequences and CB_TEST.

In the first experiment, *ALK Classifier* was interrupted upon reaching a given set of confidence thresholds for kNN $\tau = \mu - \sigma \in \big\{0.70, 0.75, 0.80, 0.85, 0.90, 0.92, 0.95, 0.98\big\}$ (i.e. the z parameter for standard deviation was -1). For threshold selection details see subsection 4.6.3. And after the interruption at the highest threshold 0.98, we resumed the algorithm and let it run till completion. In the second and third experiments, *ALK Classifier* was interrupted upon guaranteeing the exact solution where the fn_reuse was *plurality* and *distance-weighted* vote respectively.

When *ALK Classifier* is run uninterrupted or configured to interrupt with exact solution, calling `SuggestSolution` (Algorithm 5.2) after the algorithm stops would give us the exact solution. On the other hand, we were intrigued by the question as to the accuracy of solution when *ALK Classifier* is interrupted by kNN confidence threshold. In particular, we wondered how accurate the solution suggested by best-so-far kNN is when the algorithm is interrupted at a high threshold like 0.98. In this case, best-so-far kNN are expected to be almost as close to the query as the exact kNN. So, would their solutions also be the same? To find the answer to this question, in the first experiment detailed above, we called `SuggestSolution` after each interruption of the algorithm upon reaching a confidence threshold. And later, we compared the solutions at interruptions with the exact solution which we had upon the completion of the algorithm. When the solutions were the same, we called this a *solution hit*.

5.4 Results

The average gain for interruption with exact solution can be compared to the average gain upon interruption at confidence thresholds in Table 5.1 and Table 5.3, given for plurality and distance-

[1]We remind that *ALK* has exactly the same gain with its predecessor *Lazy kNN* when it is run uninterrupted.

weighted voting respectively. The average *solution hits* for the same experiments are given in Table 5.2 and Table 5.4. A solution hit occurs when the suggested solution with best-so-far kNN is equal to the solution with exact kNN.

The results show that interruption with exact solution increases the *gain* of *ALK Classifier* compared to its uninterrupted runs. This is an expected result as we do not continue the search to find remaining exact kNN members whenever we can guarantee the solution with current exact NNs. We also see that distance-weighted vote provides a slightly better gain compared to the plurality vote. This result is because, for $k = 9$, while we need at least five exact NNs to decide the exact solution with plurality vote, we may guarantee the exact solution with even only two exact NNs with distance-weighted vote depending on NNs' distances to the query. Specifically, if the nearest neighbor is too close and the second nearest neighbor is too far to the query, seven remaining NNs may not change the result since they are bound by the distance of the second exact NN.

On the other hand, although interruption with exact solution increases the gain of the algorithm, we observe that this additional gain is usually minuscule regardless of the voting method. This is due to the fact that the majority of the similarity assessments in kNN search are conducted to find 'the' nearest neighbor compared to the rest of the kNN members as discussed in section 4.8.

Of course, every time we interrupt *ALK Classifier* upon guaranteeing exact solution, we have a *solution hit* (see 'w/soln' column in Tables 5.2 and 5.4). When we interrupt the algorithm upon reaching confidence thresholds, however, the average solution hit in experiments are variable. For many CBs, we have high average hit percentage even for interruptions at relatively low confidence thresholds. And for a few other experiments, we have very low hit percentage. Virtually all of these low hit % experiments are with $w = 40$ and $step = 10$ setting which introduces more difference between two consecutive problems. Another observation of interest with hits is that we do not necessarily have a higher hit rate upon interruption at a higher confidence threshold. In some experiments, e.g. $\big(SweedishLeaf, Expanding, 1\big)$ in Table 5.2, the average solution hit % is higher for interruptions at the threshold of 0.70 compared to the average hit for the threshold 0.98. So, although best-so-far kNN for a higher confidence threshold are nearer to the query than the best-so-far kNN for a lower threshold, this does not necessarily imply that the solutions will be more accurate for the higher threshold. These results indicate us that the accuracy of solution with best-so-far kNN depends on the cluster structure in the CB solution space. In particular, when there are no clear boundaries between cases of different classes, further approximate kNN of a target query may suggest the same class with the exact solution while nearer kNN can give a different class. And the results show that this was the case for some of the CBs that we generated for our experiments.

Therefore, we can conclude that before using *ALK Classifier* for classification task in a CBR system, it would be a good practice to carry out solution hit experiments detailed in this chapter for the CB and select the confidence thresholds for interruption accordingly. If low solution hits are observed even for interruptions at high thresholds, this may indicate that we do not have clear boundaries in the solution space of the CB and/or there may be 'noise' cases. And to be sure of the underlying reason, we can perform clustering methods to the solution space. And if necessary, we can apply case-base maintenance methods that we briefly mentioned in section 2.1.

Table 5.1: **Average gain** upon interruptions by exact solution with **plurality vote** and at confidence thresholds.

	Time window		Unint.	Intrpt.	Interruption at μ-$\sigma =$				Efficiency	
				w/Soln	0.98	0.95	0.85	0.70	$\overline{\eta}$	$\overline{\sigma_\eta}$
Dataset	w	$step$								
PowerCons	40	1	71.81	73.14	97.42	98.47	99.16	99.46	0.92	0.08
		10	36.53	37.50	74.83	81.50	85.30	96.04	0.97	0.05
	Expanding	1	75.56	76.76	97.75	98.62	99.17	99.45	0.92	0.07
		10	47.80	48.55	75.40	84.36	89.46	93.47	0.96	0.06
SwedishLeaf	40	1	80.35	81.07	98.97	99.14	99.44	99.66	0.92	0.07
		10	37.83	38.34	63.75	75.32	95.32	96.37	0.94	0.07
	Expanding	1	75.48	76.21	98.75	99.14	99.43	99.66	0.92	0.07
		10	46.93	47.37	75.98	79.94	87.00	96.36	0.94	0.07
Strawberry	40	1	88.97	89.57	98.95	99.15	99.43	99.57	0.93	0.06
		10	58.24	58.81	79.89	83.69	92.07	98.34	0.94	0.07
	Expanding	1	81.59	82.01	98.71	99.10	99.39	99.57	0.92	0.06
		10	56.48	56.87	74.82	80.25	87.06	97.07	0.94	0.06
EOGHorizontalSignal	40	1	96.54	96.66	99.40	99.47	99.73	99.73	0.95	0.03
		10	87.61	87.70	99.08	99.21	99.29	99.68	0.91	0.08
	Expanding	1	91.14	91.41	99.51	99.60	99.73	99.73	0.95	0.03
		10	76.21	76.37	98.84	98.96	99.41	99.64	0.94	0.07
InsectWingbeatSound	40	1	84.65	85.02	95.34	96.19	97.37	97.55	0.95	0.05
		10	40.26	40.43	42.76	47.32	83.86	94.12	0.91	0.08
	Expanding	1	81.35	82.37	95.22	96.09	97.34	97.51	0.94	0.05
		10	56.81	57.76	58.49	62.43	78.73	93.44	0.89	0.09
ECG5000	40	1	80.28	81.18	93.49	95.34	97.07	97.45	0.94	0.05
		10	48.91	49.69	72.64	81.31	85.52	89.06	0.92	0.08
	Expanding	1	77.15	77.96	92.26	94.58	97.10	97.46	0.93	0.05
		10	48.08	48.70	48.08	48.08	65.45	95.64	0.94	0.07
UWaveGestureLibraryX	40	1	91.53	91.78	98.12	98.76	98.96	99.00	0.96	0.03
		10	63.98	64.13	90.70	93.67	94.20	98.10	0.93	0.06
	Expanding	1	84.28	84.98	98.14	98.66	98.91	98.95	0.96	0.03
		10	62.17	62.90	84.34	87.12	93.03	98.07	0.91	0.07
Yoga	40	1	92.71	93.15	95.44	96.69	97.21	97.28	0.96	0.03
		10	63.80	64.19	70.57	89.11	92.92	95.86	0.93	0.06
	Expanding	1	86.40	87.52	95.24	96.67	97.22	97.28	0.96	0.03
		10	66.35	67.81	78.69	85.21	90.92	96.40	0.89	0.08
Phoneme	40	1	64.27	64.80	96.33	97.55	97.58	97.58	0.97	0.02
		10	26.21	26.31	94.06	94.63	96.26	97.51	0.94	0.05
	Expanding	1	89.12	89.30	96.27	97.54	97.56	97.57	0.97	0.02
		10	71.51	71.63	90.47	92.92	96.34	97.53	0.92	0.06
Mallat	40	1	92.47	92.88	93.13	93.61	93.83	93.93	0.97	0.02
		10	64.16	64.43	66.39	69.39	82.86	86.82	0.93	0.05
	Expanding	1	90.10	90.92	90.75	91.07	91.25	91.30	0.97	0.02
		10	73.14	74.64	73.42	74.48	83.20	86.98	0.91	0.06
MixedShapesRegularTrain	40	1	95.22	95.45	97.94	98.56	98.67	98.68	0.98	0.02
		10	76.00	76.19	95.41	95.73	97.62	98.56	0.95	0.04
	Expanding	1	90.55	91.54	97.92	98.60	98.66	98.67	0.97	0.02
		10	74.45	75.95	96.22	96.93	98.06	98.58	0.92	0.06

Table 5.2: **Average solution hit %** upon interruptions by exact solution with **plurality vote** and at confidence thresholds.

	Time window		Intrpt.	Interruption at μ-$\sigma =$			
			w/Soln	0.98	0.95	0.85	0.70
Dataset	Width	Step					
PowerCons	40	1	100.00	95.09	94.37	92.59	93.50
		10	100.00	81.45	76.82	74.96	72.69
	Expanding	1	100.00	95.30	92.37	91.52	92.83
		10	100.00	86.23	78.26	59.24	71.29
SwedishLeaf	40	1	100.00	87.62	85.99	85.11	86.77
		10	100.00	65.59	48.40	31.89	43.41
	Expanding	1	100.00	86.75	83.18	85.16	90.46
		10	100.00	89.86	81.99	70.69	75.08
Strawberry	40	1	100.00	95.41	94.08	92.02	90.88
		10	100.00	81.01	74.57	70.05	70.21
	Expanding	1	100.00	91.09	83.02	83.79	86.02
		10	100.00	98.99	97.84	97.07	98.52
EOGHorizontalSignal	40	1	100.00	98.32	98.24	97.87	97.35
		10	100.00	61.40	61.16	59.22	54.82
	Expanding	1	100.00	99.35	98.46	96.32	95.76
		10	100.00	68.07	75.04	82.44	86.62
InsectWingbeatSound	40	1	100.00	93.38	92.65	91.37	91.29
		10	100.00	51.85	47.97	27.52	22.62
	Expanding	1	100.00	97.73	97.36	96.74	96.64
		10	100.00	100.00	99.55	96.14	74.95
ECG5000	40	1	100.00	99.85	99.79	99.64	99.62
		10	100.00	78.27	69.88	64.51	73.78
	Expanding	1	100.00	99.73	99.70	99.69	99.63
		10	100.00	100.00	100.00	98.80	99.00
UWaveGestureLibraryX	40	1	100.00	95.78	94.87	94.71	94.45
		10	100.00	44.81	44.18	40.47	39.88
	Expanding	1	100.00	98.25	98.20	98.19	97.88
		10	100.00	99.16	98.91	89.12	84.26
Yoga	40	1	100.00	99.49	98.32	98.02	98.04
		10	100.00	90.24	67.90	63.08	60.81
	Expanding	1	100.00	99.53	99.30	99.19	99.15
		10	100.00	99.38	98.96	98.23	92.51
Phoneme	40	1	100.00	92.17	91.38	91.38	91.38
		10	100.00	32.33	31.47	27.50	26.81
	Expanding	1	100.00	99.87	99.80	99.76	99.76
		10	100.00	99.85	99.86	96.97	91.51
Mallat	40	1	100.00	94.96	96.91	96.73	96.62
		10	100.00	80.15	73.71	60.22	53.70
	Expanding	1	100.00	94.00	97.80	97.95	97.95
		10	100.00	100.00	100.00	100.00	99.79
MixedShapesRegularTrain	40	1	100.00	98.73	98.15	98.07	98.08
		10	100.00	64.11	63.16	58.35	57.97
	Expanding	1	100.00	99.84	99.84	99.79	99.78
		10	100.00	99.75	99.59	99.38	95.44

Table 5.3: **Average gain** upon interruptions by exact solution with **distance-weighted vote** and at confidence thresholds.

	Time window		Unint.	Intrpt.	Interruption at μ-$\sigma =$				Efficiency	
Dataset	w	$step$		w/Soln	0.98	0.95	0.85	0.70	$\overline{\eta}$	$\overline{\sigma_\eta}$
PowerCons	40	1	71.52	74.50	97.37	98.44	99.16	99.46	0.92	0.08
		10	34.99	36.90	74.70	81.51	85.34	96.04	0.98	0.05
	Expanding	1	75.46	77.65	97.74	98.61	99.17	99.45	0.92	0.07
		10	47.70	48.88	75.52	84.37	89.47	93.47	0.96	0.07
SwedishLeaf	40	1	80.08	80.98	98.97	99.15	99.44	99.66	0.92	0.07
		10	36.19	36.84	63.58	75.35	95.32	96.37	0.94	0.07
	Expanding	1	75.78	76.43	98.79	99.14	99.43	99.66	0.92	0.07
		10	47.07	47.67	75.94	79.94	87.00	96.36	0.94	0.07
Strawberry	40	1	89.23	89.80	98.96	99.15	99.43	99.57	0.92	0.06
		10	58.44	59.11	79.65	83.54	92.11	98.35	0.94	0.07
	Expanding	1	81.58	82.14	98.72	99.10	99.39	99.57	0.92	0.06
		10	56.53	57.09	74.97	80.32	87.10	97.06	0.94	0.06
EOGHorizontalSignal	40	1	96.17	96.33	99.39	99.47	99.73	99.73	0.95	0.03
		10	86.76	86.90	99.08	99.21	99.29	99.68	0.91	0.08
	Expanding	1	91.17	91.53	99.51	99.60	99.73	99.73	0.95	0.03
		10	76.19	76.41	98.84	98.96	99.41	99.64	0.94	0.07
InsectWingbeatSound	40	1	85.37	85.89	95.47	96.29	97.43	97.61	0.95	0.05
		10	44.55	44.86	46.72	50.23	83.88	94.06	0.91	0.08
	Expanding	1	81.18	82.57	95.22	96.09	97.34	97.50	0.94	0.05
		10	55.99	57.27	57.48	61.76	78.49	93.54	0.89	0.09
ECG5000	40	1	79.92	80.82	93.53	95.37	97.08	97.45	0.94	0.05
		10	49.89	50.65	73.28	81.95	86.15	89.33	0.92	0.08
	Expanding	1	76.90	77.91	92.28	94.60	97.11	97.47	0.93	0.05
		10	47.88	48.65	47.88	47.88	65.30	95.58	0.94	0.07
UWaveGestureLibraryX	40	1	91.39	91.70	98.11	98.75	98.95	98.99	0.96	0.03
		10	65.08	65.27	90.73	93.68	94.21	98.10	0.93	0.06
	Expanding	1	84.32	85.26	98.15	98.66	98.91	98.95	0.96	0.03
		10	61.90	62.79	84.46	87.24	93.10	98.08	0.91	0.07
Yoga	40	1	92.35	92.84	95.38	96.68	97.20	97.27	0.96	0.03
		10	64.33	64.79	70.89	89.19	92.96	95.89	0.92	0.06
	Expanding	1	86.73	88.39	95.21	96.64	97.21	97.28	0.96	0.03
		10	66.55	67.95	78.80	85.33	90.96	96.43	0.89	0.08
Phoneme	40	1	55.54	56.09	96.26	97.53	97.56	97.57	0.97	0.02
		10	31.87	32.04	94.06	94.63	96.26	97.50	0.94	0.05
	Expanding	1	89.06	89.20	96.28	97.55	97.57	97.57	0.97	0.02
		10	71.70	71.79	90.55	93.02	96.41	97.53	0.92	0.06
Mallat	40	1	93.45	94.26	94.05	94.36	94.58	94.68	0.97	0.02
		10	66.37	67.15	68.31	70.85	82.98	86.93	0.93	0.05
	Expanding	1	89.70	90.54	90.42	90.78	90.96	91.01	0.97	0.02
		10	72.87	75.08	73.40	74.27	83.30	87.00	0.91	0.06
MixedShapesRegularTrain	40	1	95.47	95.72	97.95	98.56	98.67	98.68	0.98	0.02
		10	75.86	76.10	95.37	95.70	97.61	98.56	0.95	0.04
	Expanding	1	90.69	91.86	97.92	98.60	98.66	98.67	0.97	0.02
		10	74.65	76.27	96.22	96.93	98.06	98.58	0.92	0.06

Table 5.4: **Average solution hit %** upon interruptions by exact solution with **distance-weighted vote** and at confidence thresholds.

Dataset	Time window Width	Time window Step	Intrpt. w/Soln	Interruption at μ-$\sigma =$ 0.98	0.95	0.85	0.70
PowerCons	40	1	100.00	96.94	96.37	94.15	92.71
		10	100.00	80.65	72.47	73.37	67.51
	Expanding	1	100.00	96.35	94.22	91.07	90.02
		10	100.00	94.40	89.52	74.68	79.69
SwedishLeaf	40	1	100.00	90.71	89.01	87.20	90.75
		10	100.00	64.93	46.39	34.56	42.74
	Expanding	1	100.00	85.93	81.63	82.40	89.96
		10	100.00	89.35	80.27	69.18	78.13
Strawberry	40	1	100.00	94.80	93.24	90.48	89.09
		10	100.00	84.37	79.13	72.78	73.44
	Expanding	1	100.00	92.09	84.65	84.96	85.17
		10	100.00	99.16	97.98	96.92	98.89
EOGHorizontalSignal	40	1	100.00	98.79	98.78	98.50	98.01
		10	100.00	61.48	61.08	58.90	55.06
	Expanding	1	100.00	99.84	99.02	96.79	96.24
		10	100.00	69.61	76.49	84.27	87.19
InsectWingbeatSound	40	1	100.00	95.95	95.65	95.25	95.12
		10	100.00	63.46	66.91	33.83	28.49
	Expanding	1	100.00	98.38	98.08	97.92	97.82
		10	100.00	100.00	100.00	98.88	87.14
ECG5000	40	1	100.00	99.97	99.94	99.93	99.82
		10	100.00	81.04	70.52	64.54	74.18
	Expanding	1	100.00	100.00	100.00	100.00	99.99
		10	100.00	100.00	100.00	98.80	99.17
UWaveGestureLibraryX	40	1	100.00	97.46	96.64	96.43	96.20
		10	100.00	44.35	43.88	44.46	41.43
	Expanding	1	100.00	99.59	99.58	99.56	99.36
		10	100.00	99.00	98.96	89.94	84.26
Yoga	40	1	100.00	99.43	98.23	97.90	97.87
		10	100.00	88.46	69.30	64.24	63.67
	Expanding	1	100.00	99.94	99.92	99.81	99.80
		10	100.00	99.88	99.90	99.21	90.67
Phoneme	40	1	100.00	89.12	88.40	88.38	88.39
		10	100.00	35.14	33.41	30.36	29.86
	Expanding	1	100.00	99.95	99.92	99.91	99.91
		10	100.00	99.85	99.86	97.20	89.95
Mallat	40	1	100.00	97.95	97.64	97.93	97.96
		10	100.00	89.47	79.62	64.04	58.76
	Expanding	1	100.00	99.55	99.85	99.85	99.85
		10	100.00	100.00	100.00	100.00	100.00
MixedShapesRegularTrain	40	1	100.00	98.84	98.37	98.28	98.28
		10	100.00	63.70	63.46	59.40	58.89
	Expanding	1	100.00	99.87	99.86	99.80	99.81
		10	100.00	99.96	99.84	99.44	95.89

5.5 Approximate solution confidence

In CBR, confidence for a solution is an indicator of its correctness and it helps to judge how much we should trust that particular solution (Cheetham, 2000). For example, a classification solution would be more trustworthy when all kNN are very close to the query and they are all of the same class; compared to a classification where the solution is suggested by a simple majority vote among further neighbors. Many confidence indicators have been studied and corresponding confidence measures have been proposed by the CBR community (e.g. Cheetham and Price, 2004; Delany et al., 2005). Regarding classification, to the best of our knowledge, virtually all of them take into account similarities and/or solutions of exact neighbors of the query.

Accordingly, solution confidence becomes especially important when *ALK Classifier* provides an exact solution while not all kNN members are exact NNs. In this case, although the exact solution will remain the same, the solution confidence for best-so-far kNN will possibly be different from the confidence that would have been calculated for the exact kNN. For example, for a confidence measure that only takes into account the classes of kNN, the confidence value would fluctuate while approximate kNN are being replaced by other NNs of different classes during kNN search.

This section describes how we can transform some of the well-known confidence measures in CBR literature by incorporating *ALK Classifier*'s ability to provide the exact solution without the need to find all exact kNN. To give concrete examples, we will extend the measures in Delany et al. (2005)'s work that were suggested for a case-based spam e-mail filter.

While defining extensions to these confidence measures, we will assume that the target query t is classified exactly by the *ALK Classifier* and thus the solution class cannot change afterwards even if we continued the kNN search, as explained in section 5.1. The original versions of the measures that we extend use exact NNs of a given query and provide a single confidence value for the solution. However, when *ALK Classifier* terminates with an exact solution prior to finding all of the exact kNN, some of the kNN members will still be approximate NNs. Consequently, we will calculate the exact contribution of the already found exact NNs to the solution confidence. On the other hand, the remaining exact NNs that are still to-be-found will help us define the lower (LB) and upper-bound (UB) of the confidence value for each measure. In particular, we will be calculating the limits of the solution confidence range by taking into account the number and/or possible similarities of the still to-be-found exact NNs. For all these measures, a greater return value means a higher confidence.

5.5.1 Nomenclature

The following notation will be used in the definition of the extended confidence measures. Some terms are akin to the original nomenclature in (ibid.). Where t is the target query:

- An *already-found* exact NN of t is a NN whose rank within t's kNN is ascertained by *ALK Classifier* and cannot change afterwards;
- A *to-be-found* exact NN of t is a NN that is yet undecided and is still to be ascertained by *ALK Classifier*;
- A *like* neighbor of t is a neighbor sharing the same class with t;

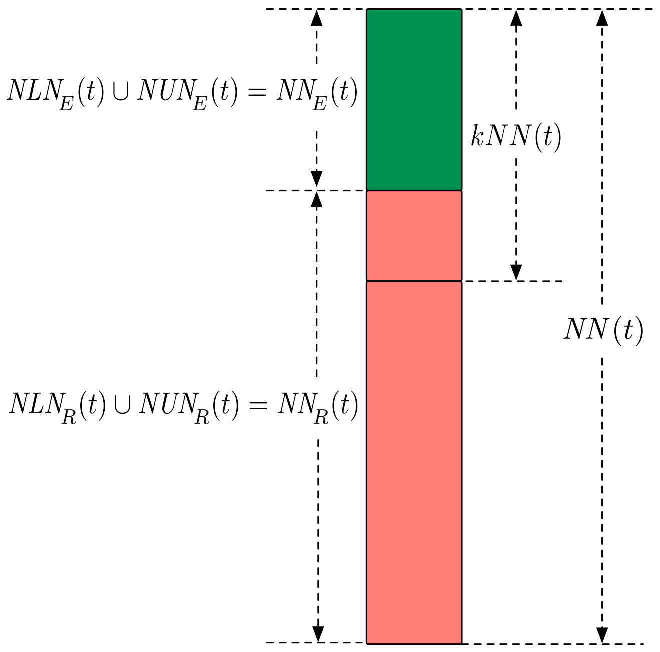

$NLN_E(t) \cup NUN_E(t) = NN_E(t)$
$kNN(t)$
$NN(t)$
$NLN_R(t) \cup NUN_R(t) = NN_R(t)$

Figure 5.2: **Neighbor sets for the target query *t*.** $NLN_E(t)$ and $NUN_E(t)$ are the *already-found like* and *unlike* exact neighbors of t that altogether guaranteed an exact solution before finding all of the exact kNN. $NLN_R(t)$ and $NUN_R(t)$ are the *to-be-found like* and *unlike* exact neighbors of t, and $NN(t)$ is the set of all evaluated kNN candidates for t.

– An *unlike* neighbor of t is a neighbor having a different class from t;

– $NLN_E(t)$ is the set of *already-found* exact *like* neighbors of t,

– $NUN_E(t)$ is the set of *already-found* exact *unlike* neighbors of t;

– $NN_E(t) = NLN_E(t) \cup NUN_E(t)$ is the set of all *already-found* exact neighbors of t which altogether guaranteed the exact solution and so the algorithm was terminated (possibly prior to completion of the kNN search);

– $NLN_R(t)$ is the set of still *to-be-found* exact *like* neighbors of t;

– $NUN_R(t)$ is the set of still *to-be-found* exact *unlike* neighbors of t;

– $NN_R(t) = NLN_R(t) \cup NUN_R(t)$ is the set of all *to-be-found* exact neighbors of t, and when the algorithm runs to completion, $NN_R(t) = \emptyset$;

– $NN(t) = NN_E(t) \cup NN_R(t)$ is the set of all evaluated kNN candidates for t in the CB by *ALK Classifier* so far, and when the algorithm runs to completion, $NN(t) \equiv NN_E(t)$;

– $kNN(t)$ is the set of best-so-far kNN of t, and when the algorithm runs to completion, all kNN are exact;

– All of the sets above are assumed to be sorted in descending order regarding the similarities of their contained cases to t;

- $NLN^i_E(t)$, $NUN^i_E(t)$, $NN^i_E(t)$, $NLN^i_R(t)$, $NUN^i_R(t)$, $NN^i_R(t)$ are the i^{th} members of each set respectively;
- $Index(a)$ is the index (i.e. ordinal rank) of case a in $NN(t)$;
- $d_{LB} \in [0, 1]$ is the normalized distance of the furthest *already-found* exact NN to t;
- $sim(a, b) \in [0, 1]$ is the calculated similarity between cases a and b;
- $1(a, b)$ is an indicator function that returns 1 if cases a and b are of the same class, or 0 otherwise;
- $min(v_1, v_2)$ returns the smaller of the two given values v_1 and v_2;
- κ is the number of neighbors used in a confidence measure and may optionally be different from the k parameter of *ALK Classifier*.

The sets mentioned above are depicted in Figure 5.2. In the original definitions of the measures that we will be extending in below subsections, the same letter k is used for both kNN search and for the number of neighbors taken into account by confidence measures. We find this usage confusing because of the fact that these two values may optionally be different as suggested by the authors. Especially, in the case of *ALK Classifier* where the kNN search may be interrupted before termination, this might lead to more confusion. Nevertheless, we stick to the original notation for the sake of coherence. However, we use the uppercase κ instead to mark the difference. Accordingly, while quoting an original definition, we replace k with κ and use the latter in the extended definition.

Avg NUN Index

"The Average Nearest Unlike Neighbour Index (Avg NUN Index) is a measure of how close the first κ NUNs are to the target case t" (Delany et al., 2005, p.180).

If *ALK Classifier* terminates with the exact solution before completion of the kNN search, only top $|NN_E(t)|$ number of kNN members will be exact and thus, only their ranks will be certain. Consequently, if $\kappa > |NN_E(t)|$, we will not have found κ exact unlike neighbors needed by this measure yet. Therefore, to be able to use $AvgNUNIndex$ as a solution confidence measure for *ALK Classifier*, we need to extend this measure by taking into account to-be-found exact neighbors. The extended definition is given in Eq. (5.1):

$$AvgNUNIndex(t, \kappa) = \frac{\sum_{i=1}^{min(\kappa, |NUN_E(t)|)} Index(NUN^i_E(t)) + \sum_{i=1}^{\kappa - |NUN_E(t)|} Index(NUN^i_R(t))}{\kappa} \tag{5.1}$$

In the left part of the dividend of the above equation, we sum the indices of the already-found unlike neighbors of t. Since, these neighbors are already ascertained by the incremental *ALK Classifier*, their contributions to the confidence value are fixed and will not change even if the kNN search is resumed. The second part of the dividend is the summation of the indices of the to-be-found unlike neighbors of t.

The $min(\kappa, |NUN_E(t)|)$ expression in the upper-bound of the left summation is to handle the occasions where $\kappa < k$. In this case, $|NUN_E(t)|$ might be bigger than the κ value, and this occasion would also make the second summation in the dividend obsolete.

While calculating the measure value for the more expected occasions where $|NUN_E(t)| < \kappa$, we need to refer to to-be-found unlike neighbors, i.e. $NUN_R(t)$. But, since to-be-found neighbors are still not ascertained, their exact contributions to confidence remains unknown. However, as mentioned previously, *ALK Classifier* allows us to define the lower and upper-bounds of their contributions. Specifically, for the lower-bound of $AvgNUNIndex$, all to-be-found neighbors in kNN (i.e. $kNN(t) - NN_E(t)$) are supposed to be unlike neighbors. And for its upper-bound, unlike neighbors are supposed to be the furthest cases to t in the CB. Formally:

$$\begin{aligned}
AvgNUNIndex(t,\kappa) &\xrightarrow{LB} Index(NUN_R^i(t)) = |NN_E(t)| + i, \forall i \leq \kappa - |NUN_E(t)|;\\
AvgNUNIndex(t,\kappa) &\xrightarrow{UB} Index(NUN_R^i(t)) = |CB| - i - 1, \forall i \leq \kappa - |NUN_E(t)|
\end{aligned}$$

Similarity Ratio

"The Similarity Ratio measure calculates the ratio of the similarity between the target case t and its κ NLNs to the similarity between the target case and its κ NUNs" (Delany et al., 2005, p.181). The extension for this measure is given in Eq. (5.2):

$$SimRatio(t,\kappa) = \frac{\sum_{i=1}^{min(\kappa,|NLN_E(t)|)} sim(t, NLN_E^i(t)) + \sum_{i=1}^{\kappa-|NLN_E(t)|} sim(t, NLN_R^i(t))}{\sum_{i=1}^{min(\kappa,|NUN_E(t)|)} sim(t, NUN_E^i(t)) + \sum_{i=1}^{\kappa-|NUN_E(t)|} sim(t, NUN_R^i(t))} \tag{5.2}$$

The lower-bound of $SimRatio$ is obtained by supposing the extreme case that the similarities of the rest of the κ like neighbors to-be-found are all zero and the rest of the κ unlike neighbors to-be-found have the same similarity with the already-found furthest exact neighbor. Accordingly, the upper-bound is obtained by assuming that the rest of the κ like neighbors to-be-found have the same similarity with the already-found furthest exact neighbor and that the similarities of the rest of the κ unlike neighbors to-be-found are near zero.[2] Formally:

$$\begin{aligned}
SimRatio(t,\kappa) \xrightarrow{LB} & \; sim(t, NLN_R^i(t)) = 0, \forall i \leq \kappa - |NLN_E(t)|,\\
& \; sim(t, NUN_R^i(t)) = 1 - d_{LB}, \forall i \leq \kappa - |NUN_E(t)|;\\
SimRatio(t,\kappa) \xrightarrow{UB} & \; sim(t, NLN_R^i(t)) = 1 - d_{LB}, \forall i \leq \kappa - |NLN_E(t)|,\\
& \; sim(t, NUN_R^i(t)) = \epsilon, \forall i \leq \kappa - |NUN_E(t)|
\end{aligned}$$

where $\epsilon \approx 0$.

[2] ϵ is used to avoid the case where $|NUN_E(t)| = 0$. In this case, $UB \to \infty$.

Sum of NLN Similarities

This measure corresponds to the "Sum of NN Similarities" in Delany et al.'s work and is defined as "the Sum of NN Similarities measure is the total similarity of the NLNs in the first k neighbours of the target case t" (Delany et al., 2005, p.182). The extended definition of this measure is in Eq. (5.3):

$$\begin{aligned} SumNLNSim(t,\kappa) = & \sum_{i=1}^{min(\kappa,|NN_E(t)|)} sim(t, NN_E^i(t)) 1(t, NN_E^i(t)) + \\ & \sum_{i=1}^{\kappa-|NN_E(t)|} sim(t, NN_R^i(t)) 1(t, NN_R^i(t)) \end{aligned} \tag{5.3}$$

For the lower-bound of this measure, we suppose that all of the kNN members to-be-found will be unlike neighbors. Conversely, for the upper-bound, we suppose that all of the kNN members to-be-found will be like neighbors and they will have the same similarity with the already-found furthest exact neighbor. Formally:

$$\begin{aligned} SumNLNSim(t,\kappa) \xrightarrow{LB} & NLN_R(t) \cap \{NN_R^i(t) : \forall i \leq \kappa - |NN_E(t)|\} = \emptyset; \\ SumNLNSim(t,\kappa) \xrightarrow{UB} & NUN_R(t) \cap \{NN_R^i(t) : \forall i \leq \kappa - |NN_E(t)|\} = \emptyset, \\ & sim(t, NLN_R^i(t)) = 1 - d_{LB}, \forall i \leq \kappa - |NN_E(t)| \end{aligned}$$

Similarity Ratio Within K

"The Similarity Ratio Within K is similar to the Similarity Ratio as described above except that, rather than consider the first κ NLNs and the first κ NUNs of a target case t, it only uses the NLNs and NUNs from the first κ neighbours" (ibid., p.181). The extended definition of this measure is given in Eq. (5.4):

$$SimRatioWithinK(t,\kappa) = \frac{SumNLNSim(t,\kappa)}{1 + SumNUNSim(t,\kappa)} \tag{5.4}$$

where $SumNLNSim(t,\kappa)$ is essentially the measure defined in Eq. 5.3 and,

$$\begin{aligned} SumNUNSim(t,\kappa) = & \sum_{i=1}^{min(\kappa,|NN_E(t)|)} sim(t, NN_E^i(t))(1 - 1(t, NN_E^i(t))) + \\ & \sum_{i=1}^{\kappa-|NN_E(t)|} sim(t, NN_R^i(t))(1 - 1(t, NN_R^i(t))) \end{aligned}$$

For the lower-bound of this measure, we suppose that all of the kNN members to-be-found will be unlike neighbors and they will have the same similarity with the already-found furthest exact neighbor. And for the upper-bound, we suppose that all of the kNN members to-be-found will be like neighbors and they will have the same similarity with the already-found furthest exact neighbor. Formally:

$$\begin{aligned}
SimRatioWithinK(t,\kappa) \xrightarrow{LB}\ & NLN_R(t) \cap \{NN_R^i(t) : \forall i \leq \kappa - |NN_E(t)|\} = \emptyset, \\
& sim(t, NUN_R^i(t)) = 1 - d_{LB}, \forall i \leq \kappa - |NN_E(t)| \\
SimRatioWithinK(t,\kappa) \xrightarrow{UB}\ & NUN_R(t) \cap \{NN_R^i(t) : \forall i \leq \kappa - |NN_E(t)|\} = \emptyset, \\
& sim(t, NLN_R^i(t)) = 1 - d_{LB}, \forall i \leq \kappa - |NN_E(t)|
\end{aligned}$$

Avg of NLN Similarity

This measure corresponds to the "Avg of NN Similarity" in Delany et al.'s work and is defined as "the Average NN Similarity measure is the average similarity of the NLNs in the first κ neighbours of the target case t" (Delany et al., 2005, p.182). The extended definition of this measure is given in Eq. (5.5):

$$AvgNLNSim(t,\kappa) = \frac{SumNLNSim(t,\kappa)}{\sum\limits_{i=1}^{min(\kappa,|NN_E(t)|)} 1(t, NN_E^i(t)) + \sum\limits_{i=1}^{\kappa-|NN_E(t)|} 1(t, NN_R^i(t))} \tag{5.5}$$

Since the sim measure that we use returns a normalized value $\in [0,1]$, having more like neighbors in $NLN_E(t)$ lowers the $AvgNLNSim$ value. This is because, additional like neighbors can have a maximum similarity of $1 - d_{LB}$ that is defined by the furthest already-found exact neighbor. Hence, as the divisor integer value increases, the result of the division decreases.

Therefore, contrary to $SumNLNSim$, for the lower-bound of $AvgNLNSim$, we suppose that all of the kNN members to-be-found will be like neighbors and they will have a similarity of zero to t. And for the upper-bound, we suppose that all of the kNN members to-be-found will be unlike neighbors. Formally:

$$\begin{aligned}
AvgNLNSim(t,\kappa) \xrightarrow{LB}\ & NUN_R(t) \cap \{NN_R^i(t) : \forall i \leq \kappa - |NN_E(t)|\} = \emptyset, \\
& sim(t, NLN_R^i(t)) = 0, \forall i \leq \kappa - |NN_E(t)|; \\
AvgNLNSim(t,\kappa) \xrightarrow{UB}\ & NLN_R(t) \cap \{NN_R^i(t) : \forall i \leq \kappa - |NN_E(t)|\} = \emptyset
\end{aligned}$$

5.6 Summary

In this chapter, we have extended *Anytime Lazy kNN* to be used as a classifier. *ALK Classifier* is able to suggest a solution for both approximate and exact kNN of a target query by using the parametric *reuse* method.

As another enhancement to its predecessor, *ALK Classifier* also benefits from the solution space and offers the option to interrupt the algorithm *upon guaranteeing the exact solution* without the need to find all exact kNN when possible. We showed that interruption with exact solution increases the gain of the algorithm compared to its uninterrupted runs. We also carried our experiments to look into *solution hits* when the algorithm is interrupted prior to guarantee the exact solution.

As a desired property of CBR systems, beside attaching *confidence* to best-so-far kNN like its predecessors, *ALK Classifier* also provides the *confidence* value for a suggested solution. Confidence for kNN are estimated by the algorithm itself as described in section 4.5. The solution confidence is calculated by a given parametric measure.

Although there are a multitude of solution confidence measures in CBR literature, to the best of our knowledge, they all take into account exact kNN. To tackle situations when *ALK Classifier* suggests a solution with best-so-far kNN instead, we gave formal examples of how we can transform some of the confidence measures in literature so that they can incorporate approximate NNs as well. Beside calculating the confidence value, the extended versions of these measures also provide lower and upper-bounds of solution confidence to better help the expert in his/her decision making. For example, if the solution confidence range is too wide, the expert has the option to resume *ALK Classifier*. Naturally, resuming the algorithm after interruption with exact solution would not change the solution itself. However, the confidence value would be more precise as more exact neighbors are found.

Chapter 6

Conclusions

Case-Based Reasoning (CBR) is a prominent methodology for solving problems in domains where reasoning is inherently based on remembering and adapting past experience. Since 1980's till today, CBR has proved its value with many successful applications in a vast spectrum of fields. Thanks to being a multifaceted paradigm, sometimes it served as a standalone intelligent system, other times it smoothly integrated as a component into a larger application.

Nevertheless, from its very early years, CBR has been facing the *faster performance vs richer knowledge* dilemma caused by growing case bases (CBs). As new problem-solving experiences are learned and retained in the case base, CBR is expected to give more accurate solutions. However, its lazy-learning approach to retrieving similar past problems for a present query considerably slows down the overall system performance. This slow-down is primarily caused by the increase in the computational cost of expensive similarity assessments with increased number of cases in the CB. The main approach of the CBR community that addressed this problem by an efficient control of CB growth is being questioned by CBR researchers due to the abundance of digital data today. And the approach has started to shift from avoiding large data towards benefiting from it. New techniques are recently being sought after to improve CBR's performance in large-scale case bases. We consider our thesis as a contribution to this line of research.

In our research, we focused on *temporal case bases*, a particular—yet increasingly important—type of case base where a sequence of cases captures the evolution of a phenomenon. Most common examples of temporal case bases are found in (but not limited to) medical CBR systems, one of the major application areas of the research field. We believe that the boom in the availability of data-producing devices will eventually make temporal case bases more common in many other fields; on the condition that CBR community finds efficient ways to deal with CBs of unprecedented scales.

Thanks to its seamless fit in CBR methodology, k-nearest neighbors (kNN) algorithm is the most used retrieval method in CBR. Alas, kNN's run-time complexity practically prohibits its use with big data when response time is a constraint. Current approaches to tackle this complexity are using data structures that partitions the search space a-priori and/or use approximation techniques. However, we showed how we can remarkably speed-up kNN search by exploiting the evolution of cases in temporal case bases without the need to a create and maintain an overall partitioning structure. Specifically, we demonstrated how we can identify true kNN candidates of a problem in *metric space*s and discard bulks of non-candidate cases all-at-once just by leveraging the kNN search history of predecessor cases of a present problem. To account for time critical applications where the speed-up for exact kNN search may not suffice and for applications where exact kNN are not even necessary, we addressed both *exact* and *approximate* kNN search by developing a fully-fledged *anytime* kNN search algorithm, namely *Anytime Lazy kNN* (*ALK*). *ALK* finds exact kNN of a query when it is allowed to run to termination. Or, if it is interrupted, it provides best-so-far kNN and a *confidence* value for each of these neighbors. The confidence of *ALK* is based on a probabilistic model and reflects the expected quality of an approximate neighbor regarding its similarity to the query compared to the exact neighbor. The confidence monotonically increases and we showed that this increase is more pronounced in the early stages of computation. This means that we get near-to-exact neighbors very quickly long before the termination of the algorithm. We also used our model to *automatize interruption* of the algorithm upon reaching *confidence thresholds*. The thresholds can be chosen with an adjustable optimism for model's estimation. Moreover, we devised a means to measure the *efficiency* of confidence estimation itself.

To have a platform-independent speed-up measure, we calculate the *gain* of *ALK* in terms of the percentage of avoided similarity calculations compared to a brute-force kNN in the CB. We evaluated *ALK* on numerous small to large-scale CBs with thousands to millions of cases that we generated out of publicly available time series datasets from diverse real-world domains. The results empirically demonstrate that *ALK* speeds up exact kNN search for all CBs of all datasets from a notable to remarkable level. And, when resorting to approximate kNN is an option or inevitable for time limitations despite the speed-up, *ALK* further boosts retrieval with superior speed-up even when the approximation is near exact kNN. One key observation in the results is that *ALK*'s gain is higher for the larger CBs generated from the same dataset. We also note that the experiments were conducted on generated CBs without any knowledge engineering applied to them. The distance *metric* was also the standard euclidean distance. With a carefully crafted metric which better discriminates the cases of an application domain, *ALK* could provide even higher speed-up.

Overall, the empirical results meet our very goals that we set for the speed-up in both exact and approximate kNN search. And, we believe the results strongly encourage the use of *ALK* in CBR applications with large-scale temporal case bases of millions of cases.

Furthermore, since most CBR systems are expected to suggest a solution for a problem and not just find similar past problems, we addressed the solution space for *classification* which is a common task for CBR systems in domains of our interest. For this purpose, we developed *ALK Classifier* as an extension to *ALK*. We showed that when an exact solution is demanded, *ALK Classifier* can further improve the gain of *ALK* by automatically terminating upon guaranteeing the exact solution. We provided the solution accuracy for classification with approximate kNN. Although our aim was not time series classification (which is a research field of its own) and we continued to use only the euclidean distance, the solution accuracy largely overlapped with the confidence in approximate kNN.

As *ALK* provides confidence for best-so-far kNN, we also extended some existing 'solution confidence' measures used in CBR literature for classification so that they can be used with approximate kNN. We formalized the extensions of these measure in order to provide a confidence range for the solution. To the best of our knowledge, these are the first examples to *approximate solution confidence* measures in CBR literature.

We believe our contribution is more than a set of fast algorithms. Our methodology both for the search for kNN candidates and for the building of anytime algorithm components are quite extendable. As long as the candidacy assessment is preserved, these components can be tailored for the domain of application. We also gave examples to alternative search for kNN candidates to show that even a domain-specific search method implementation can be integrated to our algorithms. We discussed how we can adjust the accuracy of confidence and also, in the case of automatic interruption, how we can set confidence thresholds in terms of an adjustable optimism with model's estimation. On the other hand, we mainly focused on its use in a CBR context, but *ALK* can be used as a standalone kNN algorithm in any domain that exhibits temporal relation between examples in a metric search space.

Complete source code of *ALK*, *ALK Classifier* and alternative search methods for kNN candidates together with simple instructions to launch the scripts to repeat all experiments and reproduce all plots and result tables that are detailed in this dissertation are publicly available at the online repository https://github.com/IIIA-ML/alk.

In the following two sections, we give recommendations for the application of our algorithms in a specific domain and directions for future work, especially to obtain further speed-up.

6.1 Recommendations for *ALK* practitioners

Following recommendations are for the use of *ALK* in a CBR application or as a standalone kNN algorithm. They include advice for extending our open source code. They are given for *ALK* but they hold for *ALK Classifier* as well.

Similarity metric As in any CBR system design, the choice of the similarity measure is highly important. The more discriminative it is between the cases, the less candidates will show up and thus, the faster will work the algorithm. The measure should be a true *metric* as a ***prerequisite*** for *ALK*.

Initial problem *ALK* speeds up search starting from the first consecutive problem in a problem sequence. The search for the initial problem in a sequence is parametric and any search function can be passed as an argument to *ALK*. This search can be an exhaustive exact search or it can be an approximate search. The only ***prerequisite*** is that the function should return all cases in the CB sorted regarding their exact or upper-bound of similarities to the query.

Curse of dimensionality kNN search in general is prone to this phenomenon. High dimensionality is not necessarily a curse per se, especially if the data forms dense clusters in the search space (see Beyer et al., 1999). On the other hand, it can occur in relatively low number of dimensions too. For a better performance of *ALK* on a particular CB, the problem space should be analyzed (e.g. by examining the distribution of distance between cases), and if necessary, *dimensionality reduction* and/or *temporal abstraction* techniques should be applied. In this case, *ALK* should work on reduced dimensions.

Space complexity *ALK* uses an instance of the internal data structure $RANK$ for every problem sequence to benefit from the kNN search history of predecessor problems. At any time, this structure has exactly $|CB|$ entries which are references to the cases and their similarities to past updates. $RANK$ should be saved to and loaded from a persistence layer. Furthermore, if the application allows concurrent multiple queries, in order to save space in Random-Access Memory, the $Stage$s in $RANK$ can be read sequentially from the persistence layer instead of loading whole $RANK$ into memory.

On the other hand, the Performance Distribution Profile (PDP) of *ALK* can occupy a large space on disk if very small intervals are used in the discretization of the quality and calculation ranges to obtain high precision in confidence estimation. In this case, it is better to use a *sparse matrix* instead of a regular multidimensional array for PDP representation.

Up-to-date Performance Distribution Profile Representativeness of the PDP of *ALK* in an application domain plays a key role in the accuracy of confidence estimation. As the CB grows, PDP is likely to become outdated and this may cause a decrease in the accuracy of estimations. To avoid this, PDP can be regenerated on a regular basis (e.g. when the CB grows by a certain percentage) or the efficiency of confidence estimations can be used as an indicator to decide when to update the PDP. If *efficiency* becomes $\gg 1$ on average although we are not being optimistic when choosing interruption thresholds, this means that PDP is giving overconfident estimations due to the fact that current size of the CB is larger than the CB used for building the PDP. Therefore, time has come to regenerate PDP for the current CB. Efficiency can be monitored as follows.

Efficiency monitoring This would be a nice example for Introspective CBR techniques. To measure the efficiency of confidence estimation for the approximate kNN of a query, when *ALK* is interrupted, it can later be resumed as an offline process to have the exact kNN in order to calculate the efficiency. These offline processes can be carried out for arbitrary or all queries, and the efficiency results can be saved and analyzed automatically.

Confidence thresholds The definition of an acceptable confidence for best-so-far kNN depends on the query, application and/or domain. Especially if *ALK* is to be interrupted automatically, confidence thresholds for interruptions should be chosen adequately. In any case, a good practice could be interrupting *ALK* at a lower confidence threshold, and if the approximate results are not good enough, the algorithm could be *resumed* and interrupted at a higher threshold until a desired quality in kNN is achieved within time limitations, if any. Interrupting and resuming *ALK* do not imply any significant computational overload.

6.2 Directions for future work

Planned applications of *ALK* We are planning to use *ALK* in IIIA-CSIC's healthcare projects *Innobrain* (2017) and *Play&Sing* (2018) when their data are available for use. Both projects will provide us with large-scale temporal case bases. For the latter project in particular, we are planning to integrate *ALK* in the AI platform that we have been developing for home-based music supported therapy of chronic stroke patients (Sanchez-Pinsach et al., 2019). There, while planning the next therapy session for a patient, *ALK* would be responsible for the fast retrieval of similar past sessions in the CB.

Other CBR tasks with *ALK* We want to implement a new extension of our algorithm, *ALK Regressor*, and test it for *regression* tasks. This new extension can be utilized in a forecasting application on medical records and/or other time series data.

Flexible time window choice In the current implementation of *ALK*, each problem sequence can use a different time window setting. However, when a time window choice is made for a particular problem sequence, it stays fixed during the evolution of that sequence. It would be interesting to explore the possibilities to apply a flexible time window setting for the queries throughout the updates of the same sequence. This would give the expert an option to apply a different time window of choice for each update. Besides, he/she could run *ALK* for the same query with different time window settings and compare the results before a final decision. We have not explored this idea yet but we note it here as food for thought.

Better accuracy for confidence estimation A way to increase both the accuracy and efficiency of confidence estimation for best-so-far kNN can be to discretize the calculation range of PDP logarithmically. Thus, more importance would be attributed to the beginning of the calculation range given the fact that NNs are actually found by *ALK* in the early stages of execution for the domains of our interest.

Further speed-up by parallelization *ALK* can be easily converted to a multithreaded application (or a multiproccessing one for that matter). A separate thread can be dedicated to iterate a unique $Stage$ in $RANK$, or multiple threads can iterate over the same $Stage$ simultaneously. In the latter option, when a thread encounters a non-candidate case, others should be warned gracefully in order to jump to the next $Stage$ all together.

Further speed-up by a different $RANK$ design The *triangle inequality* helps us to define an upper-bound of similarity to identify true kNN candidates of a query. However, this upper-bound does not bear any sense of direction. More specifically, the Δ between two consecutive problems does not help us to judge if a neighbor of a prior problem has become closer or further to the current problem. We want to explore other representations of $RANK$ that might give us a sense of direction of change between sequence updates so that we can use them in discarding more non-candidates in search. In particular, we are interested in exploring *proximity graphs* (e.g. Toussaint, 2005) for this purpose.

Further speed-up by introspection *Introspective CBR* systems use metareasoning (Cox and Raja, 2011) to improve their performances (e.g. Arcos, Mülâyim, and Leake, 2011). As a proactive introspection approach, we previously investigated *dubious future problems* (DFPs) for a CBR system. A DFP is a synthetically generated problem near existing cases in a CB and it yields a *low solution confidence* (Mülâyim and Arcos, 2007). DFPs are created by an evolutionary algorithm using domain ontology in order to predict possible future reasoning deficiencies of a CBR system. Each DFP holds the information of its neighbor cases and their similarities to it, and the cases reciprocally hold the information of their neighbor DFPs. Neighborhood of a DFP is defined by a given similarity threshold δ. Likewise, Map of Dubious Regions (*MDR*) is a graph where the vertices are DFPs and the edges connect neighbor DFPs that are within a given similarity threshold δ' to each other (Mülâyim and Arcos, 2008; Mülâyim and Arcos, 2010).

We can exploit *MDR* to further speed up *ALK* in the search of kNN candidates as follows. During the kNN search for a target query t, let us assume that we evaluate a kNN candidate case A which happens to be a neighbor of a DFP D. Then, for candidacy assessment, instead of continuing the search for the next kNN candidate in the current `Stage` in $RANK$, we can give priority to the cases within the same DFP neighborhood with A. By giving the priority to these cases, we hope that some of them will be closer to t than A as we already know that they are in the same neighborhood of a DFP. For example, if C is one of D's neighbor cases and it is a true kNN candidate—$UB(C, t) > sim(A, t)$ (see Definition 3.1), we calculate C's actual similarity to t—$sim(C, t)$. If C proves to be closer to t than A, then the kNN candidacy threshold increases and this may result in having fewer future candidates. Thus, the *gain* of *ALK* may increase.

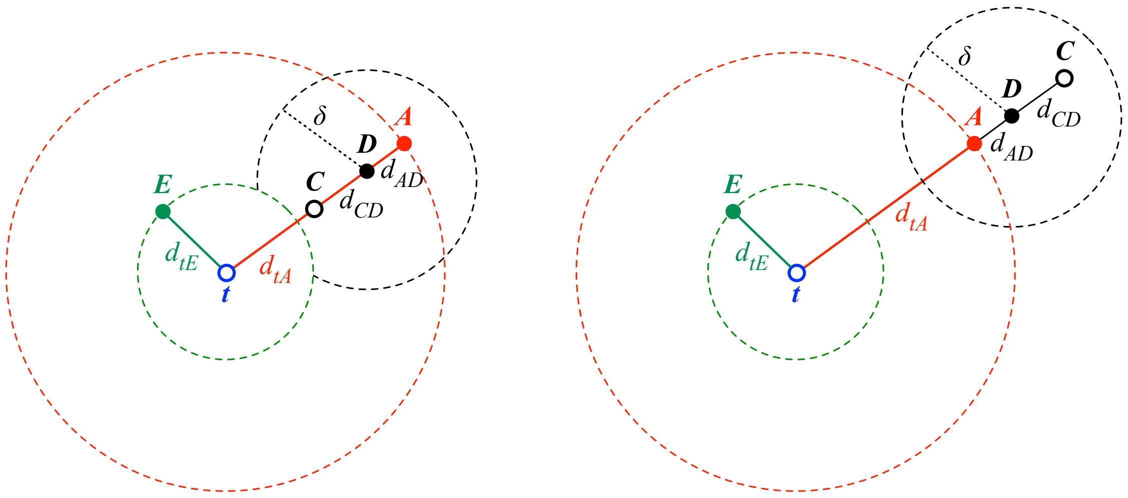


Figure 6.1: **Exploiting dubiosity neighborhood**. E is the furthest exact neighbor and A is the closest approximate neighbor of the target query t. D is a *dubious future problem* (DFP) that was previously found in the vicinity of A. And C is a case which shares the neighborhood of D—defined by δ—with A. d_{AD} and d_{CD} distances are known from the *Map of Dubious Regions* that holds the DFPs and their neighbors. d_{tE} marks the lower-bound of distance for the remaining exact NNs. The left and right images show the closest and furthest points that C can be to t respectively.

Figure 6.1 illustrates an example for the closest and furthest possible distances of C to the target query t where there is at least one already-found *exact* NN E. Due to the incremental nature of *ALK*, the distance of E to t—d_{tE}—marks the lower-bound of distance for the remaining exact NNs. While searching for the second NN, we find an *approximate* NN A. MDR indicates that A has a DFP neighbor D and that C lies in the same neighborhood. MDR also gives us the distances d_{AD} and d_{CD} in Figure 6.1. Although we do not know where A and C exactly fall inside D's neighborhood circle, we can still estimate the lower and upper-bound of the distance of C to t (i.e. d_{tC}) as follows:

$$max(d_{tE},\ d_{tA} - d_{AD} - d_{CD}) \leq d_{tC} \leq d_{tA} + d_{AD} + d_{CD}$$

If we want to be fairer with all candidates, before accessing C's information in $RANK$, we can compare the lower-bound of d_{tC} to the lower-bound of distance of the next candidate in $RANK$ derived from the Definition 3.1, and evaluate the actual similarity of the better candidate among these two.

To be able to access to C within $RANK$ for candidacy assessment without a time overload, we can maintain a hash table $RANK\_HASH$ as an attribute of the `AnytimeLazyKNN` class. This would be the same hash table that we used for the *Exploit Approaching Candidates* alternative $RANK$ iteration explained in section 4.9.

This line of investigation poses two engineering challenges. In our previous research, DFPs were found for CBR applications with rather small case bases ($<$ 1,000 cases). The evolutionary algorithm that we had implemented to find DFPs should now be scaled to work with millions of cases. Also, an efficient way to maintain $RANK\_HASH$ structure should be ensured as previously mentioned for the *Exploit Approaching Candidates* iteration.

Glossary

Case *see* Definition 2.6. 16

Evaluating a case Calculating the similarity of the case to the target query. 23

kNN candidate *see* Definition 3.2. 23

Lazy evaluation of a case Evaluating a case only when the case is deemed a kNN candidate. 23, *see also* Evaluating a case

Problem *see* Definition 2.4. 16

Problem sequence *see* Definition 2.1. 15

Query *see* Definition 2.5. 16

Sequence . *see* Problem sequence

Temporal case base *see* Definition 2.8. 16

Temporally related cases *see* Definition 2.7. 16

Time window *see* Definition 2.3. 15

Update *see* Definition 2.2. 15

Bibliography

Acorn, Timothy L and Sherry H Walden (1992). “SMART: Support Management Automated Reasoning Technology for Compaq Customer Service”. In: *Proceedings of the Fourth Conference on Innovative Applications of Artificial Intelligence*. IAAI’92. San Jose, California: AAAI Press, pp. 3–18 (cit. on p. 10).

Agnar, Aamodt and Enric Plaza (1994). “Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches”. In: *AI Communications* 7.1, pp. 39–59. DOI: 10.3233/AIC-1994-7104 (cit. on pp. 8, 9).

Aha, David W., ed. (1997). *Lazy Learning*. Dordrecht: Springer Netherlands. DOI: 10.1007/978-94-017-2053-3 (cit. on pp. 4, 8).

Aha, David W., Dennis Kibler, and Marc K. Albert (1991). “Instance-based learning algorithms”. In: *Machine Learning* 6.1, pp. 37–66. DOI: 10.1007/BF00153759 (cit. on pp. 4, 11).

Andoni, Alexandr and Piotr Indyk (2006). “Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions”. In: *IEEE Symposium on Foundations of Computer Science*. IEEE, pp. 459–468. DOI: 10.1109/FOCS.2006.49 (cit. on p. 13).

Arcos, Josep Lluís, Ramon López De Mántaras, and Xavier Serra (1998). “SaxEx: A case-based reasoning system for generating expressive musical performances”. In: *Journal of New Music Research* 27.3, pp. 194–210. DOI: 10.1080/09298219808570746 (cit. on p. 10).

Arcos, Josep Lluís, Mehmet Oğuz Mülâyim, and David B. Leake (2011). “Using Introspective Reasoning to Improve CBR System Performance”. In: *Metareasoning: Thinking about Thinking*. Ed. by Michael T Cox and Anita Raja. The MIT Press. Chap. 11, pp. 167–182. DOI: 10.7551/mitpress/9780262014809.003.0011 (cit. on p. 86).

Arcos, Josep Lluís and Enric Plaza (1997). “Noos: an integrated framework for problem solving and learning”. In: *Knowledge Engineering: Methods and Languages* (cit. on p. 10).

Arefin, Ahmed Shamsul, Carlos Riveros, Regina Berretta, and Pablo Moscato (2012). “GPU-FS-kNN: A Software Tool for Fast and Scalable kNN Computation Using GPUs”. In: *PLoS ONE* 7.8. DOI: 10.1371/journal.pone.0044000 (cit. on p. 13).

Armengol, Eva, A. Palaudàries, and Enric Plaza (2001). “Individual Prognosis of Diabetes Long-term Risks: A CBR Approach”. In: *Methods of Information in Medicine* 40.01, pp. 46–51. DOI: 10.1055/s-0038-1634463 (cit. on p. 10).

Baccigalupo, Claudio and Enric Plaza (2007). “A Case-Based Song Scheduler for Group Customised Radio”. In: *Case-Based Reasoning Research and Development*. Ed. by Rosina O Weber and Michael M Richter. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 433–448. DOI: 10.1007/978-3-540-74141-1_30 (cit. on p. 10).

Bagnall, Anthony, Jason Lines, William Vickers, and Eamonn Keogh (2018). *The UEA & UCR Time Series Classification Repository*. URL: http://www.timeseriesclassification.com (visited on 07/14/2020) (cit. on p. 29).

Begum, Shahina, Mobyen Uddin Ahmed, Peter Funk, Ning Xiong, and Mia Folke (2011). “Case-based reasoning systems in the health sciences: A survey of recent trends and developments”. In: *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews* 41.4, pp. 421–434. DOI: 10.1109/TSMCC.2010.2071862 (cit. on pp. 3, 10).

Bellman, Richard (1957). *Dynamic Programming*. 1st ed. Princeton, NJ, USA: Princeton University Press (cit. on p. 12).

Bentley, Jon Louis (1975). “Multidimensional binary search trees used for associative searching”. In: *Communications of the ACM* 18.9, pp. 509–517. DOI: 10.1145/361002.361007 (cit. on p. 13).

Beyer, Kevin, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft (1999). “When Is “Nearest Neighbor” Meaningful?” In: *Database Theory — ICDT’99*. Ed. by Catriel Beeri and Peter Buneman. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 217–235 (cit. on pp. 13, 84).

Bichindaritz, Isabelle and E. Conlon (1996). “Temporal knowledge representation and organization for case-based reasoning”. In: *Proceedings Third International Workshop on Temporal Representation and Reasoning (TIME ’96)*, pp. 152–159. DOI: 10.1109/TIME.1996.555694 (cit. on p. 13).

Bichindaritz, Isabelle and Cindy Marling (2010). “Case-Based Reasoning in the Health Sciences: Foundations and Research Directions”. In: *Computational Intelligence in Healthcare 4*. Berlin Hei. Vol. 309. Springer-Verlag, pp. 127–157. DOI: 10.1007/978-3-642-14464-6_7 (cit. on p. 3).

Boddy, Mark and Thomas Dean (1989). “Solving Time-Dependent Planning Problems”. In: *Eleventh International Joint Conference on Artificial Intelligence*, pp. 979–984 (cit. on p. 18).

Bozkaya, Tolga and Meral Ozsoyoglu (1997). “Distance-based indexing for high-dimensional metric spaces”. In: *Proceedings of the 1997 ACM SIGMOD international conference on Management of data - SIGMOD ’97*. SIGMOD ’97. New York, USA: ACM Press, pp. 357–368. DOI: 10.1145/253260.253345 (cit. on p. 13).

Bridge, Derek, Mehmet H. Göker, Lorraine McGinty, and Barry Smyth (2005). “Case-based recommender systems”. In: *The Knowledge Engineering Review* 20.3, pp. 315–320. DOI: 10.1017/S0269888906000567 (cit. on p. 10).

Broder, Alan J. (1990). “Strategies for efficient incremental nearest neighbor search”. In: *Pattern Recognition* 23.1-2, pp. 171–178. DOI: 10.1016/0031-3203(90)90057-R (cit. on p. 39).

Burke, Robin (1999). “The Wasabi Personal Shopper: A Case-Based Recommender System”. In: *Proceedings of the 11th Conference on Innovative Applications of Artificial Intelligence*. AAAI ’99/IAAI ’99. Menlo Park, CAx: AAAI Press, pp. 844–849 (cit. on p. 10).

Chávez, Edgar, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín (2001). “Searching in Metric Spaces”. In: *ACM Computing Surveys* 33.3, pp. 273–321. DOI: 10.1145/502807.502808 (cit. on pp. 4, 12).

Cheetham, William (2000). “Case-Based Reasoning with Confidence”. In: *European Workshop on Advances in Case-Based Reasoning*. 1898. Springer, pp. 15–25. DOI: 10.1007/3-540-44527-7_3 (cit. on pp. 38, 46, 63, 75).

Cheetham, William and Joseph Price (2004). “Measures of Solution Accuracy in Case-Based Reasoning Systems”. In: *European Conference on Case-Based Reasoning*, pp. 106–118. DOI: 10.1007/978-3-540-28631-8_9 (cit. on p. 75).

Cheetham, William and Ian Watson (2005). “Fielded applications of case-based reasoning”. In: *The Knowledge Engineering Review* 20.3, pp. 321–323. DOI: 10.1017/S0269888906000580 (cit. on p. 10).

Cover, Thomas M. and P. E. Hart (1967). “Nearest Neighbor Pattern Classification”. In: *IEEE Transactions on Information Theory* 13.1, pp. 21–27. DOI: 10.1109/TIT.1967.1053964 (cit. on pp. 1, 11).

Cox, Michael T. and Anita Raja, eds. (2011). *Metareasoning: Thinking about Thinking*. The MIT Press. DOI: 10.7551/mitpress/9780262014809.001.0001 (cit. on p. 86).

Cunningham, Pádraig, Barry Smyth, and Andrea Bonzano (1998). “An incremental retrieval mechanism for case-based electronic fault diagnosis”. In: *Knowledge-Based Systems* 11.3-4, pp. 239–248. DOI: 10.1016/S0950-7051(97)00049-X (cit. on p. 40).

Dean, Jeffrey and Sanjay Ghemawat (2004). “MapReduce: Simplified data processing on large clusters”. In: *OSDI 2004 - 6th Symposium on Operating Systems Design and Implementation*. USENIX Association, pp. 137–149 (cit. on p. 12).

Dean, Thomas and Mark Boddy (1988). “An Analysis of Time-Dependent Planning”. In: *Seventh AAAI National Conference on Artificial Intelligence*. Vol. 88. AAAI, pp. 49–54 (cit. on pp. 2, 13).

Delany, Sarah Jane, Pádraig Cunningham, Dónal Doyle, and Anton Zamolotskikh (2005). “Generating estimates of classification confidence for a case-based spam filter”. In: *International Conference on Case-Based Reasoning*. Ed. by Héctor Muñoz-Ávila and Francesco Ricci. Vol. 3620 LNAI. Berlin, Heidelberg: Springer, pp. 177–190. DOI: 10.1007/11536406_16 (cit. on pp. 75, 77–80).

Deza, Michel Marie and Elena Deza (2009). *Encyclopedia of distances*, pp. 1–590. DOI: 10.1007/978-3-642-00234-2 (cit. on p. 14).

Díaz-Agudo, Belén, Pedro A González-Calero, Juan A Recio-García, and Antonio A Sánchez-Ruiz-Granados (2007). “Building CBR systems with jCOLIBRI”. In: *Science of Computer Programming* 69.1-3, pp. 68–75. DOI: 10.1016/j.scico.2007.02.004 (cit. on p. 10).

Eastman, C. M. and S. F. Weiss (1982). “Tree structures for high dimensionality nearest neighbor searching”. In: *Information Systems* 7.2, pp. 115–122. DOI: 10.1016/0306-4379(82)90023-0 (cit. on p. 12).

Francis, Anthony G. and Ashwin Ram (1993). “The Utility Problem in Case-Based Reasoning”. In: *AAAI Case-Based Reasoning Workshop*. Washington DC: AAAI Press, p. 160 (cit. on pp. 1, 11).

Funk, Peter and Ning Xiong (2006). “Case-based reasoning and knowledge discovery in medical applications with time series”. In: *Computational Intelligence* 22.3-4, pp. 238–253. DOI: 10.1111/j.1467-8640.2006.00286.x (cit. on p. 4).

Garcia, Vincent, Eric Debreuve, and Michel Barlaud (2008). “Fast k nearest neighbor search using GPU”. In: *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops* 2, pp. 1–6. DOI: 10.1109/CVPRW.2008.4563100 (cit. on p. 13).

Gervás, Pablo, Belén Díaz-Agudo, Federico Peinado, and Raquel Hervás (2005). “Story Plot Generation based on CBR”. In: *Applications and Innovations in Intelligent Systems XII*. Ed. by Ann Macintosh, Richard Ellis, and Tony Allen. London: Springer London, pp. 33–46 (cit. on p. 10).

Goel, Ashok K. and Belén Díaz-Agudo (2017). “What’s Hot in Case-Based Reasoning”. In: *Thirty-First AAAI Conference on Artificial Intelligence*. AAAI Press (cit. on pp. 1, 3, 10, 12).

Grass, Joshua (1996). “Reasoning about computational resource allocation”. In: *XRDS: Crossroads, The ACM Magazine for Students* 3.1, pp. 16–20. DOI: 10.1145/332148.332154 (cit. on p. 18).

Grass, Joshua and Shlomo Zilberstein (1996). “Anytime algorithm development tools”. In: *ACM SIGART Bulletin* 7.2, pp. 20–27. DOI: 10.1145/242587.242592 (cit. on p. 18).

Hajebi, Kiana, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang (2011). “Fast Approximate Nearest-Neighbor Search with k-Nearest Neighbor Graph”. In: *International Joint Conference on Artificial Intelligence*. AAAI Press, pp. 1312–1317. DOI: 10.5591/978-1-57735-516-8/IJCAI11-222 (cit. on p. 13).

Hinkle, David and Christopher N. Toomey (1994). “Clavier: Applying Case-Based Reasoning to Composite Part Fabrication”. In: *Innovative Applications of Artificial Intelligence*. AAAI, pp. 55–62 (cit. on p. 10).

Indyk, Piotr and Rajeev Motwani (1998). “Approximate nearest neighbors”. In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing - STOC ’98*. New York, New York, USA: ACM Press, pp. 604–613. DOI: 10.1145/276698.276876 (cit. on p. 12).

Innobrain (2017). *New Technologies for the Innovation in Cognitive Stimulation and Rehabilitation*. URL: https://iiia.csic.es/research/project/?project_id=6 (visited on 08/31/2020) (cit. on pp. ix, 85).

Jære, Martha Dørum, Agnar Aamodt, and Pål Skalle (2002). “Representing Temporal Knowledge for Case-Based Prediction”. In: *European Conference on Case-Based Reasoning*. Springer, Berlin, Heidelberg, pp. 174–188. DOI: 10.1007/3-540-46119-1_14 (cit. on p. 13).

Jalali, Vahid and David B. Leake (2015). “CBR Meets Big Data: A Case Study of Large-Scale Adaptation Rule Generation”. In: *International Conference on Case-Based Reasoning*. Vol. 9343. Springer, pp. 181–196. DOI: 10.1007/978-3-319-24586-7_13 (cit. on pp. 2, 12).

Jiang, Fei, Yong Jiang, Hui Zhi, Yi Dong, Hao Li, Sufeng Ma, Yilong Wang, Qiang Dong, Haipeng Shen, and Yongjun Wang (2017). “Artificial intelligence in healthcare: Past, present and future”. In: *Stroke and Vascular Neurology* 2.4, pp. 230–243. DOI: 10.1136/svn-2017-000101 (cit. on p. 3).

Juarez, Jose M., Susan Craw, J. Ricardo Lopez-Delgado, and Manuel Campos (2018). “Maintenance of case bases: Current algorithms after fifty years”. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, {IJCAI-18}*. International Joint Conferences on Artificial Intelligence Organization, pp. 5457–5463. DOI: 10.24963/ijcai.2018/770 (cit. on p. 11).

Jurisica, Igor, Janice Glasgow, and John Mylopoulos (2000). “Incremental Iterative Retrieval and Browsing for Efficient Conversational CBR Systems”. In: *Applied Intelligence* 12.3, pp. 251–268. DOI: 10.1023/A:1008375309626 (cit. on p. 40).

Kibriya, Ashraf M. and Eibe Frank (2007). “An Empirical Comparison of Exact Nearest Neighbour Algorithms”. In: *Knowledge Discovery in Databases: PKDD 2007*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 140–151. DOI: 10.1007/978-3-540-74976-9_16 (cit. on p. 12).

Kolodner, Janet L. (1988). “Retrieving events from a case memory: A parallel implementation”. In: *Proc. of 1988 Case-Based Reasoning*, pp. 233–249 (cit. on p. 11).

— (1994). “Understanding creativity: A case-based approach”. In: *Topics in Case-Based Reasoning*. Ed. by Stefan Wess, Klaus-Dieter Althoff, and Michael M Richter. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–20 (cit. on p. 10).

— (1996). “Making the Implicit Explicit: Clarifying the Principles of Case-Based Reasoning”. In: *Case-Based Reasoning: Experiences, Lessons & Future Directions*. Ed. by David B. Leake. Menlo Park: The AAAI Press/The MIT Press. Chap. 2, pp. 349–370 (cit. on p. 2).

Kolodner, Janet L., Michael T. Cox, and Pedro A. González-Calero (2005). “Case-based reasoning-inspired approaches to education”. In: *The Knowledge Engineering Review* 20.3, pp. 299–303. DOI: 10.1017/S0269888906000634 (cit. on p. 10).

Kolodner, Janet L. and David B. Leake (1996). "A Tutorial Introduction to Case-Based Reasoning". In: *Case-Based Reasoning: Experiences, Lessons & Future Directions*. Ed. by David B. Leake. Menlo Park: The AAAI Press/The MIT Press. Chap. 2, pp. 31–66 (cit. on pp. 3, 8).

Kranen, Philipp and Thomas Seidl (2009). "Harnessing the strengths of anytime algorithms for constant data streams". In: *Data Mining and Knowledge Discovery* 19.2, pp. 245–260. DOI: 10.1007/s10618-009-0139-0 (cit. on p. 13).

Kwiatkowska, Mila and Margaret Stella Atkins (2004). "Case representation and retrieval in the diagnosis and treatment of obstructive sleep apnea: a semio-fuzzy approach". In: *Proceedings of the Seventh European Conference on Case-Based Reasoning*. Madrid, Spain, pp. 25–35 (cit. on p. 10).

Leake, David B., ed. (1996a). *Case-Based Reasoning: Experiences, Lessons & Future Directions*. 1st. Cambridge, MA, USA: MIT Press (cit. on p. 10).

— (1996b). "CBR in Context: The Present and Future". In: *Case-Based Reasoning: Experiences, Lessons & Future Directions*. Ed. by David B. Leake. Menlo Park: The AAAI Press/The MIT Press. Chap. 1, pp. 3–30 (cit. on pp. 3, 8, 9).

Leake, David B., Barry Smyth, David C. Wilson, and Qiang Yang (2001). "Introduction To the Special Issue on Maintaining Case-Based Reasoning Systems". In: *Computational Intelligence* 17.2, pp. 193–195. DOI: 10.1111/0824-7935.00139 (cit. on p. 11).

López De Mántaras, Ramon, David McSherry, Derek Bridge, David B. Leake, Barry Smyth, Susan Craw, Boi Faltings, Mary Lou Maher, Michael T. Cox, Kenneth Forbus, Mark Keane, Agnar Aamodt, and Ian Watson (2005). "Retrieval, Reuse, Revision and Retention in Case-based Reasoning". In: *Knowledge Engineering Review* 20.3, pp. 215–240. DOI: 10.1017/S0269888906000646 (cit. on p. 8).

López, Beatriz and Enric Plaza (1993). "Case-based planning for medical diagnosis". In: *Methodologies for Intelligent Systems*. Ed. by Jan Komorowski and Zbigniew W Raś. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 96–105. DOI: 10.1007/3-540-56804-2_10 (cit. on p. 10).

López, Beatriz, Carles Pous, Pablo Gay, Albert Pla, Judith Sanz, and Joan Brunet (2011). "eXiT*CBR: A framework for case-based medical diagnosis development and experimentation". In: *Artificial Intelligence in Medicine* 51.2, pp. 81–91. DOI: 10.1016/j.artmed.2010.09.002 (cit. on p. 10).

Mark, William, Evangelos Simoudis, and David Hinkle (1996). "Case-Based Reasoning: Expectations and Results". In: *Case-Based Reasoning: Experiences, Lessons, and Future Directions*. Ed. by David B. Leake. AAAI Press/MIT Press, Menlo Park, CA. Chap. 14, pp. 269–294 (cit. on pp. 10, 11).

Marling, Cindy, Jay Shubrook, and Frank Schwartz (2008). "Case-Based Decision Support for Patients with Type 1 Diabetes on Insulin Pump Therapy". In: *Advances in Case-Based Reasoning*. Ed. by Klaus-Dieter Althoff, Ralph Bergmann, Mirjam Minor, and Alexandre Hanft. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 325–339. DOI: 10.1007/978-3-540-85502-6_22 (cit. on p. 10).

McSherry, David (2002). "Diversity-Conscious Retrieval". In: *Advances in Case-Based Reasoning*. Ed. by Susan Craw and Alun Preece. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 219–233. DOI: 10.1007/3-540-46119-1_17 (cit. on p. 10).

Montani, Stefania, Riccardo Bellazzi, Luigi Portinale, Stefano Fiocchi, and Mario Stefanelli (1998). "A Case-Based Retrieval System for Diabetic Patients Therapy". In: *Proceedings of IDAMAP*, pp. 64–70 (cit. on p. 15).

Montani, Stefania and Luigi Portinale (2006). “Accounting for the temporal dimension in case-based retrieval: A framework for medical applications”. In: *Computational Intelligence* 22.3-4, pp. 208–223. DOI: 10.1111/j.1467-8640.2006.00284.x (cit. on pp. 2, 4, 14, 17).

Mueen, Abdullah, Eamonn Keogh, Qiang Zhu, Sydney Cash, and Brandon Westover (2009). “Exact Discovery of Time Series Motifs”. In: *SIAM International Conference on Data Mining*. Philadelphia, PA: Society for Industrial and Applied Mathematics, pp. 473–484. DOI: 10.1137/1.9781611972795.41 (cit. on pp. 15, 31).

Muja, Marius and David G Lowe (2014). “Scalable Nearest Neighbor Algorithms for High Dimensional Data”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36.11, pp. 2227–2240. DOI: 10.1109/TPAMI.2014.2321376 (cit. on p. 13).

Mülâyim, Mehmet Oğuz and Josep Lluís Arcos (2007). “Exploring Dubious Future Problems”. In: *UK Workshop on Case-Based Reasoning*. Ed. by Miltos Petridis. CMS Press, University of Greenwich, pp. 52–63 (cit. on p. 86).

— (2008). “Understanding Dubious Future Problems”. In: *Advances in Case-Based Reasoning, ECCBR 2008*. Ed. by Klaus-Dieter Althoff, Ralph Bergmann, Mirjam Minor, and Alexandre Hanft. Vol. LNCS, 5239. Springer Berlin Heidelberg, pp. 385–399. DOI: 10.1007/978-3-540-85502-6_26 (cit. on p. 86).

— (2010). “Predicting Dubiosity in CBR Systems”. In: *Expert UPDATE* 10.2, pp. 1–8 (cit. on p. 86).

— (2018). “Perks of Being Lazy: Boosting Retrieval Performance”. In: *International Conference on Case-Based Reasoning*. Ed. by Michael T Cox, Peter Funk, and Shahina Begum. Vol. LNAI, 11156. Springer Verlag, pp. 309–322. DOI: 10.1007/978-3-030-01081-2_21 (cit. on p. 5).

— (2020). “Fast Anytime Retrieval with Confidence in Large-Scale Temporal Case Bases”. In: *Knowledge-Based Systems* 206, p. 106374. DOI: 10.1016/j.knosys.2020.106374 (cit. on p. 5).

Muñoz-Avila, Héctor (1999). “A Case Retention Policy based on Detrimental Retrieval”. In: *Case-Based Reasoning Research and Development*. Ed. by Klaus-Dieter Althoff, Ralph Bergmann, and L.Karl Branting. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 276–287 (cit. on pp. 1, 11).

Nakhaeizadeh, Gholamreza (1994). “Learning Prediction of Time Series. A Theoretical and Empirical Comparison of CBR with some other Approaches”. In: *AAAI Technical Report*, pp. 67–71. DOI: 10.1007/3-540-58330-0_77 (cit. on pp. 4, 13).

Play&Sing (2018). *Playing and Singing for the Recovering Brain: Efficacy of Enriched Social-Motivational Musical Interventions in Stroke Rehabilitation*. URL: https://iiia.csic.es/research/project/?project_id=7 (visited on 08/31/2020) (cit. on pp. ix, 85).

Raghupathi, Wullianallur and Viju Raghupathi (2014). “Big data analytics in healthcare: promise and potential”. In: *Health Information Science and Systems* 2.1, pp. 1–10. DOI: 10.1186/2047-2501-2-3 (cit. on p. 11).

Ram, Ashwin and J.C. Santamaría (1997). “Continuous case-based reasoning”. In: *Artificial Intelligence* 90.1-2, pp. 25–77. DOI: 10.1016/S0004-3702(96)00037-9 (cit. on p. 13).

Reinsel, David, John Gantz, and John Rydning (2018). “The Digitization of the World - From Edge to Core”. In: *IDC White Paper* November (cit. on p. 4).

Ricci, Francesco, Bora Arslan, Nader Mirzadeh, and Adriano Venturini (2002). “ITR: A Case-Based Travel Advisory System”. In: *Advances in Case-Based Reasoning*. Ed. by Susan Craw and Alun Preece. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 613–627. DOI: 10.1007/3-540-46119-1_45 (cit. on p. 10).

Ricci, Francesco and Paolo Avesani (1995). “Learning a local similarity metric for case-based reasoning”. In: *International Conference on Case-Based Reasoning*. Vol. 1010. Springer Berlin Heidelberg, pp. 301–312. DOI: 10.1007/3-540-60598-3_27 (cit. on p. 18).

Riesbeck, Christopher K. (1996). “What Next? The Future of Case-Based Reasoning in Postmodern AI”. In: *Case-Based Reasoning: Experiences, Lessons & Future Directions*. Ed. by David B. Leake. Menlo Park: The AAAI Press/The MIT Press. Chap. 17, pp. 371–388 (cit. on p. 2).

Rissland, Edwina L., Kevin D. Ashley, and L. Karl Branting (2005). “Case-based reasoning and law”. In: *The Knowledge Engineering Review* 20.3, pp. 293–298. DOI: 10/b5swps (cit. on p. 10).

Sánchez-Marré, Miquel, Ulises Cortés, Montse Martínez, Joaquim Comas, and Ignasi Rodríguez-Roda (2005). “An Approach for Temporal Case-Based Reasoning: Episode-Based Reasoning”. In: *Case-Based Reasoning Research and Development*. Ed. by Héctor Muñoz-Ávila and Francesco Ricci. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 465–476. DOI: 10.1007/11536406_36 (cit. on pp. 4, 13, 17).

Sanchez-Pinsach, David, Mehmet Oğuz Mülâyim, Jennifer Grau-Sánchez, Emma Segura, Berta Juan-Corbella, Josep Lluís Arcos, Jesús Cerquides, Monique Messaggi-Sartor, Esther Duarte, and Antoni Rodriguez-Fornells (2019). “Design of an AI Platform to Support Home-Based Self-Training Music Interventions for Chronic Stroke Patients”. In: *Frontiers in Artificial Intelligence and Applications*. Ed. by Jordi Sabater-Mir, Vicenç Torra, Isabel Aguiló, and Manuel González-Hidalgo. Vol. 319. IOS Press, pp. 170–175. DOI: 10.3233/FAIA190120 (cit. on p. 85).

Schaaf, Jörg Walter (1995). “”Fish and Sink” An Anytime-Algorithm to Retrieve Adequate Cases”. In: *International Conference on Case-Based Reasoning*. Springer, pp. 538–547. DOI: 10.1007/3-540-60598-3_50 (cit. on p. 18).

— (1996). “Fish and Shrink. A next step towards efficient case retrieval in large scaled case bases”. In: *European Workshop on Advances in Case-Based Reasoning*. Vol. 1168, pp. 362–376. DOI: 10.1007/BFb0020623 (cit. on p. 15).

Schank, Roger C. (1982). *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*. New York: Cambridge University Press (cit. on pp. 1, 8).

Schank, Roger C., Richard Osgood, Matt Brand, Robin Burke, Eric Domeshek, Daniel Edelson, William Ferguson, Michael Freed, Menachem Jona, Brue Krulwich, E Ohmayo, and L Pryor (1990). *A Content Theory of Memory Indexing*. Technical Report No. 1. Institute for the Learning Sciences, Northwestern University (cit. on p. 11).

Schmitt, Gerhard (1993). “Case-based design and creativity”. In: *Automation in Construction* 2.1, pp. 11–19. DOI: 10.1016/0926-5805(93)90031-R (cit. on p. 10).

Serrà, Joan and Josep Lluís Arcos (2014). “An empirical evaluation of similarity measures for time series classification”. In: *Knowledge-Based Systems* 67, pp. 305–314. DOI: 10.1016/j.knosys.2014.04.035 (cit. on p. 31).

Shahar, Yuval (1997). “A framework for knowledge-based temporal abstraction”. In: *Artificial Intelligence* 90.1-2, pp. 79–133. DOI: 10.1016/S0004-3702(96)00025-2 (cit. on p. 14).

Smyth, Barry (1998). “Case-Base Maintenance”. In: *Proceedings of the Eleventh International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*. Springer, Berlin, Heidelberg, pp. 507–516. DOI: 10.1007/3-540-64574-8_436 (cit. on p. 48).

Smyth, Barry and Pádraig Cunningham (1996). “The Utility Problem Analysed: A Case-Based Reasoning Perspective”. In: *European Workshop on Advances in Case-Based Reasoning*. Springer, pp. 392–399. DOI: 10.1007/BFb0020625 (cit. on p. 11).

Smyth, Barry and Mark T. Keane (1995). “Remembering to Forget: A Competence-preserving Case Deletion Policy for Case-based Reasoning Systems”. In: *International Joint Conference on Artificial Intelligence*. Vol. 1, pp. 377–382 (cit. on pp. 1, 11).

Sox, Harold C., Michael C. Higgins, and Douglas K. Owens (2013). *Medical Decision Making*. Chichester, UK: John Wiley & Sons, Ltd. DOI: 10.1002/9781118341544 (cit. on p. 13).

Stahl, Armin and Thomas R Roth-Berghofer (2008). “Rapid Prototyping of CBR Applications with the Open Source Tool myCBR”. In: *Advances in Case-Based Reasoning*. Ed. by Klaus-Dieter Althoff, Ralph Bergmann, Mirjam Minor, and Alexandre Hanft. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 615–629. DOI: 10.1007/978-3-540-85502-6_42 (cit. on p. 10).

Torrent-Fontbona, Ferran and Beatriz López (2019). “Personalized adaptive CBR bolus recommender system for type 1 diabetes”. In: *IEEE Journal of Biomedical and Health Informatics* 23.1, pp. 387–394. DOI: 10.1109/JBHI.2018.2813424 (cit. on p. 10).

Toussaint, Godfried (2005). “Geometric proximity graphs for improving nearest neighbor methods in instance-based learning and data mining”. In: *International Journal of Computational Geometry and Applications* 15.2, pp. 101–150. DOI: 10.1142/S0218195905001622 (cit. on p. 86).

Troiano, Luigi, Alfredo Vaccaro, and Maria Carmela Vitelli (2016). “On-line smart grids optimization by case-based reasoning on big data”. In: *IEEE Workshop on Environmental, Energy, and Structural Monitoring Systems (EESMS)*. IEEE, pp. 1–6. DOI: 10.1109/EESMS.2016.7504842 (cit. on p. 12).

Ueno, Ken, Xiaopeng Xi, Eamonn Keogh, and Dah-Jye Lee (2006). “Anytime classification using the nearest neighbor algorithm with applications to stream mining”. In: *IEEE International Conference on Data Mining, ICDM*. IEEE, pp. 623–632. DOI: 10.1109/ICDM.2006.21 (cit. on pp. 13, 17).

Van den Branden, Martijn, Nirmalie Wiratunga, Dean Burton, and Susan Craw (2011). “Integrating case-based reasoning with an electronic patient record system”. In: *Artificial Intelligence in Medicine* 51.2, pp. 117–123. DOI: 10.1016/j.artmed.2010.12.004 (cit. on pp. 14, 17).

Wang, Xiaoyue, Abdullah Mueen, Hui Ding, Goce Trajcevski, Peter Scheuermann, and Eamonn Keogh (2013). “Experimental comparison of representation methods and distance measures for time series data”. In: *Data Mining and Knowledge Discovery* 26.2, pp. 275–309. DOI: 10.1007/s10618-012-0250-5 (cit. on p. 31).

Wess, Stefan, Klaus-Dieter Althoff, and Guido Derwand (1993). “Using k-d trees to Improve the retrieval step in case-based reasoning”. In: *European Workshop on Case-Based Reasoning*. Springer, pp. 167–181 (cit. on p. 12).

Wettschereck, Dietrich, David W. Aha, and Takao Mohri (1997). “A Review and Empirical Evaluation of Feature Weighting Methods for a Class of Lazy Learning Algorithms”. In: *Lazy Learning*. Dordrecht: Springer Netherlands, pp. 273–314. DOI: 10.1007/978-94-017-2053-3_11 (cit. on p. 14).

Wilson, David C. and David B. Leake (2001). “Maintaining Case-Based Reasoners: Dimensions and Directions”. In: *Computational Intelligence* 17.2, pp. 196–213. DOI: 10.1111/0824-7935.00140 (cit. on pp. 11, 12).

Woodbridge, Jonathan, Bobak Mortazavi, Alex A.T. Bui, and Majid Sarrafzadeh (2016). “Improving biomedical signal search results in big data case-based reasoning environments”. In: *Pervasive and Mobile Computing* 28, pp. 69–80. DOI: 10.1016/j.pmcj.2015.09.006 (cit. on p. 12).

Xu, Weijia, Daniel Miranker, Rui Mao, and Smriti Ramakrishnan (2008). “Anytime K-Nearest Neighbor Search for Database Applications”. In: *IEEE International Conference on Data Engineering Workshop*. IEEE, pp. 426–435. DOI: 10.1109/ICDEW.2008.4498354 (cit. on pp. 13, 15).

Yianilos, Peter N. (1993). “Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces”. In: *SODA, ACM-SIAM Symposium on Discrete Algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, pp. 311–321 (cit. on p. 12).

Zilberstein, Shlomo (1993). “Operational Rationality Through Compilation of Anytime Algorithms”. PhD thesis. University of California at Berkeley (cit. on p. 18).

— (1996). “Using Anytime Algorithms in Intelligent Systems”. In: *AI Magazine* 17.3, pp. 73–83. DOI: 10.1609/aimag.v17i3.1232 (cit. on pp. 2, 13, 18, 38, 40).