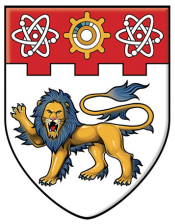
Trajectory Data Simplification, Similarity Search and Inference with Deep Learning



NANYANG TECHNOLOGICAL UNIVERSITY

Zheng Wang

School of Computer Science and Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfilment of the requirement for the degree of
Doctor of Philosophy (Ph.D)

February 6, 2023

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

20 Apr 2022

. . . . . .

Date

. . . . . .

Wang Zheng

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

02 June 2022

.....

Date

.....

Asst Prof Long Cheng

Authorship Attribution Statement

This thesis contains material from 7 paper(s) published/submitted in the following peer-reviewed conferences and journals in which I am listed as an author.

Chapter 3 is published as **Zheng Wang**, Cheng Long, Gao Cong, Qianru Zhang "Error-Bounded Online Trajectory Simplification with Multi-agent Reinforcement Learning", *In Proceedings of the 27th SIGKDD conference on Knowledge Discovery and Data Mining* (**KDD 2021**), Pages 1758-1768.

The contributions of the co-authors are as follows:

- I proposed the key technical solution, wrote all of the source code, conducted all experiments, and prepared the manuscript draft.
- Prof Cheng Long proposed the research problem, provided technical suggestions, and polished the manuscript draft.
- Prof Gao Cong provided useful comments on the techniques and experiments, and revised/edited the draft.
- Ms Qianru Zhang provided some discussions and carefully proofread the draft.

Chapter 4 is published as **Zheng Wang**, Cheng Long, Gao Cong "Trajectory Simplification with Reinforcement Learning", *In Proceedings of the 37th IEEE International Conference on Data Engineering* (**ICDE 2021**), Pages 684-695.

The contributions of the co-authors are as follows:

- I proposed the key technical solution, wrote all of the source code, conducted all experiments, and prepared the manuscript draft.
- Prof Cheng Long proposed the research problem, provided technical suggestions, and polished the manuscript draft.
- Prof Gao Cong provided useful comments on the techniques and experiments, and revised/edited the draft.

Chapter 5 is submitted as **Zheng Wang**, Cheng Long, Gao Cong, Qianru Zhang, Christian S. Jensen "Query Accuracy Driven Trajectory Simplification with Deep Reinforcement Learning", *Under review by the 49th International Conference on Very Large Data Bases* (**PVLDB 2023**).

The contributions of the co-authors are as follows:

- I proposed the key technical solution, wrote all of the source code, conducted all experiments, and prepared the manuscript draft.
- Prof Cheng Long proposed the research problem, provided technical suggestions, and polished the manuscript draft.
- Prof Gao Cong provided useful comments on the techniques and experiments, and revised/edited the draft.
- Ms Qianru Zhang provided some discussions and carefully proofread the draft.
- Prof Christian S. Jensen provided comments on the problem, techniques and revised/edited the draft.

Chapter 6 is published as **Zheng Wang**, Cheng Long, Gao Cong, Yiding Liu "Effective and Efficient Subtrajectory Similarity Computation with Deep Reinforcement Learning", *In Proceedings of the VLDB Endowment* (**PVLDB 2020**), Volume 13, Number 12, Pages 2312-2325.

The contributions of the co-authors are as follows:

- I proposed the key technical solution, wrote all of the source code, conducted all experiments, and prepared the manuscript draft.
- Prof Cheng Long proposed the research problem, provided technical suggestions, and polished the manuscript draft.
- Prof Gao Cong provided useful comments on the techniques and experiments, and revised/edited the draft.
- Dr Yiding Liu provided some initial discussions and feedback about my initial idea, and proofread the draft with some writing suggestions.

Chapter 7 is published as **Zheng Wang**, Cheng Long, Gao Cong, Ce Ju "Effective and Efficient Sports Play Retrieval with Deep Representation Learning", *In Proceedings of the 25th SIGKDD conference on Knowledge Discovery and Data Mining* (**KDD 2019**), Pages 499-509.

The contributions of the co-authors are as follows:

- I proposed the key technical solution, wrote all of the source code, conducted all experiments, and prepared the manuscript draft.
- Prof Cheng Long proposed the research problem, provided technical suggestions, and polished the manuscript draft.
- Prof Gao Cong provided useful comments on the techniques and experiments, and revised/edited the draft.
- Mr Ce Ju provided some initial discussions and assisted in the revision of the mathematics.

Chapter 8 is published as **Zheng Wang**, Cheng Long, Gao Cong "Similar Sports Play Retrieval with Deep Reinforcement Learning", *IEEE Transactions on Data Engineering* (**TKDE 2021**), Volume 1: Regular Paper, Accepted.

The contributions of the co-authors are as follows:

- I proposed the key technical solution, wrote all of the source code, conducted all experiments, and prepared the manuscript draft.
- Prof Cheng Long proposed the research problem, provided technical suggestions, and polished the manuscript draft.
- Prof Gao Cong provided useful comments on the techniques and experiments, and revised/edited the draft.

Chapter 9 is submitted as **Zheng Wang**, Mingrui Liu, Cheng Long, Qianru Zhang, Jiangneng Li "On Inferring User Socioeconomic Status with Mobility Records", *The 2022 IEEE International Conference on Big Data* (**BigData 2022**), Regular Paper, Accepted.

The contributions of the co-authors are as follows:

- I proposed the key technical solution, designed all experiments, and prepared the manuscript draft.
- Mr Mingrui Liu and Ms Qianru Zhang collected and preprocessed the datasets.
- I and Mr Mingrui Liu implemented all of the source code and conducted all experiments.
- Prof Cheng Long proposed the research problem, provided useful comments on the techniques and experiments, and revised/edited the draft.
- Ms Qianru Zhang and Mr Jiangneng Li provided some initial discussions and carefully proofread the draft.

18 Dec 2022 ..... Wang Zheng .....

Date Wang Zheng

Acknowledgments

I would like to express my gratitude to my main supervisor Prof. Long Cheng and co-supervisor Prof. Cong Gao for their patient guidance and help during my Ph.D. study. I gained a lot on how to conduct research with critical thinking.

I also would like to thank our members in Data Management and Analytics Lab (DMAL) including Feng Kaiyu, Li Xiucheng, Chen Zhida, Liu Yiding, Chen Jinyao, Chen Yile, Chen Yue, Balsebre Pasquale, Gu Tu, Hettige Kethmi Hirushini, Kim Junghoon, Li Jiangneng, Li Yi, Liu Shuai, Meng Zizhong, Shi Jiachen, Yang Jingyi, Zhang Chi, Zhang Zhaoqi, Zhao Yue, Zhou Haicang, Huang Weiming, Jiang Yue, Kong Weilong, Liu Shang, Wang Hao, Gao Jianyang, Zhu Qiuyu, Xu Qianxiong, Zhang Liang, Zhou Shaowen, Liu Kaijun, Wang Kaixin, Yu Kaiqiang, Liu Xuanyi, Ruan Sijie, Zhou Wei, Ma Ziyi, Liu Chenxi and Liu Mingrui. They make my daily life rich and thanks for their discussion of my research works. I would also like to thank the technician Mr. Loo Kian Hock in DMAL for his technical support when I encounter software issues in the lab. Last but not least, I would also like to extend my thanks to my girlfriend Zhang Qianru. It is my pleasure of meeting her.

Abstract

With the proliferation of GPS devices, trajectory data is being generated at an unprecedented speed, and the interest in the management and analysis of trajectory data has also grown dramatically. Trajectory data corresponds to a sequence of positions and timestamps to capture the traces of moving objects such as vehicles, pedestrians, sports players, etc. It embeds rich spatial and temporal information of moving objects, and how to manage and analyze the data efficiently and effectively has attracted much research attention in recent years. In this thesis, we explore effective and efficient solutions for trajectory data simplification, similarity search, and inference with emerging deep learning techniques.

First, we study three trajectory simplification problems, i.e., (1) error-bounded trajectory simplification, (2) size-bounded trajectory simplification, and (3) query accuracy driven trajectory simplification. Trajectory simplification is a common practice in trajectory data pre-processing, which aims to drop some of the points in a trajectory to release the burden on trajectory transmission, storage, and query processing. To be specific, *for (1) and (2)*, existing algorithms usually involve some decision making tasks (e.g., deciding which point to drop), for which, some human-crafted rules are used. Motivated by this, we propose to learn a policy for the decision making tasks via reinforcement learning (RL) and develop trajectory simplification methods based on the learned policy. Compared with existing algorithms, our RL-based methods are data-driven and can adapt to different dynamics underlying the problem. *For (3)*, existing techniques rely mainly on hand-craft error measures when deciding which point to drop when simplifying a trajectory. While the hope may be that such simplification affects the subsequent usability of the data only minimally, the usability of the simplified data remains largely unexplored. Instead of using error measures that indirectly may to some extent yield

simplified trajectories with high usability, we propose a direct approach to simplification and present the first study of query accuracy driven trajectory simplification, where the direct objective is to achieve a simplified trajectory database that preserves the query accuracy of the original database as much as possible.

Second, we study two similarity search problems, namely (1) *subtrajectory similarity search (SimSub)* and (2) *multi-trajectory similarity search*. The two problems are variants of the similar trajectory search problem, which is a fundamental problem of trajectory management, and have many applications such as those that take subtrajectories and/or groups of trajectories as basic units for analysis (e.g., sports play analytics). To be specific, for (1), we develop a suite of algorithms including both exact and approximate ones. Among those approximate algorithms, two that are based on deep reinforcement learning stand out and outperform those non-learning based algorithms in terms of effectiveness and efficiency. For (2), we propose a deep learning approach to learn the representations of sports plays, called play2vec, which is robust against noise and takes only linear time to compute the similarity between two sports plays. In addition, we extend the second problem to a database of games and a query play, the problem is to find those fragments of the games (i.e., plays), which are the most similar to the query play. This new problem setting is more aligned with real application scenarios, since the raw spatiotemporal sports data is collected in the units of games, but not plays.

Third, we study the problem of inferring user socioeconomic status based on their trajectories. For this task, we propose a novel learning framework incorporating a Deep Network and a Recurrent Network for user socioeconomic status inference, which extracts the features of the mobility records from three aspects, namely spatiality, temporality and activity. Compared with existing studies, we make the following contributions in this line of research. First, we study a novel problem of inferring users' economic statuses based on their GPS mobility records. This problem is new and has practical applications in real life (e.g., risk assessment for car loan applications/managements). Second, we propose a novel analytical framework called DeepSEI for the problem, which is a supervised deep learning model and incorporates two neural networks to capture the features from three aspects of users' mobility records.

In summary, the thesis aims at leveraging deep learning techniques for effective and efficient trajectory data simplification, similarity search and inference.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the advancement of sensor and mobile devices, massive spatial trajectory data is being generated by various moving objects (e.g., people, vehicles) in spatial spaces. Such collected trajectory data records the mobility traces of moving objects at different timestamps, which reflects diverse mobility patterns of objects. Accurate spatial trajectory data analysis serves as the key technical component for a wide spectrum of spatial-temporal applications, including intelligent transportation [1], location-based recommendation service [2], pandemic tracking [3], and crime prevention for public safety [4]. The large amount trajectory data also attracts much research efforts.

A typical trajectory data management and analytic system consists of several fundamental components, namely pre-processing, querying and analytics, as explained below.

- **Pre-processing.** Pre-processing on trajectory data is to process raw trajectory data before performing query processing and data mining tasks on the data. For example, one important task in this component is called trajectory simplification, which is to reduce the size of the raw trajectory data so as to save the storage costs and speed up query processing on the data.

- **Querying.** Querying on trajectory data is to answer queries from users based on the trajectory data. Some common queries on trajectory data include range query, similarity search, and trajectory join. In particular, the similarity search query is to search from a trajectory database those that are similar to a query trajectory based on some trajectory similarity measurement.

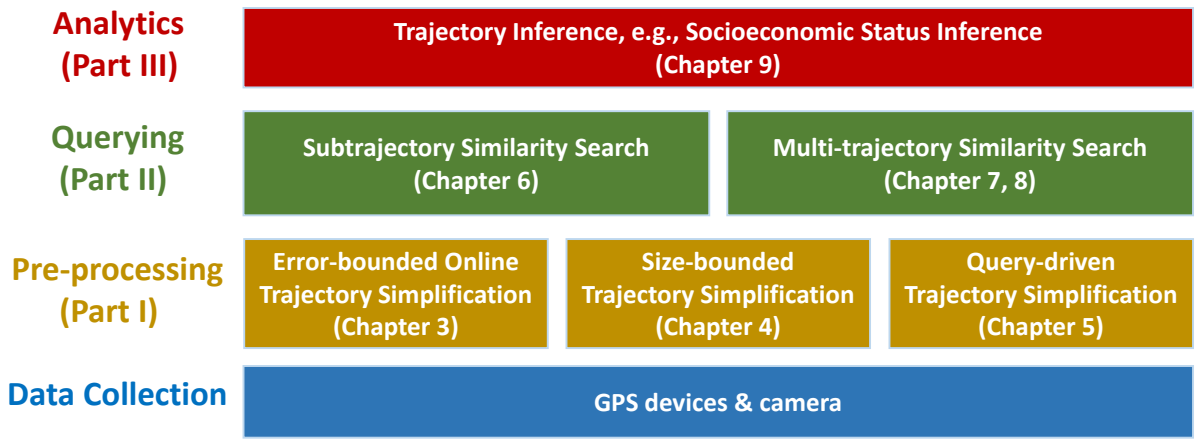


Figure 1.1: A typical architecture of a trajectory management and analytics system.

- **Analytics.** Analytics on trajectory data is to analyze the trajectory data in order to discover patterns and knowledge embedded in the data. For example, knowledge inference from trajectory data is a typical task in trajectory data analytics.

Figure 1.1 shows a typical architecture of a trajectory management and analytics system.

In this thesis, we study quite a few research problems spanning over different components of the trajectory data management and analytics system, including (1) error-bounded, size-bounded and query-driven trajectory simplification in the pre-processing component; (2) subtrajectory similarity search and multi-trajectory similarity search in the querying component; and (3) users' socioeconomic statuses inference from trajectory data in the analytics component. A summary of the research problems studied in this thesis is also included in Figure 1.1. Next, we provide preliminaries in terms of the above three parts in Section 1.1, 1.2 and 1.3, respectively.

1.1 Trajectory Simplification

1.1.1 Background

Trajectory data is a data type that captures traces of moving objects such as vehicles, pedestrians, robots, etc. It is central to many applications such as urban mobility analysis, logistics, transportation, sports games, etc. Trajectory data is typically generated continuously and collected by remote sensors such as GPS devices. One typical scenario is that a sensor periodically checks the coordinates and time, which corresponds

to a time-stamped location (called spatio-temporal point or simply point), and stores the point in a buffer. Typically, a sensor has a small storage budget, low computation capability, and limited network bandwidth. A consequent issue is that the buffer would become occupied frequently and the workload of transmitting the points is high. In addition, in some applications, there could be hundreds of thousands of sensors, which collect trajectory data simultaneously. Once the trajectory data collected by all these sensors is accumulated at a server, the volume would be huge, which has been illustrated by several existing studies (see the survey paper [5]). A consequent issue is that the huge volume of trajectory data would increase the storage cost and more importantly make the query processing on the data expensive.

A common practice that has been used to deal with the aforementioned two issues is to conduct trajectory simplification, which essentially is to drop some points of a given trajectory and keep the remaining ones as a simplified trajectory. Specifically, in the online mode, the trajectory data is inputted point by point and once a point is dropped, it is no longer accessible. In the batch mode, the trajectory data is inputted completely once and remains accessible during the whole course of trajectory simplification. The online mode and batch mode are for different application scenarios. In the online mode, a trajectory is fed point by point in an online fashion, the dropped points will no longer be accessible. In the batch mode, all the points in a trajectory are fed together, and remain accessible during the simplification process. The rationale behind trajectory simplification is two-fold. First, not all points of a trajectory carry equal amount of information and some carry little or even no information. For example, when an object moves along a straight line at a constant speed, all points except for the first and last ones carry no information and could be dropped. Second, with some point dropped, the burden on the transmission, storage, and query processing would be lowered down significantly. In this thesis, we consider three problems of trajectory simplification, and discuss the relationships among the three pieces of work.

1.1.2 Research Problems

Error-Bounded Online Trajectory Simplification. We consider the problem error-bounded online trajectory simplification (EB-OTS), which is to drop as many points as

possible (or equivalently to keep as few points as possible) such that the "error" of the simplified trajectory, is bounded by an error tolerance.

Size-Bounded Trajectory Simplification. The dual problem of error-bounded trajectory simplification (i.e., min-size) is called size-bounded trajectory simplification, which is to drop at least a certain number of points (or equivalently to keep at most a certain number of points) such that the information loss, captured as the "error" of the simplified trajectory is minimized.

Query Accuracy Driven Trajectory Simplification. We propose a new trajectory simplification problem, called *Query accuracy Driven Trajectory Simplification (QDTS)*. Given a trajectory database D and a storage budget, the problem is to find a simplified trajectory database D' that preserves the accuracy of query results as much as possible, compared to the query results on D. We remark that the query studied in this thesis is in database-level, e.g., Range Query, kNN Query, Similarity Query, and Clustering (the detailed definition can be found in Section 5.2.3), which return a set of trajectories to respond to the query, and those are widely studied on trajectory data [5, 6].

Here, we use the F_1-score and accuracy to define the query accuracy, which measures the difference between query results on an original database D and those on a simplified database D', by taking the results on D as the ground truth. The detailed definition can be found in Section 5.2.3. We note that the measures for query accuracy are based on the trajectory IDs returned via query processing. This is affected by how much the trajectories are deformed, which relates to the measures (e.g., PED or DAD) in error/size-bounded simplification. In particular, if a trajectory is an answer to a query yet it is deformed too much (i.e., the PED or DAD degrades), it may not be returned for the query based on a simplified database (i.e., the query accuracy degrades).

Relationships among the three problems. The size-bounded problem is the dual problem of the error-bounded problem, which is also called the *Min-Error* problem. Interested readers are referred to a survey paper [5] and a concurrent work [7] and the references therein for details. For the size-bounded problem, we target the problem in both the online mode and the batch mode and proposes a reinforcement learning method for the problem. For the error-bounded problem, we target the problem in the online mode and develop a multi-agent reinforcement learning method. We investigate the

two problems separately as the solutions are largely different with each other by taking different constraints as the inputs (e.g., a given storage budget for the size-bounded problem, and a given error tolerance for the error-bounded problem).

In addition, query accuracy driven problem differs from the existing size-bounded problem. First, it considers a different objective of trajectory simplification, namely that of preserving query accuracy directly, rather than through minimizing an error measure, as in the size-bounded problem. Second, it is global in nature and takes a trajectory database as input and outputs a simplified trajectory database that satisfies a specified storage budget as a whole, instead of simplifying each trajectory in isolation according to a budget, as do existing size-bounded techniques.

1.1.3 Methodology Overview

Methodology of Error-Bounded Online Trajectory Simplification. Many existing algorithms have been developed for EB-OTS problem, including `OPW` [8], `CISED-S` [9], `BQS` [10, 11], `FBQS` [10, 11], `OPERB` [12], `Intersect` [13], `Angular` [14], and `Interval` [15]. They all adopt the following three-step process. Step 1: it opens a window involving the first and second points. Step 2: it checks whether the segment linking the first point and the last point in the window can approximate the trajectory consisting of all points in the window within the error tolerance. If so, it marks the last point of the window as a *safe point*, expands the window by including one following point outside the window, and repeats Step 2; otherwise, it proceeds to Step 3. Step 3: it (1) chooses a point among all safe points in the window based on some *pre-defined rules*, (2) drops all the points between the first point and the chosen safe point in the window, both exclusively (note that error caused by dropping these points should be bounded by the error tolerance), (3) re-opens a new window involving the chosen safe point and the one following the chosen point, and goes back to Step 2. An example illustrating the three-step process is shown in Figure 1.2.

The existing algorithms differ from each other in (1) adopting different strategies for checking the error in Step 2 and (2) using different pre-defined rules in Step 3. For example, the `OPW` algorithm computes the error *exactly* in Step 2 and explores two pre-defined rules in Step 3, namely one to choose the last point of the window and one to

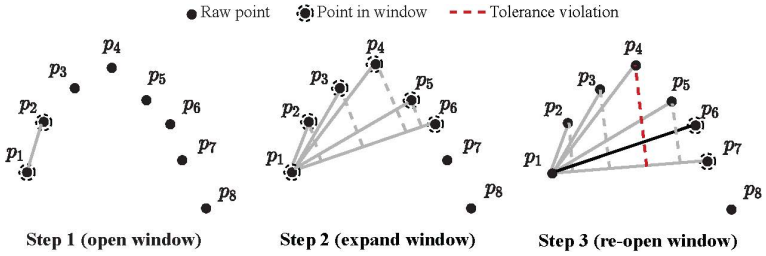


Figure 1.2: Illustration of the three-step process. Step 1: it opens a window with p_1 and p_2. Step 2: the window is repeatedly expanded to p_6, and the points p_2, p_3, p_4, p_5, p_6 are marked as safe points because the errors on segment $\overline{p_1p_2}, \overline{p_1p_3}, \overline{p_1p_4}, \overline{p_1p_5}, \overline{p_1p_6}$ are within a given error tolerance. Step 3: point p_4 causes the error tolerance violation when checking segment $\overline{p_1p_7}$. Then, (1) point p_6 is chosen among all safe points based on a pre-defined rule, (2) points between p_1 and p_6 are dropped, (3) re-opens a new window involving the chosen safe point p_6 and its following point p_7.

choose the point such that dropping it would cause the greatest error. The `CISED-S` algorithm computes only an *upper bound* of the error in Step 2 for better efficiency and only expands the window if the upper bound is bounded by the error tolerance, and chooses the last point in the window in Step 3.

These existing algorithms suffer from two issues. First, in Step 2, they expand the window by one each time, for which it needs to check the error caused by dropping a sequence of points, which incurs expensive computation costs. Note that for algorithms such as `OPW`, they compute the error exactly for each point, which takes $O(n)$ time, and as a result, they have the time complexity of $O(n^2)$, where n is the number of points of the inputted trajectory. Second, in Step 3, they use pre-defined rules for choosing a safe point for re-opening a window, e.g., choosing the last safe point in the window. These rules are human-crafted and cannot adapt to different problem instances and inputted data. There is no theoretical ground supporting that these rules would achieve the objective of the EB-OTS problem, and as a result, the effectiveness of the algorithms with these rules is not guaranteed.

We observe that the aforementioned three-step process in EB-OTS corresponds to a *sequential decision process*, i.e., it sequentially make decisions on how to expand a window (if possible) and on how to choose a safe point for re-opening a window (if expanding a window is not possible). All existing algorithms utilize pre-defined rules this sequential decision process, e.g., expanding a window by one point in Step 2 and choosing the last safe point in a window in Step 3. Inspired by the fact that reinforcement

learning (RL) has been shown to work effectively for sequential decision making problems such as those in robotics [16] and gaming [17], we propose to leverage RL for the EB-OTS problem. Specifically, we introduce two agents, namely Agent-E and Agent-R, for expanding a window and re-opening a window in the three-step process, respectively. Agent-E decides how many points to be included for expanding a window. Agent-R decides which safe point to choose for re-opening a window. For both Agent-E and Agent-R, we carefully define their underlying Markov decision process (MDP) [18] model, including states, actions, transitions and rewards such that the states can be computed efficiently and capture essential information for decision making and the model can be learned effectively. In particular, we let the two agents share their rewards so that they can work cooperatively for achieving effective online trajectory simplification. We call this multi-agent RL-based algorithm `MARL4TS`. `MARL4TS` is applicable to all error measurements that are commonly used, while most existing algorithms are designed for specific error measurement (e.g., `CISED-S` is applicable only when the *synchronous Euclidean distance [8, 19–21] is used for measuring the error of a simplified trajectory*). In addition, we develop two techniques to further boost the efficiency and effectiveness of `MARL4TS`, namely one of imposing a maximum size over windows to be considered (for improving efficiency) and one of delaying to re-open a window for a certain number of points so as to potentially include more safe points for consideration when re-opening a window (for improving the effectiveness).

Methodology of Size-Bounded Trajectory Simplification. The size-bounded trajectory simplification is also a widely used solution to reduce the storage cost and boost query processing for trajectory data. Quite a few algorithms exist for the size-bounded problem in the online mode, including STTrace [19], SQUISH [20], and SUISH-E [21]. All these algorithms share the idea that it maintains a buffer of a certain size and keeps storing points in the buffer, and whenever the buffer becomes full, it picks one point from the buffer and drops it. In addition, they all make the decision on which point to drop by defining some "importance value" of each point and always dropping the point with the least importance value. Roughly, the importance value of a point is defined as some form of error that would be introduced when the point is dropped since a smaller error means that the point is less important and should be dropped. When a point is dropped,

the importance values of the remaining points should be updated, in which the existing algorithms differ from one another. STTrace simply re-computes the values, while SQUISH distributes the value of a point that has been dropped to its neighboring points (so that the importance values would be carried on) and SQUISH-E is similar to SQUISH with only some slight refinements on the update procedure. As could be noticed, these algorithms are mainly based on some human-crafted rules for deciding which point to drop.

We propose the reinforcement learning (RL) methods for the size-bounded trajectory. We treat the trajectory simplification problem (in the online mode) as one of a sequential decision process, i.e., it scans a trajectory sequentially, and whenever the buffer is full, it makes a decision on which point to drop. We then model the process as a *Markov decision process* (MDP), learn the policy for the MDP via a widely-used *policy gradient* method [22–24], and develop a trajectory simplification method based on the learned policy. We call this method *RLTS*. Compared with existing algorithms, our RLTS method computes simplified trajectories with smaller errors, which is mainly due to its data-driven nature and also its capability of adapting to different dynamics of the inputted points. The RLTS method has the time complexity of $O((n-W)\log W)$, where n is the number of points in the inputted trajectory and W is the buffer size, which is the same as that of existing algorithms STTrace, SQUISH, and SQUISH-E.

We also propose a variant of RLTS, called RLTS-Skip, by augmenting the MDP with additional actions of *skipping* points from being scanned. The rationale is that some points may carry little information so that they can be dropped immediately without being inserted in the buffer when they are being scanned, as RLTS does. The benefit is that the efforts of deciding and taking actions for these points that are skipped are saved and the efficiency is boosted. In addition, RLTS-Skip accepts a parameter J, which controls the maximum number of points that are allowed to be skipped. Therefore, RLTS-Skip provides a tunable trade-off between efficiency and effectiveness with different settings of J. Note that when $J = 0$, RLTS-Skip reduces to RLTS.

We further investigate the problem under the batch mode, which assumes more data access than under the online mode. Specifically, we have access to all points of a trajectory during the whole course of trajectory simplification in the batch mode, while we can

only access those points that are stored in the buffer in the online mode. With the increased data access, we have more options for defining the states of the MDP. We investigate three state definitions, which capture different portions of the points of a trajectory, and correspondingly develop three different categories of algorithms, namely (1) RLTS and RLTS-Skip, (2) RLTS+ and RLTS-Skip+, and (3) RLTS++ and RLTS-Skip++. RLTS and RLTS-Skip are exactly the same as those for the online mode, whose underlying states are defined based on those points stored in the buffer only. RLTS+ and RLTS-Skip+ are enhanced versions of RLTS and RLTS-Skip, respectively, where their states are defined based on all those points that have been scanned (including those that are stored in the buffer and those that have been dropped). In addition, RLTS-Skip+ augments the states further with the information of the J points that follow the point that is being scanned. RLTS++ and RLTS-Skip++ ultimately consider all points for defining states. From RLTS and RLTS-Skip, to RLTS+ and RLTS-Skip+, to RLTS++ and RLTS-Skip++, more information is captured for defining the states and correspondingly, the MDPs become more complex and the methods take more time costs. Specifically, the time complexities of the three categories of methods are $O((n-W)\log W)$, $O((n-W)(n'+\log W))$, $O((n-W)(n'+\log n))$, respectively, where n' is the cost of computing the value of a point and is bounded by n (in practice, $n' \leq n$).

Methodology of Query Accuracy Driven Trajectory Simplification. The extent to which a collection of simplified trajectories enables accurate query results has been used widely as a measure of the quality of a simplification technique in empirical studies of trajectory simplification [5, 25, 26]. For example, Zhang et al. [5] evaluate existing simplification techniques in terms of their ability to produce simplified trajectories that affect the accuracy of range, kNN, and join queries as well as clustering minimally. While there are many proposals for trajectory simplification [7, 19–21, 27–29], they all assume a storage budget and aim to produce simplified trajectories that minimize the difference from the original trajectories according to a given difference notion. No proposals aim to optimize directly the query accuracy offered by the simplified trajectories. We call this line of study *Error-Driven Trajectory Simplification (EDTS)*.

In addition, existing simplification techniques are local in nature and operate on a per-trajectory basis as opposed to being global in nature and operating on a database

of trajectories. Specifically, they aim to simplify a given trajectory T within a budget $r \cdot |T|$, where r is the compression ratio. When these techniques are used to simplify a database of trajectories according to a compression ratio r, they simplify each trajectory in the database *separately* according to compression ratio r. This is likely sub-optimal in cases where trajectories have different sampling rates or different complexities. Intuitively, trajectories with higher sampling rates or lower complexity are candidates for simplification with larger compression ratios.

We develop a solution to the Query Accuracy Driven Trajectory Simplification (QDTS) problem with reinforcement learning. It starts with the most simplified database, in which each simplified trajectory T' of an original trajectory T consists of only the first and last points of T. It then introduces original points into the simplified database iteratively until the budget is exhausted. This solution differs from existing solutions in two ways. (1) It considers all trajectories in the database *collectively* rather than *separately* as do all existing algorithms. In addition, it partitions the database into spatio-temporal cubes. Then, each time it needs to select a point to introduce, it first identifies a few cubes and considers only points in these cubes as candidates. This improves efficiency. (2) It leverages reinforcement learning to learn a policy, instead of using heuristics as do most of the existing algorithms, for selecting points to introduce. The learned policies are query accuracy aware such that they select points that optimize query accuracy. We call the resulting solution `RL4QDTS`.

1.1.4 Contributions

In summary, we make the following contributions in trajectory simplification.

Firstly, we develop the first multi-agent RL-based method `MARL4TS` for the EB-OTS problem. `MARL4TS` works generally for *four* error measurements and achieves the best effectiveness compared with *all* existing algorithms under *different* parameter settings. In addition, the objective of EB-OTS is perfectly captured by that of `MARL4TS`. This provides some theoretical ground/explanation on the superior effectiveness over existing algorithms, which are based on pre-defined rules and do not optimize EB-OTS's objective directly. `MARL4TS` runs in linear time, which provides theoretical justifications of its competitive efficiency.

Secondly, we develop an RL-based method for the size-bounded trajectory simplification problem. The RL-based method achieves competitive effectiveness and efficiency simultaneously. In particular, for the batch mode, the RL-based method outperforms *all* existing approximate methods in terms *both* effectiveness and efficiency, under *four* error measurements, across *different* parameter settings, and on *all* real datasets tested. Furthermore, we propose a variant of RLTS, i.e., RLTS-Skip, which runs faster than RLTS and provides a controllable trade-off between the effectiveness and efficiency. Particularly for the batch mode, we investigate three variants of the method given the increased data access.

Thirdly, we propose the QDTS problem, which aims to find a simplified trajectory database within a given storage budget that preserves query accuracy as much as possible. This is the first systematic study of this line of trajectory simplification. We develop an RL-based solution called `RL4QDTS` to the problem. We show that the objective of the QDTS problem can be optimized explicitly by `RL4QDTS`, and provide theoretical arguments.

1.2 Trajectory Similarity Search

1.2.1 Background

Trajectory data, which corresponds to a type of data for capturing the traces of moving objects, is ubiquitous. It has been used for various types of queries such as clustering [30–32] and similarity search [33–38]. Here, we discuss two typical queries (i.e., subtrajectory similarity search and multi-trajectory similarity search), and further develop an indexing technique for multi-trajectory similarity search.

In *subtrajectory similarity search*, the majority of existing studies take *a trajectory as a whole* for analysis [33–38]. Motivated by the phenomenon that two trajectories could be dissimilar to each other if each is considered a whole but similar if only some portion of each is considered, there have been a few studies, which take *a portion of a trajectory* as a basic entity for analysis [30–32, 39, 40]. Some examples include subtrajectory clustering [30–32] and subtrajectory join [39, 40]. For example, the subtrajectory clustering method in [31] first partitions raw trajectories into different subtrajectories using some principle and then groups those subtrajectories that are similar to one another into clusters. In quite a few real-life applications, subtrajectories are naturally considered as basic units for analysis, e.g., subtrajectory search [41], subtrajectory join [39], subtrajectory clustering [32], etc. One application is the subtrajectory search query on sports play data. In sports such as soccer and basketball, a common practice nowadays is to track the movements of players and/or the ball using some special-purpose camera and/or GPS devices [42]. The resulting trajectory data is used to capture the semantics of the plays and for different types of data analyses. One typical task on such sports play data is to search for a portion/segment of play from a database of plays, with its trajectories of players and/or its trajectory of the ball similar to those and/or that of a given query play [41]. This task is essentially one of searching for similar subtrajectories. Another potential application is detour route detection. It first collects those routes that have been reported by passengers to be detour routes and then searches for those subtrajectories of taxis' routes, which are similar to a detour route. The found subtrajectories are probably detour routes as well.

In *multi-trajectory similarity search*, a typical application is similar play retrieval. Similar play retrieval is a process of finding those plays from a database that are similar

to a query play, where a play corresponds to a fragment of a game and has its duration varying from seconds to minutes. It is widely used in some emerging sports applications such as ESPN and Team Stream to recommend similar plays to sports fans. Besides, it could help sports club managers and coaches to improve team tactics when preparing for an upcoming match [43].

1.2.2 Research Problems

Subtrajectory Similarity Search. We consider a query with its goal to search for a portion of a trajectory from a database storing many trajectories called data trajectories, which is the most similar to a given trajectory called query trajectory. In this query, a portion of a trajectory, called subtrajectory, is considered as a basic entity and a query trajectory is taken as a whole for analysis. Therefore, it captures trajectory similarity in a finer-grained way than conventional similar trajectory search. We call this problem the *similar subtrajectory search* (SimSub) problem.

Multi-trajectory Similarity Measurement. We consider measuring the similarity between two plays, which is non-trivial because each play involves multiple trajectories. Here, we aim to learn a vector representation for each play such that the similarities among plays are well captured by the Euclidean distances in the vector space. Without loss of generality, our proposed representation learning techniques can be applied to measure any multi-trajectory similarities, and sports play retrieval is a typical application to instantiate the technique.

Multi-trajectory Similarity Search. We further study a problem of searching for plays from a database of games, which are similar to a query play, where a *play* corresponds to a fragment of a game and has its duration varying from seconds to minutes. The problem is called *similar sub-game retrieval* or simply *similar play retrieval* (SimPlay).

1.2.3 Methodology Overview

Methodology of Subtrajectory Similarity Search. A key problem that is involved in answering the query mentioned above is to find a subtrajectory of a data trajectory, which is the most similar to a given query trajectory. While there are many existing studies on the similar trajectory search problem with each trajectory considered a whole,

there are very few studies on the SimSub problem. Let T be a data trajectory involving n points and T_q be a query trajectory involving m points. To answer the similar subtrajectory search, we design an exact algorithm, which enumerates all possible subtrajectories of T, computes the similarity between each subtrajectory and the query trajectory, and returns the one with the greatest similarity. We further adopt an *incremental* strategy for computing the similarities that are involved in the exact algorithm, which helps to improve the time complexity by $O(n)$. We also follow some existing studies on subsequence matching [44, 45] and design an algorithm, which considers only those subtrajectories with their sizes similar to that of the query trajectory and controlled by a user parameter. This would provide a controllable trade-off between efficiency and effectiveness.

To push the efficiency further up, we propose several algorithms, which share the idea of splitting a data trajectory into some subtrajectories to be candidate solutions to the problem and differ in using different methods for splitting the data trajectory. Specifically, the process is to scan the points of a data trajectory one by one sequentially and for each one, it decides whether to split the data trajectory at the point. Some of them use pre-defined heuristics, e.g., a greedy one. Others model the process as a *markov decision process* (MDP) [18] and use deep reinforcement learning to learn an optimal policy for the MDP, which is then used for splitting the data trajectory. These splitting-based algorithms have time complexities much lower than the exact algorithm in general, e.g., for measurements such as t2vec, each splitting-based algorithm runs in $O(n)$ time while the exact algorithm runs in $O(nm)$ time.

Methodology of Multi-trajectory Similarity Measurement. Existing solutions for this problem all adopt a two-step approach: (1) it *aligns* the trajectories in one play to those in the other; and (2) it computes the similarity between each pair of trajectories that have been aligned to each other using some trajectory similarity metrics such as the Dynamic Time Warping (DTW) [38] and then sums up all similarities to be one between the two plays [41]. Two strategies have been considered for the alignment step [41]. The first one is an enumeration-based strategy which (conceptually) considers all possible alignments and picks the one with the best similarity score. In practice, the method with this alignment strategy could be implemented by finding the optimal matching between the two sets of trajectories using algorithms such as the *Hungarian* algorithm, where

the weight between a pair of trajectories is set to be the similarity between them. The second one is a role-based strategy which first detects the role of each player, which is hidden, and then aligns the trajectories of two players who share their roles. Here, the role of a player detected may have semantics such as center forward, midfield, etc. in a soccer game and could change from time to time throughout the game. While these existing solutions have some merits in transforming the original problem for plays to one for trajectories with the alignment strategy, they are insufficient in three aspects.

The first one is the effectiveness. These methods rely on the assumption that human beings would check individual pairs of trajectories separately and then combine their perceptions of similarity on trajectories together using a sum function, which, however, remains not justified. For example, these methods assume that each pair of trajectories contributes equally to the final similarity between two plays, but it is more intuitive to assign more weights to those pairs of trajectories that are closer to the ball.

The second one is the efficiency. These methods all involve the procedure of computing the similarity between two trajectories, which has a time cost at least $O(n^2)$ for those well-known trajectory metrics (including the DTW one) where n is the length of the longer trajectory. Since a typical similar play retrieval scenario would usually need to compute the similarity between the query play and many plays in the database, which is huge in practice (e.g., a typical soccer game involves about 4.14 million sampled positions and one season of soccer games in the English Premier League involves about 1.57 billion sampled positions), this quadratic time complexity would impose a big challenge on the efficiency.

The third one is the robustness (against sampling errors and measurement errors). Recall that the spatiotemporal sports data is collected by sampling the locations of the players and the ball with devices such as GPS. Two types of errors are inevitable in the data, namely the errors due to the sampling nature and those due to the measurement of devices. Existing methods use the locations in the form of coordinates and thus some minor changes on the coordinates may result in an obvious change on the similarity, i.e., they are not robust against the errors.

We propose to learn representations of plays in a low-dimensional space using deep models, which we call *play2vec*, such that the (Euclidean) distances in the space capture

the similarities among the plays well. The core idea of our approach is to treat a play as a sequence of play segments with uniform durations and then design a denoising sequential encoder-decoder (DSED) model for extracting a feature vector from the sequence. There is a gap that needs to close up to make this idea work since a play segment, which corresponds to a fragment of a play, has the same form as a play - it consists of a set of multiple trajectories (of shorter lengths) while an encoder-decoder model typically accepts inputs in the form of vectors. We achieve this by embedding each play segment as a vector. Specifically, we treat each play segment as a word or token and each play as a sentence in a natural language context and extend the Skip-Gram with Negative Sampling (SGNS) model [46] to embed the words (or play segments) as vectors. Again, there is a gap here since a play segment is in a continuous space (there exists an infinite number of possible trajectories) while a word is in a discrete one. We close this gap by (1) mapping each play segment to a binary matrix, where an entry of the matrix is set to be 1 if its corresponding cell in a grid that partitions the pitch is traveled through by some trajectories in the play segment and 0 otherwise; and (2) assigning one token to those matrices that are similar (for restricting the token space size). In summary, our approach first maps all play segments to tokens with a spatial grid, then embeds the tokens as vectors by extending the SGNS model, and then extracts a feature vector from each play with the DSED model that is designed to fit our context.

Our new method has obvious advantages over existing ones in the aforementioned three aspects. Regarding the effectiveness, our method is based on the popular encoder-decoder deep model which is widely known to perform well in extracting features from sequential data. Regarding the efficiency of computing the similarity of two plays, our method runs in $O(n+d)$ time while existing ones have time complexities at least $O(n^2)$, where n is the length of the longest trajectory in a play and d is the size of a learned feature vector which is small. Regarding the robustness, our method involves two mechanisms to deal with sampling errors and measurement errors. First, in the step of mapping play segments to tokens, a grid is used such that it is not sensitive to errors. Second, in the encoder-decoder model, the data is first injected with some noises and then denoised for training which would help mitigate the problems caused by errors.

Methodology of Multi-trajectory Similarity Search. Existing studies on similar play retrieval usually assume that some candidate plays called *data plays* are stored in

a database for query processing [41, 42]. For example, in the proposal [41], all possible fractions of a game are materialized as data plays, whose duration fall in a range, e.g., from 1s to 5s, and are stored in a database for query processing. While this strategy is simple, it is not an ideal solution. First, with a small range used for materializing data plays, it restricts the space of possible plays for query processing, i.e., only those with the length in the small range are considered. When the length of a query play is outside the range, those plays that are similar to the query play may be missed (since they should have similar lengths as the query play but are not materialized). Second, with a very large range, more data plays will be stored in the database, and the cost of both materializing all possible data plays and searching for similar data plays would be increased accordingly.

We propose the SimPlay, namely we do not materialize data plays from games, but search games directly for plays that are similar to a given query play. Specifically, given a database of games and a query play, the problem is to find those fragments of the games (i.e., plays), which are the most similar to the query play. Different from existing studies [41, 42], which assume a database of plays, *each as a whole being considered*, this problem takes as input a database of games, *each with its fragments being considered*. This problem setting has two advantages over existing ones. First, the materialization of plays is no longer necessary and thus the specification of a duration range is avoided. Second, this is more aligned with real application scenarios, since the raw spatiotemporal sports data is collected in the units of games, but not plays.

There are two challenges in solving the SimPlay problem: (1) *how to search for sub-games (plays) from a specific game, which are similar to a query play (given that a game involves many possible plays)*, and (2) *how to search over a database of many games for plays, which are similar to a query play (given that there are usually many games in the database)*. We develop techniques to solve the three challenges in SimPlay, as explained as follows.

(1) Search Similar Play Within a Game. We adopt *play2vec* as introduced above to measure the similarity between two plays. Then, a simple method is to enumerate all possible plays of a game, compute the similarity between each play and the query play based on the *play2vec*, and return the play with the greatest similarity. For better

efficiency, we borrow the ideas of existing techniques that have been proposed for the sub-trajectory similarity search problem in [47] for searching similar plays within a game.

(2) Search Similar Play Within a Database of Games. An intuitive solution to solve the SimPlay problem is to scan the games in the database, and for each one, compute its most similar play (using one of the methods presented in Section 6.3) and update the most similar play found so far. Nevertheless, there are usually many games in a database in practice, and thus this method would not meet the efficiency requirement for real applications. In Chapter 8, we develop a deep metric learning based technique for determining the order of the games in a database for searching the most similar play within a game and ignoring those that are behind others for better efficiency. Specifically, we define a score for each game given a query play to be the maximum similarity between a play of the game and the query play. The rationale of the score is that the games in the databases would be taken in a descending order of their score for searching for the similar plays to a query play. To infer the score of a game efficiently, we develop a deep metric learning method based on a triplet network [48] to learn embeddings of games such that the order based on their scores is preserved. When performing the similar play retrieval, we first only consider a fraction r of the games with the top scores for searching for the most similar play, where $r \in (0, 1]$ is a user parameter for controlling the trade-off between effectiveness and efficiency.

1.2.4 Contributions

In summary, we make the following contributions in trajectory similarity search.

Firstly, we propose the SimSub problem, and this to the best of our knowledge, corresponds to the first systematic study on searching subtrajectories that are similar to a query trajectory. The SimSub problem relies on a trajectory similarity measurement, which could be instantiated with any existing measurement. we develop a suite of algorithms for the SimSub problem including (1) one exact algorithm, (2) one approximate algorithm, which provides a controllable trade-off between efficiency and effectiveness, and (3) several splitting-based algorithms including both heuristics-based ones and deep reinforcement learning based ones. These algorithms cover a wide spectrum of application scenarios in terms of efficiency and effectiveness.

Secondly, we develop an unsupervised deep learning model, called play2vec, to learn representations of plays (a set of multiple trajectories of players and the ball), which is superior to existing ones in aspects of effectiveness, efficiency, and robustness.

Thirdly, we propose a new problem of searching similar plays for a given query play from a database of games. This is different from existing studies, which assume a database of materialized plays, and is more aligned with real applications. We develop a suite of algorithms for searching the play of a game, which is the most similar to a query play. Among them, two use policies learned via reinforcement learning for forming candidate plays. They correspond to adaptions of the techniques in [47], which search for similar sub-trajectories, for searching similar plays within a game. We develop a deep metric learning based method for deciding the order of searching the games in a database and searching only a fraction r of the games that are before others, where $r \in (0, 1]$ is a user parameter for controlling the trade-off of effectiveness and efficiency.

1.3 Socioeconomic Status Inference from Trajectory Data

1.3.1 Background

With the rapid development of GPS devices and/or mobile technologies, recent years have witnessed an unprecedented growth in mobility data. The large data has attracted much research efforts to acquire knowledge of human mobility behaviors. More specifically, extensive studies have been conducted on user profiling from mobility records. For example, it has been explored to infer users' demographic attributes from their check-ins [49], users' ethics and gender from their photo sharing data with geo tags [50], passengers' employment status from their smart card data [51], and users' demographics from their trajectories [52], etc.[1]

While these techniques are extensive and have some merits, there still exist some scenarios that have been overlooked and/or cannot be adequately solved by them. For example, in some real-life applications such as car loans, the mobility records that are available are not those based on smart phones as most existing studies assume, but those based on GPSs installed on their cars. In these scenarios, only when users have a movement, the mobility records reflect those of the users. For these applications, it is interesting to infer the economic profiles of the users such as their socioeconomic status, yet these are mostly overlooked by existing studies [49–52].

1.3.2 Research Problems

We aim to explore a central problem: can the mobility records be used for inferring users' economic statuses, and if so, to which extent and how? This is motivated by the following considerations including (1) users' economic statuses are closely linked to where they live or work, both of which could be potentially reflected by their mobility records; and (2) users' economic statuses can sometimes be disclosed by the places they visit, especially those they visit during weekends, which again could be revealed by their mobility records. To this end, we investigate techniques for (1) taking the GPS mobility

[1]We remark that those research efforts are generally conducted on the data publicly available, and thus no privacy will be broken.

records as inputs, (2) incorporating economic contexts for learning user socioeconomic status, (3) extracting various types of features from the inputted mobility records data and context data, and (4) inferring the users' socioeconomic statuses that are unknown with their mobility records. Each of these elements makes it distinctive from existing studies. We note that we study a general problem in trajectory inference (i.e., inferring users' economic statuses). Here, a user's socioeconomic status can refer to many different indicators, such as the price range of the user's living house [53, 54], the likelihood that the user will pay a car loan installment on time, or the user's income, etc. Constrained by the availability of datasets and privacy concerns, we infer the home location of a user based on his/her mobility records (i.e., Geolife) and then crawl the house price data from the Web based on the home location as the proxy of the user's socioeconomic status. Since both the mobility records data and the house price data are publicly available, no privacy will be broken in this study. In addition, we note that the application is ethical. In general, there is an agreement need to be approved by users in advance before the research activity. For example, users the in car loan business need to sign an agreement of collecting their GPS trajectories for this research purpose.

1.3.3 Methodology Overview

We propose a deep model for user socioeconomic status inference, called `DeepSEI`. The `DeepSEI` model incorporates two networks for the task, i.e., *Deep Network* and *Recurrent Network*. The Deep Network aims to capture some statistics based on users' mobility records and the Recurrent Network aims to capture the sequential patterns behind users' mobility records. We train our model in a supervised manner by concatenating the outputs of the two networks, and training with the collected economic contexts (e.g., house price) as labels to infer their different socioeconomic classes.

In the deep network, we capture the features including spatiality diversity, temporality diversity and activity diversity. The rationale of designing the features is that (1) spatiality diversity is typically to capture the spatial information in the territory where users' daily activities are conducted; (2) temporality diversity is designed to capture the temporal regularity of users, which can potentially help to indicate their professions, e.g.,

self-employers tend to stay at home and only go out occasionally, while some users working at a government department would commute more regularly; (3) activity diversity reveals the diversity of movements among users' activity locations, which are associated with users' economic statuses as shown in [52, 53].

In the recurrent network, we employ a hierarchical LSTM to capture the users' sequential activities, where each activity involves spatial, temporal and semantic features. For the hierarchical LSTM, the activities within a day are modeled in low-level LSTM and the activities within days are modeled in high-level LSTM. The hierarchical LSTM brings two advantages. First, users' mobility records are generally long, and the hierarchical structure can reduce the length and alleviate the issue of degraded performance for handling long sequences. Second, users' mobility records are on a daily basis, and the two-level LSTMs preserve the users' periodic information. The rationale of designing the features is that (1) spatial and (2) temporal features are two fundamental features to indicate where and when the activities are conducted; (3) semantic features, e.g., working or shopping, infers the activity types and provides the context for understanding a user's daily routine.

1.3.4 Contributions

In summary, we make the following contributions in socioeconomic status inference from trajectory data. We study a novel problem of inferring users' economic statuses based on their GPS mobility records. This problem is new and has practical applications in real life (e.g., risk assessment for car loan applications/managements). We propose a novel analytical framework called `DeepSEI` for the problem, which is a supervised deep learning model and incorporates two neural networks (i.e., deep network and recurrent network) to capture the features from three aspects of users' mobility records, i.e., spatiality, temporality and activity.

Chapter 2

Literature Review

In this chapter, we briefly introduce the background of trajectory data, and review the existing literature on trajectory data simplification, similarity search and inference of socioeconomic status. We review the issues of existing works, and point out literature for our research problems in this direction. We then survey the related work of deep reinforcement learning and deep representation learning. In addition, we emphasize the differences between our studies and existing studies.

The movement of an object is usually captured by sampling its locations at a certain frequency with tracking technologies such as those based on GPS devices. As a result, the movement of an object corresponds to a sequence of time-stamped locations, which is called a trajectory. In recent years, we have witnessed an unprecedented growth in trajectory data. This big amount of data has attracted many research activities to acquire knowledge from the data, which serves many spatial-temporal applications, including intelligent transportation [1], location-based recommendation service [2], pandemic tracking [3], and crime prevention for public safety [4].

2.1 Trajectory Simplification

Trajectory simplification is a common practice in trajectory data pre-processing. It drops some of the points in a trajectory when they are being collected at the sensor side (called online mode) or after they are accumulated at the server side (called batch mode). There are two dual problems in trajectory simplification, namely Error-Bounded Trajectory Simplification and Size-Bounded Trajectory Simplification.

2.1.1 Error-Bounded Trajectory Simplification

Recall that in the Error-Bounded problem, it aims to drop as many points as possible from a trajectory and keep the information loss bounded by a given error tolerance. We review the literature in terms of two different settings (i.e., online mode and batch mode). In the online mode, the trajectory is fed point by point in an online fashion and those points that have been dropped during the trajectory simplification process will no longer be accessible. In the batch mode, all the points in the trajectory are fed together and remain accessible during the process.

Online Mode. Many existing algorithms have been developed for error-bounded trajectory simplification (i.e., min-size problem), including `OPW` [8], `CISED-S` [9], `BQS` [10, 11], `FBQS` [10, 11], `OPERB` [12], `Intersect` [13], `Angular` [14], and `Interval` [15]. They all adopt the three-step process, but adopt different strategies for expanding and re-opening a window in Step 2 and Step 3, respectively. Specifically, `BOPW`, `NOPW` and `MOPW` are three variants of `OPW`. They check the error exactly and expand a window by including one point in Step 2. When re-opening a window in Step 3, `BOPW` chooses the last safe point, `NOPW` chooses the first point that would cause the tolerance violation, and `MOPW` chooses the safe point such that dropping it incurs the greatest error. These three algorithms have the time complexity of $O(n^2)$ and are applicable to multiple error measurements. `CISED-S` proposes an upper bound estimation via a spatiotemporal cone intersection technique for error checking and expands a window by including one point in Step 2 and adopts the same strategy as BOPW for Step 3. It has the time complexity of $O(n)$ and works for the synchronous Euclidean distance (SED) error measurement only. `BQS` adopts the same strategies for expanding and re-opening windows as BOPW and improves the error checking procedure in Step 2. The time complexity remains $O(n^2)$ in the worst case. Further, `FBQS` reduces the time complexity of `BQS` to $O(n)$ by estimating an upper bound of the error in Step 2. `OPERB` adopts the same strategies as BOPW and achieves a fast upper bound computation in Step 2 based on a local distance checking method. `OPERB` has the time complexity of $O(n)$. `BQS`, `FBQS` and `OPERB` are applicable for the perpendicular Euclidean distance (PED) error measurement only. `Intersect`, `Augular` and `Interval` adopt the same strategies as BOPW. They differ from each other in estimating a different upper bound in Step 2 and each of them has a time complexity of $O(n)$. They are

applicable to the direction-based distance (DAD) error measurement only. Our solution MARL4TS is based on a learned policy for decision making via multi-agent reinforcement learning instead of utilizing some pre-defined rules as these existing algorithms do. In addition, MARL4TS is applicable to multiple error measurements.

Batch Mode. There exist other related work on trajectory simplification that does not target the online scenario, i.e., they require the access to all data points in a trajectory throughout the trajectory simplification process, which is different with the online setting studied in this thesis. For example, in [13], SP is proposed to exactly solve a trajectory simplification problem via the shortest path algorithm. DP [27] is proposed to iteratively split the whole trajectory into two sub-trajectories and continues the process until each of the sub-trajectories cannot be approximated by a line linking its starting point and ending point under the given error tolerance. Bottom-Up [28] is a reversed version of DP by repetitively dropping a point that would introduce the smallest error from the whole trajectory until dropping any point violates the tolerance. SQUISH-E [21] follows the framework of Bottom-Up, but estimates an upper bound error on each point.

2.1.2 Size-Bounded Trajectory Simplification

Recall that in the Size-Bounded problem, it drops some points of a trajectory and keeps the information loss as small as possible. Similarly, we discuss the literature in terms of online mode and batch mode.

Online Mode. The size-bounded trajectory simplification problem is to decide which points to be dropped and correspondingly which to be kept in the buffer and sent to the server's side later on. The problem is to find a simplified trajectory within a given storage budget and its error minimized.

In the online mode, streaming trajectory data is continuously collected by sensors and stored in a local buffer. Among the existing studies, [19–21] target the size-bounded problem. In [19, 55], the STTrace algorithm is proposed, which processes incoming points one by one and for each one, it first decides whether to drop it. If so, it moves to the next one; and if not, it drops one existing point in the buffer, and then inserts the point that is being processed into the buffer. The decision making is based on some heuristic values defined for the points. In [20], the authors propose the SQUISH algorithm and in [21]

they propose an enhanced version of SQUISH called SQUISH-E. SQUISH and SQUISH-E follow the framework of STTrace, but use different definitions for the heuristic values of points. Each of these algorithms has the time complexity of $O((n-W)\log W)$. Our solution differs from these methods in that it is based on a policy learned via reinforcement learning instead of human-crafted heuristic values for decision making.

Batch Mode. In the batch mode, all the points in the trajectory that is to be simplified are inputted together and remain accessible throughout the simplification process. Among the existing studies, [8, 28, 29, 56] target the size-bounded problem. Specifically, in [56], the authors propose a dynamic programming algorithm called *Bellman*. Bellman runs in at least cubic time, which is prohibitively expensive for large datasets. In [8, 28, 55], the authors explore different approximate algorithms for the problem, including *Top-Down* and *Bottom-Up*. Top-Down is inspired by the traditional *Douglas-Peucker* [57] algorithm and the idea is to start with two points (the first one and the last one) and then repetitively include a point that has the largest error until the number of points reaches the storage budget. Top-Down has the time complexity of $O(Wn)$. Bottom-Up starts with all points of the inputted trajectory and repetitively drops a point that would introduce the smallest error until the number of points left is within the storage budget. Bottom-Up has the time complexity of $O((n-W)(n'+\log n))$, where n' is bounded by n. In [29], the authors propose an approximate algorithm called *Span-Search*, which is specifically designed for the error measurement direction-aware distance (DAD). Span-Search has the time complexity of $O(cn\log^2 n)$, where c is a moderate constant. Again, these methods are mainly based on human-crafted rules, while our method is based on a learned policy.

2.1.3 Other Trajectory Compression Studies

Trajectory compression [58–61] is a related but different problem from trajectory simplification. Trajectory compression typically exploits underlying road networks. Specifically, raw trajectories are initially map-matched to an underlying road network to obtain map-matched trajectories that consist of sequences of road segments. The map-matched trajectories are treated as strings, where each road segment is considered as a character.

Then, string compression algorithms such as Huffman coding [62] can be utilized to compress the trajectories with or without information loss. In contrast, we study the general trajectory simplification problem without assuming the availability of a road network, which is used widely to compress trajectories in free space, such as animal trajectories [13] and pedestrian trajectories [63].

Trajectory quantization [64] is another method to encode each trajectory point with a small number of bits, and it does not assume the availability of a road network. The main difference with trajectory simplification is in two aspects. First, it operates the compression in a global nature (i.e., compressing a database of trajectories), and it cannot support the online scenario studied in trajectory simplification, where the trajectory points arrive in streaming within a limited storage budget. Second, it targets a query-friendly compression (e.g., a high accuracy of trajectory path query on compressed trajectories). This is not suitable for some applications studied in trajectory simplification, where it specifies some error metrics, e.g., DAD, whose information needs to be preserved for a concrete application.

2.2 Trajectory Similarity Computation and Search

2.2.1 Trajectory Similarity Measurements

Measuring the similarity between trajectories is a fundamental problem and has been studied extensively. Some classical solutions focus on indexing trajectories and performing similarity computation by the alignment of matching sample points. For example, DTW [38] is the first attempt at solving the local time shift issue for computing trajectory similarity. Frechet distance [65] is a classical similarity measure that treats each trajectory as a spatial curve and takes into account the location and order of the sampling points. Further, ERP [33] and EDR [34] are proposed to improve the ability to capture the spatial semantics in trajectories. However, these point-matching methods are inherently sensitive to noise and suffer from quadratic time complexity. EDS [35] and EDwP [36] are two segment-matching methods, which operate on segments for matching two trajectories. In recent years, some learning-based algorithms were proposed to speed up the similarity computation. Li et al. [37] propose to learn representations of trajectories in the form of vectors and then measure the similarity between two trajectories as

the Euclidean distance between their corresponding vectors. Some other studies [66–68] define similarity measurements on trajectories based on road segments, to which the trajectories are matched. Yao et al. [69] employ deep metric learning to approximate and accelerate trajectory similarity computation. In addition, Ma et al. [70] propose a similarity measurement called p-distance for uncertain trajectories and study the problem of searching for top-k similar trajectories to a given query trajectory based on p-distance. Different specialized index techniques are developed for these similarity measures, such as DTW distance [38, 71], LCSS [72], ERP [33], EDR [34], and EDwP [36]. However, these index techniques do not generalize to other similarity measures or subtrajectory similarity search. Here, we assume an abstract trajectory similarity measurement, which could be instantiated with any of these existing similarity measurements and our techniques still apply.

2.2.2 Subtrajectory Similarity Related Problems

Measuring subtrajectory similarity is also a fundamental functionality in many tasks such as clustering [30–32] and similarity join [39]. Lee et al [31] propose a general partition and group framework for subtrajectory clustering. Further, Buchin et al. [32] show the hardness of subtrajectory clustering based on Frechet distance, and Agarwal et al. [30] apply the trajectory simplification technique to approximate discrete Frechet to reduce the time cost of subtrajectory clustering. Recently, Tampakis et al. [39, 40] proposed a distributed solution for subtrajectory join and clustering by utilizing the MapReduce programming model. Although these algorithms need to consider subtrajectory similarity, similarity computation is not their focus and they usually first segment a trajectory into subtrajectories and employ an existing measure, such as Frechet distance.

2.2.3 Subsequence (Substring) Matching

Subsequence matching is a related but different problem to similar subtrajectory search. It aims to find a subsequence that has the same length as the query in a given candidate sequence, which usually contains millions or even trillions [73, 74] of elements. Efficient pruning algorithms [73–79] have been proposed for the matching, and these pruning

algorithms are generally designed for a specific similarity measure, such as DTW [73–77, 80, 81] and Euclidean distance [78, 79], and cannot generalize to other measures. On the other hand, substring matching [44, 45] often focuses on approximate matching based on the Edit distance. It aims to find a substring in a string to best match the query. Similar subtrajectory search problem differs from the substring matching problem mainly in two aspects. First, characters in a string have an exact match (0 or 1) in the alphabet; however, the points of a trajectory are different. Second, substring matching techniques are usually designed based on the characteristics of strings. e.g., grammar structure patterns, or word concurrence patterns; however, a trajectory does not have such patterns.

2.2.4 Multi-trajectory Similarity Related Problems

Another related study is regarding multi-trajectory (or a set of trajectories) similarity. He et al. [82] study trajectory set similarity on road networks, in which the idea of the Earth Mover's Distance (EMD) is leveraged to capture both spatial and temporal characteristics of trajectories. Sha et al [41, 83] measure the similarity between two plays (a set of multiple trajectories of players and the ball) by first aligning trajectories from two plays (based on the extracted roles of trajectories) and then aggregating the similarities between aligned trajectories as one between the two plays. Compared with these studies, we develop the first unsupervised deep learning model, play2vec, to learn representations of plays (or multi-trajectory), which is superior over existing ones in aspects of effectiveness, efficiency, and robustness.

2.3 Trajectory Analytics

2.3.1 Human Mobility and Socioeconomic Status Inference

Previous works are related to the studies on inferring socioeconomic status from users' mobility records. Earlier studies [84–88] examine the relationships between human travel behaviors with their profiles, including gender [84, 87, 88], age [87, 88], race [87], employment status and income [85]. For example, Hanson et al. [85] suggest that individual's employment status and income have a positive impact on his/her travel frequency. Kwan

et al. [87] reveal that men are more inclined to visit recreational places than women. These earlier works make a great impact on the subsequent research; however, their works are conducted on travel survey data, which needs to be collected manually, and consequently the findings are mainly based on a small number of volunteers for a short period of time.

With the proliferation of communication techniques, mobile phone data becomes a new data source for conducting this type of research. Mobile phone data is collected from cellphone users, where the mobile phone records track the user id, the user's location when he/she makes a phone call (the location is reported as the longitude and latitude), and the timestamp at which the phone call starts. With the mobile phone data, Xu et al. [89] present a home-based approach to analyze human activities in Shenzhen, and find that people who live in the northern part of Shenzhen are generally with a small activity space around their home, and people with a larger activity space mainly live in the southern part, where the economy is highly developed. Further, they propose an analytical framework for understanding the relationships between human mobility and socioeconomic status [53]. Specifically, they take two cities: Singapore and Boston for case studies, and reveal an interesting finding that the richer tend to travel shorter in Singapore but longer in Boston. Blumenstock et al. [90] reveal that mobile phone data can be used to predict individual socioeconomic status, and further the regional socioeconomic status (e.g., poverty and wealth levels). In addition, Huang et al. [91] explore possible factors that may influence individual daily activities, and the results demonstrate that socioeconomic status, urban spatial structure, work place and region geographical layout all play a critical role. Kelly et al. [92] use the location data collected from mobile sensors to identify some predictability patterns that can be linked to users' demographics such as age, gender and social meeting contacts, etc. Different from these studies, we propose to infer users' socioeconomic statuses with their GPS trajectories and develop a deep learning based model called DeepSEI.

2.3.2 Mobility and Temporality Prediction

We review the existing studies regarding the prediction task, where mobility and temporality information is involved. For mobility prediction, with the proliferation of location-based services, it has been a hot research topic in recent years. Mobility prediction aims

at predicting the next location for the user, and POI recommendation aims to predict the following several locations that the user will visit. The earlier works [93, 94] for the prediction task are based on exacting the historical mobility patterns with prior knowledge. In recent years, many learning-based methods [95–97] are proposed to model the users' mobility patterns in a data-driven manner. For example, DeepMove [96] is an attentional recurrent neural network based model to capture the user's periodical patterns for his/her mobility prediction, which is effective with two attention mechanisms. One is to attend to historical records with the users' current status, and the other one is to preserve the sequential information among historical records with the recurrent module. In addition, Wang et al. [97] propose a hybrid predictive model for mobility prediction, which incorporates both the regularity and conformity of human mobility into the model. Chen et al. [95] propose a context-aware deep model called DeepJMT to jointly predict where and when a user will visit the next location, and thus DeepJMT takes both user's spatial and temporal patterns into the prediction.

For temporality prediction, many existing studies [98–101] adopt the temporal point process [102] to model the time as a sequence of discrete random events, and then jointly predict the next event type and timestamp. Du et al. [100] apply RNN to learn a representation of influences from the event history for the prediction, and then develop an efficient stochastic gradient algorithm to speed up model training on millions of data points. Further, Xiao et al. [99] synergically model the time series and event sequence for the prediction, where time series and event sequence are fed into two RNNs, and then the outputs of the two RNNs are concatenated to an embedding mapping layer to predict the next event type and timestamp. In addition, Yan et al. [101] propose to improve the temporal point process with discriminative and adversarial learning, which aims to generate the temporal sequences that are close to the real distribution. Our problem differs from these studies mainly in that we aim to predict users' socioeconomic statuses but not their mobility and/or temporality.

2.4 Deep Learning Techniques

2.4.1 Deep Reinforcement Learning

The goal of reinforcement learning is to guide agents on how to take actions to maximize a cumulative reward [103] in an environment, and the environment is generally modeled as a Markov decision process (MDP) [18]. Recently, RL models have been utilized successfully to solve some database related problems. For example, Zhang et al. [104] and Li et al. [105] use the RL model for automatic DBMS tuning. It utilizes a policy gradient method to find the optimal configurations of knob settings with a limited number of samples to accomplish the task. Trummer et al. [106] use RL to learn optimal join orders in the SkinnerDB system. It divides a query execution into many small time slices, where different join orders are tried with RL in different time slices. Then, it merges the result according to different join orders until an overall result is obtained. Wang et al. [107] design an effective RL-based algorithm for bipartite graph matching. Specifically, it studies a dynamic bipartite graph matching problem to better align with the real-world taxi dispatching applications, where the passengers need be assigned dynamically to taxi drivers within a limited waiting time. It utilizes a RL-based algorithm to conduct the dynamic task assignment, which aims to yield the highest overall revenue. Overall, there are two types of popular reinforcement learning methods: (1) model-based methods [108, 109] that require understanding the environment and learning the parameters of the MDP in advance, and (2) model-free methods [17, 110] that make no effort to learn a model and get feedback from the environment step by step.

2.4.2 Multi-agent Reinforcement Learning

In recent years, multi-agent reinforcement learning [111, 112] has also attracted much research attention. For example, a cooperative RL model is proposed in [113] to solve the problem of the express system. Specifically, it partitions a city into many independent regions, and assigns the same number of couriers to serve the express requests cooperatively. As a request comes, a proposed cooperative multi-agent RL model is learned to guide where should each courier serve for a short period. And this is further used for courier dispatching policies in express system [114]. To alleviate traffic congestion,

Lin et al [111] design a contextual multi-agent model to manage the vehicles in a city. Specifically, it targets a large-scale fleet management problem, where the environment is with complex dynamics between demand and supply. The proposed multi-agent RL model aims to coordinate vehicles adaptive to different contexts.

2.4.3 Deep Representation Learning

Inspired by the success of word2vec, the idea of representation learning [115] is widely used for many tasks such as natural language processing [116] and graph embedding [117]. The Skip-Gram with Negative Sampling (SGNS) model [46] is one of the common methods of word2vec which is based on the assumption in linguistics that words frequently occurring in a sentence tend to share more statistical information. Seq2Seq based learning model has achieved good performance on spatiotemporal data [37, 118]. For example, Li et al. [37] adapt a Seq2Seq framework based on RNN and takes the final hidden vector of the encoder to represent a trajectory. It computes the similarity between two trajectories based on the Euclidean distance between their representations as vectors. The ability to capture the local spatial correlation makes it inherently applicable to various downstream analysis tasks. Our proposed model play2vec for play retrieval (i.e., multi-trajectory similarity search) is inspired by the Seq2Seq model and the SGNS architecture.

2.5 Summary

In summary, we review the literature in terms of three research problems studied in this thesis (i.e., trajectory simplification, trajectory similarity computation and search, and trajectory analytics). In addition, we review the deep learning techniques (i.e., deep reinforcement learning, multi-agent reinforcement learning, and deep representation learning) behind the problems. Next, we present more technical details for the problems.

Part I

Trajectory Data Simplification

Chapter 3

Multi-agent Reinforcement Learning for Error-Bounded Online Trajectory Simplification

3.1 Overview

In this chapter, we study the problem of Error-Bounded Online Trajectory Simplification (EB-OTS) [1]. We propose the first multi-agent RL-based method MARL4TS for the EB-OTS problem. This chapter is organized as follows. We review some basic concepts and provide the problem definition in Section 3.2. We present our `MARL4TS` algorithm in Section 3.3 and present the experimental results in Section 3.4. Finally, we conclude our work in Section 3.5.

3.2 Problem Statement

(1) Trajectory. A trajectory, denoted by T, consists of a sequence of time-stamped locations to capture the trace of a moving object. Let $T =< p_1, p_2, ..., p_n >$ be a trajectory, where p_i has the form of a triple (x_i, y_i, t_i), which denotes a moving object is at i^{th} location (x_i, y_i) and t_i is the time stamp of the location. We use $|T|$ to denote the size of the trajectory T (i.e., $|T| = n$). We use $T[i : j]$ $(i \leq j)$ to denote the subtrajectory of T, which starts form point p_i and ends at point p_j. Let $\overline{p_i p_{i+1}}$ $(1 \leq i \leq n-1)$ denote the

[1] This chapter was published as Error-Bounded Online Trajectory Simplification with Multi-Agent Reinforcement Learning [119].

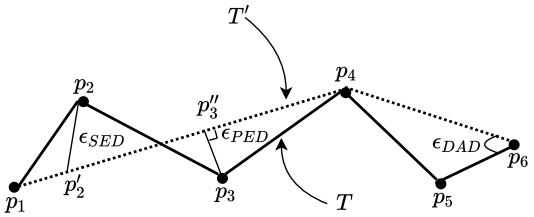


Figure 3.1: Error measurements.

line segment which connects two locations (x_i, y_i) and (x_{i+1}, y_{i+1}), meaning that an object moves from (x_i, y_i) to (x_{i+1}, y_{i+1}) along the line segment at a constant speed. Therefore, the speed can be calculated as $\frac{d(p_i,p_{i+1})}{t_{i+1}-t_i}$, where $d(\cdot,\cdot)$ denotes the Euclidean distance.

(2) Trajectory Simplification. A new trajectory resulted from dropping some points from an original trajectory T corresponds to a simplified trajectory of T. A simplified trajectory of T is denoted by T', where T' consists of $< p_{s_1}, p_{s_2}, ..., p_{s_m} >$, $m \leq n$ and $1 = s_1 < s_2 ... < s_m = n$. We take Figure 3.1 for example, $T =< p_1, p_2, p_3, p_4, p_5, p_6 >$ is an original trajectory and $T' =< p_1, p_4, p_6 >$ is the simplified trajectory of T by dropping points p_2, p_3 and p_5.

(3) Error Measurements. Within the time period $[t_{s_j}, t_{s_{j+1}}]$ $(1 \leq j \leq m-1)$, based on the simplified trajectory T', it represents that the object moves along the segment $\overline{p_{s_j} p_{s_{j+1}}}$, while based on the original trajectory T, it represents that the object moves along a sequence of segments which are formed by a sequence of points, that is $p_{s_j}, p_{s_j+1}, ..., p_{s_{j+1}}$. The segment $\overline{p_{s_j} p_{s_{j+1}}}$ in simplified trajectory T' approximates a sequence of segments in the original trajectory starting from the points $p_{s_j}, p_{s_j} + 1, ..., p_{s_{j+1}-1}$. The segment $\overline{p_{s_j} p_{s_{j+1}}}$ is denoted as the anchor segment of points $p_{s_j}, p_{s_j+1}, ..., p_{s_{j+1}-1}$. Thus, for points in the original trajectory T, except the point p_n, all points including $p_1, p_2, ..., p_{n-1}$ have anchor segments. For example, in Figure 1, $\overline{p_1 p_4}$ approximates a sequence of three segments $\overline{p_1 p_2}$, $\overline{p_2 p_3}$ and $\overline{p_3 p_4}$ in T and corresponds to the anchor segment of points p_1, p_2 and p_3. Note that the movement along the the simplified trajectory T' (e.g., $\overline{p_1 p_4}$) and original trajectory T (e.g., $\overline{p_1 p_2}$, $\overline{p_2 p_3}$ and $\overline{p_3 p_4}$) has a discrepancy, and we measure the discrepancy as the error of T' wrt T by $\epsilon(T')$. There are several error measurements proposed for defining $\varepsilon(T')$, and they share the following common ideas: (1) The error is defined on a segment $\overline{p_{s_j} p_{s_{j+1}}}$ wrt a point p_i $(s_j \leq i < s_{j+1})$, whose anchor segment is $\overline{p_{s_j} p_{s_{j+1}}}$, denoted by $\epsilon(\overline{p_{s_j} p_{s_{j+1}}} | p_i)$; (2) The error is defined as the maximum error among

all points on the anchor segment $\overline{p_{s_j}p_{s_{j+1}}}$, that is $\epsilon(\overline{p_{s_j}p_{s_{j+1}}}) = \max_{s_j \le i < s_{j+1}} \epsilon(\overline{p_{s_j}p_{s_{j+1}}}|p_i)$; (3) The error of simplified trajectory T' is then defined as the maximum error among all segments in T', i.e., $\epsilon(T') = \max_{1 \le j \le m-1} \epsilon(\overline{p_{s_j}p_{s_{j+1}}})$. That is, these error measurements, including Synchronized Euclidean Distance (SED) [8, 19–21], Perpendicular Euclidean Distance (PED) [8, 10, 11, 56], Direction-aware Distance (DAD) [13–15, 29], and Speed-aware Distance (SAD) [21], define $\epsilon(T')$ as follows.

$$\epsilon(T') = \max_{1 \le j \le m-1} \max_{s_j \le i < s_{j+1}} \epsilon(\overline{p_{s_j}p_{s_{j+1}}}|p_i) \tag{3.1}$$

Note that the definition of maximum error of a segment in T' is followed by almost all existing studies, and very few work [20] try to define it by summation because it is sensitive to the size of trajectory, i.e., the error would be accumulated with a trajectory gets longer. To calculate $\epsilon(\overline{p_{s_j}p_{s_{j+1}}}|p_i)$, different measurements use different functions, which can be found in [5, 7] and omitted here.

(4) Problem Definition (EB-OTS). We study the EB-OTS problem, which is defined as follows.

Problem 1 (EB-OTS) *Given a raw trajectory $T =< p_1, p_2, ..., p_n >$ that is inputted in a streaming fashion and an error tolerance ε_t, the EB-OTS problem is to find a simplified trajectory T' of T such that $\epsilon(T') \le \varepsilon_t$ and $|T'|$ is minimized.Here, $\epsilon(T')$ could be instantiated with SED, PED, DAD, and SAD.*

3.3 Proposed Method

We observe that the three-step process as described in Section 1.1.3 corresponds to a *sequential decision process*, i.e., it sequentially makes decisions on how to expand a window if possible and how to choose a safe point for re-opening a window if expanding a window is not possible. Therefore, we propose to leverage reinforcement learning (RL) to help the decision making of the three-step process. In particular, RL is widely used to guide agents on how to take actions to maximize a cumulative reward in an environment, and the environment is generally modelled as a *Markov decision process* [18] (MDP). In this chapter, we introduce two agents, namely Agent-E and Agent-R, to help with expanding a window and re-opening a window, respectively. We capture the decision

problems of the two agents with MDP models (Section 3.3.1 and Section 3.3.2), learn the optimal policies for the MDPs via Deep-Q-Network (DQN)[17] (Section 3.3.3), and utilize the learned policies of Agent-E and Agent-R for expanding a window and re-opening a window in the three-step process, respectively (Section 3.3.4). We call the resulting algorithm `MARL4TS`.

3.3.1 MDP for Expanding a Window (Agent-E)

Consider the task of expanding a window in Step 2 of the three-step process. Existing algorithms all adopt a pre-defined rule of expanding a window by including one following point outside the window. While this strategy gives each point a chance to be chosen for re-opening a window in Step 3 if it is a safe point, it is somehow a bit too conservative. For instance, in a scenario where the underlying object moves along a straight line and at a constant speed, we have much confidence to drop the corresponding sampled points. This motivates us to include a variable number of points for expanding a window so as to reduce the number of times of checking the error of dropping points from a window. Specifically, we introduce an agent called Agent-E for deciding how many points to be included for expanding a window. We capture this decision making problem as a MDP, involving states, actions, transitions and rewards, which are defined as follows.

(1) States. We denote a state of Agent-E's MDP by s^e. Suppose the window involves points $p_l, p_{l+1}, ..., p_r$, which we denote by $W[l, r]$. Intuitively, the state should capture essential information of the window for deciding how many points to be included when expanding the window. We propose to define the state as a pair of two numbers, namely, (1) the ratio between the distance from p_l to p_r when traveled along the raw trajectory and that when traveled along the segment $\overline{p_l p_r}$ and (2) the number of points in the window. Formally, the state s^e is defined as follows.

$$s^e := (\frac{\sum_{l \leq h < r} d(p_h, p_{h+1})}{d(p_l, p_r)}, r - l + 1) \tag{3.2}$$

The intuition of the above definition is as follows. The first number captures how smooth the underlying trace is to some extent, e.g., if it is close to 1, it means that the trace is close to a straight line, and if it is significantly larger than 1, it means that trace involves big turns. The second number captures the size the window. In addition, with

this definition, the state can be updated in $O(1)$ time, which is important since the main purpose of Agent-E is to boost the efficiency of the three-step process.

(2) Actions. We denote an action of Agent-E's MDP by a^e. Suppose the current window is $W[l, r]$. We define J possible actions for expanding $W[l, r]$, namely, (1) expanding to $W[l, r + 1]$, (2) expanding $W[l, r + 2]$, ..., (J) expanding to $W[l, r + J]$. Here, J is a hyperparameter, which is to be decided via empirical studies. Formally, we define a^e as follows.

$$a^e := j \quad (1 \leq j \leq J) \tag{3.3}$$

where $a^e = j$ means to expand $W[l, r]$ to $W[l, r + j]$. Note that when J is set to 1, it reduces to the conventional rule of always expanding a window by including one following point outside the window.

(3) Transitions. Let s^e be a state and $W[l, r]$ be the window at s^e. Suppose we take an action $a^e = j$, which expands the window to $W[l, r + j]$. There are two cases. Case 1: the error of dropping the points in the window, i.e., $\epsilon(\overline{p_l p_r})$, is bounded by the error tolerance ϵ_t. In this case, it would continue to expand the window $W[l, r+j]$ (in Step 2). A new state can be computed based on the window $W[l, r + j]$. Case 2: it corresponds to the other case. In this case, it would re-open a window (in Step 3) and then expand it (Step 2). A new state can be computed based on the re-opened window. Note that a re-opened window can always be expanded further since it involves exactly two points.

(4) Rewards. When expanding a window, it does not drop any points and thus the rewards cannot be immediately observed. Since Agent-E and Agent-R cooperate for the same objective, i.e., to drop as many points as possible without violating the error tolerance, we let Agent-E and Agent-R share their rewards. Specifically, we set the reward of performing an action of expanding a window to be equal to that of the following action of re-opening a window (which will be introduced in Section 3.3.2).

3.3.2 MDP for Re-Opening a Window (Agent-R)

Consider the task of re-opening a window in Step 3 of the three-step process. The key is to choose a safe point in the window since once it has been chosen, a window can be re-opened by including the chosen safe point and the point following the chosen

safe point. Existing algorithms all use pre-defined rules for choosing a safe point, e.g., choosing the *last* safe point in the windows. These pre-defined rules are simple heuristics and cannot adapt to different inputs with different settings. In addition, there is no theoretical ground supporting that they optimize the objective of the EB-OTS problem. We introduce an agent called Agent-R for deciding which safe point in the window to choose for re-opening a window. We capture this decision making problem as a MDP, involving states, actions, transitions and rewards, which are defined as follows.

(1) States. We denote a state of Agent-R's MDP by s^r. Suppose that the current window is $W[l, r]$ and M safe points have been marked in Step 2 since the current window is re-opened. We denote these safe points by p_{s_1}, p_{s_2}, ..., p_{s_M}. We need to choose a point among these M safe points. An immediate idea is to incorporate all M safe points for defining the state. Nevertheless, it would have two issues. First, the state defined in this would be M-dependent and cannot be used for other cases where the number of safe points is not M. Second, M is typically a large integer and causes a huge state space, which makes the model difficult to train.

We propose to focus on the last K safe points, i.e., $p_{s_{M-K+1}}$, $p_{s_{M-K+2}}$, ..., p_{s_M}, for defining a state, where K is a hyperparameter that can be tuned. The rationale is that when re-opening a window to start at a safe point that appear later would mean to drop more points. In case that there are fewer than K safe points, we simply duplicate the last safe point a few times to obtain K safe points. For each safe point p_{s_i} considered, we define a pair of two numbers, namely (1) the error of the segment linking p_l and p_{s_i} and (2) the numer of points that would be dropped if point p_{s_i} is chosen for re-opening a window. We denote the pair by $c(p_{s_i})$, i.e., $c(p_{s_i}) = (\epsilon(\overline{p_l p_{s_i}}), s_i - l + 1)$. Note that the number $\epsilon(\overline{p_l p_{s_i}})$ has already been computed when checking whether the window $W[l, s_i]$ can be expanded in Step 2. These numbers capture the the consequence of choosing p_{s_i} for re-opening a window to some extent. Formally, we define s^r as follows.

$$s^r := \{c(p_{s_{M-K+1}}), c(p_{s_{M-K+2}}), ..., c(p_{s_M})\}. \tag{3.4}$$

The above definition is M-independent. In addition, with this definition, the state space is controllable via the hyperparameter K. Note that we would normalize the values of the state in each dimension to avoid the potential data scale issues.

(2) Actions. We denote an action of Agent-R's MDP by a^r. We define K actions, each to choose one of the last K safe points for re-opening a window. The rationale is the same as that of defining the state s^r. Formally, we defne a^r as follows.

$$a^r := k \quad (0 \leq k \leq K-1) \tag{3.5}$$

where the action $a^r = k$ means to choose the safe point $p_{s_{M-k}}$, drop all the points between p_l and $p_{s_{M-k}}$ (both exclusively), and re-open a window involving $p_{s_{M-k}}$ and $p_{s_{M-k}+1}$, i.e., $W[s_{M-k}, s_{M-k}+1]$.

(3) Transitions. After we take an action a^r at a state s^r, which re-opens a window. It would continue to expand the window iteratively until it is not possible to do so. By then, some safe points would have been marked, based on which, we can compute a new state $s^{r'}$, which corresponds to the next state of s^r.

(4) Rewards. Consider that Agent-R performs an action a^r at a state s^r and then it arrives at a new state $s^{r'}$. We define the reward associated with this transition from state s^r to state $s^{r'}$, which we denote by R, as follows. Let $W[l, r]$ and $W[l', r']$ be the windows before and after the action a^r is taken, respectively. Recall that the action a^r involves the process of dropping the points between p_l and $p_{l'}$. We therefore define the reward to be equal to the number of points dropped. Formally, we define R as follows.

$$R := l' - l - 1 \tag{3.6}$$

The intuition is that if more points are dropped, it is more towards the objective of the EB-OTS problem and thus the reward should be larger. It is clear that with this reward definition, the objective of the Agent-R's MDP, i.e., maximizing the accumulative rewards, is equivalent to that of the EB-OTS problem, i.e., maximizing the number of points dropped. A formal verification of this equivalence is as follows. Suppose that the Agent-R goes through a sequence of N states, s_1^r, s_2^r, ..., s_N^r and correspondingly it receives a sequence of $N-1$ rewards, R_1, R_2, ..., R_{N-1}. Let $W[l_1, r_1]$, $W[l_2, r_2]$, ..., $W[l_N, r_N]$ be the windows at the N states. Note that $l_1 = 1$ and N is the number of points kept among the first l_N points. Then, the accumulative rewards can be computed as follows.

$$\sum_{t=1}^{N-1} R_t = \sum_{t=1}^{N-1} (l_{t+1} - l_t - 1) = l_N - l_1 - N + 1 = l_N - N \tag{3.7}$$

where $l_N - N$ corresponds to the number of points dropped before point p_{l_N}.

3.3.3 Learning Policies of MDPs

We adopt the Deep-Q-Network (DQN) [120] method for learning the policies from the MDPs of Agent-E and Agent-R, which has been widely used for MPDs with continuous state spaces. The major idea of DQN is as follows. It first defines an optimal action-value function $Q^*(s, a)$ (called Q function), which represents the maximum expected accumulative rewards it would receive by following any policy after taking the action a at the state s. Formally, $Q^*(s, a)$ is defined as follows.

$$Q^*(s, a) = E_{s'}[R + \gamma \cdot \max_{a'} Q^*(s', a')] \tag{3.8}$$

where s' is a possible next state and R is the reward observed after action a is taken at a given state s, and γ is a discount factor. It then estimates $Q^*(s, a)$ by a parameterized neural network denoted by $Q^*(s, a; \theta)$ (for details, please refer to [120]). It finally returns the policy, which always chooses for a given state s the action a that maximizes $Q^*(s, a; \theta)$. We denote the estimated Q functions of Agent-E and Agent-R by $Q^*(s, a; \theta^e)$ and $Q^*(s, a; \theta^r)$, respectively.

3.3.4 The MARL4TS Algorithm

The `MARL4TS` algorithm follows the three-step process and uses Agent-E and Agent-R for expanding and re-opening a window, respectively. We present `MARL4TS` in Algorithm 1. Specifically, it first opens a window $W[1, 2]$ (line 1-2). It then enters a while loop (line 3-22). Let $W[l, r]$ be the current window (line 5). It checks whether $\epsilon(\overline{p_l p_r})$ is bounded by the error tolerance. If so, it performs the following steps (line 7-14). It first marks p_r as a safe point (line 7). It then terminates the loop if p_r is the last point by dropping all the points in the window except for p_l and p_r (line 8-11) and expands the window with Agent-E otherwise (line 12-14). If not, it chooses a safe point with Agent-R (line 17-18), drops all points between the first point and the chosen safe point in the window (line 19) and re-opens a new window involving the chosen safe point and the point following the chosen safe point (line 20). Finally, it returns a simplified trajectory involving all points that have not been dropped (line 23).

We illustrate the MARL4TS algorithm with an example shown in Table 3.1, where the inputted trajectory is shown in Figure 3.2. It first opens a window $W[1, 2]$. Since

Algorithm 1: The MARL4TS algorithm

Input: A trajectory $T =< p_1, p_2, ..., p_n >$ which is inputted in an online fashion; A given error tolerance ε_t;
Output: A simplified trajectory T' under $\varepsilon(T') \leq \varepsilon_t$;

```
1  //Step 1
2  Open a window W[1, 2];
3  while true do
4      //Step 2
5      Let W[l, r] denote the current window;
6      if ε(p_l p_r) ≤ ε_t then
7          Mark p_r as a safe point;
8          if p_r is the last point, i.e., p_n then
9              Drop all points between p_l and p_r;
10             Break;
11         end
12         Construct a state s^e (as in Equation (3.2));
13         Choose an action a^e = arg max_a Q(s^e, a; θ^e);
14         Expand the window to W[l, r + j] where a^e = j (Round r + j to be n if p_{r+j} will not
             arrive, i.e., r + j > n);
15     else
16         //Step 3
17         Construct a state s^r (as in Equation (3.4));
18         Choose an action a^r = arg max_a Q(s^r, a; θ^r);
19         Drop all points between p_l and the chosen safe point indicated by a^r;
20         Re-open a window involving the chosen safe point and the point following the chosen
             safe point;
21     end
22 end
23 Return trajectory T' consisting of all points that have not been dropped;
```

$\epsilon(\overline{p_1 p_2})$ is bounded by ε_t, it marks p_2 as a safe point and uses Agent-E to expand the window to $W[1, 3]$. Similarly, it expands the window to $W[1, 8]$ with Agent-E. Since $\epsilon(\overline{p_1 p_8})$ is not bounded by ε_t, it uses Agent-R to choose a safe point p_7, drops the points p_2, p_3, p_4, p_5, p_6 and re-opens a window $W[7, 8]$. Since $\epsilon(\overline{p_7 p_8})$ is bounded by ε_t and p_8 is the last point, it breaks from the loop and returns the simplified trajectory $T' =< p_1, p_7, p_8 >$.

We further develop two techniques for boosting the effectiveness and efficiency of MARL4TS as follows. First, in Step 3 of the three-step process, before we choose a safe point, we check for each of the D points that follow p_r, i.e., the last point in the window, whether it is a safe point. Here, D is a hyperparameter. We then focus on the last K safe points among all safe points in the window and those from the D points for defining a state and choosing among them a safe point for re-opening a window. The rationale

Table 3.1: Running example of MARL4TS with PED.

Initial	$l = 1$, $r = 2$ and $\varepsilon_t = 1.0$				
$W[l, r]$	$\epsilon(\overline{p_l p_r})$	Safe points	Agent	State	Action
$W[1, 2]$	$\epsilon(\overline{p_1 p_2}) = 0.0 < \varepsilon_t$	$< p_2 >$	E	$s_1^e = \{1.0, 2\}$	Expand to p_3
$W[1, 3]$	$\epsilon(\overline{p_1 p_3}) = 0.5 < \varepsilon_t$	$< p_2, p_3 >$	E	$s_2^e = \{1.118, 3\}$	Expand to p_5
$W[1, 5]$	$\epsilon(\overline{p_1 p_5}) = 0.5 < \varepsilon_t$	$< p_2, p_3, p_5 >$	E	$s_3^e = \{1.160, 5\}$	Expand to p_7
$W[1, 7]$	$\epsilon(\overline{p_1 p_7}) = 0.5 < \varepsilon_t$	$< p_2, p_3, p_5, p_7 >$	E	$s_4^e = \{1.177, 7\}$	Expand to p_8
$W[1, 8]$	$\epsilon(\overline{p_1 p_8}) = 1.029 > \varepsilon_t$	$< p_2, p_3, p_5, p_7 >$	R	$s_1^r = \{(0.5, 5), (0.5, 7)\}$	Re-open at p_7
$W[7, 8]$	$\epsilon(\overline{p_7 p_8}) = 0.0 < \varepsilon_t$	$< p_8 >$	-	-	-
Output	Return $T' =< p_1, p_7, p_8 >$				

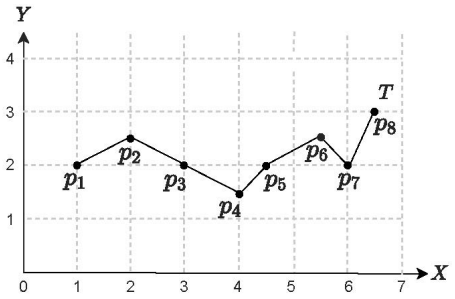


Figure 3.2: A problem input.

is to consider potentially more safe points for more effective trajectory simplification, which is verified via experiments. Second, in Step 2 of the three-step process, we impose a constraint that the window to be expanded has its size bounded by C. Here, C is a hyperparameter. In the case the size of an expanded window exceeds C, its right end would be shifted towards the left until the size reaches C. The rationale is that the cost of checking the error of dropping all points in a window (except for the first and the last points) depends on the size of the window, i.e., it is $O(n)$ in the worst-case if the size of a window is not bounded. With a bounded window size, the cost would be $O(C)$. The benefit of this technique for improving the efficiency is verified in experiments.

Time complexity. The time complexity of the `MARL4TS` algorithm is $O(n)$, where n denotes the number of points in the inputted trajectory T, which we analyze as follows. The cost is dominated by those of Step 2 and Step 3. In Step 2, the cost of one transition consists of (1) that of checking the error of a segment, which is bounded by $O(1)$ given that C is a constant and (2) that of constructing a state and sampling an action for Agent-E, which is $O(1)$. In Step 3, the cost of one transition includes (1) that of constructing

Table 3.2: Dataset statistics I.

Statistics	Geolife	T-Drive	Indoor
# of trajectories	17,621	10,359	529,397
Total # of points	24,876,978	17,740,902	330,117,253
Ave. # of points per trajectory	1,412	1,713	624
Sampling rate	1s ~ 5s	177s	0.03s ~ 0.06s
Average distance	9.96m	623m	0.037m

the state based on the last K safe points for Agent-R, which is $O(1)$ given that K is a constant and the errors of segments have already been computed during Step 2 and (2) that of sampling an action, choosing a safe point, dropping points, which is $O(1)$. In addition, Step 2 involves at most nC transitions and Step 3 involves no more transitions than Step 3. Therefore, the time complexity of `MARL4TS` is $O(nC) \times O(1) = O(n)$.

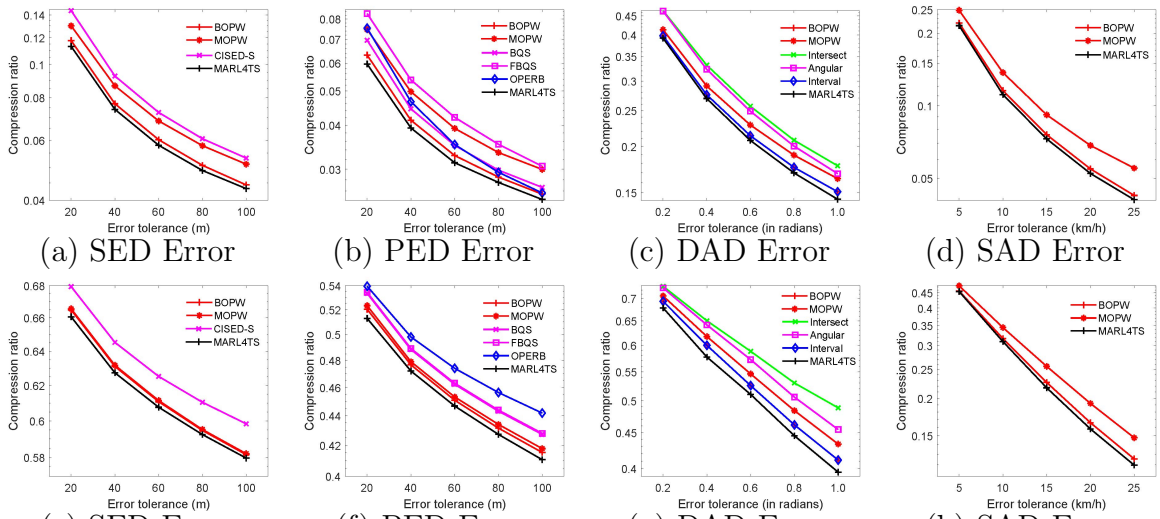
3.4 Experiments

3.4.1 Experimental Setup

Dataset. We conduct our experiments on three real-world trajectory datasets, namely Geolife, T-Drive and Indoor, which are widely used for evaluating the performance of trajectory simplification in previous work [12, 13, 29] including the recent dedicated evaluation work [5]. We show the statistics of datasets in Table 3.2. Specifically, Geolife[2] records the outdoor trajectories of 182 users in different transportation modes like walking or driving for a period of five years. T-Drive [3] tracks the trajectories of 10,357 taxis in Beijing, and Indoor [4] contains the trajectories of visitors in the "ATC" shopping center in Osaka, Japan from October 2012 to November 2013.

Baselines. To evaluate the effectiveness and efficiency of the proposed multi-agent reinforcement learning method (called `MARL4TS`), we have carefully examined the evaluation paper [5] and the recent literature to cover all baselines that are proposed or can be adapted for the EB-OTS problem, including `OPW` [8], `CISED-S` [9], `BQS` [10, 11],

[2]http://research.microsoft.com/en-us/downloads/b16d359d-d164-469e-9fd4-daa38f2b2e13/
[3]http://research.microsoft.com/apps/pubs/?id=152883
[4]http://www.irc.atr.jp/crest2010_HRI/ATC_dataset/



(a) SED Error (b) PED Error (c) DAD Error (d) SAD Error

(e) SED Error (f) PED Error (g) DAD Error (h) SAD Error

Figure 3.3: Effectiveness with varying the error tolerance ε_t (Geolife (a)-(d), T-Drive (e)-(h)).

`FBQS` [10, 11], `OPERB` [12], `Intersect` [13], `Angular` [14], and `Interval` [15]. The descriptions of these algorithm can be found in Section 2.1.1.

Parameter Setting. The neural network used in the `MARL4TS` involves two agents, i.e., Agent-E and Agent-R. For Agent-E, we use a feedforward neural network with 2 layers. In the first layer, we use the tanh function with 23 neurons, and in the second layer, we use the linear function with J neurons as the output corresponding to different actions. In order to avoid the data scale issues if any, batch normalization is employed before the activation. For Agent-R, it involves one hidden layer followed by one output layer. The hidden layer has 25 neurons with the tanh function as the activation and the output layer has K neurons corresponding to different actions. Before training the neural networks, we shuffle the states which is organized by the order of K tuples to improve the model quality [121]. The default hyperparameters J, K, D, C are set as 2, 5, 5 and 50, respectively. The effects of these parameters are studied in experiments. We randomly sample 40%, 40%, 2% trajectories from Geolife, T-Drive, Indoor dataset for training and choose the best models during the process, and the remaining trajectories are used for testing. For both agents in the training process, the size of replay memory M is set at 2000. In addition, we train the model using Adam stochastic gradient descent with an initial learning rate of 0.01. The minimal ϵ is set at 0.1 with decay 0.99 for the ϵ-greedy strategy, and the reward discount rate γ is set at 0.99.

Table 3.3: Ablation study of the learned policy (Geolife).

Measurement	SED		PED	
Algorithm	Ratio	Time (s)	Ratio	Time (s)
MARL4TS	11.864%	0.88	6.304%	0.46
w/o Agent-E	11.857%	0.97	6.299%	0.56
w/o Agent-R	13.386%	0.61	6.744%	0.33
w/o Agent-E and Agent-R	13.019%	0.70	6.632%	0.36

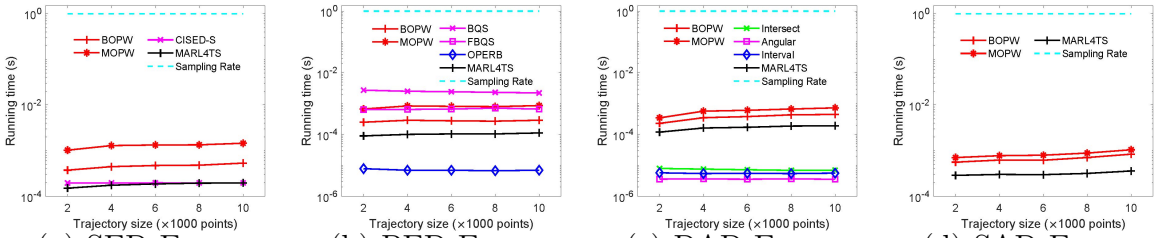
Evaluation Platform. All the methods are implemented in Python 3.6. The implementation of MARL4TS is based on Keras 2.2.0. The experiments are conducted on a machine with Intel(R) Xeon(R) CPU E5-1620 v2 @3.70GHz 16.0GB RAM and one Nvidia GeForce GTX 1070 GPU.

3.4.2 Experimental Results

(1) Effectiveness evaluation (comparison with existing approximate algorithms). We randomly sample 1000 trajectories from a dataset and vary the error tolerance ε_t from 20m to 100m for SED and PED by following [9, 12], 0.2 radian to 1.0 radian for DAD by following [13], and 5km/h to 25km/h for SAD by following [21]. Figure 3.3 show the average compression ratios of MARL4TS and the existing algorithms on Geolife and T-Drive. MARL4TS consistently outperforms all the existing algorithms under all error measurements due to its data-driven nature. For example, BOPW is the best baseline for SED, PED and SAD, and MARL4TS outperforms it by 4% (resp. 5% and 4%) for SED (resp. PED and SAD); Interval is the best for DAD, and MARL4TS outperforms it by 3%. Note that the improvement contributes a lot in practice because the storage overhead of trajectories is generally huge.

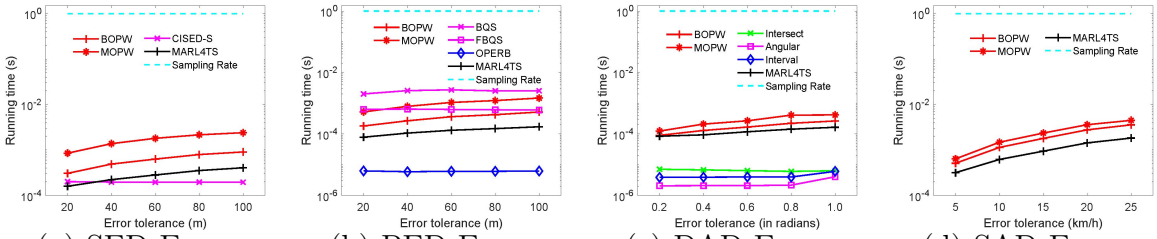
(2) Effectiveness evaluation (ablation study of the learned policy). To show the effect of the learned policy in MARL4TS model, we conduct an ablation study by replacing the policy of Agent-E, Agent-R, or both Agent-E and Agent-R with the greedy rules. Table 3.3 reports the average compression ratio and running time on 1,000 randomly sampled trajectories (ε_t is fixed at 20m) under the SED and PED error from Geolife. We can see all these components contribute to the effectiveness or efficiency. For example, Agent-E improves the efficiency by 9.3% and Agent-R improves the effectiveness by

11.4%. As expected, these components illustrate a trade-off between the efficiency and effectiveness. The results based on other error measurements or datasets have a similar trend and thus omitted.



(a) SED Error (b) PED Error (c) DAD Error (d) SAD Error

Figure 3.4: Efficiency with varying the trajectory length $|T|$ (Geolife).



(a) SED Error (b) PED Error (c) DAD Error (d) SAD Error

Figure 3.5: Efficiency with varying the error tolerance ε_t (Geolife).

(3) Efficiency evaluation (varying the trajectory length $|T|$). To evaluate the effect of trajectory length $|T|$ on efficiency, we follow [12, 13] by varying $|T|$ from 20,000 to 100,000. For each setting, we randomly select 100 trajectories and fix ε_t at 20m for SED and PED, 0.4 radian for DAD, and 5km/h for SAD. We report the average processing time per point in Figure 3.4, since a trajectory is fed point by point for the online scenario. In addition, we show the sampling rate (1s) of the Geolife dataset with a dotted line for ease of reference. We observe that `MARL4TS` is slower than some baselines that are designed for a specific error (i.e., `OPERB` for PED, `Intersect`, `Angular` and `Interval` for DAD), though they have the same time complexity. This is because these algorithms employ some specific techniques to an error measurement, e.g., `OPERB` developed a local distance checking method for PED, while `MARL4TS` is based on a general design that can adapt to different error measurements. In addition, we notice `MARL4TS` is much faster than the opening window algorithms (i.e., `BOPW`, `NOPW` and `MOPW`) since it employs Agent-E to prune the computation of error checking and impose the window

size, thus the time cost is saved correspondingly. Overall, `MARL4TS` runs faster than most of existing baselines and would meet the practical needs: for a trajectory with about 10,000 points, it takes around 0.1ms per point, which is 10,000 times faster than the sampling rate (1s). The results on the other datasets are qualitatively similar as those reported and are omitted.

(4) Efficiency evaluation (varying the error tolerance ε_t). To study the efficiency with the varying error tolerance ε_t, We set ε_t from 20m to 100m for PED and SED by following [12], 0.2 radian to 1.0 radian for DAD by following [13], and 5km/h to 25km/h for SAD on Geolife. For each setting, we randomly sample 100 trajectories with a fixed $|T|$ at 50,000. Figure 3.5 shows the average running time of different algorithms for each measure. Similarly, `MARL4TS` is slower than some algorithms that are specifically designed for one error measure, but run reasonably fast. For example, it takes around 1ms per point for a trajectory with 50,000 points. The running time for all the methods slightly increases with ε_t. This is because the window would be incrementally extended with a larger setting of ε_t, and the corresponding time cost on checking the window increases.

More experiments. More experimental results regarding (5) effectiveness evaluation on Indoor, (6) comparison with the exact algorithm SP, (7) effectiveness evaluation (varying parameter J, K, D, C), (8) case study, (9) training time and (10) transferability test can be found in [119].

3.5 Summary

In this chapter, we study the error-bounded online trajectory simplification called EB-OTS, which is to drop as many points as possible in a streaming fashion and keep the information loss, captured as the "error", is bounded by an error tolerance. We propose a multi-agent RL-based method called MARL4TS. Compared with existing algorithms, MARL4TS is data-driven and can adapt different error measurements. We conduct extensive experiments on real-world trajectory datasets, which show the MARL4TS simplifies trajectories with consistently lower compression ratios and runs comparably fast.

Chapter 4

Deep Reinforcement Learning for Size-Bounded Trajectory Simplification

4.1 Overview

In this chapter, we study the problem of Size-Bounded Trajectory Simplification [1]. We propose a RL-based method for the trajectory simplification problem. This chapter is organized as follows. We give the problem definition in Section 4.2. We introduce our algorithms for the online and batch modes in Section 4.3.1 and in Section 4.3.2, respectively. We present our experimental results in Section 4.4 and finally conclude our work in Section 4.5.

4.2 Problem Statement

The detailed definitions of trajectory, trajectory simplification and error measurements can be found in Section 3.2. We study the size-bounded problem as defined below.

Problem 2 (Size-Bounded) *Given a trajectory* $T =< p_1, p_2, ..., p_n >$ *and a storage budget* W*, which is an integer, the* ***Size-Bounded*** *problem is to find a simplified trajectory* $T' =< p_{s_1}, p_{s_2}, ..., p_{s_m} >$ *where* $m \leq n$ *and* $1 = s_1 < s_2 < ... < s_m = n$ *such that* $|T'| \leq W$ *and* $\epsilon(T')$ *is minimized, and* $\epsilon(T')$ *can be defined by any of those existing error measurements including SED, PED, DAD, and SAD.*

[1] This chapter was published as Trajectory Simplification with Reinforcement Learning [7].

The Size-Bounded problem has two modes, namely the online mode and the batch mode, for different application scenarios.

Online mode. In the online mode, the trajectory to be simplified is fed to the system point by point in an online fashion and those points that have been dropped during the trajectory simplification process will no longer be accessible. This mode is commonly used in applications such as remote sensing, where the sensors collect points from time to time and are constrained by storage budget, network bandwidth and energy.

Batch mode. In the batch mode, all the points in the trajectory to be simplified are fed together, and remain accessible during the simplification process. This mode is usually used at a server's side and the purpose is to reduce the storage cost (e.g., after the simplification, the original trajectory is discarded) and/or the query processing cost (e.g., it performs queries on the simplified trajectory data instead of the original data).

4.3 Proposed Method

4.3.1 Algorithms for Online Mode

In this section, by Size-Bounded, we mean the problem in the online mode unless specified otherwise. In the online mode, points are inputted one by one in an online fashion, while only a buffer with size W is available, i.e., at most W points can be retained throughout the trajectory simplification process. We adopt an existing strategy [19–21] that for the first W points, we store them in the buffer directly and for each of the remaining points, since the buffer is already full, we drop one point in the buffer to release some space and then store the new point in the buffer. Different from those existing strategies, which use some human-crafted heuristic values for deciding which point to drop when the buffer is full, we aim to achieve a more intelligent method for this decision-making task. Specifically, we treat the trajectory simplification problem as a *sequential decision making process* and model it as a *Markov decision process* (MDP) [18] (Section 6.3.2.1), use a policy gradient method [22–24] for learning an optimal policy for the MDP (Section 4.3.1.2), and then develop an algorithm called *RLTS*, which uses the learned policy for the Size-Bounded problem (Section 4.3.1.3). In Section 4.3.1.4, we present a variant of RLTS, called *RLTS-Skip*, which boosts the efficiency of RLTS via skipping some points from being scanned.

4.3.1.1 Size-Bounded Modeled as an MDP

We model the Size-Bounded problem as an MDP, which consists of four components, namely *states*, *actions*, *transitions*, and *rewards* as defined below.

(1) States. Consider a situation where there are W points $p_{s_1}, p_{s_2}, ..., p_{s_W}$ in the buffer and a newly inputted point p_i ($i > W$) is to be inserted into the buffer next. The task is to drop one point from the buffer and then insert the point p_i into the buffer. Conceptually, it is equivalent to the process that we first append the point p_i to the buffer, i.e., the buffer becomes $p_{s_1}, p_{s_2}, ..., p_{s_W}, p_{s_{W+1}}$, where $p_{s_{W+1}} = p_i$, and then we drop one point p_{s_j} ($2 \leq j \leq W$) from the buffer. Note that by the definition of trajectory simplification, we are not allowed to drop point p_{s_1}, i.e., p_1. An intuitive idea is to drop one of those points such that the error that is introduced as a consequence of the dropping operation is small. If we drop the point p_{s_j} ($2 \leq j \leq W$), two existing segments $\overline{p_{s_{j-1}} p_{s_j}}$ and $\overline{p_{s_j} p_{s_{j+1}}}$ would be destroyed, one new segment $\overline{p_{s_{j-1}} p_{s_{j+1}}}$ would be created, and other segments are unchanged. Since the error of a simplified trajectory is determined by those of its segments and the error of a segment is further determined by its errors wrt the points in the original trajectory that take the segment as their anchor segments, the error of the newly created segment $\overline{p_{s_{j-1}} p_{s_{j+1}}}$ wrt the point p_{s_j}, i.e., $\epsilon(\overline{p_{s_{j-1}} p_{s_{j+1}}} | p_{s_j})$, captures the consequence of dropping p_{s_j} well.

Motivated by this, we define for each point p_{s_j} ($2 \leq j \leq W$), a *value*, denoted by $v(p_{s_j})$, as follows.

$$v(p_{s_j}) := \epsilon(\overline{p_{s_{j-1}} p_{s_{j+1}}} | p_{s_j}) \tag{4.1}$$

We note that for DAD and SAD, $\epsilon(\overline{p_{s_{j-1}} p_{s_{j+1}}} | p_{s_j})$ depends on segment $\overline{p_{s_j} p_{s_j+1}}$. In the online mode, p_{s_j+1} may not be accessible, and thus we use segment $\overline{p_{s_j} p_{s_{j+1}}}$ instead (i.e., we measure the angular and speed difference between segment $\overline{p_{s_{j-1}} p_{s_{j+1}}}$ and segment $\overline{p_{s_j} p_{s_{j+1}}}$). A lower value means that once the point is dropped, the introduced error tends to be smaller and thus it should be dropped with a higher chance. Then, we define the state of the situation, which we denote by s, based on the values of the points in the buffer. An immediate idea is to incorporate the values of all $(W - 1)$ points p_{s_j} ($2 \leq j \leq W$) in the buffer for defining the state. However, this definition has two issues. First, since W is an input to the problem and for different problem instances, W is usually

different. With this definition, the model that is defined for one input W would not be usable for other inputs different from W. Second, W is typically a moderate to large integer, e.g., it could be in thousands. With this definition, the state space would be huge and the model be hard to train.

We propose to define the state s as the set containing k lowest values, where k ($k \leq W-1$) is hyper-parameter that could be tuned, instead of the set containing all $(W-1)$ values. Specifically, we let π denote the permutation of $s_2, ..., s_W$ such that $v(p_{\pi(1)}), v(p_{\pi(2)}), ..., v(p_{\pi(W-1)})$ is a list of the values in an ascending order. Then, we define state s as follows.

$$s := \{v(p_{\pi(1)}), v(p_{\pi(2)}), ..., v(p_{\pi(k)})\} \tag{4.2}$$

With this definition, a state is of a fixed size that is independent from the problem input. In addition, the state space is controllable via the parameter k.

(2) Actions. Suppose that there are $W+1$ points $p_{s_1}, p_{s_2}, ..., p_{s_{W+1}}$ in the buffer (conceptually) and $s = \{v(p_{\pi(1)}), v(p_{\pi(2)}), ..., v(p_{\pi(k)})\}$ is the corresponding state. Essentially, the task is to pick a point among $p_{s_2}, ..., p_{s_W}$ and drop it. An immediate idea is to define $(W-1)$ actions, each for a point p_{s_j} $(2 \leq j \leq W)$, but then there would be two issues similar to those when we discuss a straightforward method for defining a state, namely (1) the definition would be W-dependent and thus it is not flexible and (2) the action space would be large and thus the model is hard to train. In fact, it is intuitive to restrict our attention to those points with small values since dropping one of these points incurs a small consequent error. Therefore, we focus on those points with their values maintained in the state, i.e., $p_{\pi(1)}, p_{\pi(2)}, ..., p_{\pi(k)}$.

With all these, we define an action space containing k actions, each meaning to drop a point $p_{\pi(j)}$ $(1 \leq j \leq k)$. Formally, we define an action, which we denote by a, as follows.

$$a := j \quad (1 \leq j \leq k) \tag{4.3}$$

where the action $a = j$ means that it drops the point $p_{\pi(j)}$.

(3) Transitions. Suppose a is an action to drop point p_{s_j} $(2 \leq j \leq W)$. After the action a is taken, W points are left in the buffer. When a new point p_i, is inserted, we need to compute a new state, which we denote by s', i.e., state s' would be the next state when

action a is taken at state s. We update the state s to state s' as follows. Recall that a state is mainly about the values of the points in the buffer, and in order to compute the state s', we examine how the points in the buffer and their corresponding values would have changed after p_{s_j} is dropped and a new point p_i is inputted.

First, we consider the consequence of dropping p_{s_j}. After the point p_{s_j} is dropped, only two neighboring points, namely $p_{s_{j-1}}$ and $p_{s_{j+1}}$, could have their anchor segment changed. Specifically, point $p_{s_{j-1}}$'s (if $j-1 \geq 2$) anchor segment would be changed from $\overline{p_{s_{j-2}}p_{s_j}}$ to $\overline{p_{s_{j-2}}p_{s_{j+1}}}$ and point $p_{s_{j+1}}$'s (if $j+1 \leq W-1$) anchor segment would be changed from $\overline{p_{s_j}p_{s_{j+2}}}$ to $\overline{p_{s_{j-1}}p_{s_{j+2}}}$. Therefore, the values of the two points need to be updated, which we do as follows.

$$v(p_{s_{j-1}}) = \max\{\epsilon(\overline{p_{s_{j-2}}p_{s_{j+1}}}|p_{s_{j-1}}), \epsilon(\overline{p_{s_{j-2}}p_{s_{j+1}}}|p_{s_j})\} \tag{4.4}$$

$$v(p_{s_{j+1}}) = \max\{\epsilon(\overline{p_{s_{j-1}}p_{s_{j+2}}}|p_{s_{j+1}}), \epsilon(\overline{p_{s_{j-1}}p_{s_{j+2}}}|p_{s_j})\} \tag{4.5}$$

Here, we include the two errors (i.e., $\epsilon(\overline{p_{s_{j-2}}p_{s_{j+1}}}|p_{s_j})$ and $\epsilon(\overline{p_{s_{j-1}}p_{s_{j+2}}}|p_{s_j})$) wrt the point p_{s_j} for updates. The rationale is as follows. Recall that $v(p_{s_{j-1}})$ is defined to capture the consequence of dropping $p_{s_{j-1}}$. Ideally, $v(p_{s_{j-1}})$ should be defined as the error of the segment $\overline{p_{s_{j-2}}p_{s_{j+1}}}$, i.e., $\max_p \epsilon(\overline{p_{s_{j-2}}p_{s_{j+1}}}|p)$, where p is a point that takes segment $\overline{p_{s_{j-2}}p_{s_{j+1}}}$ as the anchor segment. In the online mode, among those points that take segment $\overline{p_{s_{j-2}}p_{s_{j+1}}}$ as the anchor segment, only $p_{s_{j-1}}$ and p_{s_j} are accessible when p_{s_j} is being dropped. Therefore, we include $\epsilon(\overline{p_{s_{j-2}}p_{s_{j+1}}}|p_{s_j})$ for defining $v(p_{s_{j-1}})$. Similarly, we include $\epsilon(\overline{p_{s_{j-1}}p_{s_{j+2}}}|p_{s_j})$ for defining $v(p_{s_{j+1}})$.

Second, we consider the consequence of inserting point p_i. With point p_i inserted to the buffer, there would be $W+1$ points, which we still denote by $p_{s_1}, p_{s_2}, ..., p_{s_{W+1}}$. Note that $p_{s_{W+1}}$ corresponds to p_i. The value of p_{s_W}, which is previously not defined (since it is the last point in the buffer before p_i is inserted), needs to be defined as follows.

$$v(p_{s_W}) = \epsilon(\overline{p_{s_{W-1}}p_{s_{W+1}}}|p_{s_W}) \tag{4.6}$$

The values of all other points are unchanged. Based on these values, we compute the state s' in the same way as we compute the state s (Section 4.3.1.1).

(4) Rewards. Consider that we perform an action a at a state s and then we arrive at a new state s'. We define the reward associated with this transition from state s to

state s', which we denote by r, as follows. At state s, we have W points in the buffer and a new point p_i to be inserted. We consider those points in the buffer, which constitute a trajectory that corresponds to a simplified trajectory of the trajectory fed so far, i.e., $T[1:i-1]$. We denote this simplified trajectory by T'. Similarly, at state s', we have a simplified trajectory of $T[1:i]$, which we denote by T''. We then define the reward r as follows.

$$r = \epsilon(T') - \epsilon(T'') \tag{4.7}$$

where $\epsilon(T')$ is wrt $T[1:i-1]$ and $\epsilon(T'')$ is wrt $T[1:i]$. The intuition is that if the error of the simplified trajectory resulted from the action, i.e., $\epsilon(T'')$, is smaller, then the reward is larger. With this definition, it would favor those actions that lead to simplified trajectories with smaller errors. In fact, it could be verified that with this reward definition, the goal of the MDP problem is well aligned with that of the trajectory simplification problem. To see this, suppose that we go through a sequence of states $s_1, s_2, ..., s_N$ and correspondingly, we receive a sequence of rewards $r_1, r_2, ..., r_{N-1}$. In the case that the future rewards are not discounted, we have

$$\sum_{t=1}^{N-1} r_t = \sum_{t=1}^{N-1} (\epsilon(T'_t) - \epsilon(T''_t)) = \epsilon(T'_1) - \epsilon(T''_{N-1}) = -\epsilon(T''_{N-1}) \tag{4.8}$$

where T'_t (resp. T''_t) is the simplified trajectory at the state s_t before (resp. after) the action a_t is performed. Note that $\epsilon(T'_1) = 0$ since at the start state, no points have been dropped and thus the error is equal to 0, and $\epsilon(T''_{N-1})$ corresponds to the error of the simplified trajectory of T.

Remarks. We note that for the computation of the reward r, which involves the computations of the errors of two simplified trajectories, is only required for the learning process. Therefore, we can use a repository of trajectories for the learning process, and once a policy has been learned, we use it for trajectory simplification. In addition, we note that we can compute $\epsilon(T'_t)$'s and $\epsilon(T''_t)$'s *incrementally* except for $\epsilon(T'_1)$ since T''_t corresponds to T'_t with one point dropped ($1 \le t \le N$) and T'_{t+1} corresponds to T''_t with one point inserted ($1 \le t \le N-1$).

4.3.1.2 Policy Learning on the MDP

The core problem of an MDP is to find an optimal *policy* for the agent, which corresponds to a function that specifies the action that the agent should choose at a specific state so as to maximize the accumulative rewards. We learn the policy for the MDP via a *policy gradient* (PNet) method, which is widely used [22–24]. PNet models a stochastic policy as $\pi_\theta(a|s)$, which means the probability of selecting an action a for a given state s. PNet parameterizes $\pi_\theta(a|s)$ using a neural network as follows.

$$\pi_\theta(a|s) = \sigma(\boldsymbol{W}s + \boldsymbol{b}) \tag{4.9}$$

where σ denotes the softmax function and $\theta = \{\boldsymbol{W}, \boldsymbol{b}\}$ denotes the parameters of the neural network. Then, PNet computes the gradients of some performance measure wrt the parameters θ as follows.

$$\nabla_\theta J(\theta) = \sum_{t=1}^{N} \frac{R_t - \overline{R}}{\sigma_R} \nabla_\theta \ln \pi_\theta(a_t|s_t) \tag{4.10}$$

where $s_1, s_2, ..., s_N$ is a sequence of states, $a_1, a_2, ..., a_{N-1}$ is a sequence of actions by sampling $\pi_\theta(a|s_t)$, R_t denotes the accumulative reward since action a_t is taken, $\overline{R}$ is the mean of R_t's, and σ_R is the standard deviation of R_t's. PNet repeatedly updates the parameters θ via *gradient ascent* based on the gradients computed in Equation (4.10) until some stopping criterion specified by users is satisfied. Note that PNet corresponds to a variant of the REINFORCE algorithm with baseline [22–24] and the normalization mechanism based on the mean and standard deviation helps reduce the variance of the computed gradients in Equation (4.10).

4.3.1.3 The RLTS Algorithm

Our RLTS algorithm is based on the learned policy for the MDP that models the Size-Bounded problem, which is presented in Algorithm 2. Specifically, it stores the first W points in the buffer directly (Line 1 - 3). It initializes an index t for a sequence of states and actions to be traversed (Line 4); Then, for each of the following points, says, p_i, it proceeds as follows (Line 5). It computes (incrementally if possible) the values of the points in the buffer except for the first one, i.e., $v(p_{s_j})$ $(2 \leq j \leq W)$, and maintains

Algorithm 2: The RLTS algorithm

Input: A trajectory $T =< p_1, p_2, ..., p_n >$ which is inputted in an online fashion; A buffer with a storage budget W ($W < n$);
Output: A simplified trajectory T' of T with $|T'| \leq W$;

```
1  for i = 1, 2, ..., W do
2  |  Store point p_i into the buffer;
3  end
4  t ← 1;
5  for i = W + 1, W + 2, ..., n do
6  |  Compute (incrementally if possible) the values of the points in the buffer except for the
   |    first one, i.e., v(p_{s_j}) (2 ≤ j ≤ W) and maintain the values in a min-priority queue with
   |    the ascending permutation denoted by π;
7  |  Construct a state s_t ← {v(p_{π(1)}), v(p_{π(2)}), ..., v(p_{π(k)})};
8  |  Sample an action a_t ~ π_θ(a|s);
9  |  Drop the point p_{π(j)} from the buffer where a_t = j;
10 |  Insert the point p_i into the buffer;
11 |  t ← t + 1;
12 end
13 Trajectory T' ← the sequence of points in the buffer;
14 Return trajectory T';
```

the values in a min-priority queue with the ascending permutation denoted by π (Line 6). It then constructs a state s_t containing the set of k lowest values of points (Line 7), and samples an action a_t based on the stochastic policy that has been learned (Line 8). Based on the sampled action $a_t = j$, it drops the point with the j^{th} lowest value, i.e., $p_{\pi(j)}$ (Line 9). It then inserts the point p_i that is being processed (Line 10). Finally, it returns a simplified trajectory T' which has the W points in the buffer (Line 13 - 14).

Time complexity. The time complexity of the RLTS algorithm is $O((n - W) \log W)$, where n is the number of points in the input trajectory T and W denotes the storage budget of the buffer. To see this, the complexity is dominated by the part of processing the last $(n - W)$ points (Line 5 - 12 in Algorithm 2), and the time cost of processing one point consists of (1) that of computing the state whose cost is $O(1)$ incrementally (Line 6); (2) that of maintaining the min-priority queue is $O(\log W)$ (Line 6); and (3) that of constructing a state, sampling an action, dropping a point, inserting a point and updating the index is $O(1)$ assuming k is a constant (Line 7 - 10). We note that this time complexity is the same as those of existing algorithms [19–21] and all algorithms can meet practical requirements as shown in our experiments (e.g., the time of processing one point is much less than 1ms on a moderate machine). Compared with existing algorithms,

RLTS is based on a learned policy but not some human-crafted rules, and thus it could return simplified trajectories with smaller errors.

To analyze the time cost of learning the policy, we first consider the time cost of learning on one trajectory of length n for one epoch. The time cost should be $O((n - W)(n + \log W))$ since compared with the RLTS algorithm, the training process involves two additional costs, namely that for computing the reward after an action is taken and that for performing the gradient decent over a neural network. The former has the cost of $O(n)$ and the latter has a constant cost given that the neural network we use has a small size that is independent of the problem size. Let N_t be the number of trajectories and E be the number of epochs for training the policy. The time cost of training is $O(E \cdot N_t \cdot (n - W)(n + \log W))$.

4.3.1.4 The RLTS-Skip Algorithm

In the RLTS algorithm, each point is inserted to the buffer for sure after it is scanned. It may then be dropped when some following points are being scanned. Consequently, when each point is being scanned, some efforts are spent on deciding which existing point in the buffer to be dropped by going through a neural network in RLTS. While this strategy gives each point a chance to be included in the buffer and thus exploring a large space of possible simplified trajectories, it may be too conservative. Consider a scenario where the points that are scanned most recently constitute a trajectory that indicates a movement along a straight line with a constant speed. In this scenario, we have much confidence to drop a certain number of points in a row without including them one by one to the buffer and then dropping some of them at later stages. This would help save the efforts for deciding and taking actions when scanning these points and at the same time, the effectiveness should not be affected much.

Motivated by this, we propose to augment the MDP that is defined in Section 6.3.2.1 by introducing J additional actions when scanning a point p_i, namely (1) dropping p_i and continuing to scan p_{i+1}, (2) dropping p_i and p_{i+1} and continuing to scan p_{i+2}, ..., and (J) dropping p_i, p_{i+1}, ..., and p_{i+J-1} and continuing to scan p_{i+J}. Here, J is a hyper-parameter, which could be tuned. These actions essentially mean (1) skipping 1 point, (2) skipping 2 points, ..., and (J) skipping J points during the process of scanning the

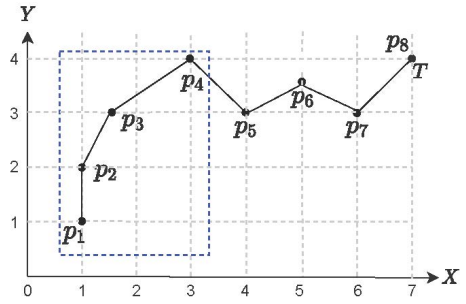


Figure 4.1: A running example.

points of a trajectory sequentially. As a result, the augmented MDP involves $(k + J)$ actions, namely the k actions of the original MDP as defined in Section 4.3.1.1 and the J actions as newly introduced in this section. Note that these $(k + J)$ actions are exclusive and at each state, only one of them could be taken. For an action of dropping a point, the reward is defined the same as in the online mode. For an action of skipping j points, the reward is still defined by Equation (4.7), but with T'' changed to be the points stored in the buffer plus p_{i+j}, which corresponds to a simplified trajectory of $T[1 : i + j]$.All other components of the original MDP remain unchanged. We call the reinforcement learning algorithm based on this augmented MDP as *RLTS-Skip*. We note that when J is set to 0, RLTS-Skip reduces to RLTS.

The RLTS-Skip algorithm is illustrated in Table 4.1 with the inputted data trajectory shown in Figure 4.1 and $W = 4$. It first stores 4 points p_1, p_2, p_3 and p_4 in the buffer. It then scans point p_5, observes the first state s_1 and takes the action of dropping the point p_3. It then scans point p_6, updates the values of two neighboring points of p_3, namely p_2 and p_4, and computes the value of the newly inserted point p_5. It then observes the second state s_2 and takes the action of skipping the next 2 points, i.e., p_6 and p_7. It then scans p_8 and updates the value of p_5. It observes the third state s_3, takes an action of dropping p_2, and then terminates. At the end, it returns a simplified trajectory $T' =< p_1, p_4, p_5, p_8 >$, which has the PED error 0.693.

As could be verified, RLTS-Skip has the same time complexity as RLTS. But in practice, RLTS-Skip should have better efficiency than RLTS for two reasons: (1) At each state, RLTS-Skip can take either a "dropping" action (among the k actions) or a "skipping" action (among the J actions), RLTS can only take a "dropping" action, and

Table 4.1: Illustration of the RLTS-Skip algorithm with PED.

Initial	Store $< p_1, p_2, p_3, p_4 >$ into the buffer with the initial reward 0.0				
Point	State	Action	Buffer	Reward	Accum Reward
p_5	$s_1 = \{v(p_2) = 0.243, v(p_3) = 0.354, v(p_4) = 1.0\}$	Drop p_3	$< p_1, p_2, p_4, p_5 >$	-0.354	-0.354
p_6	$s_2 = \{v(p_2) = 0.693, v(p_5) = 0.728, v(p_4) = 1.265\}$	Skip p_6 and p_7	$< p_1, p_2, p_4, p_5 >$	-0.278	-0.632
p_7	-	-	-	-	-
p_8	$s_3 = \{v(p_2) = 0.693, v(p_5) = 1.0, v(p_4) = 1.265\}$	Drop p_2	$< p_1, p_4, p_5, p_8 >$	-0.061	-0.693
Output	Return $T' =< p_1, p_4, p_5, p_8 >$ with $\epsilon(T') = 0.693$				

the cost incurred by a "skipping" action is smaller than that by a "dropping" action (since for a "dropping" action, the values of three points need to be updated (Equation (4.4), (4.5), and (4.6)) while for a "skipping" action, only the value of one point need to be updated (Equation (4.6)); and (2) For RLTS-Skip, for those points that are skipped, the efforts for deciding and taking an action are saved.

4.3.2 Algorithms for Batch Mode

In the batch mode, we have more data access than in the online mode. Specifically, we have access to all points of a trajectory during the course of trajectory simplification in the batch mode, while in the online mode, we can only access those points that are stored in the buffer. With the increased data access, we have more options of defining the states of the MDP. In the following, we investigate three state definitions, which capture different portions of the points of a trajectory, and correspondingly develop three different sets of algorithms, namely (1) RLTS and RLTS-Skip, (2) RLTS+ and RLTS-Skip+, and (3) RLTS++ and RLTS-Skip++.

(1) RLTS and RLTS-Skip. The RLTS and RLTS-Skip algorithms that are designed for the online mode can immediately carry over for the batch mode. Same as the case of online mode, the states of the MDP underlying RLTS and RLTS-Skip are defined based on those points that are stored in the buffer only. The time complexity of RLTS and RLTS-Skip is $O((n - W) \log W)$.

(2) RLTS+ and RLTS-Skip+. In RLTS, a state is defined based on the values of the points in the buffer and the value of a point is defined as the error of its anchor segment wrt the point only. That is, the errors of the point's anchor segment wrt other points that take the segment as an anchor segment are ignored since those points have been dropped already and are no longer accessible in the online mode. In the batch mode, this is not the case. In fact, we can make use of these errors for defining the value of a

point so as to capture richer information. Specifically, we define the value of a point p_{s_j} ($2 \leq j \leq W$) in the buffer as the maximum error of the p_{s_j}'s anchor segment wrt a point that takes the segment as its anchor segment as follows.

$$v(p_{s_j}) := \max_{s_{j-1} \leq i < s_{j+1}} \epsilon(\overline{p_{s_{j-1}} p_{s_{j+1}}} | p_i) \tag{4.11}$$

With this new definition of the value of a point, we would have correspondingly a new MDP, a new learned policy and a new algorithm for trajectory simplification. We call this algorithm *RLTS+*. Essentially, RLTS+ is an adapted version of RLTS with the states enhanced with additional information of those points that have been dropped before. The time complexity of RLTS+ is $O((n - W)(n' + \log W))$, where n' is the cost of computing the value of a point and is bounded by n (in practice, $n' \leq n$).

Similarly, we refine the state definition of the MDP for RLTS-Skip by (1) using the new definition of the value of a point (Equation (4.11)); and (2) appending J values to the original k values, each corresponding to the error incurred by dropping j points $p_i, ..., p_{i+j-1}$ for $1 \leq j \leq J$, when scanning point p_i. We then develop an algorithm based on the MDP with the refined state. We call the resulting algorithm *RLTS-Skip+*. The time complexity of RLTS-Skip+ is the same as RLTS+.

(3) RLTS++ and RLTS-Skip++. In both RLTS and RLTS+, a buffer of a *fixed* size (i.e., W) is maintained and only those points stored in the buffer are considered as candidates to drop. This is necessary in the online mode due to the restricted data access. In the batch mode, all points can be accessed throughout the course of trajectory simplification. Therefore, an alternative design is to use a buffer of a *variable* size. Specifically, we put all the points in the buffer at the beginning and each time we drop one point from the buffer until only W points remain in the buffer. Based on this design, we can define the states in the same way as we do for RLTS+ except that the buffer is of a variable size. Correspondingly, we can obtain a MDP, a learned policy and a trajectory simplification algorithm. We call the resulting algorithm *RLTS++*. Similarly, we can replace the buffer of RLTS-Skip+ with one of a variable size and to obtain a new algorithm, which we call *RLTS-Skip++*. For RLTS-Skip++, all points are stored in the buffer at the beginning, an action of skipping j points means dropping j points.

RLTS++ and RLTS-Skip++ come with an increased cost of computing a state since it needs to maintain the k lowest values among $O(n)$ values instead of W values as RLTS or

Table 4.2: Dataset statistics II.

Statistics	Geolife	T-Drive	Truck
# of trajectories	17,621	10,359	10,110
Total # of points	24,876,978	17,740,902	10,059,685
Ave. # of points per trajectory	1,412	1,713	995
Sampling rate	1s ~ 5s	177s	3s ~ 60s
Average distance	9.96m	623m	82.74m

RLTS+ does. As a result, the time complexity of RLTS++ and RLTS-Skip++ becomes $O((n-W)(n' + \log n))$, where n' is the cost of computing the value of a point and is bounded by n (in practice, $n' \leq n$).

Comparisons and Analysis. From RLTS, to RLTS+, to RLTS++, more information is captured for defining the states, and correspondingly, the resulting algorithms for trajectory simplification have more time costs, i.e., from $O((n-W)\log W)$, to $O((n-W)(n' + \log W))$, to $O((n-W)(n' + \log n))$. This holds for RLTS-Skip, RLTS-Skip+, and RLTS-Skip++ as well. As will be investigated in Section 4.4.1, these algorithms provide different trade-offs of effectiveness and efficiency. Among RLTS, RLTS+ and RLTS++, RLTS++ has the best effectiveness and RLTS has the best efficiency. Among RLTS-Skip, RLTS-Skip+ and RLTS-Skip++, RLTS-Skip++ has the best effectiveness and RLTS-Skip has the best efficiency. As can be verified, the time complexity of training for RLTS, RLTS+, RLTS-Skip, and RLTS-Skip+ is $O(E \cdot N_t \cdot (n-W)(n + \log W))$ and that for RLTS++ and RLTS-Skip++ is $O(E \cdot N_t \cdot (n-W)(n + \log n))$, where E is the number of epochs and N_t is the number of trajectories used for training.

4.4 Experiments

4.4.1 Experimental Setup

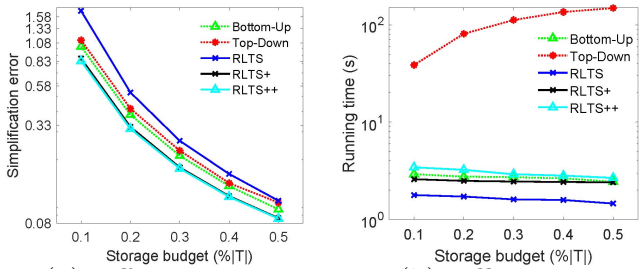
Dataset. Our experiments are conducted on three real-world trajectory datasets, namely Geolife, T-Drive and Trucks, where Geolife and T-Drive datasets have been introduced in Section 3.4.1. Truck [2] records the GPS trajectories of 10,368 trucks in China during a period from March to October, 2015. The three datasets are widely used in evaluating

[2]http://mashuai.buaa.edu.cn/traj.html

trajectory simplification [12, 13, 29] including the recent dedicated evaluation work [5] and the detailed statistics are summarized in Table 4.2.

Algorithms for Comparison. We review the trajectory simplification literature thoroughly and identify seven methods for comparison, including (1) STTrace [19], (2) SQUISH [20] and (3) SQUISH-E [21] for the online mode and (4) Bellman [56], (5) Top-Down [27], (6) Bottom-Up [13, 28], and (7) Span-Search [29] for the batch mode. We cross check these algorithms against those covered in a recent survey [5] on trajectory simplification so that all existing methods that are proposed for the Size-Bounded problem are included for comparison. Note that we do not compare with the adaptions of the algorithms that are designed for the dual problem of Size-Bounded (via binary search) since these adapted algorithms would have the time complexity at least $O(n^2 \log n)$, e.g., for DAD, the time complexity is $O(n^2 C \log n)$, where $C < n$, which are clearly higher than those of RLTS and RLTS-Skip and not scalable on large datasets.

Parameter Setting and Policy Learning. The neural network used in the RLTS and RLTS-Skip (RLTS+ and RLTS-Skip+) methods involves one input layer, one hidden layer and one output layer, where the hidden layer involves 20 neurons and uses the tanh function as the activation function. In order to avoid the data scale issues, batch normalization provided by tensorflow is employed before the activation. For RLTS (RLTS+), the output layer involves k neurons and k is set as 3 by default. For RLTS-Skip (RLTS-Skip+), the output layer involves $(k + J)$ neurons and k and J are set as 3 and 2, respectively. We randomly sample 1,000 trajectories from a training dataset, and

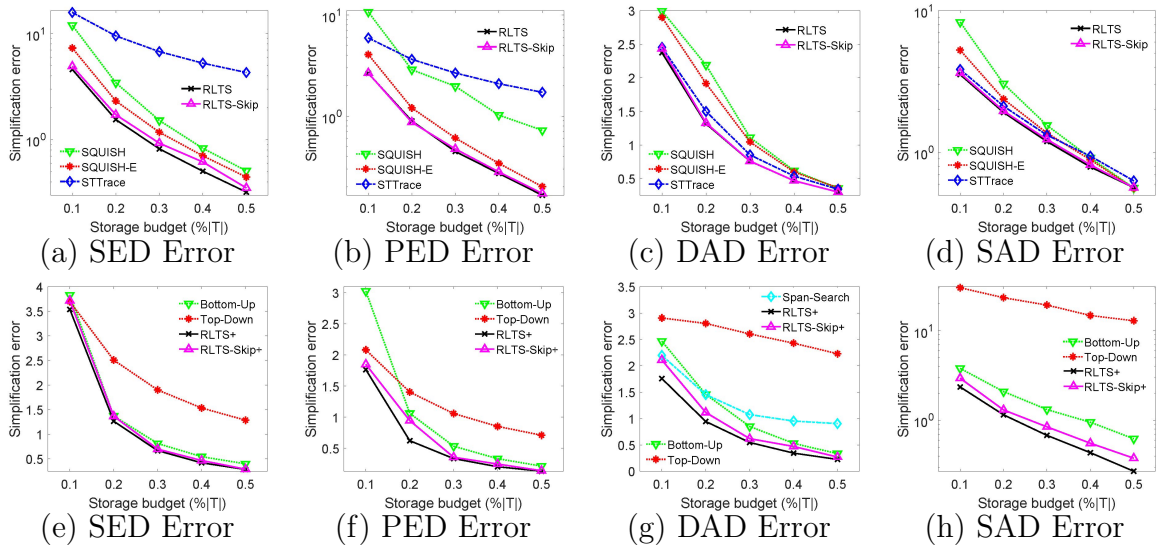


(a) Effectiveness (b) Efficiency

Figure 4.2: Variants of RLTS (Batch mode)

for each trajectory, we generate 10 episodes for policy learning. Since each trajectory involves around 1,000 points, there would be about 10 million transition steps in the learning process. In addition, we use the Adam stochastic gradient descent with the

learning rate of 0.001 based on empirical findings. For the reward discount factor, we tried several settings, and since the results are similar, we set it as 0.99. We take the policy, which gives the maximum reward per episode and use it for trajectory simplification. For the online mode, we sample an action with the probability outputted by the softmax function at each state, and for the batch mode, we take the action with the maximum probability based on empirical findings. The units of SED, PED, DAD, and SAD are 10m, 10m, 1 radian and 10m/s, respectively.



(a) SED Error (b) PED Error (c) DAD Error (d) SAD Error

(e) SED Error (f) PED Error (g) DAD Error (h) SAD Error

Figure 4.3: Effectiveness evaluation with varying W ((a)-(d): Online mode, (e)-(h): Batch mode, Geolife).

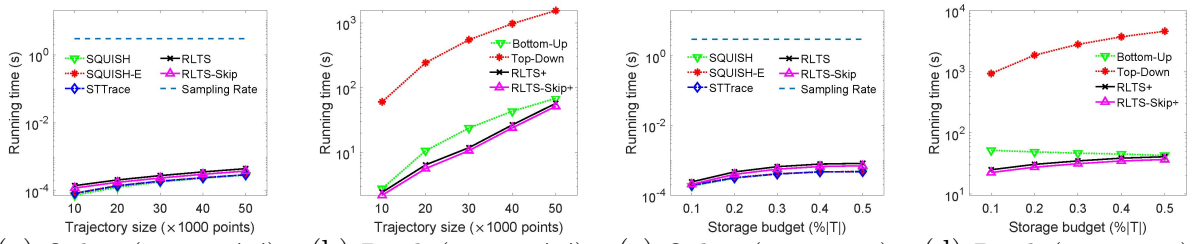
Evaluation Platform. All the methods are implemented in Python 3.6. The implementation of RLTS and RLTS-Skip (RLTS+ and RLTS-Skip+) is based on tensorflow 1.8.0. The experiments are conducted on a machine with Intel(R) Xeon(R) CPU E5-1620 v2 @3.70GHz 16.0GB RAM and one Nvidia GeForce GTX 1070 GPU.

4.4.2 Experimental Results

(1) Comparison among variants (batch mode). We randomly sample 1,000 trajectories from Geolife each with 5,000 points, run the algorithms, and report the average SED error and running time. The results are shown in Figure 4.2. We observe that the effectiveness increases and the efficiency drops from RLTS, to RLTS+, to RLTS++. This is because from RLTS, to RLTS+, to RLTS++, more information is captured for

defining the states, and correspondingly the MDP is more powerful yet takes more time to run. Besides, only RLTS+ dominates the best existing algorithm, i.e., Bottom-UP, in terms of both effectiveness and efficiency. Therefore, in the following experiments, we focus on RLTS+ and RLTS-Skip+ for the batch mode.

(2) Effectiveness evaluation (comparison with existing approximate algorithms). We randomly sample 1,000 trajectories T from a dataset and vary the storage budget W from $0.1 \times |T|$ to $0.5 \times |T|$ by following [29]. Figure 4.3 show the results for both online and batch modes on the Geolife dataset. Overall, the results clearly show that RLTS (RLTS+) consistently outperforms existing algorithms under all error measurements for both online and batch modes and on all datasets. For RLTS-Skip (RLTS-Skip+), it beats all baselines in the online mode, and provides comparable performance in the batch mode. In addition, the effectiveness of RLTS-Skip (RLTS-Skip+) is slightly worse than RLTS (RLTS+), but still better than that of the baselines due to its data-driven nature.



(a) Online (varying $|T|$) (b) Batch (varying $|T|$) (c) Online (varying W) (d) Batch (varying W)

Figure 4.4: Efficiency evaluation on varying $|T|$ and W on Truck.

(3) Efficiency evaluation (varying the trajectory length $|T|$). We follow [5, 29] by varying $|T|$ from 10,000 to 50,000. For each setting, we randomly select 100 trajectories with the size around the setting from Truck. We fix W at $0.1|T|$. Figure 4.4(a)-(b) shows the results for both online and batch modes under SED. In the online mode, we show the average running time per point because the processing time of a single point is important for an online scenario. In addition, the sampling rate (3s) of the Truck dataset is shown with a dotted line in the figure for reference. We observe that RLTS and RLTS-Skip are slightly slower than the three baseline algorithms though all of them have the same time complexity. This is because learning-based algorithms employ the learning models to make the decision (i.e., dropping or skipping) while the other three algorithms use a simple comparison operation for the same task. In addition, RLTS-Skip runs faster

than RLTS since the time cost of constructing states and choosing actions are saved for those points that have been skipped. Overall, RLTS and RLTS-Skip are fast enough (very close to other algorithms) and far meets the practical needs, e.g., for a trajectory with about 10,000 points, they take less than 0.15ms per point, which is 20,000 times faster than the sampling rate (3s). In the batch mode, both RLTS+ and RLTS-Skip+ are faster than Top-Down and Bottom-Up, and the gaps of efficiency are aligned with the time complexities. We omit the running time of Span-Search because it has been shown to be slower than Top-Down in the existing studies [5, 29].

(4) Scalability test. We conduct the scalability test on long trajectories in the datasets. Here, we present the running time results on the longest trajectory, which involves around 383,000 points, that RLTS-Skip+, RLTS+, Bottom-Up, and Top-Down run in 2,843s, 3,412s, 4,952s, 98,427s, respectively.

(5) Efficiency evaluation (varying the budget size W). We vary W from $0.1|T|$ to $0.5|T|$ on Truck and fix $|T|$ at 40,000 using SED as the error measurement. Figure 4.4(c)-(d) shows the result for both the online and batch modes. Similarly, in the online mode, RLTS and RLTS-Skip are a bit slower than SQUISH, SQUISH-E and STTrace, but run reasonably fast. For example, they take less than 1ms per point point for a trajectory with 40,000 points. The running times of all the methods slightly increase with W. In the batch mode, both RLTS+ and RLTS-Skip+ run faster than Top-Down by around two orders of magnitude. They also run faster than Bottom-Up, and the gap reduces as W increases since they take $O(\log W)$ time to construct states while Bottom-Up takes $O(\log n)$ time to decide which segments to merge.

More experiments. More experimental results regarding (6) comparison with the exact algorithm Bellman (Batch mode), (7) effectiveness evaluation (with and without the learned policy), (8) effectiveness evaluation (varying parameter J, K), (9) case study and (10) training time can be found in [7].

4.5 Summary

In this chapter, we study the trajectory simplification problem in both online and batch modes. We propose a reinforcement learning (RL)-based method called RLTS for both

modes. Compared with existing algorithms, which are mainly heuristic-based, our RLTS method is data-driven and can adapt to different dynamics of the underlying points. We conduct extensive experiments, which show that RLTS computes simplified trajectories with consistently lower errors and runs comparably fast in the online mode and faster in the batch mode, compared with existing algorithms.

Chapter 5

Deep Reinforcement Learning for Query Accuracy Driven Trajectory Simplification

5.1 Overview

In this chapter, we study the problem of Query Accuracy Driven Trajectory Simplification (QDTS). This chapter is organized as follows. We cover preliminaries and define the QDTS problem in Section 5.2. We introduce the `RL4QDTS` solution in Section 5.3, and we report on the experimental study in Section 5.4. We conclude in Section 5.5.

5.2 Problem Statement

5.2.1 Preliminaries

Trajectory Database and Data Cubes. A *trajectory database* D is a set of trajectories, and a trajectory database is bounded by a spatial-temporal cube represented by $(x_{min}, x_{max}, y_{min}, y_{max}, t_{min}, t_{max})$, such that each point $p_i = (x_i, y_i, t_i)$ in D satisfies $x_{min} \leq x_i \leq x_{max}$, $y_{min} \leq y_i \leq y_{max}$ and $t_{min} \leq t_i \leq t_{max}$. We define N to be the total number of points in D (i.e., $N = |D|$). The data cubes refer to the partitions of the database, which are three-dimensional cubes for the spatial and temporal data.

5.2.2 Problem Definition

We define a new problem, called Query Accuracy Driven Trajectory Simplification (QDTS), that aims to simplify a trajectory database D such that its simplified database D' preserves the accuracy of query processing as much as possible when compared to the accuracy on D.

Problem 3 (QDTS) *Given a trajectory database D and a storage budget W indicating the fraction of original points that can be retained,* ***Query-Driven Trajectory Simplification*** *aims to find a trajectory database D' of simplified trajectories, such that the difference between query results on D and D' (i.e., $diff(Q(D), Q(D'))$) is minimized, where $Q(\cdot)$ denotes a query workload.*

The QDTS problem relies on (1) a query workload $Q(\cdot)$ of queries on D and D', and (2) a quality measure $diff(\cdot,\cdot)$ that captures the difference of their query results. To establish the former, we review the literature, including evaluation papers on trajectory simplification [5, 25, 26] and a recent trajectory survey [6], and identify four widely-used queries: Range Query, kNN Query, Similarity Query, and Clustering. For the latter, we define query-based quality measures by following [5].

5.2.3 Trajectory Queries and Quality Measures

Range Query. Given a trajectory database D, range query with parameters $(q_{x_{min}}, q_{x_{max}}, q_{y_{min}}, q_{y_{max}}, q_{t_{min}}, q_{t_{max}})$, finds all trajectories that contain at least one point $p_i = (x_i, y_i, t_i)$ such that $q_{x_{min}} \leq x_i \leq q_{x_{max}}$, $q_{y_{min}} \leq y_i \leq q_{y_{max}}$, and $q_{t_{min}} \leq t_i \leq q_{t_{max}}$.

kNN Query. Given a trajectory database D, a kNN query takes a trajectory T_q and a time window $[t_s, t_e]$ as parameters and returns a set of k trajectories (denoted by R) such that $\forall T_i \in R, \forall T_j \in D - R$, $\Theta(T_q[t_s, t_e], T_i[t_s, t_e]) \leq \Theta(T_q[t_s, t_e], T_j[t_s, t_e])$. A kNN query relies on a distance metric $\Theta(\cdot,\cdot)$ between two trajectories. We adopt EDR [34] and t2vec [37] to instantiate $\Theta(\cdot,\cdot)$, which represent non-learning and learning based trajectory similarity measures [5, 6], respectively. Note that our solution is orthogonal to the similarity measure used.

Similarity Query. Given a trajectory database D, a similarity query takes a trajectory T_q and a time window $[t_s, t_e]$ as parameters and returns a set of trajectories (denoted

as R), such that $d(T_q[i], T_j[i]) \leq \delta$ for any $t_s \leq i \leq t_e$, where $T_j \in R$, $d(T_q[i], T_j[i])$ is the Euclidean distance between two points $T_q[i]$ and $T_j[i]$ and δ is a given distance threshold.

Trajectory Clustering. We adopt an existing definition of trajectory clustering [31]. Given a trajectory database, we partition each trajectory into subtrajectories and then cluster subtrajectories according to their perpendicular distance, parallel distance, and angle distance).

Quality Measures. We use the F_1-score and accuracy for measuring the difference between query results on the original database D and the simplified database D' (denoted as $diff(Q(D), Q(D'))$). The main idea is to use the results on D as the ground truth and then measure the quality of the results on D' using the F_1-score and accuracy. The larger the F_1-score and accuracy are, the smaller the difference $diff(Q(D), Q(D'))$ is.

For the range query and the similarity query, we denote by R_o and R_s the trajectory sets returned on D and D', respectively. We define precision (P) and recall (R), and compute F_1-score as follows.

$$\mathrm{P} = \frac{|R_o \cap R_s|}{|R_s|}, \quad \mathrm{R} = \frac{|R_o \cap R_s|}{|R_o|}, \quad F_1 = 2 \cdot \frac{\mathrm{P} \cdot \mathrm{R}}{\mathrm{P} + \mathrm{R}} \tag{5.1}$$

For the kNN query, since $|R_o| = |R_s| = k$, we define the accuracy follows.

$$Accuracy = \frac{|R_o \cap R_s|}{k} \tag{5.2}$$

For the clustering query, let T_i and T_j denote a trajectory pair in D. We consider $(T_i, T_j) \in R_o$ if they are clustered into the same group on D. Similarly, we define R_s that contains the simplified trajectory pairs returned on D'. Then, we calculate the F_1-score by following Equation 5.1.

5.3 Proposed Method

5.3.1 Problem Analysis

Recall the trajectory simplification problem: Given a database D of M trajectories with a total of N points, we aim to produce a simplified database D' that contains a simplified

trajectory for each original trajectory in D and D' contains at most $W = r \cdot N$ points ($W \leq N$).

An immediate solution is to reduce this *database-level* simplification problem to a *trajectory-level* problem by applying simplification to each trajectory *separately* with a proportional budget. Specifically, it simplifies each trajectory T with the budget of $r \cdot |T|$. It is obvious that the resulting simplified database contains M simplified trajectories and has at most $r \cdot \sum_{T \in D} |T| = r \cdot N$ points. While this solution needs minimal design efforts, it has two notable shortcomings. Issue 1: Uniform compression ratio: It applies the same compression ratio to each trajectory, which is sub-optimal in trajectories with different sampling rates of different complexities. Intuitively, trajectories with higher sampling rates or lower complexities should be simplified with higher compression ratios. Issue 2: Query accuracy unawareness: Existing algorithms (all of which operate at trajectory-level) aim to optimize some form of error metric that quantifies the difference between an original and a simplified trajectory and are query-unaware. As a consequence, simply using any of these algorithms only helps indirectly to optimize the query accuracy of the simplified database.

Another solution is to consider all trajectories in the database D *collectively* during the course of simplification. Consider a top-down approach where we start with the most simplified database, in which each simplified trajectory T' of an original trajectory T contains only the first and last points of T. We then iteratively introduce points from the original database according to some selection criterion until the budget is exhausted. This method is a counterpart to the Top-Down method for the EDTS problem [27], except that it operates at the database level. Alternatively, we can adopt an opposite bottom-up approach where we start from D and iteratively drop points until the budget is satisfied. This solution considers all trajectories *collectively* and thus it avoids the first issue above. Still, this solution does not contend with the second issue. Furthermore, the solution operates at the database level, whose scale is typically much larger than that of a single trajectory. For example, the Geolife dataset used in our experiment contains millions of points, while individual trajectories contain only around one thousand points. This brings up a third issue. Issue 3: Lack of scalability: Operating at the database level, it must repeatedly select a point from among a very large set of points.

Motivated by the above discussion, we propose a new solution for trajectory database simplification that avoids all the three issues. The solution is based on two key ideas. (1) It considers all trajectories in the database *collectively* and partitions the database into spatio-temporal cubes. Then, to achieve efficiency, whenever it needs to select a point, it selects a few cubes and considers only points within these cubes. (2) It leverages reinforcement learning to learn a query accuracy aware policy for selecting a point within the chosen cubes such the selected points optimize the query accuracy. The first idea enables the solution to avoid the first and third issues, and the second idea enables it to avoid the second issue. Specifically, for (1), we partition the database into data cubes with a predefined granularity and use the number of trajectories in each cube to represent the data distribution over cubes. We sample two cubes with a simple strategy (leaving others for future exploration), namely one by following the data distribution and the other via uniform sampling. Next, we present (2) in detail in Section 5.3.2 and our `RL4QDTS` algorithm based on the learned policy for trajectory simplification in Section 5.3.3.

5.3.2 MDP for Choosing Points in a Cube

We refer the sampled cubes as one general cube denoted by B for simplicity. Recall that the overall trajectory simplification process is to sample some cubes and select a point from the cubes repetitively until the budget is exhausted. As could be noted, it is a sequential process, during which we need make decisions on which point from a (general cube) to select many times. It is well known that reinforcement learning (RL) is powerful for sequential decision making, which is typically modelled as a Markov decision process (MDP) [103]. Therefore, in this chapter, we model the sequential process of selecting points from cubes as an MDP and adopt RL for learning a policy based on the MDP for selecting points from cubes. Specifically, a MDP mainly involves states, actions and rewards, and in this sequel, we first introduce the MDP for our task of selecting a point from a cube and then present the policy learning procedure on the MDP.

(1) States. Let N_B (resp. M_B) denote the number of points (resp. trajectories) in the B. We next need to include a point among the N_B points into the simplified database D'. To define the state, an idea is to define by incorporating all N_B points. However, this idea has two issues. (1) The definition in this way is N_B-dependent, which is not

suitable for other cases when the number of points is not N_B. (2) N_B is generally very large. With the definition, the state space would be huge and the model is hard to train.

We design the states such that these two issues are avoided as follow. First, let $p_{s_j}^{T_i}$ be a point of a trajectory T_i in the cube B, where $1 \leq i \leq M_B$. Let $p_{s_a}^{T_i}$ and $p_{s_b}^{T_i}$ denote the first point and last point of T_i within the cube, respectively, where $s_a \leq s_j \leq s_b$. For each point $p_{s_j}^{T_i}$ in the cube, we define a pair of two values, denoted by $v(p_{s_j}^{T_i})$. The first one, denoted by $v_s(p_{s_j}^{T_i})$, is equal to the "spatial" distance between $p_{s_j}^{T_i}$ and the synchronous point on the current simplified trajectory of T_i within the cube B. The second one, denoted by $v_t(p_{s_j}^{T_i})$, is equal to the "temporal" difference between the time of $p_{s_j}^{T_i}$ and the time of $p_{s_j}^{T_i}$'s closest point on the current simplified trajectory of T_i within the cube B. That is, we have

$$v(p_{s_j}^{T_i}) = (v_s(p_{s_j}^{T_i}), v_t(p_{s_j}^{T_i})). \tag{5.3}$$

The intuition of the two values is to capture the features of the point $p_{s_j}^{T_i}$ from both the spatial and temporal aspects given the context of trajectory simplification.

Among all points in each trajectory T_i, we then find a point (denoted as $p_{s_*}^{T_i}$) which has the maximum v_s, where s_* denotes its index, that is

$$s_* = \arg\max_{s_a \leq s_j \leq s_b} v_s(p_{s_j}^{T_i}). \tag{5.4}$$

Finally, the state s is defined as the set of K largest v_s values of $v(p_{s_*}^{T_i})$ among the B_M trajectories, that is

$$s = \{v(p_{s_*}^{T_{\pi(1)}}), v(p_{s_*}^{T_{\pi(2)}}), ..., v(p_{s_*}^{T_{\pi(K)}})\}, \tag{5.5}$$

where π denotes the permutation of $T_1, T_2, ..., T_{M_B}$ such that $v(p_{s_*}^{T_{\pi(1)}}), v(p_{s_*}^{T_{\pi(2)}}), ..., v(p_{s_*}^{T_{\pi(M_B)}})$ is a list of the values in a descending order wrt v_s. K ($K \leq M_B$) is a hyper-parameter that can be tuned empirically to controls the size of the state space.

Our state design avoids the two aforementioned issues. First, K is generally much smaller than N_B or M_B. With this design, a state has a fixed size that is independent from the number of trajectories in the cubes. Second, the state design considers both trajectory level features and point level features.

(2) Actions. Let a denote an action of MDP, which is to choose one of K points $p_{s_*}^{T_{\pi(1)}}, p_{s_*}^{T_{\pi(2)}}, ..., p_{s_*}^{T_{\pi(K)}}$ into D'. The design of actions is consistent with the design of state

s, and the actions are defined as follows:

$$a = k \quad (1 \leq k \leq K), \tag{5.6}$$

where action $a = k$ means to insert point $p_{s_*}^{T_{\pi(k)}}$ into D'.

(3) Rewards. The reward is associated with transitions to reflect the quality of actions taken at given states. A larger reward indicates a better quality of the performed actions. Since our objective is to learn a simplified database that serves queries more effectively (i.e., minimizing the difference of the query results on the original database and the simplified database), the reward design is expected to reflect the improvement of query performance as more points are included in simplified database.

To do this, we randomly generate a set of range queries, where each query location is randomly sampled by either following the data distribution or uniformly and the widths of the queries are set via empirical studies. One option is to perform the queries after each point is inserted to the simplified database, which is associated with the transition from the current state s to the next state s' when an action a is taken. However, it would be prohibitively costly to perform queries for each inserted point. In addition, since the simplified database D' has not been fully constructed, the query improvement with inserting just one point is often negligible and it is hard to demonstrate the quality of the action.

In our design, we choose to perform the queries after Δ (e.g., $\Delta = 50$) points are inserted for achieving accumulative effects. Specifically, we denote the reward by R. At state s_i, we consider a simplified database with i points so far (denoted as D'). Similarly, at state $s_{i+\Delta}$, we have a simplified database with $i + \Delta$ points (denoted as D''). We then define the reward as follows.

$$R = \mathit{diff}(Q(D), Q(D')) - \mathit{diff}(Q(D), Q(D'')), \tag{5.7}$$

where $\mathit{diff}(Q(D), Q(D'))$ measures the difference between the results of queries on the original database D and the simplified database D'. The intuition is that if the difference of the simplified database D'', is smaller, then the reward is larger. Further, the reward R will be shared by all transitions that lead the state to traverse from s_i to $s_{i+\Delta}$.

With the reward definition, the objective of the MDP, i.e., maximizing the accumulative rewards, is equivalent to that of the QDTS problem, i.e., minimizing the difference between queries on the original database and the simplified database. To see this, suppose we traverse through a sequence of N' states $s_1, s_2, ..., s_{N'}$. Correspondingly, we receive a sequence of rewards $R_1, R_2, ..., R_{N'-1}$. We assume that the future rewards are accumulated without discounted rates, and thus the accumulative reward is calculated as follows.

$$\begin{aligned}\sum_{t=1}^{N'-1} R_t &= \sum_{t=1}^{N'-1} \mathit{diff}(Q(D), Q(D'_t)) - \mathit{diff}(Q(D), Q(D''_t)) \\ &= \mathit{diff}(Q(D), Q(D'_1)) - \mathit{diff}(Q(D), Q(D''_{N'-1})) \\ &= -\mathit{diff}(Q(D), Q(D''_{N'-1})),\end{aligned} \tag{5.8}$$

where D'_t (resp. D''_t) denotes the simplified database at the state s_t before (resp. after) the action a_t is performed. We set the initial term $\mathit{diff}(Q(D), Q(D'_1)) = 0$, since no points have been inserted at that state. Therefore, the objective of the MDP is to maximize $-\mathit{diff}(Q(D), Q(D''_{N'-1}))$ or equivalently to minimize $\mathit{diff}(Q(D), Q(D''_{N'-1}))$, which is exactly the objective of QDTS.

(4) Policy Learning via DQN. The core problem of a MDP is to find an optimal policy, which guides an agent to chooses an action at a specific state, such that the accumulative reward is maximum. Due to the continuous state space in our problem, we adopt the Deep-Q-Networks (DQN) [17] for learning a policy from the MDPs. Specifically, we adopt the deep Q learning with replay memory [17] for learning the policy, denoted as $\pi_\theta(a|s)$, which samples an action a at a given state s via DQN, whose parameters are denoted by θ.

5.3.3 The `RL4QDTS` Algorithm

Algorithm 3 details the `RL4QDTS` algorithm with the learned policy for the QDTS problem. Specifically, `RL4QDTS` starts with inserting the first and the last points of each trajectory into D' (lines 1 – 3). It initializes an index t to record a sequence of states and actions during the MDP (line 4). The remaining budget $W - 2M$ is utilized in lines 5 – 13. First, it samples a cube B by following data distribution and uniform distribution via cube sampling (line 6). Within B, it computes the value of $v(p_{s_*}^{T_j})$ for each trajectory T_j using

Algorithm 3: The RL4QDTS algorithm

Input: A trajectory database $D = \langle T_1, T_2, ..., T_M \rangle$; a given storage budget W;
Output: A simplified trajectory database D' of D with $|D'| \leq W$;

1 **for** *i=1,2,...,M* **do**
2 | Insert point $p_1^{T_i}$ and $p_{|T_i|}^{T_i}$ into D';
3 **end**
4 $t \leftarrow 1$
5 **for** *i=2M+1,2M+2,...,W* **do**
6 | Sample a cube B;
7 | Compute $v(p_{s_*}^{T_j})$ $(1 \leq j \leq M_B)$ for each trajectory T_j within B;
8 | Maintain $v(p_{s_*}^{T_j})$ in a max-priority queue with the descending permutation π within B;
9 | Construct a state $s_t \leftarrow \{v(p_{s_*}^{T_{\pi(1)}}), v(p_{s_*}^{T_{\pi(2)}}), ..., v(p_{s_*}^{T_{\pi(K)}})\}$
10 | Sample an action $a_t \sim \pi_\theta(a|s_t)$;
11 | Insert the point $p_{s_*}^{T_{\pi(k)}}$ into D' where $a_t = k$ $(1 \leq k \leq K)$;
12 | $t \leftarrow t + 1$;
13 **end**
14 **Return** simplified database D';

Equations 5.3 and 5.4, where $1 \leq j \leq M_B$ (line 7). Then, it maintains the values in a max-priority queue with the descending permutation π within each cube (line 8). Next, it constructs a state s_t using Equation 5.5 (line 9), and samples an action with the learned policy $\pi_\theta(a|s_t)$, which takes s_t as the input (line 10). Let $a_t = k$ $(1 \leq k \leq K)$ denote the sampled action, which inserts the point $p_{s_*}^{T_{\pi(k)}}$ into D' (line 11). The process continues until the remaining budget is exhausted. Finally, the RL4QDTS algorithm returns the simplified trajectory database D', which contains W points (line 14).

Time complexity. The time complexity of the RL4QDTS algorithm is $O(W(n + \log M_B))$, where W, n, and M_B denote the storage budget, the maximum number of points in the input trajectories, and the maximum number of trajectories in the data cubes. The complexity is detailed as follows. The processing of the remaining budget of $W - 2M$ points dominates the complexity (lines 5 – 13), which includes (1) sampling data cubes with cost $O(1)$ (line 6); (2) computing states with cost $O(n)$ (line 7); (3) maintaining the min-priority queue with cost $O(\log M_B)$ (line 8); (4) constructing a state, sampling an action, and inserting a point with cost $O(1)$ assuming K is a small constant (lines 9 – 12). We note that the RL4QDTS algorithm has the same complexity as have the error-driven algorithms [7, 27, 28] for simplifying a set of trajectories. In addition, we note

that simplification is normally performed once offline, after which the simplified database is used for online querying.

5.4 Experiments

5.4.1 Experimental Setup

Dataset. We conduct the experiments on two real-world trajectory datasets, Geolife and T-Drive, where they have been introduced in Section 3.4.1.

Baselines. To evaluate the effectiveness of `RL4QDTS`, we consider four error-driven methods: Top-Down [27], Bottom-Up [28], RLTS+ [7] and Span-Search [29]. Among these, Top-Down, Bottom-Up and RLTS+ are general frameworks that can be applied with different error measures, while Span-Search works only with DAD. In addition, given that the QDTS problem is to simplify a database instead of a single trajectory, we adapt the Top-Down, Bottom-Up, and RLTS+ in two ways. The first is to simplify each trajectory in the database one by one by calling one of the algorithms (this adaption is denoted as "E"). The second is to consider the database as a whole and simplify the database by inserting or dropping points among all points in the database as it simplifies a trajectory (this adaption is denoted as "W"). In summary, for each of the algorithms Top-Down, Bottom-Up, and RLTS+, we get 8 ($= 4 \cdot 2$) adaptions as baselines, each corresponding to a combination of an error measure SED, PED, DAD, or SAD, and an adaption method ("E" and "W"). In total, we get 25 baselines including 24 ($=3 \cdot 8$) adaptions of Top-Down, Bottom-Up, and RLTS+ and Span-Search (for Span-Search "W" is not possible).

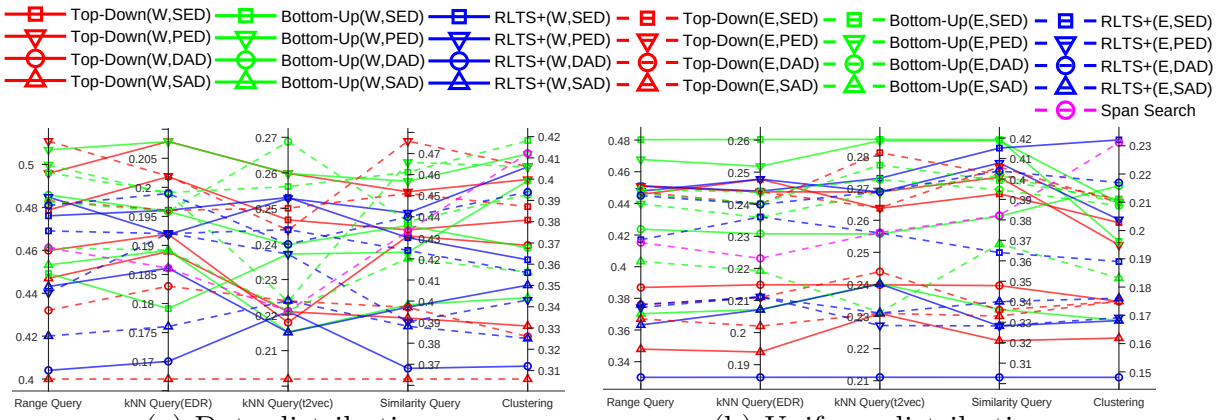
Evaluation Platform. We implement `RL4QDTS` and the baselines in Python 3.6 and Keras 2.2.0. The experiments are conducted on a 10-cores server with an Intel(R) Core(TM) i9-9820X CPU @3.30GHz 64.0GB RAM and an Nvidia GeForce RTX 2080 GPU.

Parameter Settings. We implement `RL4QDTS` using two-layers feedforward neural networks. The first layer involves 25 neurons with the tanh activation function. The second layer involves K neurons corresponding to the action space, with the linear function as the activation, where K is set to 2. We employ batch normalization in the neural networks

to avoid the potential data scale issue. For training, we randomly sample 6,000 trajectories from each dataset and use the remaining trajectories for testing. From the 6,000 trajectories, we randomly prepare 12 databases and with 500 trajectories each. Further, we generate 5 episodes for each database for training the policy, and the best model is chosen during the training. In addition, for every 50 points that have been inserted, we perform 100 range queries with a range of 2 km and 7 days cubes, to obtain the reward, and the discount rate is set to 0.99. The `RL4QDTS` model is trained via Adam stochastic gradient descent with an initial learning rate of 0.01. The minimal ϵ is set to 0.1 with decay 0.99 for ϵ-greedy in DQN, and the replay memory is set to 2000. To sample cubes for the `RL4QDTS` model, we use 2 km and 4 days as the spatial and temporal granularity to partition a database, whose effects are shown in Section 5.4.2. Next, `RL4QDTS` involves random sampling of cubes. For each result of `RL4QDTS`, we run the algorithm 50 times and collect the averages and standard deviations of the query metrics. For query processing, we set the range query to a cube with 2 km and 7 days. We use 7 days as the window length for both kNN and similarity queries. For the kNN query, we set $k = 1$ (i.e., nearest neighbor), and we use EDR as the distance metric with the threshold of 2 km, and we also use t2vec and set it described in the original paper [37]. For similarity query, the distance threshold is set to 5 km. For clustering, we adopt the TRACLUS algorithm by following the original paper [31].

5.4.2 Experimental Results

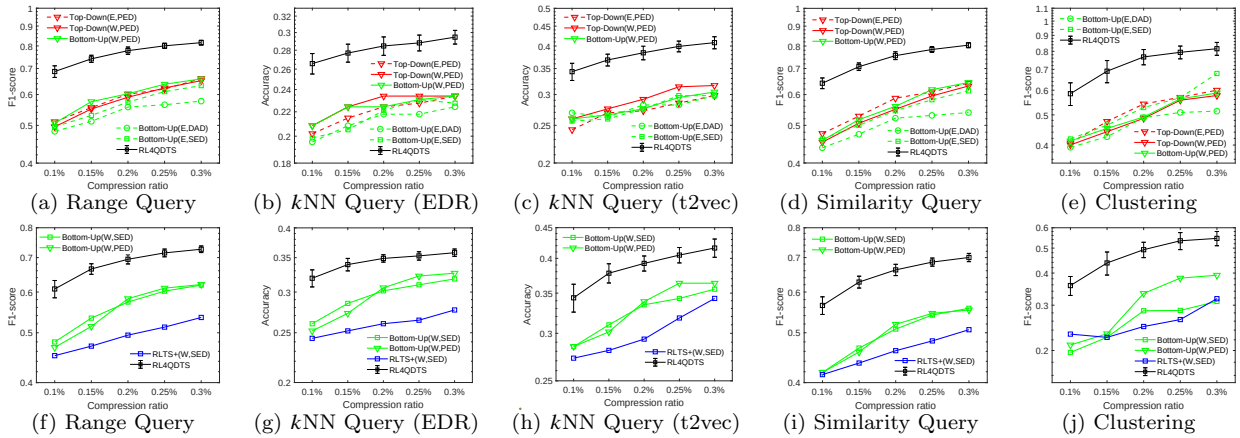
(1) Effectiveness evaluation (skyline selection of existing algorithms). Since we have 25 baselines, we select the skylines of the baselines for each query task to achieve more targeted comparisons. In addition, we consider two possible ways to generate queries for each task. One is to generate queries such that they follow the data distribution, i.e., more queries appear in regions where the data trajectories are concentrated. We call this the *data query*. The other is to generate queries such that they are uniformly distributed in the trajectory database. We call this the *uniform query*. We randomly sample trajectories from Geolife and construct a trajectory database D containing around 1.5 million points, and the storage budget for the simplification is set to $W = 0.1\% \cdot |D|$. In Figure 5.1, we illustrate the effectiveness of the algorithms for five query tasks: range



(a) Data distribution (b) Uniform distribution

Figure 5.1: Skyline selection with existing algorithms.

query, kNN Query(EDR), kNN Query(t2vec), Similarity Query, and clustering, on the two query distributions. For each task, we query 100 times and report the average results of the F_1-score and accuracy as described in Section 5.2.1. For the data distribution, we observe that Top-Down(E,PED), Top-Down(W,PED), Bottom-Up(W,PED), Bottom-Up(E,DAD) and Bottom-Up(E,SED) are on the skyline. For the uniform distribution, we observe that Bottom-Up(W,SED), Bottom-Up(W,PED) and RLTS+(W,SED) are on the skyline.

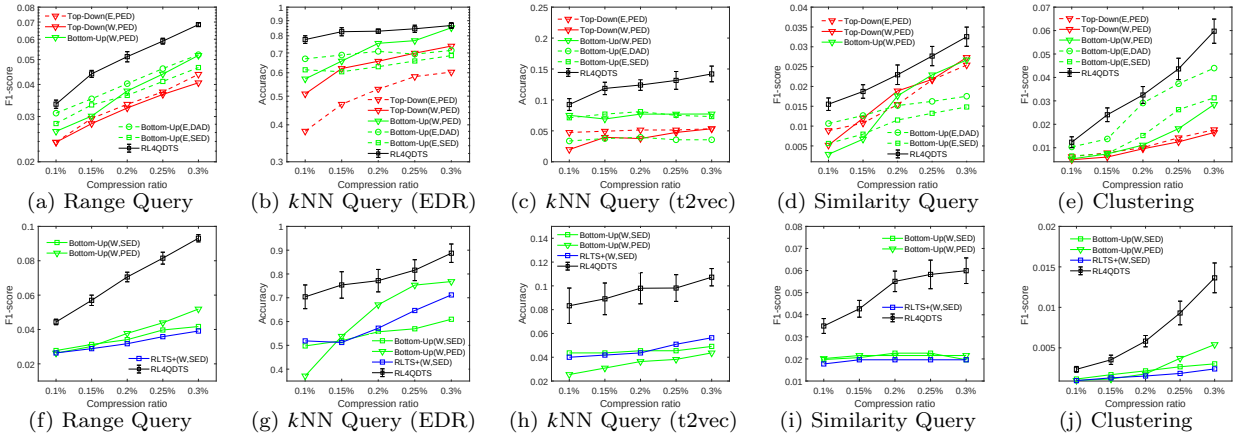


(a) Range Query (b) kNN Query (EDR) (c) kNN Query (t2vec) (d) Similarity Query (e) Clustering

(f) Range Query (g) kNN Query (EDR) (h) kNN Query (t2vec) (i) Similarity Query (j) Clustering

Figure 5.2: Comparison with skylines on Geolife (data distribution (a)-(e) and uniform distribution (f)-(j)).

(2) Effectiveness evaluation (comparison with skyline). We compare RL4QDTS with the selected skyline methods at each query task. We vary the storage budget W



(a) Range Query (b) kNN Query (EDR) (c) kNN Query (t2vec) (d) Similarity Query (e) Clustering

(f) Range Query (g) kNN Query (EDR) (h) kNN Query (t2vec) (i) Similarity Query (j) Clustering

Figure 5.3: Comparison with skylines on T-Drive (data distribution (a)-(e) and uniform distribution (f)-(j)).

from $0.1\% \cdot |D|$ to $0.3\% \cdot |D|$. Figure 5.2 shows the results on the two query distributions on Geolife. For `RL4QDTS`, we show its error bars obtained by running the algorithm 50 times as described in Section 5.4.1. The results based on T-Drive, shown in Figure 4, demonstrate trends similar to those seen on Geolife. Overall, we observe that `RL4QDTS` consistently outperforms the existing error-driven methods across the different storage budgets, different query tasks with different generation distributions, and real datasets. Specifically, `RL4QDTS` outperforms the best skyline(s) by 34.6% (resp. 27.7%, 27.7%, 34.8%, and 39.9%) for range query (resp. kNN Query(EDR), kNN Query(t2vec), Similarity Query, and clustering) in data distribution, and by 26.7% (resp. 22.8%, 20.6%, 34.8% and 54.1%) for range query (resp. kNN Query(EDR), kNN Query(t2vec), Similarity Query and clustering) in uniform distribution. This is because `RL4QDTS` takes the query quality as the objective for trajectory simplification; while existing methods aim to minimize a given error measure, while query quality is no considered directly.

(3) Effectiveness evaluation (ablation study). We conduct an ablation study to investigate the effects of individual components in `RL4QDTS`. We drop the cube sampling and construct the states in the cube of the whole database (denoted w/o cube sampling); we replace the trajectory-based error state, where a state is defined on all points within the cubes instead of the points, with the maximum value on each trajectory; we drop the learned policy and instead insert the point with the maximum value into a simplified database. Table 5.1 reports the average results of 100 data distribution range queries

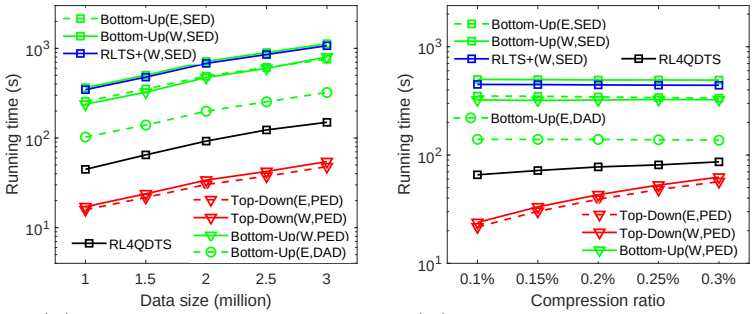
on a randomly sampled trajectory database with around 1.5 million points from Geolife. Overall, all components contribute to the final result. Specifically, we observe that cube sampling has the largest effect, since it captures the potential query signals and guides the model to make a decision within a small space (i.e., the space in the selected cubes). In addition, we notice that it also contributes to the efficiency, since maintaining all data points for constructing states incur high time cost, which is also indicated by the time complexity.

Table 5.1: Ablation study for `RL4QDTS`.

Effectiveness	Range Query	Time (s)
`RL4QDTS`	0.721 ± 0.023	76.61
w/o cube sampling	0.551	91.68
w/o traj-based error state	0.691 ± 0.019	84.06
w/o learned policy	0.682 ± 0.021	74.54

(4) Efficiency evaluation (varying the data size N). We study efficiency when varying the database size for trajectory simplification on Geolife. We compare with the skyline methods and vary the trajectory database size N from 1 million to 3 million points, with a fixed storage budget $W = 0.1\% \cdot N$. The running times are shown in Figure 5.4(a). Overall, we observe that `RL4QDTS` runs faster than quite a few of the existing methods. It is slower than the methods adapted from Top-Down, because these methods use heuristic values with a simple operation for trajectory simplification. In contrast, `RL4QDTS` employs a learned policy for that task to improve effectiveness, and this incurs time costs for constructing states and sampling actions. The results on T-Drive show similar trends and are thus omitted.

(5) Efficiency evaluation (varying the budget size W). We further study the effect of budget size W from $0.1\% \cdot N$ to $0.3\% \cdot N$, with a fixed N of 1.5 million points. Figure 5.4(b) illustrates the running time on Geolife. `RL4QDTS` is slower than the Top-Down adaptions, but is faster than the Bottom-Up adaptions by at least a factor of two times. In addition, we observe that the running times of the Top-Down adaptions increase, while those of the Bottom-Up adaptions decrease slightly as W increases, which may be explained by their strategies for conducting simplification (i.e., inserting vs dropping



(a) Varying data size (b) Varying budget size

Figure 5.4: Efficiency evaluation

Table 5.2: Impacts of cube grid size for `RL4QDTS` (Geolife).

Grid Size	1 km	2 km	3 km	4 km	5 km
#Cubes	22,094	11,515	7,846	5,834	4,759
Range Query	0.697 ± 0.019	**0.721 ± 0.023**	0.691 ± 0.018	0.679 ± 0.021	0.651 ± 0.019
Time (s)	75.19	76.61	77.75	77.99	78.63

points), and this is consistent with their time complexities. Again, the results on T-Drive are qualitatively similar and are thus omitted.

(6) Effectiveness evaluation (varying cube granularity). We evaluate the effect of the cube size, considering the spatial aspect (i.e., the grid size) in Table 5.2 and the temporal aspect (i.e., time duration) in Table 5.3, We report the average results of 100 range queries with data distribution on a database with around 1.5 million points. In Table 5.2, we observe that a moderate size brings the best effectiveness. The reason is that smaller cubes contain fewer candidate points, making the model miss potential points to be inserted. Further, while larger cubes contain many candidates, the sampled cubes may not capture the query signals accurately, e.g., in the extreme case of setting the grid size to cover the whole database, the capability of cube sampling is removed. In addition, the time cost increases as the grid size grows since more trajectories need to be maintained within each cube for constructing states. As expected, similar trends can be observed for the temporal duration as shown in Table 5.3.

(7) Effectiveness evaluation (varying parameter K). We study the effect of parameter K, which controls the state space for decision-making. We construct a database with around 1.5 million points randomly sampled from Geolife, and report the average

Table 5.3: Impacts of cube duration for RL4QDTS (Geolife).

Duration	2 days	4 days	6 days	8 days	10 days
#Cubes	12,085	11,515	11,106	10,803	10,559
Range Query	0.692 ± 0.015	**0.721 ± 0.023**	0.719 ± 0.026	0.714 ± 0.021	0.714 ± 0.021
Time (s)	71.62	76.61	77.22	79.39	79.48

Table 5.4: Impacts of parameter K for RL4QDTS (Geolife).

Parameter	$K = 1$	$K = 2$	$K = 3$	$K = 4$	$K = 5$
Range Query	0.682 ± 0.021	**0.721 ± 0.023**	0.716 ± 0.026	0.716 ± 0.028	0.712 ± 0.028
Time (s)	74.54	76.61	76.77	77.65	78.16

results of 100 range queries with data distribution and the corresponding running time. In Table 5.4, we observe that the setting of $K = 2$ gives the best effectiveness. When K becomes larger, the model performance degrades slightly, since it becomes more difficult to train the model to choose a point among a larger number of candidate points. In addition, the time cost increases as K increases because more time is spent on constructing states. We set K to 2 by default, since this provides a reasonable trade-off between effectiveness and efficiency.

(8) Training time. In Table 5.5, we show the training time of RL4QDTS on Geolife with the default settings described in Section 5.4.1. We randomly construct 5 training sets with 2,000 to 5,000 trajectories. For each training set, we report the time cost and effectiveness based on the selected model during the training. We observe that the training generally takes several hours and that the cost increases almost linearly with the number of trajectories. The effectiveness improves slightly when the number of trajectories exceeds 5,000. We use the setting of 6,000 trajectories to train the RL4QDTS model, which is enough to obtain a good model with a reasonable time cost.

Table 5.5: Training cost (hours) on Geolife.

Training size	2,000	3,000	4,000	5,000	6,000	7,000
Range Query	0.675 ± 0.026	0.691 ± 0.021	0.702 ± 0.019	0.712 ± 0.021	**0.721 ± 0.023**	0.721 ± 0.023
Time Cost	0.89	1.33	1.70	2.13	2.48	2.73

5.5 Summary

In this chapter, we propose the query accuracy driven trajectory simplification problem, which aims to find a simplified trajectory database, such that the difference between query results on the original database and simplified database is minimized. We develop the first solution to the problem, called RL4QDTS, which is based on reinforcement learning. Compared with existing trajectory simplification algorithms, `RL4QDTS` is able to optimize the objective of the QDTS problem directly. Extensive experiments on two real-world trajectory datasets show that our solution is able to consistently outperform the existing EDTS algorithms for five query processing operations.

Part II

Trajectory Data Similarity Search

Chapter 6

Deep Reinforcement Learning for Subtrajectory Similarity Search

6.1 Overview

In this chapter, we study the problem of Similar Subtrajectory Search (SimSub) [1]. We propose the SimSub problem, and this to the best of our knowledge, corresponds to the first systematic study on searching subtrajectories that are similar to a query trajectory, and we develop a suite of algorithms for the SimSub problem. This chapter is organized as follows. We formulate the SimSub problem in Section 6.2. Section 6.3 presents non-learning based algorithms and reinforcement learning based algorithms. We report experimental results in Section 6.4 and conclude in Section 6.5.

6.2 Problem Statement

The trace of a moving object such as a vehicle and a mobile user is usually captured by a trajectory. Specifically, a trajectory T has its form as a sequence of time-stamped locations (called points), i.e., $T =< p_1, p_2, ..., p_n >$, where point $p_i = (x_i, y_i, t_i)$ means that the location is (x_i, y_i) at time t_i. The size of trajectory T, denoted by $|T|$, corresponds to the number of points of T.

Given a trajectory $T =< p_1, p_2, ..., p_n >$ and $1 \leq i \leq j \leq n$, we denote by $T[i, j]$ the portion of T that starts from the i^{th} point and ends at the j^{th} point, i.e., $T[i, j] =<$

[1] This chapter was published as Efficient and Effective Similar Subtrajectory Search with Deep Reinforcement Learning [47].

$p_i, p_{i+1}, ..., p_j >$. Besides, we say that $T[i, j]$ for any $1 \leq i \leq j \leq n$ is *subtrajectory* of T. There are in total $\frac{n(n+1)}{2}$ subtrajectories of T. Note that any subtrajectory of a trajectory T belongs to a trajectory itself.

6.2.1 Problem Definition

Suppose we have a database of many trajectories, which we call *data trajectories*. One common application scenario would be that a user has a trajectory at hand, which we call a *query trajectory* and would like to check what is the portion of the data trajectories that is the most similar to the one at his/her hand. Note that in some cases, by looking each data trajectory as whole, none is similar enough to the query trajectory, e.g., all data trajectories are relatively long while the query trajectory is relatively short.

We note that a more general query is to find the *top-k* similar subtrajectories to a query trajectory, which reduces to the user's query as described above when $k = 1$. In this chapter, we stick to the setting of $k = 1$ since extending the techniques for the setting of $k = 1$ to general settings of k is straightforward. Specifically, the techniques for the setting $k = 1$ in this chapter are all based on a search process, which maintains the most similar subtrajectory found so far and updates it when a more similar subtrajectory is found during the process. These techniques could be adapted to general settings of k by simply maintaining the k most similar subtrajectories and updating them when a subtrajectory that is more similar than the k^{th} most similar subtrajectory.

An intuitive solution to answer the user's query is to scan the data trajectories, and for each one, compute its subtrajectory that is the most similar to the query one based on some similarity measurement and update the most similar subtrajectory found so far if necessary. This solution could be further enhanced by employing indexing techniques such as the R-tree based index and the inverted-file based index for pruning [66, 69], e.g., the data trajectories that do not have any overlap with the query trajectory could usually be pruned. The key component of this solution (no matter whether indexing structures are used or not) is to compute for a given data trajectory, its subtrajectory that is the most similar to a query trajectory. We formally define the problem corresponding to this procedure as follows.

Problem 4 (Similar Subtrajectory Search) *Given a data trajectory* $T =< p_1, p_2, ..., p_n >$ *and a query trajectory* $T_q =< q_1, q_2, ..., q_m >$*, the* ***similar subtrajectory search (SimSub)*** *problem is to find a subtrajectory of* T*, denoted by* $T[i^*, j^*]$ $(1 \leq i^* \leq j^* \leq n)$*, which is the most similar to* T_q *according to a trajectory similarity measurement* $\Theta(\cdot, \cdot)$*, i.e.,* $[i^*, j^*] = \arg\max_{1 \leq i \leq j \leq n} \Theta(T[i, j], T_q)$.

The SimSub problem relies on a similarity measurement $\Theta(T, T')$, which captures the extent to which two trajectories T and T' are similar to each other. The larger the similarity $\Theta(T, T')$ is, the more similar T and T' are. In the literature, several "dissimilarity measurements" have been proposed for $\Theta(\cdot, \cdot)$ such as DTW [38], Frechet [65], LCSS [72], ERP [33], EDR [34], EDS [35], EDwP [36], and t2vec [37]. Different measurements have different merits and suit for different application scenarios. In this chapter, we assume an abstract similarity measurement $\Theta(\cdot, \cdot)$, which could be instantiated with any of these existing measurements by applying some *inverse operation* such as taking the ratio between 1 and a distance.

6.2.2 Trajectory Similarity Measurements

The SimSub problem assumes an abstract similarity measurement and the techniques developed could be applied to any existing measurements. Since the time complexity analysis of the algorithms proposed in this chapter relies on the time complexities of computing a specific measurement in several different cases, in this part, we review three existing measurements, namely t2vec [37], DTW [38], and Frechet [65], and discuss their time complexities in different cases as background knowledge. The first two are the most widely used measurements and the last one is the most recently proposed one, which is a data-driven measurement.

We denote by Φ the time complexity of computing the similarity between a general subtrajectory of T and T_q *from scratch*, Φ_{inc} be the time complexity of computing $\Theta(T[i, j], T_q)$ $(1 \leq i < j \leq n)$ *incrementally* assuming that $\Theta(T[i, j-1], T_q)$ has been computed already, and Φ_{ini} the time complexity of computing $\Theta(T[i, i], T_q)$ $(1 \leq i \leq n)$ from scratch since it cannot be computed incrementally. As will be discussed later, Φ_{inc} and Φ_{ini} are usually much smaller than Φ across different similarity measurements.

Table 6.1: Time complexities of computing the similarity between a subtrajectory of T and T_q in three cases

Time complexities	t2vec	DTW	Frechet
Φ (general)	$O(n+m)$	$O(n \cdot m)$	$O(n \cdot m)$
Φ_{inc} (incremental)	$O(1)$	$O(m)$	$O(m)$
Φ_{ini} (initial)	$O(1)$	$O(m)$	$O(m)$

t2vec [37]. t2vec is a data-driven similarity measure based on representation learning. It adapts a sequence-to-sequence framework based on RNN [122] and takes the final hidden vector of the encoder [123] to represent a trajectory. It computes the similarity between two trajectories based on the Euclidean distance between their representations as vectors.

Given T and T_q, it takes $O(n)$ and $O(m)$ time to compute their hidden vectors, respectively and $O(1)$ to compute the Euclidean distance between two vectors [37]. Therefore, we know $\Phi = O(n+m+1) = O(n+m)$. Since in the context studied in this chapter, we need to compute the similarities between many subtrajectories and a query trajectory T_q, we assume that the representation of T_q under t2vec is computed once and re-used many times, i.e., the cost of computing the representation of T_q, which is $O(m)$, could be amortized among all computations of similarity and then that for each one could be neglected. Because of the sequence-to-sequence nature of t2vec, given the representation of $T[i, j-1]$, it would take $O(1)$ to compute that of $T[i, j]$ $(1 \leq i < j \leq n)$. Therefore, we know $\Phi_{inc} = O(1)$. Besides, we know $\Phi_{ini} = O(1)$ since the subtrajectory involved in the computation of similarity, i.e., $T[i, i]$ $(1 \leq i \leq n)$, has its size equal to 1.

DTW [38]. Given a data trajectory $T =< p_1, p_2, ..., p_n >$ and a query trajectory $T_q =< q_1, q_2, ..., q_m >$, the DTW distance is defined as below

$$D_{i,j} = \begin{cases} \sum_{h=1}^{i} d(p_h, q_1) & \text{if } j = 1 \\ \sum_{k=1}^{j} d(p_1, q_k) & \text{if } i = 1 \\ d(p_i, q_j) + & \\ \min(D_{i-1,j-1}, D_{i-1,j}, D_{i,j-1}) & \text{otherwise} \end{cases} \tag{6.1}$$

where $D_{i,j}$ denotes the DTW distance between $T[1, i]$ and $T_q[1, j]$ and $d(p_i, q_j)$ is the distance between p_i and q_j (typically the Euclidean distance, which could be computed

in $O(1)$). Consider Φ. It is clear that $\Phi = O(n \cdot m)$ since it needs to compute all pairwise distances between a point in a subtrajectory of T and a point in T_q and in general, the subtrajectory has its size of $O(n)$ and T_q has its size of m. Consider Φ_{inc}. This should be the same as the time complexity of computing $D_{i,m}$ given that $D_{i-1,m}$ has been computed. Since $D_{i-1,m}$ has been computed, we can safely assume that $D_{i-1,1}$, $D_{i-1,2}$, ..., $D_{i-1,m}$ have been computed also according to Equation 6.1 (note that we can always make this hold by enforcing that we compute $D_{i-1,m}$ or any other DTW distance in this way). Therefore, in order to compute $D_{i,m}$, we compute $D_{i,1}, D_{i,2}, ..., D_{i,m}$ sequentially, each of which would take $O(1)$ time with the information of $D_{i-1,k}$ $(1 \leq k \leq m)$ all available. That is, it takes $O(m)$ to compute $D_{i,m}$, and thus we know that $\Phi_{inc} = O(m)$. Consider Φ_{ini}. We know $\Phi_{ini} = O(m)$ since $T[i,i]$ $(1 \leq i \leq n)$ has its size always equal to 1 and T_q has its size of m.

Frechet [65]. Given a data trajectory $T =< p_1, p_2, ..., p_n >$ and a query trajectory $T_q =< q_1, q_2, ..., q_m >$, the Frechet distance is defined as below

$$F_{i,j} = \begin{cases} \max_{h=1}^{i} d(p_h, q_1) & \text{if } j = 1 \\ \max_{k=1}^{j} d(p_1, q_k) & \text{if } i = 1 \\ \max(d(p_i, q_j), & \\ \min(F_{i-1,j-1}, F_{i-1,j}, F_{i,j-1})) & \text{otherwise} \end{cases} \tag{6.2}$$

where $F_{i,j}$ denotes the Frechet distance between $T[1,i]$ and $T_q[1,j]$ and $d(p_i, q_j)$ is the distance between p_i and q_j (typically the Euclidean distance, which could be computed in $O(1)$). When the Frechet distance is used, we have $\Phi = O(n \cdot m)$, $\Phi_{inc} = O(m)$, and $\Phi_{ini} = O(m)$, based on similar analysis as for the DTW distance.

The summary of Φ, Φ_{inc} and Φ_{ini} for the similarity measurements corresponding to the distance measurements DTW, Frechet and t2vec is presented in Table 6.1.

6.3 Proposed Method

6.3.1 Non-learning based Algorithms

In this part, we introduce three types of algorithms, namely an exact algorithm ExactS, an approximate algorithm SizeS, and splitting-based algorithms including PSS, POS

Table 6.2: Time complexities of algorithms ($n_1 << n$)

Algorithms	abstract similarity measurement	t2vec	DTW	Frechet
ExactS	$O(n \cdot (\Phi_{ini} + n \cdot \Phi_{inc}))$	$O(n^2)$	$O(n^2 \cdot m)$	$O(n^2 \cdot m)$
SizeS	$O(n \cdot (\Phi_{ini} + (m + \xi) \cdot \Phi_{inc}))$	$O((\xi + m) \cdot n)$	$O((\xi + m) \cdot n \cdot m)$	$O((\xi + m) \cdot n \cdot m)$
PSS, POS, POS-D	$O(n_1 \cdot \Phi_{ini} + n \cdot \Phi_{inc})$	$O(n)$	$O(n \cdot m)$	$O(n \cdot m)$
RLS, RLS-Skip (learning-based)	$O(n_1 \cdot \Phi_{ini} + n \cdot \Phi_{inc})$	$O(n)$	$O(n \cdot m)$	$O(n \cdot m)$

and POS-D. The ExactS algorithm is based on an exhaustive search with some careful implementation and has the highest complexity, the SizeS algorithm is inspired by existing studies on subsequence matching [44, 45] and provides a tunable parameter for controlling the trade-off between efficiency and effectiveness, and the splitting-based algorithms are based on the idea of splitting the data trajectory for constructing subtrajectories as candidates of the solution and run the fastest. A summary of the time complexities of these algorithms is presented in Table 6.2.

6.3.1.1 The ExactS Algorithm

Let T be a data trajectory and T_q be a query trajectory. The ExactS algorithm enumerates all possible subtrajectories $T[i, j]$ $(1 \leq i \leq j \leq n)$ of the data trajectory T and computes the similarity between each $T[i, j]$ and T_q, i.e., $\Theta(T[i, j], T_q)$, and then returns the one with the greatest similarity. For better efficiency, ExactS computes the similarities between the subtrajectories and T_q *incrementally* as much as possible as follows. It involves n iterations, and in the i^{th} iteration, it computes the similarity between each subtrajectory starting from the i^{th} point and the query trajectory in an ascending order of the ending points, i.e., it computes $\Theta(T[i, i], T_q)$ (from scratch) first and then computes $\Theta(T[i, i+1], T_q)$, ..., $\Theta(T[i, n], T_q)$ sequentially and incrementally. During the process, it maintains the subtrajectory that is the most similar to the query one, among those that have been traversed so far. As could be verified, it would traverse all possible subtrajectories after n iterations. The ExactS algorithm with this implementation is presented in Algorithm 4.

Consider the time complexity of ExactS. Since there are n iterations and in each iteration, the time complexity of computing $\Theta(T[i, i], T_q)$ is Φ_{ini} and the time complexity of computing $\Theta(T[i, i+1], T_q)$, ..., and $\Theta(T[i, n], T_q)$ is $O(n \cdot \Phi_{inc})$, we know that the overall time complexity is $O(n \cdot (\Phi_{ini} + n \cdot \Phi_{inc}))$.

Algorithm 4: ExactS

Input: A data trajectory T and query trajectory T_q;
Output: A subtrajectory of T that is the most similar to T_q;

$T_{best} \leftarrow \emptyset$; $\Theta_{best} \leftarrow 0$;
forall $1 \leq i \leq |T|$ **do**
  compute $\Theta(T[i,i], T_q)$;
  if $\Theta(T[i,i], T_q) > \Theta_{best}$ **then**
    $T_{best} \leftarrow T[i,i]$; $\Theta_{best} \leftarrow \Theta(T[i,i], T_q)$;
  end
  forall $i+1 \leq j \leq |T|$ **do**
    compute $\Theta(T[i,j], T_q)$ based on $\Theta(T[i,j-1], T_q)$;
    if $\Theta(T[i,j], T_q) > \Theta_{best}$ **then**
      $T_{best} \leftarrow T[i,j]$; $\Theta_{best} \leftarrow \Theta(T[i,j], T_q)$;
    end
  end
end
return T_{best};

We note that for some specific similarity measurement, there may exist algorithms that have better time complexity than ExactS. For example, the Spring algorithm [124], which finds the most similar subsequence of a data time series to a query one, is applicable to the SimSub problem and has the time complexity of $O(nm)$. The major idea of Spring is a dynamic programming process for computing the DTW distance between the data time series and the query one, where the latter is padded with a fictitious point that could be aligned with any point of the data time series with distance equal to 0 (so as to cover all possible suffixes of the data time series). Nevertheless, Spring is designed for the specific similarity DTW while ExactS works for an abstract one that could be instantiated to be any similarity.

6.3.1.2 The SizeS Algorithm

ExactS explores all possible $\frac{n(n+1)}{2}$ subtrajectories, many of which might be quite dissimilar from the query trajectory and could be ignored. For example, by following some existing studies on subsequence matching [44, 45], we could restrict our attention to only those subtrajectories, which have similar sizes as the query one for better efficiency. Specifically, we enumerate all subtrajectories that have their sizes within the range $[m-\xi, m+\xi]$, where $\xi \in [0, n-m]$ is a pre-defined parameter that controls the trade-off between the efficiency and effectiveness of the algorithm. Again, we adopt the

strategy of incremental computation for the similarities between those subtrajectories starting from the same point and the query trajectory. We call this algorithm *SizeS* and analyze its time complexity as follows. The time complexity of computing the similarities between all subtrajectories starting from a specific point and having their sizes within the range $[m-\xi, m+\xi]$ is $O(\Phi_{ini}+(m-\xi-1)\cdot\Phi_{inc}+2\xi\cdot\Phi_{inc})$, where $\Phi_{ini}+(m-\xi-1)\cdot\Phi_{inc}$ is cost of computing $\Theta(T[i, i+m-\xi-1], T_q)$ and $2\xi\cdot\Phi_{inc}$ is the cost of computing $\Theta(T[i, j], T_q)$ for $j \in [i+m-\xi, i+m+\xi-1]$. It could be further reduced to $O(\Phi_{ini} + (m+\xi)\cdot\Phi_{inc})$. Therefore, the overall time complexity of SizeS is $O(n\cdot(\Phi_{ini}+(m+\xi)\cdot\Phi_{inc}))$. For example, when DTW or Frechet is used, it is $O(n\cdot(m+(m+\xi)\cdot m)) = O((\xi+m)\cdot n\cdot m)$ and when t2vec is used, it is $O(n\cdot(1+(\xi+m)\cdot 1)) = O((\xi+m)\cdot n)$.

In summary, SizeS achieves a better efficiency than ExactS at the cost of its effectiveness. Besides, SizeS still needs to explore $O(\xi\cdot n)$ subtrajectories, which restricts its application on small and moderate datasets only. Unfortunately, SizeS may return a solution, which is arbitrarily worse than the best one.

6.3.1.3 Splitting-based Algorithms

The ExactS algorithm is costly since it explores $O(n^2)$ subtrajectories. The SizeS algorithm runs faster than ExactS since it explores about $O(\xi\cdot n)$ subtrajectories ($\xi << n$). Thus, an intuitive idea to push the efficiency further up is to explore fewer subtrajectories. In the following, we design a series of three approximate algorithms, which all share the idea of splitting a data trajectory into several subtrajectories and returning the one that is the most similar to the query trajectory. These algorithms differ from each other in using different heuristics for deciding where to split the data trajectory. With this splitting strategy, the number of subtrajectories that would be explored is bounded by n and in practice, much smaller than n. We describe these algorithms as follows.

(1) Prefix-Suffix Search (PSS). The PSS algorithm is a greedy one, which maintains a variable T_{best} storing the subtrajectory that is the most similar to the query trajectory found so far. Specifically, it scans the points of the data trajectory T in the order of $p_1, p_2, ..., p_n$. When it scans p_i, it computes the similarities between the two subtrajectories that would be formed if it splits T at p_i, i.e., $T[h, i]$ and $T[i, n]$, and the query trajectory T_q, where p_h is the point following the one, at which the last split was done if

any and p_h is the first point p_1 otherwise. In particular, we replace the part of computing the similarity between the suffix $T[i,n]$ and the query trajectory with that between their reversed versions, denoted by $T[i,n]^R$ and T_q^R, respectively. This is because (1) $\Theta(T[i,n]^R, T_q^R)$ could be computed incrementally based on $\Theta(T[i+1,n]^R, T_q^R)$ and (2) $\Theta(T[i,n]^R, T_q^R)$ and $\Theta(T[i,n], T_q)$ are equal for some similarity measurements such as DTW and Frechet and positively correlated for others such as t2vec as we found via experiments. If any of these two similarities are larger than the best-known similarity, it performs a split operation at p_i and updates T_{best} accordingly; otherwise, it continues to scan the next point p_{i+1}. At the end, it returns T_{best}. The procedure of PSS is presented in Algorithm 5.

We analyze the time complexity of PSS as follows. When it scans a specific point p_i, the time costs include that of computing $\Theta(T[h,i], T_q)$ and also that of computing $\Theta(T[i,n]^R, T_q^R)$. Consider the former part. If $i = h$, it is Φ_{ini}. If $i \geq h+1$, it is Φ_{inc} since $\Theta(T[h,i], T_q)$ could be computed based on $\Theta(T[h,i-1], T_q)$ incrementally. Consider the latter part. It is simply $O(\Phi_{inc})$. In conclusion, the time complexity of PSS is $O(n_1 \cdot (\Phi_{ini} + \Phi_{inc}) + (n - n_1) \cdot \Phi_{inc}) = O(n_1 \cdot \Phi_{ini} + n \cdot \Phi_{inc})$, where n_1 is the number of points where splits are done. For example, when DTW or Frechet is used, the time complexity of PSS is $O(n_1 \cdot m + n \cdot m) = O(n \cdot m)$ and when t2vec is used, it is $O(n_1 \cdot 1 + n \cdot 1) = O(n)$.

(2) Prefix-Only Search (POS). In PSS, when it scans a point p_i, it considers two subtrajectories, namely $T[h,i]$ and $T[i,n]$. An alternative is to consider the prefix $T[h,i]$ only - one argument is that the suffix $T[i,n]$ might be destroyed when further splits are conducted. A consequent benefit is that the time cost of computing $\Theta(T[i,n], T_q)$ would be saved. We call this algorithm the POS algorithm. As could be verified, POS has the same time complexity as PSS though the former runs faster in practice.

(3) Prefix-Only Search with Delay (POS-D). POS performs a split operation whenever a prefix, which is better than the best subtrajectory known so far, is found. This looks a bit rush and may prevent a better subtrajectory to be formed by extending it with a few more points. Thus, we design a variant of POS, called *Prefix-Only Search with Delay* (POS-D). Whenever a prefix is found to be more similar to the query trajectory than the best subtrajectory known so far, POS-D continues to scan D more points and splits at one of these $D+1$ points, which has the corresponding prefix the most similar

Algorithm 5: Prefix-Suffix Search (PSS)

Input: A data trajectory T and query trajectory T_q;
Output: A subtrajectory of T that is similar to T_q;

$T_{best} \leftarrow \emptyset$; $\Theta_{best} \leftarrow 0$;
compute $\Theta(T[n,n]^R, T_q^R)$;
compute $\Theta(T[n-1,n]^R, T_q^R)$, $\Theta(T[n-2,n]^R, T_q^R)$, ..., $\Theta(T[1,n]^R, T_q^R)$ incrementally;
$h \leftarrow 1$;
forall $1 \leq i \leq |T|$ **do**
    compute $\Theta(T[h,i], T_q)$ incrementally if possible;
    if $\max\{\Theta(T[h,i], T_q), \Theta(T[i,n]^R, T_q^R)\} > \Theta_{best}$ **then**
        $\Theta_{best} \leftarrow \max\{\Theta(T[h,i], T_q), \Theta(T[i,n]^R, T_q^R)\}$;
        if $\Theta(T[h,i], T_q) > \Theta(T[i,n]^R, T_q^R)$ **then**
            $T_{best} \leftarrow T[h,i]$;
        else
            $T_{best} \leftarrow T[i,n]$;
        end
        $h \leftarrow i+1$;
    end
end
return T_{best};

to the query trajectory. It could be verified that with this delay mechanism, the time complexity of the algorithm does not change though in practice, it would be slightly higher.

While these splitting-based algorithms including PSS, POS and POS-D, return reasonably good solutions in practice, they may return solutions that are arbitrarily worse than the best one in theory.

6.3.2 Reinforcement Learning based Algorithms

A splitting-based algorithm has its effectiveness rely on the quality of the process of splitting a data trajectory. In order to find a solution of high quality, it requires to perform split operations at appropriate points such that some subtrajectories that are similar to a query trajectory are formed and then explored. The three splitting-based algorithms, namely PSS, POS and POS-D, mainly use some hand-crafted heuristics for making decisions on whether to perform a split operation at a specific point. This process of splitting a trajectory into subtrajectories is a typical *sequential decision making* process. Specifically, it scans the points sequentially and for each point, it makes a decision on whether or not to perform a split operation at the point. In this chapter, we propose

to model this process as a *Markov decision process* (MDP) [18] (Section 6.3.2.1), adopt a *deep-Q-network* (DQN) [17] for learning an optimal policy for the MDP (Section 6.3.2.2), and then develop an algorithm called *reinforcement learning based search* (RLS), which corresponds to a splitting-based algorithm that uses the learned policy for the process of splitting a data trajectory (Section 6.3.2.3) and an augmented version of RLS, called RLS-Skip, with better efficiency (Section 6.3.2.4)

6.3.2.1 Trajectory Splitting as a MDP

A MDP consists of four components, namely *states*, *actions*, *transitions*, and *rewards*, where (1) a state captures the *environment* that is taken into account for decision making by an *agent*; (2) an action is a possible decision that could be made by the agent; (3) a transition means that the state changes from one to another once an action is taken; and (4) a reward, which is associated with a transition, corresponds to some feedback indicating the quality of the action that causes the transition. We model the process of splitting a data trajectory as a MDP as follows.

(1) States. We denote a state by s. Suppose it is currently scanning point p_t. p_h denotes the point following the one, at which the last split operation happens if any and p_1 otherwise. We define the state of the current environment as a triplet $(\Theta_{best}, \Theta_{pre}, \Theta_{suf})$, where Θ_{best} is the largest similarity between a subtrajectory found so far and the query trajectory T_q, Θ_{pre} is $\Theta(T[h,t], T_q)$ and Θ_{suf} is $\Theta(T[t,n]^R, T_q^R)$. As could be noticed, a state captures the information about the query trajectory, the data trajectory, the point at which the last split happens, and the point that is being scanned, etc. Note that the state space is a three-dimensional continuous one.

(2) Actions. We denote an action by a. We define two actions, namely $a = 1$ and $a = 0$. The former means to perform a split operation at the point that is being scanned and the latter means to move on to scan the next point.

(3) Transitions. In the process of splitting a trajectory, given a current state s and an action a to take, the probability that we would observe a specific state s' is unknown. We note that the method that we use for solving the MDP in this chapter is a *model-free* one and could solve the MDP problem even with its transition information unknown.

(4) Rewards. We denote a reward by r. We define the reward associated with the transition from state s to state s' after action a is taken as $(s'.\Theta_{best} - s.\Theta_{best})$, where the $s'.\Theta_{best}$ is the first component of state s' and $s.\Theta_{best}$ is the first component of state s. With this reward definition, the goal of the MDP problem, which is to maximize the accumulative rewards, is consistent with that of the process of splitting a data trajectory, which is to form a subtrajectory with the greatest possible similarity to the query trajectory. To see this, consider that the process goes through a sequence of states $s_1, s_2, ..., s_N$ and ends at s_N. Let $r_1, r_2, ..., r_{N-1}$ denote the rewards received at these states except for the termination state s_N. Then, when the future rewards are not discounted, we have

$$\Sigma_t r_t = \Sigma_t (s_t.\Theta_{best} - s_{t-1}.\Theta_{best}) = s_N.\Theta_{best} - s_1.\Theta_{best}$$

where $s_N.\Theta_{best}$ corresponds to the similarity between the best subtrajectory found and the query trajectory T_q and $s_1.\Theta_{best}$ corresponds to the best known similarity at the beginning, i.e., 0. Therefore, maximizing the accumulative rewards is equivalent to maximizing the similarity between the subtrajectory to be found and T_q in this case.

6.3.2.2 Deep-Q-Network (DQN) Learning

The core problem of a MDP is to find an optimal *policy* for the agent, which corresponds to a function π that specifies the action that the agent should choose when at a specific state so as to maximize the accumulative rewards. One type of methods that are commonly used is those *value-based* methods [17, 103]. The major idea is as follows. First, it defines an optimal action-value function $Q^*(s, a)$ (or Q function), which represents the maximum amount of expected accumulative rewards it would receive by following any policy after seeing the state s and taking the action a. Second, it estimates $Q^*(s, a)$ using some methods such as Q-learning [110] and deep-Q-network (DQN) [17]. Third, it returns the policy, which always chooses for a given state s the action a that maximizes $Q^*(s, a)$.

In our MDP, the state space is a three dimensional continuous one, and thus we adopt the DQN method. Specifically, we use the *deep Q learning with replay memory* [120] for learning the Q functions. This method maintains two neural networks. One is called the *main network* $Q(s, a; \theta)$, which is used to estimate the Q function. The other is

Algorithm 6: Deep-Q-Network (DQN) Learning with Experience Replay

Input: A database $\mathcal{D}$ of data trajectories and a set of $\mathcal{D}_q$ of query trajectories;
Output: Learned action-value function $Q(s, a; \theta)$;
1 initialize the reply memory M;
2 initialize the main network $Q(s, a; \theta)$ with random weights θ;
3 initialize the target network $\hat{Q}(s, a; \theta^-)$ with weights $\theta^- = \theta$;
4 **for** *episode* $= 1, 2, 3, \ldots$ **do**
5 sample a data and query trajectory T, T_q;
6 $h \leftarrow 1$;
7 $\Theta_{best} \leftarrow 0$; $\Theta_{pre} \leftarrow \Theta(T[h, h], T_q)$; $\Theta_{suf} \leftarrow \Theta(T[h, n]^R, T_q^R)$;
8 observe the first state $s_1 = (\Theta_{best}, \Theta_{pre}, \Theta_{suf})$;
9 **for** *each step* $1 \leq t \leq |T|$ **do**
10 select a random action a_t with probability ϵ and select action $a_t = \arg\max_a Q(s_t, a; \theta)$ with probablity $(1 - \epsilon)$;
11 **if** $a_t = 1$ **then**
12 $h \leftarrow t + 1$;
13 **end**
14 $\Theta_{best} \leftarrow \max\{s_t.\Theta_{best}, s_t.\Theta_{pre}, s_t.\Theta_{suf}\}$;
15 **if** $t = |T|$ **then**
16 **break**;
17 **end**
18 $\Theta_{pre} \leftarrow \Theta(T[h, t + 1], T_q)$; $\Theta_{suf} \leftarrow \Theta(T[t + 1, n]^R, T_q^R)$;
19 observe the next state $s_{t+1} = (\Theta_{best}, \Theta_{pre}, \Theta_{suf})$;
20 observe the reward $r_t = s_{t+1}.\Theta_{best} - s_t.\Theta_{best}$;
21 store the experience (s_t, a_t, r_t, s_{t+1}) in the replay memory $\mathcal{M}$;
22 sample a random minibatch of experiences from $\mathcal{M}$ uniformly;
23 perform a gradient descent step on the loss as computed by Equatioin (6.3) wrt θ;
24 **end**
25 copy the main network $Q(s, a; \theta)$ to $\hat{Q}(s, a; \theta^-)$;
26 **end**

called the *target network* $\hat{Q}(s, a; \theta^-)$, which is used to compute some form of loss for training the main network. Besides, it maintains a fixed-size pool called *replay memory*, which contains the latest transitions that are sampled uniformly and used for training the main network. The intuition is to avoid the correlation among consecutive transitions. The detailed procedure of DQN for our MDP is presented in Algorithm 6, which we go through as follows. We maintain a database $\mathcal{D}$ of data trajectories and a set of $\mathcal{D}_q$ of query trajectories. It first initializes the reply memory $\mathcal{M}$ with some capacity, the main network $Q(s, a; \theta)$ with random weights, and the target network $\hat{Q}(s, a; \theta^-)$ by copying $Q(s, a; \theta)$ (Lines 1 - 3). Then, it involves a sequence of many episodes. For each episode, it samples a data trajectory T from $\mathcal{D}$ and a query trajectory T_q from $\mathcal{D}_q$, both uniformly (Lines 4 - 5). It initializes a variable h such that p_h corresponds to the point following

the one, at which the last split operation is performed if any and p_1 otherwise (Line 6). It also initializes the state s_1 (Lines 7 - 8). Then, it proceeds with $|T|$ time steps. At the t^{th} time step, it scans point p_t and selects an action using the ϵ-greedy strategy based on the main network, i.e., it performs a random action a_t with the probability ϵ ($0 < \epsilon < 1$) and $a_t = \arg\max_a Q(s_t, a; \theta)$ with the probability $(1-\epsilon)$ (Lines 9 - 10). If $a_t = 1$, it splits the trajectory at point p_t and updates h to be $t+1$ (Lines 11 - 13). It then updates Θ_{best} if possible (Line 14). If the current point being scanned is the last point p_n, it terminates (Lines 15 - 17). Otherwise, it observes a new state s_{t+1} and the reward r_t (Lines 18 - 20). It then stores the experience (s_t, a_t, r_t, s_{t+1}) in the reply memory, samples a minibatch of experiences, and uses it to perform a gradient descent step for updating θ wrt a loss function (Lines 21 - 23). The loss function for one experience (s, a, r, s') is as follows.

$$L(\theta) = (y - Q(s, a; \theta))^2 \tag{6.3}$$

where y is equal to r if s' is a termination step and $r + \gamma \cdot \max_{a'} \hat{Q}(s', a'; \theta^-)$ otherwise. Finally, it updates the target network $\hat{Q}(s, a; \theta^-)$ with the main network $Q(s, a; \theta)$ at the end of each episode (Line 25). A graphical illustration of the method is shown in Figure 6.1.

6.3.2.3 RL-based Search Algorithm (RLS)

Once we have estimated the Q functions $Q(s, a; \theta)$ via the deep Q learning with experience replay, we use the policy, which always takes for a given state s the action that maximizes $Q(s, a; \theta)$, for the process of splitting a data trajectory. Among all subtrajectories that are formed as a result of the process, we return the one with the greatest similarity to the query trajectory T_q. We call this algorithm *reinforcement learning based search* (RLS). Essentially, it is the same as PSS except that it uses a policy learned via DQN instead of human-crafted heuristics for making decisions on how to split a data trajectory.

RLS has the same time complexity as PSS since both RLS and PSS make decisions based on the similarities of the subtrajectories that are being considered when scanning a point and the best-known similarity: (1) RLS constructs a state involving them and goes through the main network of DQN with the state information, which is $O(1)$ given that the network is small-size (e.g., a few layers); and (2) PSS simply conduct some

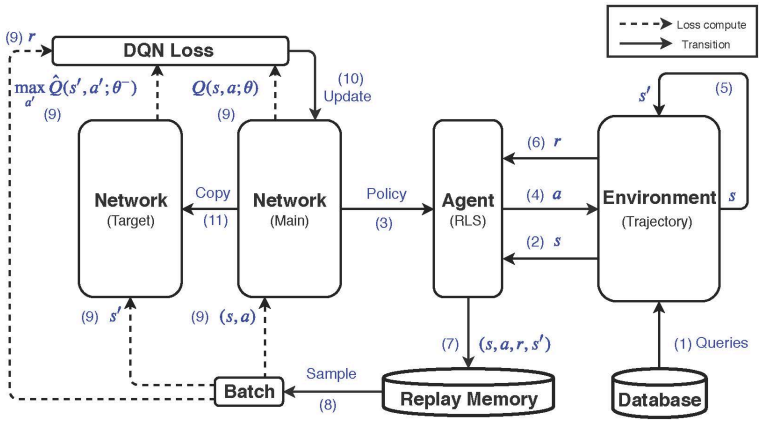


Figure 6.1: Deep Q learning with experience replay

comparisons among the similarities, which is also $O(1)$. In terms of effectiveness, RLS provides consistently better solutions than PSS as well as POS and POS-D, as will be shown in the empirical studies, and the reason is possibly that RLS is based on a learned policy, which makes decision more intelligent than simple heuristics that are human-crafted.

6.3.2.4 RL-based Search with Skipping (RLS-Skip)

In the RLS algorithm, each point is considered as a candidate for performing a split operation. While this helps to attain a reasonably large space of subtrajectories for exploration and hence achieving good effectiveness, it is somehow conservative and incurs some cost of decision marking for *each* point. An alternative is to go a bit more optimistic and *skip* some points from being considered as places for split operations. The benefit would be immediate, i.e., the cost of making decisions at these points is saved. Motivated by this, we propose to augment the MDP that is used by RLS by introducing k more actions (apart from two existing ones: scanning the next point and performing a split operation), namely skipping 1 point, skipping 2 points, ..., skipping k points. Here, k is a hyperparameter, and by skipping j points ($j = 1, 2, ..., k$), it means to skip points $p_{i+1}, p_{i+2}, ..., p_{i+j}$ and scan point p_{i+j+1} next, where p_i is the point that is being scanned. All other components of the MDP are kept the same as that for RLS. Note that when $k = 0$, this MDP reduces to the original one for RLS. We call the algorithm based on this augmented MDP *RLS-Skip*.

While the cost of making decisions at those points that are skipped (i.e., that of going through the main network of the DQN) could be saved in RLS-Skip, the cost of constructing the states at those points that are not skipped would be more or less that of constructing the states at *all* points since the state at a point involves some similarities, which are computed incrementally based on the similarities computed at those points before the point. Thus, by applying the skipping strategy alone would not help much in reducing the time cost since the cost of maintaining the states dominates that of making decisions. To fully unleash the power of the skipping strategy, we propose to ignore those points that have been skipped when maintaining the states. That is, to maintain the state $(\Theta_{best}, \Theta_{pre}, \Theta_{suf})$ at a point p_i, we compute Θ_{best} and Θ_{suf} in the same way as we do in RLS and Θ_{pre} as the similarity between the query trajectory and the subtrajectory consisting of those points that are before p_i and *have not been skipped*. Here, the prefix subtrajectory corresponds to a *simplification* of that used in RLS [13]. While RLS-Skip has the same worse-case time complexity as RLS, e.g., it reduces to RLS when no skipping operations happen, the cost of maintaining the states for RLS-Skip would be much smaller. As shown in our empirical studies, RLS-Skip runs significantly faster than RLS as well as PSS, POS and POS-D. In addition, RLS-Skip and RLS do not provide theoretical guarantees on the approximation quality due to their learning nature. Nevertheless, they work quite well in practice (e.g., RLS has the approximation ratio smaller than 1.1 for all similarity measurements and on all datasets (Figure 6.2)). In addition, the problem instances that we constructed for proving the negative results in fact rarely happen in practice, which are confirmed by the effectiveness results on real datasets.

6.4 Experiments

6.4.1 Experimental Setup

Dataset. Our experiments are conducted on three real-world trajectory datasets. The first dataset, denoted by Porto, is collected from the city of Porto [2], Portugal, which consists around 1.7 million taxi trajectories over 18 months with a sampling interval

[2]https://www.kaggle.com/c/pkdd-15-predict-taxi-service-trajectory-i/data

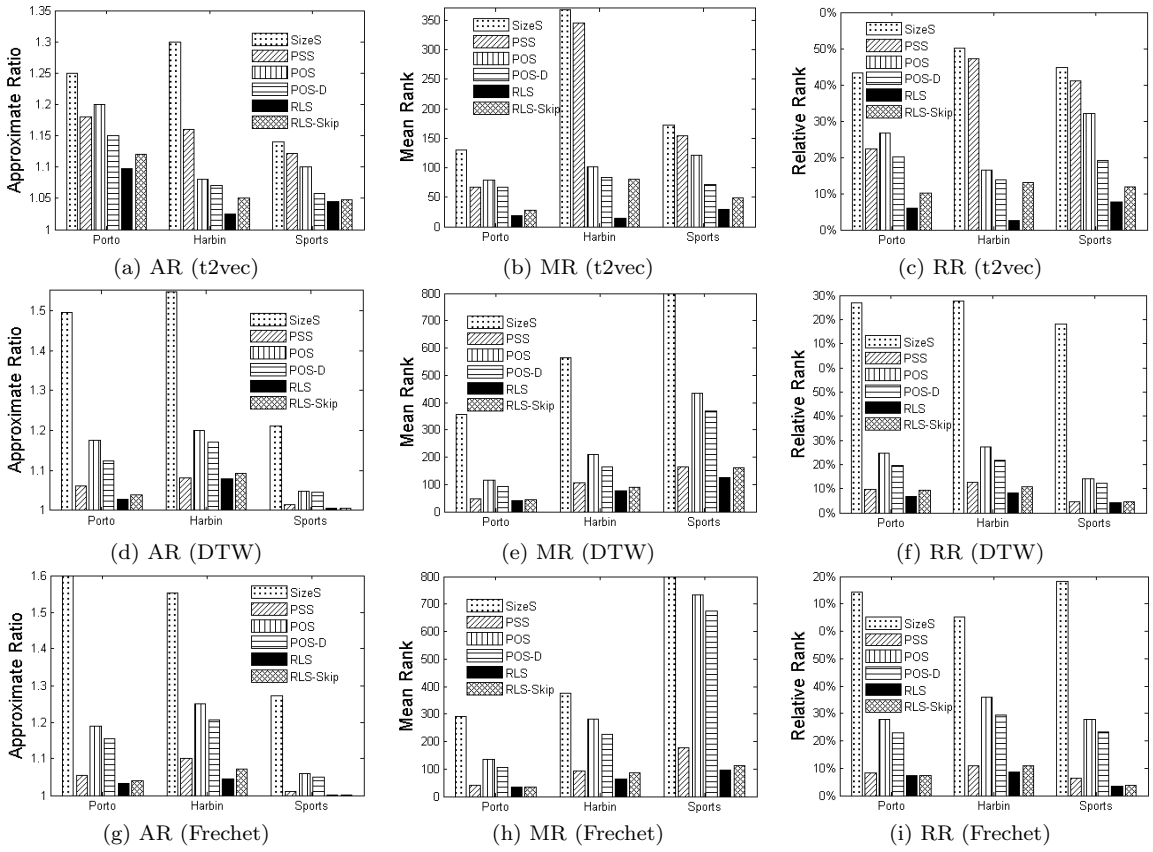
of 15 seconds and a mean length around 60. The second dataset, denoted by Harbin, involves around 1.2 million taxi trajectories collected from 13,000 taxis over 8 months in Harbin, China with non-uniform sampling rates and a mean length around 120. The third dataset, denoted by Sports, involves around 0.2 million soccer player and ball trajectories collected from STATS Sports [3] with a uniform sampling rate of 10 times per second and a mean length around 170.

Parameter Setting. For training t2vec model, we follow the original paper [37] by excluding those trajectories that are short and use their parameter settings. For SizeS, we use the setting $\xi = 5$ (with the results of its effect shown later on). For POS-D, we vary the parameter D from 4 to 7, and since the results are similar, we use the setting $D = 5$. For the neural networks involved in the RL-based algorithms, i.e., RLS and RLS-Skip, we use a feedforward neural network with 2 layers. In the first layer, we use the ReLu function with 20 neurons, and in the second layer, we use the sigmoid function with $2+k$ neurons as the output corresponding to different actions, where for RLS we use $k = 0$ and for RLS-Skip, we use $k = 3$ by default. In the training process, the size of replay memory $\mathcal{M}$ is set at 2000. We train our model on 25k random trajectory pairs, using Adam stochastic gradient descent with an initial learning rate of 0.001. The minimal ϵ is set at 0.05 with decay 0.99 for the ϵ-greedy strategy, and the reward discount rate γ is set at 0.95.

Compared Methods. We compare RL-based Search (RLS), RL-based Search with skipping (RLS-Skip) and the proposed non-learning based algorithms (Section 6.3.1), namely ExactS, SizeS, PSS, POS, and POS-D. For RLS and RLS-Skip, when t2vec is adopted, we ignore the Θ_{suf} component of a state based on empirical findings.

In addition, we consider three competitor methods, namely UCR [73, 74, 125], Spring [124], and Random-S. UCR was originally developed for searching subsequences of a time series, which are the most similar to a query time series and the similarity is based on the DTW distance. UCR enumerates all subsequences that are of the same length of the query time series and employs a rich set of techniques for pruning many of the subsequences. We adapt UCR for our similar subtrajectory search problem We note that UCR only works for DTW, but not for Frechet or t2vec. Spring is an existing algorithm for searching

[3] https://www.stats.com/artificial-intelligence (STATS, copyright 2019)



(a) AR (t2vec) (b) MR (t2vec) (c) RR (t2vec)

(d) AR (DTW) (e) MR (DTW) (f) RR (DTW)

(g) AR (Frechet) (h) MR (Frechet) (i) RR (Frechet)

Figure 6.2: Effectiveness for t2vec (a)-(c), DTW (d)-(f) and Frechet (g)-(i).

a subsequence of a time series, which is the most similar to a query time series. It is designed for DTW. Random-S randomly samples a certain number of subtrajectories of data trajectory and among them, returns the one with the highest similarity to the query trajectory. Since these methods are either not general for all similarity measurements (e.g., UCR) or involve some parameter that is difficult to set (e.g., Random-S with a parameter of sample size), we compare these two competitor methods with our RLS-Skip only in terms of effectiveness and efficiency.

Evaluation Metrics. We use three metrics to evaluate the effectiveness of an approximate algorithm. (1) Approximate Ratio (AR): It is defined as the ratio between the dissimilarity of the solution wrt a query trajectory, which is returned by an approximate algorithm, and that of the solution returned by an exact algorithm. A smaller AR indicates a better algorithm. (2) Mean Rank (MR): We sort all the subtrajectories of a data trajectory in ascending order of their dissimilarities wrt a query trajectory. MR is

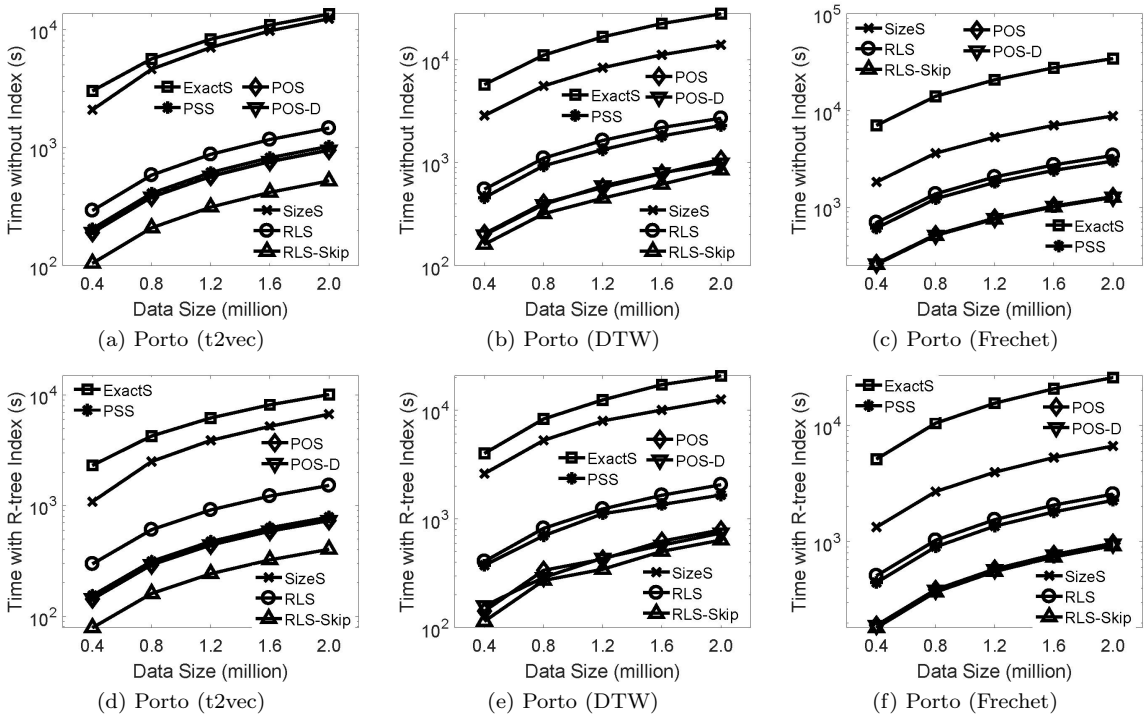
defined as the rank of the solution returned by an approximate algorithm. (3) Relative Rank (RR): RR is a normalized version of MR by the total number subtrajectories of a data trajectory. A smaller MR or RR indicates a better algorithm.

Evaluation Platform. All the methods are implemented in Python 3.6. The implementation of RLS is based on Keras 2.2.0. The experiments are conducted on a server with 32-cores of Intel(R) Xeon(R) Gold 6150 CPU @ 2.70GHz 768.00GB RAM and one Nvidia Tesla V100-SXM2 GPU.

6.4.2 Experimental Results

(1) Effectiveness results. We randomly sample 10,000 trajectory pairs from a dataset, and for each pair we use one trajectory as the query trajectory to search the most similar subtrajectory from the other one. Figure 6.2 shows the results. The results clearly show that RLS and RLS-Skip consistently outperform all other non-learning based approximate algorithms in terms of all three metrics on both datasets and under all three trajectory similarity measurements. For example, RLS outperforms POS-D, the best non-learning algorithm when using t2vec, by 70% (resp. 83%) in terms of RR on Porto (resp. Harbin); RLS outperforms PSS, the best non-learning based algorithm when using DTW, by 25% (resp. 20%) in terms of MR on Porto (resp. Harbin); RLS outperforms PSS, the best non-learning based algorithm when using Frechet, by 25% (resp. 20%) in terms of MR on Porto (resp. Harbin). Among PSS, POS, and POS-D, PSS performs the best for DTW and Frechet; However, for t2vec, PSS provides similar accuracy as POS and POS-D on Porto, but performs much worse on Harbin. The reason is that for DTW and Frechet, PSS computes exact similarity values for suffix subtrajectories, while for t2vec, it computes only approximate ones. Therefore, PSS has a relatively worse accuracy when used for t2vec. We also observe that SizeS is not competitive compared with other approximate algorithms. In addition, RLS-Skip has its effectiveness a bit worse than RLS, but still better than those non-learning based algorithms due to the fact that it is based on a learned policy for decision making.

(2) Efficiency results. We prepare different databases of data trajectories by including different amounts of trajectories from a dataset and vary the total number of points in a databases. For each database, we randomly sample 10 query trajectories from the dataset,



(a) Porto (t2vec) (b) Porto (DTW) (c) Porto (Frechet)

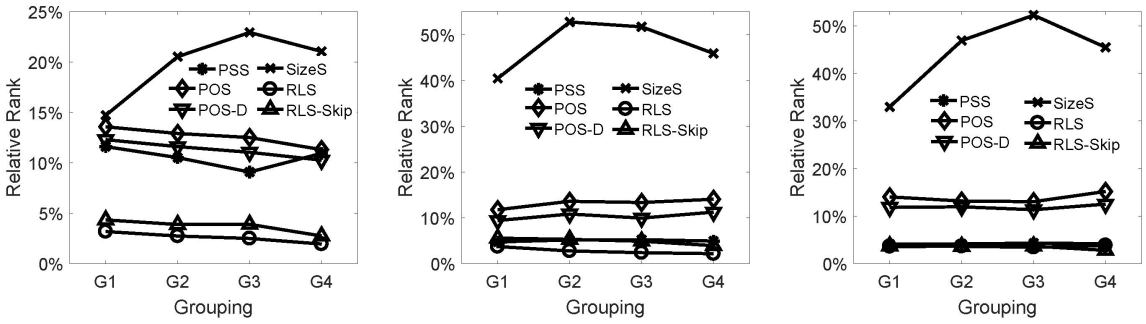
(d) Porto (t2vec) (e) Porto (DTW) (f) Porto (Frechet)

Figure 6.3: Efficiency without index (a)-(c) and with R-tree index (d)-(f) on Porto.

run for each query trajectory a query for finding the top-50 similar subtrajectories, and then collect the average running time over 10 queries. The results of running time on the Porto dataset are shown in Figure 6.3. RLS-Skip runs the fastest since on those points that have been skipped, the cost of maintaining the states and making decisions is saved. In contrast, none of the other algorithms skip points. ExactS has the longest running time, e.g., ExactS is usually around 7-15 times slower than PSS, POS, POS-D, RLS and 20-30 times slower than RLS-Skip. RLS is slightly slower than PSS, POS, POS-D. This is because RLS makes the splitting decision via a learning model while the other three use a simple similarity comparison.

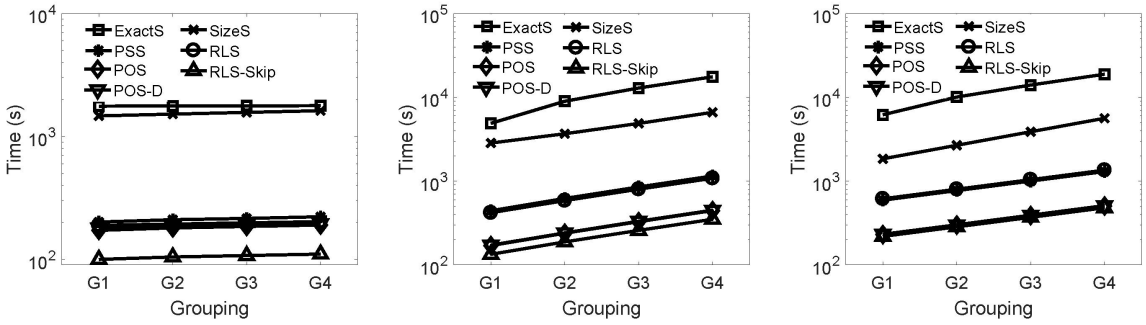
(3) Scalability. We investigate the scalability of all the algorithms based on the results reported in Figure 6.3. All those splitting-based algorithms including PSS, POS, POS-D, RLS and RLS-Skip scale well.

(4) Working with indexes. Following two recent studies [66, 69] on trajectory similarity search, we employ the Bounding Box R-tree Index for boosting the efficiency. It indexes the MBRs of data trajectories and prunes all those data trajectories whose MBRs



(a) Porto (t2vec) (b) Porto (DTW) (c) Porto (Frechet)

Figure 6.4: Effectiveness with varying query lengths.



(a) Porto (t2vec) (b) Porto (DTW) (c) Porto (Frechet)

Figure 6.5: Efficiency with varying query lengths.

do not interact with the MBR of a given query trajectory. We note that in theory, exact solutions might be filtered out by the index (e.g., the most similar subtrajectory may be part of a trajectory, whose MBR does not interact with that of a query one), but in practice, this rarely happen. For example, as found in our experiments on the Porto dataset, when DTW and Frechet are used, the results returned when using the index and those when using no indexes are exactly the same, i.e., no results are missed out. When t2vec is used, at most 20% results are missed. Furthermore, for cases of finding approximate solutions, as most of the proposed algorithms do, missing some potential solutions for better efficiency is acceptable. Compared with the results without indexes in Figure 6.3(a)-(c), the results using the R-tree index as shown in Figure 6.3(d)-(f) are lower by around 20–30%.

(5) The effect of query trajectory length. We prepare four groups of query trajectories from a dataset, namely $G1$, $G2$, $G3$, and $G4$, each with 10,000 trajectories, such that the lengths of the trajectories in a group are as follows: $G1 = [30, 45)$, $G2 = [45, 60)$,

$G3 = [60, 75)$ and $G4 = [75, 90)$. Then, for each query trajectory, we prepare a data trajectory from the dataset. Note that for a query trajectory and a corresponding data trajectory, the latter may be longer than or shorter than the former. For each group, we report the average results. The results of RR on Porto are shown in Figure 6.4. We observe that the RRs of all algorithms except for SizeS remain stable when the query length grows. For SizeS, the RRs fluctuate with the change of the query length. This is because the length of the most similar subtrajectory may not have similar length as the query trajectory, and thus it may miss high-quality results when the search space is constrained by the parameter ξ. The results of running time on Porto are shown in Figure 6.5. We notice that for t2vec, the running times of all the algorithms are almost not affected by the query length. This is because for t2vec, the time complexity of computing a similarity is constant once the vector of the query trajectory is learned. For DTW, Frechet, the time complexity of computing a similarity increase with the query length, as shown in Table 6.2.

More experiments. More experimental results regarding (6) the effect of skipping steps k, (7) the effect of parameter ξ, (8) comparison with similar trajectory search (SimTra), (9) comparison with algorithms for specific measurements (UCR and Spring), (10) comparison with Random-S and (11) training time can be found in [47].

6.5 Summary

In this chapter, we study the similar subtrajectory search (SimSub) problem and develop a suite of algorithms including an exact algorithm, an approximate algorithm, which provides a controllable trade-off between efficiency and effectiveness, and a few splitting-based algorithms, among which some are based on pre-defined heuristics and some are based on deep reinforcement learning called RLS and RLS-Skip. We conducted extensive experiments on real datasets, which verified that among the approximate algorithms, learning based algorithms achieve the best effectiveness and efficiency.

Chapter 7

Deep Representation Learning for Multi-trajectory Similarity Measurement

7.1 Overview

In this chapter, we study the problem of measuring the similarity between two sports plays (a set of multiple trajectories) [1]. We propose to learn representations of plays in a low-dimensional space using deep models, which we call play2vec, such that the (Euclidean) distances in the space capture the similarities among the plays well. This chapter is organized as follows. We give the problem definition in Section 7.2. We present our play2vec method in Section 7.3, report our experimental results in Section 7.4 and conclude the chapter in Section 7.5.

7.2 Problem Statement

We model a sports *game* by the movements of the objects involved in the game (e.g., in a soccer game, the objects include 22 players from two teams and also a ball). The movement of an object is usually captured by sampling its locations at a certain frequency with tracking technologies such as those based on GPS devices. As a result, the movement of an object corresponds to a sequence of time-stamped locations, which is called a

[1] This chapter was published as Effective and Efficient Sports Play Retrieval with Deep Representation Learning [42].

trajectory. A trajectory has its form of (x_1, y_1, t_1), (x_2, y_2, t_2), ..., where (x_i, y_i) is the i^{th} location and t_i is the time stamp of the i^{th} location. Therefore, a play corresponds to a set of multiple trajectories.

A *play* corresponds to a fragment of a game and has its duration varying from seconds to minutes, depending on users' needs. The concept of play provides the flexibility of searching finer-grained games (i.e., game fragments). Same as a game, a play corresponds to a set of multiple trajectories (with shorter lengths).

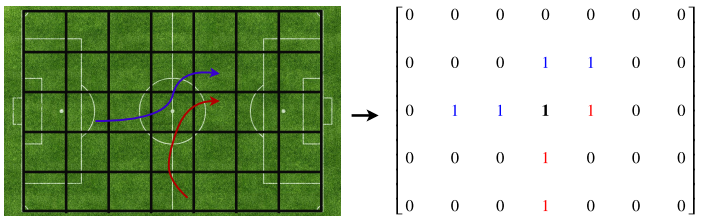
Given a database of plays $\boldsymbol{\mathcal{D}}$, we aim to learn a vector representation $\boldsymbol{v} \in \mathbb{R}^d$ for each play $\boldsymbol{\mathcal{P}} \in \boldsymbol{\mathcal{D}}$ in a d-dimensional space such that the similarities among plays are well captured by the Euclidean distances in the d-dimensional vector space, i.e., for any two plays, if they are similar, the distance between their vectors would be small.

7.3 Proposed Method

In this part, we introduce our deep learning method, play2vec, for learning vector presentations of plays. The core idea is that we break each play into a sequence of non-overlapping segments of a fixed duration, each called a *play segment*, and then design an encoder-decoder model to extract the features from the sequence as a vector. Specifically, our method first builds a corpus $\boldsymbol{\mathcal{V}}$ based on all play segments (Section 7.3.1), then adopts the Skip-Gram with Negative Sampling (SGNS) model [46] for learning distributed representations of the tokens in $\boldsymbol{\mathcal{V}}$ (Section 7.3.2), and eventually glues all distributed representations of the play segments in a play, yielding a vector representation $\boldsymbol{v}$ using the Denoising Sequential Encoder-Decoder (DSED) model that is designed in this work (Section 7.3.3).

7.3.1 Building a Sports Corpus

First, we introduce a process of mapping a play segment to a binary matrix with the help of a grid. Specifically, we divide the pitch into a grid with equal cell size γ, for which we would have a corresponding matrix called *segment matrix*, i.e., each cell in the grid has a corresponding entry in the matrix. Given a play segment, which consists of a set of trajectories, we set to 1 all those entries whose corresponding cells are traveled through



(a) soccer pitch (b) segment matrix $\mathcal{M}$

Figure 7.1: Mapping a play segment to a matrix. The soccer pitch is divided into 5×7 grid map. There are two trajectories with the color red and blue in the soccer pitch.

by the trajectories and 0 the remaining entries. An example is shown in Figure 7.1 for illustration. Note that in this example, the 5×7 grid map is determined by the cell size, which is set empirically.

Since each segment matrix has binary values only, the number of possible segment matrices is limited. A simple strategy is to create one unique token for each possible segment matrix. Nevertheless, with this strategy, the resulting corpus could be big, which may affect the effectiveness of the representation learning afterwards. Thus, we propose to scan the segment matrices one by one and for each segment matrix, we create a new token only if it is dissimilar from those segment matrices that have been scanned to a certain extent, where we use the Jaccard index for measuring the similarity between two segment matrices $\mathcal{M}$ and $\mathcal{M}'$, which is defined as follows.

$$\mathcal{J}(\mathcal{M}, \mathcal{M}') := \frac{m_{11}}{m_{01} + m_{10} + m_{11}}, \tag{7.1}$$

where m_{11} means the total number of attributes where $\mathcal{M}$ and $\mathcal{M}'$ both have a value of 1, m_{01} (or m_{10}) means the total number of attributes where $\mathcal{M}$ is 0 (or 1) and $\mathcal{M}'$ is 1 (or 0).

7.3.2 Learning Distributed Representations

In this part, we introduce a method for embedding the play segments (or their corresponding tokens) as d'-dimensional real-valued vectors. Inspired by the success of word2vec techniques in natural language processing, we adopt the Skip-Gram with Negative Sampling (SGNS) model for this task. The effectiveness of a SGNS model depends on how good the context of a token is modeled. The segment tokens occurring in the same context tend to have similar sports scenes. Hence, we use the consecutive segment tokens

after and before a token as the forward-looking and backward-looking context of the token, respectively.

In this part, we abuse the notation $\mathcal{V}$ to denote the set containing all segment tokens of the play segments involved in a database of plays. Consider a play $\mathcal{P}$ which involves L play segments. Correspondingly, there is a sequence of L segment tokens which we assume are $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_L$. Then, the m-size window context of a segment token $\mathcal{T}_t$ ($m+1 \leq t \leq L-m$), denoted by $c^m(\mathcal{T}_t)$, corresponds to $< \mathcal{T}_{t-m}, \ldots, \mathcal{T}_{t-1}, \mathcal{T}_{t+1}, \ldots, \mathcal{T}_{t+m} >$, where we say $\mathcal{T}_t$ is a target segment token and each token in $c^m(\mathcal{T}_t)$ a context segment token. Note that a segment token could be a target one and also a context one (with others as the target ones), i.e., a segment token has two types of roles, namely a target segment token and a context segment token. Given a target token $\mathcal{T}$ and a context token $\mathcal{C}$, we say the token-context pair $(\mathcal{T}, \mathcal{C})$ is positive if $\mathcal{C} \in c^m(\mathcal{T})$ and negative otherwise. Considering that a segment token has two types of role, we define for each segment token a vector $\boldsymbol{v}_{\mathcal{T}} \in \mathbb{R}^{d'}$ for cases it corresponds to a target segment token and a vector $\boldsymbol{v}_{\mathcal{C}} \in \mathbb{R}^{d'}$ for cases it corresponds to a context segment token, where d' is the embedding dimension. We aim to learn these vectors such that we could infer those context segment tokens from a target one with the maximum probability.

We explain the training data and also the loss next. The training data consists of a set of training samples. Each training sample consists of one positive token-context pair $(\mathcal{T}, \mathcal{C})$ (i.e., $\mathcal{C} \in c^m(\mathcal{T})$) and k negative pairs $(\mathcal{T}^i, \mathcal{C})$, where $\mathcal{T}^i$ for $i = 1, 2, ..., k$ is drawn from a segment token distribution $Q(\mathcal{T})$. Specifically, $Q(\mathcal{T})$ is a α-smoothed unigram distribution,

$$Q(\mathcal{T}) := \frac{f(\mathcal{T})^\alpha}{\sum_{\mathcal{T}' \in \mathcal{V}} f(\mathcal{T}')^\alpha}, \tag{7.2}$$

where $\alpha \in [0, 1]$ and $f(\mathcal{T})$ is the frequency of the segment token $\mathcal{T}$ appearing in the corpus.

The loss is explained as follows. For a token-context pair $(\mathcal{T}, \mathcal{C})$, we model the probability that the pair is positive as $\sigma\Big((\boldsymbol{v}_{\mathcal{T}})^T \cdot \boldsymbol{v}_{\mathcal{C}}\Big)$ (i.e., a sigmoid function is used with its input as the dot product between the vectors of the segment tokens) and the pair is negative as $1 - \sigma\Big((\boldsymbol{v}_{\mathcal{T}})^T \cdot \boldsymbol{v}_{\mathcal{C}}\Big)$. We then define the loss over one training sample using negative log probabilities as follows.

$$\mathcal{L}(\mathcal{T}, \mathcal{C}, \mathcal{T}^i) := -\log \sigma\Big((\boldsymbol{v}_{\mathcal{T}})^T \cdot \boldsymbol{v}_{\mathcal{C}}\Big) - \sum_{i=1}^{k} \log \sigma\Big(- (\boldsymbol{v}_{\mathcal{T}^i})^T \cdot \boldsymbol{v}_{\mathcal{C}}\Big), \tag{7.3}$$

where $(\mathcal{T}, \mathcal{C})$ is the positive token-context pair and $(\mathcal{T}^i, \mathcal{C})$, for $1 \leq i \leq k$, are the k negative pairs in the training sample. The overall loss function $\mathcal{L}$ is defined by aggregating the losses on all training samples.

The SGNS of the segmentation tokens yields a distributed representation $\boldsymbol{v}_{\mathcal{T}}$ for each $\mathcal{T} \in \boldsymbol{\mathcal{V}}$.

7.3.3 Bottom-up Gluing

In this section, we aim to glue the distributed representations of the segment tokens up together to achieve a comprehensive representation of the play. We propose a new algorithm framework called the Denoising Sequential Encoder-Decoder (DSED). The intuition is that we try to maximize the probability of recovering the most likely real (or clean) tokens from the corrupted initial inputs. For a given play segment and its corresponding token $\mathcal{T}$, we generate a corrupted version of the token, denoted by $\tilde{\mathcal{T}}$, as follows. We scan the locations of the trajectories contained in the play segment time stamp by time stamp, and at each time stamp, we keep the locations with a pre-set probability (and drop the locations with one minus the probability and continue to the next time stamp) and in the case we keep the locations, we sample for each location a noise following a normal distribution $\mathcal{N}(0, 1)$ and add the noise into the location. We then get a new set of tokens based on the corrupted trajectories.

The architecture of the model is presented in Figure 7.2. We define the encoding hidden representation $\boldsymbol{h}_t^{enc}$ at each time step t, i.e. $\boldsymbol{h}_t^{enc} := LSTM_\theta^{enc}(\boldsymbol{v}_{\tilde{\mathcal{T}}_t}, \boldsymbol{h}_{t-1}^{enc})$. The encoding hidden vector at the last time step $\boldsymbol{h}_{last}^{enc}$ denotes the target representation $\boldsymbol{v}$ and is used to be the hidden vector of the decoder at the first step, i.e. $\boldsymbol{h}_0^{dec} := \boldsymbol{h}_{last}^{enc}$. And EOS is the special token that signals the first step input of the decoding. Also, the decoding hidden representation $\boldsymbol{h}_t^{dec}$ is computed based on the distributed representation of the clean token $\boldsymbol{v}_{\mathcal{T}_{t-1}}$ and the hidden vector $\boldsymbol{h}_{t-1}^{dec}$ from the previous time step, i.e., $\boldsymbol{h}_t^{dec} := LSTM_{\theta'}^{dec}(\boldsymbol{v}_{\mathcal{T}_{t-1}}, \boldsymbol{h}_{t-1}^{dec})$. Note that LSTM is chosen as the computational unit in our model since some existing studies show that LSTM outperforms GRU in tasks requiring modeling long-distance relations [126].

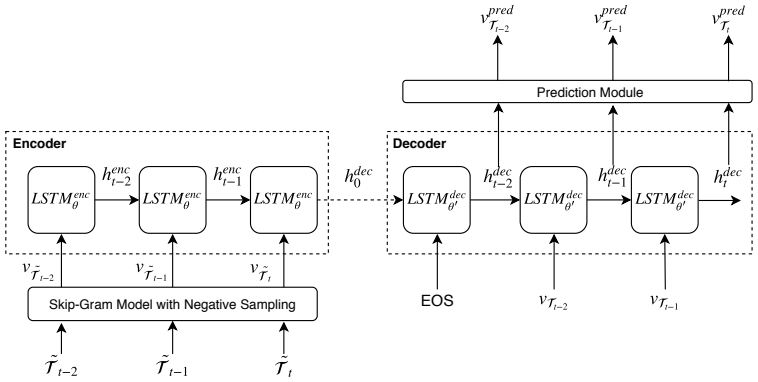


Figure 7.2: Denoising Sequential Encoder-Decoder Model. Take the sequence of the corrupted tokens $< \tilde{\mathcal{T}_{t-2}}, \tilde{\mathcal{T}_{t-1}}, \tilde{\mathcal{T}_t} >$ as an example, where EOS is a special token indicating the end of the input.

Eventually, we predict $\boldsymbol{v}_{\mathcal{T}_t}^{pred}$ by the softmax function from the decoding hidden representation $\boldsymbol{h}_t^{\boldsymbol{dec}}$ at each time step t,

$$\boldsymbol{v}_{\mathcal{T}_t}^{pred} := \frac{exp(\boldsymbol{W}^T \cdot \boldsymbol{h}_t^{\boldsymbol{dec}} + \boldsymbol{b})}{\sum_{\boldsymbol{v} \in \boldsymbol{\mathcal{V}}} exp(\boldsymbol{W}_{\boldsymbol{v}}^T \cdot \boldsymbol{h}_t^{\boldsymbol{dec}} + b_{\boldsymbol{v}})}, \tag{7.4}$$

where $\boldsymbol{W} \in \mathbb{R}^{|h_t^{dec}| \times |\boldsymbol{\mathcal{V}}|}$, $\boldsymbol{b} \in \mathbb{R}^{|\boldsymbol{\mathcal{V}}|}$, $\boldsymbol{W}_{\boldsymbol{v}} \in \mathbb{R}^{|h_t^{dec}|}$ and $b_{\boldsymbol{v}} \in \mathbb{R}$ are the weights and bias represented by η, and softmax is the activation function. We define the loss function $\mathcal{L}(\boldsymbol{v}_{\mathcal{T}}, \boldsymbol{v}_{\mathcal{T}}^{pred})$ as the average sequence cross-entropy,

$$\mathcal{L}(\boldsymbol{v}_{\mathcal{T}}, \boldsymbol{v}_{\mathcal{T}}^{pred}) := \frac{1}{L} \cdot \sum_{i=1}^{L} \mathcal{H}\Big(\boldsymbol{v}_{\mathcal{T}_i}, \boldsymbol{v}_{\mathcal{T}_i}^{pred}\Big), \tag{7.5}$$

where $\mathcal{H}$ is the cross-entropy operator. Parameter θ of the encoding function $\boldsymbol{LSTM}_{\theta}^{enc}$, θ' of the decoding function $\boldsymbol{LSTM}_{\theta'}^{dec}$ and η are trained to minimize loss $\mathcal{L}(\boldsymbol{v}_{\mathcal{T}}, \boldsymbol{v}_{\mathcal{T}}^{pred})$ over a training set with the Adam stochastic gradient descent method.

7.3.4 Complexity Analysis

The time complexity for computing the similarity between two plays consists of two parts, namely one for learning the representations of the plays and the other for computing the distance between the learned representations in the form of vectors in the d-dimensional space. The former costs $O(n)$ time, where n is the length of the longest trajectory involved in the plays, and we explain as follows: (1) it takes $O(n)$ to convert a query play to a

sequence of L segment matrices; (2) it takes $O(|\mathcal{V}|L)$ time to map the segment matrices to their corresponding segment tokens; (3) it takes $O(L)$ to map the segment tokens to their corresponding vectors; and (4) it takes roughly $O(n)$ time to fed the vectors to the DSED model and obtains the target vector representation of the play. Since $|\mathcal{V}|$ could be regarded as a constant and L is bounded by n, we know this process takes $O(n)$ time. And the latter costs $O(d)$ which is obvious. Hence, the overall time complexity is $O(n + d)$.

7.4 Experiments

7.4.1 Experimental Setup

Dataset. Our experiments are conducted on real-world soccer player tracking data [2]. The data consist of 7500 sequences and each sequence contains a segment of tracking data corresponding to actual game from a recent professional soccer league, totaling approximately 45 games worth of playing time and over 30 million data points, with redundant and "dead" situations removed. Each segment consists of the tracking data of three parts: 11 defense players, 11 attacking players and a ball. Each part contains (x, y) coordinates obtained at a sampling frequency of 10Hz. More specifically, the coordinates generally belong to the $[-52.5meters, +52.5meters]$ range along the x-axis, and $[-34meters, +34meters]$ range along the y-axis, with the very center of the pitch being $(0, 0)$.

Baselines. To evaluate the effectiveness and efficiency of our deep representation learning method (called play2vec), we compare it with the baseline similarity methods including DTW [38], Frechet [65], Chalkboard [41] and EMDT [82]. The detailed description can be found in [42].

Parameter Setting. The default size of cells is 3 meters and short duration of segments is 1 second in the experiments. After building a sports corpus via the Jaccard index (the threshold is 0.3), we got 50,465 unique tokens for our dataset. We use a 2-layer LSTM as the computational unit in LSTM-Encoder. The representation dimension of the learned segment tokens and plays are set to 20 and 50 respectively. The context window size in

[2]Data Source: STATS, copyright 2019 (https://www.stats.com/artificial-intelligence)

Table 7.1: Self-similarity (varying noise).

noise rate	0.2	0.3	0.4	0.5	0.6
DTW	24.80	33.80	44.00	73.20	90.20
Frechet	79.40	80.60	82.80	83.00	83.60
Chalkboard	77.20	77.80	78.20	78.40	78.80
EMDT	215.00	220.80	236.20	255.20	299.40
play2vec	**14.20**	**20.40**	**23.80**	**25.30**	**28.20**

Table 7.2: Self-similarity (varying drop).

dropping rate	0.2	0.3	0.4	0.5	0.6
DTW	115.60	118.20	124.80	129.00	130.80
Frechet	144.60	155.20	176.80	185.60	202.00
Chalkboard	57.80	60.40	62.40	68.20	69.60
EMDT	56.20	70.40	70.60	91.40	96.40
play2vec	**26.24**	**29.50**	**39.74**	**50.20**	**54.00**

Table 7.3: Cross-similarity (varying noise).

noise rate	0.2	0.3	0.4	0.5	0.6
DTW	0.093	0.111	0.113	0.114	0.117
Frechet	0.084	0.101	0.102	0.105	0.116
Chalkboard	0.074	0.082	0.088	**0.093**	0.112
EMDT	0.583	0.629	0.706	0.819	0.891
play2vec	**0.034**	**0.048**	**0.086**	**0.093**	**0.111**

Table 7.4: Cross-similarity (varying drop).

dropping rate	0.2	0.3	0.4	0.5	0.6
DTW	0.198	0.290	0.397	0.500	0.588
Frechet	0.251	0.253	0.254	0.256	0.257
Chalkboard	0.197	0.299	0.397	0.490	0.600
EMDT	0.295	0.302	0.303	0.304	0.322
play2vec	**0.002**	**0.008**	**0.009**	**0.017**	**0.032**

the distributed representation learning is set to 5. The α-smoother is set to 3/4 following the negative sampling in word2vec. Additionally, in the training process, we train our model on 5k generated plays with randomly noise and dropping rate and adopt Adam stochastic gradient descent with an initial learning rate of 0.01. In order to avoid the gradient vanishing problem, a maximum gradient norm constraint is used and set to 5. For the parameters of baselines, we follow their strategies described in the original papers.

Evaluation Platform. All the methods are implemented in Python 3.6. The implementation of play2vec is based on tensorflow 1.8.0. The experiments are conducted on a machine with Intel(R) Xeon(R) CPU E5-1620 v2 @3.70GHz 16.0GB RAM and one Nvidia GeForce GTX 1070 GPU.

7.4.2 Experimental Results

Overall effectiveness. We first study the effectiveness of play2vec. The lack of ground-truth makes it a challenging problem to evaluate the accuracy. To overcome it, we follow three recent studies [36, 37, 127] which propose to quantify the accuracy of trajectory similarity with Self-similarity, Cross-similarity and KNN-similarity comparisons, respectively. There are two frequently used parameters: noise rate (radius is set to 1 meter.) and dropping rate with varying values from 0.2 to 0.6. The two parameters are to measure the probabilities of adding noise or dropping sampling points of each trajectory in a play, respectively.

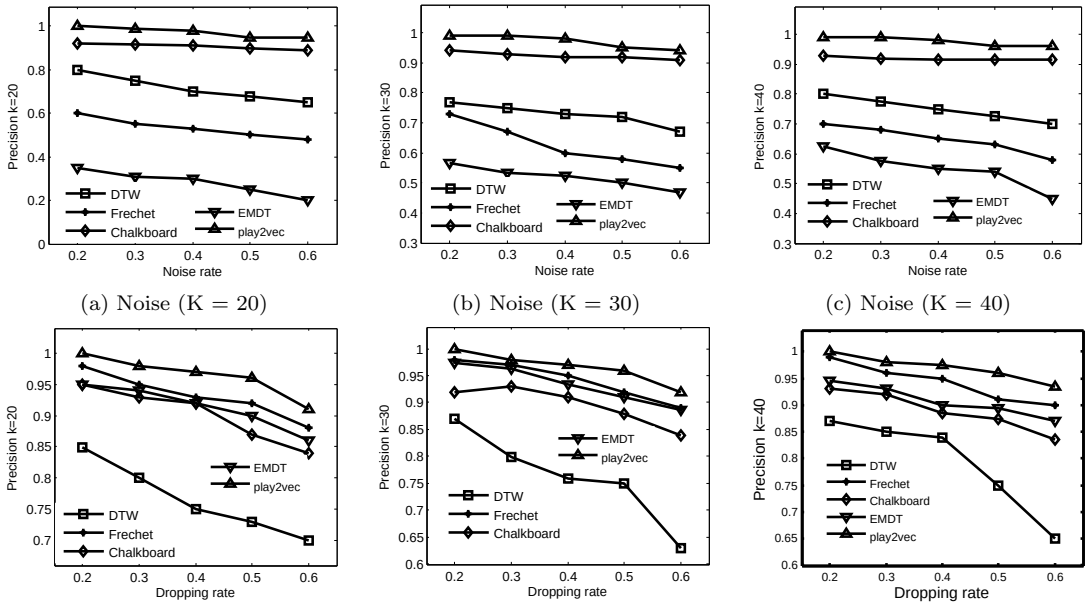
(1) Effectiveness evaluation (self-similarity comparison) In this experiment, we randomly choose 50 plays to form the query set (denoted as Q) and 1000 plays as the target database (denoted as D) from the dataset. For each play $P \in Q$, we create two sub-plays by randomly sampling 20% points for each trajectory in the play, denoted as P_a and P_b, and we use them to construct two datasets $Q_a = \{P_a\}$ and $Q_b = \{P_b\}$. Similarly, we get D_a and D_b from the target database D. Then for each query $P_a \in Q_a$, we compute the rank of P_b in the database $Q_b \cup D_b$ using different methods. Ideally, P_b should be ranked at the top since P_a and P_b are generated from the same play P. To evaluate the robustness of different approaches to noise, we consider introducing two types of noises. First, we corrupt each trajectory of each play in both Q_a and $Q_b \cup D_b$ as follows: We randomly sample a fraction of the points (denoted by noise rate r_1) and for each sample point we distort the coordinate values by adding Gaussian noises with a standard normal distribution. We vary r_1 from 0.2 to 0.6 and report the mean rank of the queries in Table 7.4.2. Second, we randomly drop a fraction of points from each trajectory of each play in both Q_a and $Q_b \cup D_b$. We vary dropping rate r_2 from 0.2 to 0.6 and report the mean rank of the queries in Table 7.4.2. Note that the mean rank in self-similarity evaluation is a rank-based metric defined as $\frac{1}{|Q_a|}\sum_{P_a} rank(P_b)$, where $rank(P_b)$ denotes the rank of P_b in $Q_b \cup D_b$ for a query $P_a \in Q_a$. We observe that play2vec consistently outperforms the other methods by a large margin as we vary the two types of noise. We also observe that most of the methods are not very sensitive to the noise rate except that DTW and EMDT degrade quickly when we increase the noise rate to 0.5. This is because the matching cost of DTW is determined by the pairwise point-matching and errors will be accumulated with noises. However, Frechet maintains an infimum of the matching cost that is robust to noise changes. With regard to Chalkboard, it splits trajectories into overlapping segments, which can alleviate the noise interruption to some degree. Moreover, with the increase of the noise rate, the gap between EMDT and other methods grows and the mean rank of EMDT is consistently very large. This is because after adding noise, the cell centers corresponding to P_a and P_b are very different, even if they are sampled from the same play. When we vary the dropping rate, the mean ranks of DTW and Frechet are significantly larger than other methods. This implies that the pairwise point-matching methods based on agent-to-agent alignment may not be able to handle the case when sampling rate is low.

(2) Effectiveness evaluation (cross-similarity comparison) A good similarity measure should preserve the distance between two plays regardless of the sampling rate or noise interference. We use a metric from the literatures [37, 127] to evaluate the effectiveness for Cross-similarity comparison, namely Cross Distance Deviation (CDD) as defined below.

$$CDD(P_a, P_b) = \frac{|S(P_{a'}(r), P_{b'}(r)) - S(P_a, P_b)|}{S(P_a, P_b)}, \tag{7.6}$$

where $S(\cdot, \cdot)$ is a similarity measure such as DTW or Frechet; P_a and P_b are two original plays; $P_{a'}(r)$ and $P_{b'}(r)$ are their variants that are obtained by randomly dropping points (or adding noise) with rate r. A small CDD value indicates that an algorithm is robust and is able to preserve the original distance well. In this experiment, we randomly select 1,000 play pairs (P_a, P_b) from the dataset. The average CDD results are reported in Table 7.4.2 and Table 7.4.2. We observe that play2vec outperforms other baselines consistently for different noise and dropping rates. Note that play2vec is very close to the ground truth over various dropping rates because a cell of the grid maps is considered occupied only if one sample point falls in the cell. Therefore, play2vec can effectively handle the low sampling issue of data.

(3) Effectiveness evaluation (KNN-similarity comparison) In this experiment, we study the accuracy and robustness of play2vec and the other baselines for KNN-similarity search on plays when we vary the dropping rate or noise rate. To circumvent the issue of lack of ground-truth, we follow the experimental methodology that is proposed by existing studies [36, 37]: We first randomly select 20 plays as the query set and 500 plays as the target database, and for each query we employ each method to find its Top-K plays as the ground-truth of each method; Then we corrupt each play in the target database by randomly dropping points or adding noise, and retrieve the Top-K plays from the corrupted database; Finally, we compare the retrieved Top-K plays against the ground-truth to compute the precision, i.e., the proportion of true Top-K plays among the retrieved Top-K plays. We vary the value of K by 20, 30, 40, and vary the dropping rate or noise rate from 0.2 to 0.6. The average precision results are reported in Figure 7.3. We observe that play2vec performs the best consistently. As expected, the precision of all methods decreases when the noise or dropping rate increases. We observe similar trends for all methods over different K values. The precision of DTW drops rapidly when the



(a) Noise (K = 20) (b) Noise (K = 30) (c) Noise (K = 40)

(d) Dropping (K = 20) (e) Dropping (K = 30) (e) Dropping (K = 40)

Figure 7.3: KNN results when varying the noise and dropping rate from 0.2 to 0.6 for K = 20, 30, 40.

dropping rate is more than 0.5. EMDT performs the worst when we inject noise into the target database due to the same reason we analyzed for the Self-similarity comparison experiment. Additionally, Frechet is more robust than other baselines when we corrupt the target database by dropping points.

(5) Efficiency Evaluation This set of experiments is to evaluate the efficiency of different methods for the sports play retrieval. Figure 7.4 shows the average cost of computing the similarity between a query play and the plays when we vary the size of the target database. Note that the y-axis is in logarithmic scale. Clearly, EMDT performs extremely slow because the overall time cost of the Earth Mover's Distance is super-cubic to the number of the cells over a soccer pitch [128]. DTW and Frechet have similar running time because they are pairwise point-matching methods, which has quadratic computational complexity. Chalkboard is the most efficient algorithm among the baselines because it adopts a fast role-based representation to avoid exhaustively comparing computation in agent-to-agent alignment. Note that in Chalkboard, we perform offline preprocessing and do not count the time for fair comparison. play2vec performs the best among all methods and is over an order of magnitude faster than the most efficient baseline Chalkboard. This is because play2vec takes linear time to retrieve similar plays as

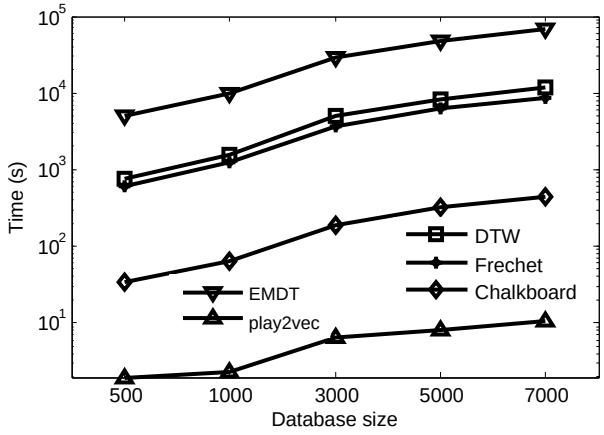


Figure 7.4: Efficiency.

discussed in Section 7.3.4. We also notice that play2vec scales linearly with the database size and the disparity between them increases as the size of the target database grows.

More experiments. More experimental results regarding (5) parameter study and (6) user study can be found in [42].

7.5 Summary

In this chapter, we present the first deep learning based method for computing the similarity between two sports plays. Inspired by the success of modeling word similarity, we extend the Skip-Gram with Negative Sampling (SGNS) model and develop a new Denoising Sequential Encoder-Decoder (DSED) framework to learn consistent representations. Our solution is robust to the non-uniform sampling rates and noises and only takes a linear time to compute the similarity. Also, we evaluate experiments on real-world soccer match data and find that it achieves better effectiveness and efficiency than the existing methods. In the future, we plan to develop indexing techniques like Locality-Sensitive Hashing to further accelerate the retrieval for large-scale datasets.

Chapter 8

Deep Metric Learning for Similar Sports Play Search

8.1 Overview

In this chapter, we study a new problem of searching similar plays for a given query play from a database of games (called SimPlay) [1]. We present the problem definition in Section 8.2. In Section 8.3, we first present algorithms for the SimPlay problem in Section 8.3.1. and then develop a deep metric learning based method for deciding the order of searching the games in a database in Section 8.3.2. We report our experimental results in Section 8.4, and conclude this chapter in Section 8.5.

8.2 Problem Statement

Given a game $\mathcal{P} =< p_1, p_2, ..., p_n >$ and $1 \leq i \leq j \leq n$, let $\mathcal{P}[i, j]$ denote the portion of $\mathcal{P}$ that starts from the i^{th} frame and ends at the j^{th} frame, i.e., $\mathcal{P}[i, j] =< p_i, p_{i+1}, ..., p_j >$, where each frame p_i contains multiple sampled locations of the players and the ball at that time stamp. We say that $\mathcal{P}[i, j]$ is a *play* of $\mathcal{P}$ for any $1 \leq i \leq j \leq n$. There are in total $\frac{n(n+1)}{2}$ plays in $\mathcal{P}$. Note that any play of a game $\mathcal{P}$ corresponds to a (shorter) game itself.

Problem definition. Suppose we have a database of many games. The problem is to search the portion of a game (i.e., a play) in the database, which is the most similar to

[1] This chapter was published as Similar Sports Play Retrieval with Deep Reinforcement Learning [7].

Table 8.1: Time complexities of SimPlay within a game.

Algorithms	**Time complexity**
ExactS	$O(n^2)$
SizeS	$O((m+\xi)\cdot n)$
PSS, POS, POS-D (Splitting-based Algorithms)	$O(n)$
RLS, RLS-Skip (Learning-based Algorithms)	$O(n)$

a query play. We call this problem SimPlay. The SimPlay problem relies on a similarity measurement for two plays, where we adopt the play2vec measurement.

We note that a more general query is to find the *top-k* similar plays to a query play, which reduces to the user's query as described above when $k = 1$. In this chapter, we stick to the setting of $k = 1$ since extending the techniques for the setting of $k = 1$ to general settings of k is straightforward. Specifically, we can adapt the techniques in this chapter to general settings of k by simply maintaining the k^{th} most similar play and updating it when a play that is more similar than it is found.

8.3 Proposed Method

8.3.1 Similar Play Search Within a Game

In this part, we introduce a suite of algorithms for the process of searching for the play of a game, which is the most similar to a query play. These algorithms were originally proposed for searching similar sub-trajectories in a single trajectory in Chapter 6. We adapt these algorithms for searching similar plays, which comprise multiple trajectories, where play2vec is used for measuring the similarity between plays.

They include (1) one exact algorithm called ExactS (Section 6.3.1.1), (2) one approximate algorithm, called SizeS, with a tunable parameter to control the trade-off between effectiveness and efficiency (Section 6.3.1.2), (3) five splitting-based algorithms including three based on heuristics (Section 6.3.1.3) and two on reinforcement learning (Section 6.3.2.3 and Section 6.3.2.4). The summary of the complexities of these algorithms is presented in Table 8.1.

8.3.2 Similar Play Search Within a Database of Games

An straightforward method for the SimPlay problem is to search the most similar play from *each* game to the query play (using a method introduced in Section 8.3.1) and then return the most similar play among the found plays. However, this method would be costly and does not meet the efficiency requirement in practice since there are usually many games in the database. For better efficiency, we present a method called *score-based search* (ScoreSearch) for the SimPlay problem. ScoreSearch computes a score for each game, which corresponds to an estimate of the maximum similarity between a play of the game and the query play, and then searches for the similar play from only a fraction r of the games with the highest scores, where r is a user parameter for controlling the trade-off between effectiveness and efficiency.

Overview of ScoreSearch. We propose to define a score for each game to reflect the *maximum* similarity between a play in the game and the query play. The intuition is that based on the scores, we can focus on a fraction of games with the highest scores only for the SimPlay problem since the most similar play is likely to be from one of these games and other games can be ignored. One possible way to compute the score of a game is to search for the most similar play of the game to the query play (by using one of the method in Section 8.3.1) and then return the corresponding similarity as the score of the game. However, this solves the problem more than necessary and would reduce to the straightforward method of searching a similar play within *each* game. A better idea is to estimate the score of a game with some light method only so that if the score is low, the game would then be pruned for searching the similar play. We propose a triplet network [48] based model, which embeds games into vectors and then uses the Euclidean distance between the vector of a game and that of the query play as the score of the game for the query play. To train the network, we use the maximum similarity between a play of a game and a query play for constructing positive and negative labels so that the learned scores capture well the maximum similarities between plays of games and query plays. Once the network is trained, given a query play, we first compute the scores of all games in the database for the query play by feeding them into the network. We can then focus on only a fraction r of games with the highest scores for searching for the most similar play to a query play.

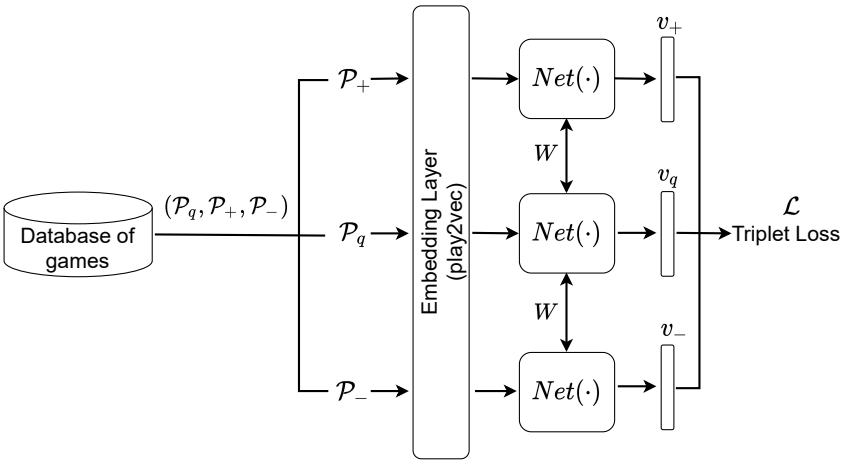


Figure 8.1: Overview of model architecture.

Algorithm 7: The ScoreSearch Algorithm

Input: Embedded vectors V of all games with the size N via preprocessing and a query play $\mathcal{P}_q$;
Output: The most similar play;
Embed the query play, whose embedding is denoted by $v_{\mathcal{P}_q}$;
Retrieve the $r \cdot N$ nearest vectors of games to $\mathcal{P}_q$ via a kNN algorithm;
forall *retrieved games* **do**
    find the play in the game, which is the most similar to the query play (e.g., with RLS-Skip);
end
return the most similar play to the query play among all found plays;

Model Architecture. Figure 8.1 illustrates the model architecture. To train the triplet network, we randomly sample triplets (i.e., an anchor sample as the query, a positive sample, and a negative sample) from the dataset. For each triplet, we use the shortest sample in the triplet as the anchor and denote it by $\mathcal{P}_q$. Among the two other samples, we choose the one which has a larger maximum similarity between one of its plays and $\mathcal{P}_q$ as the positive sample $\mathcal{P}_+$, and the other one as the negative sample $\mathcal{P}_-$. We then embed the samples to the play2vec model, and feed the corresponding embeddings to a triplet network. The triplet network is with three shared feedforward neural networks, denoted as $Net(\cdot)$, which computes two Euclidean distances. That is,

$$d_+ = ||Net(\mathcal{P}_q) - Net(\mathcal{P}_+)||_2 \tag{8.1}$$

$$d_- = ||Net(\mathcal{P}_q) - Net(\mathcal{P}_-)||_2 \tag{8.2}$$

where $Net(\mathcal{P}_q)$ (resp. $Net(\mathcal{P}_+)$ and $Net(\mathcal{P}_-)$) denotes the embedding vector of $\mathcal{P}_q$ (resp. $\mathcal{P}_+$ and $\mathcal{P}_-$) via the network $Net(\cdot)$. The network is trained via optimizing the triplet loss shown below:

$$\mathcal{L}(\mathcal{P}_q, \mathcal{P}_+, \mathcal{P}_-) = \max\{d_+ - d_- + \delta, 0\} \tag{8.3}$$

where δ denotes a desired margin between the positive and negative distances in the embedding space, which aims to enlarge the gap between positive sample $\mathcal{P}_+$ and negative sample $\mathcal{P}_-$. Empirically, we found the light architecture is easy to train, and achieves a superior performance.

The procedure of ScoreSearch algorithm is presented in Algorithm 7. Suppose all games have been fed to the network and their vectors have been computed as a preprocessing process. For a given query play, the ScoreSearch algorithm would (1) embed the query play (Line 1), (2) retrieve a fraction r of games using a nearest neighbor algorithm [129] (Line 2), and then (3) search the most similar play from each of the retrieved games using one of the algorithms presented in Section 8.3.1 (Lines 3-6).

Time Complexity Analysis. As described above, ScoreSearch involves three parts, whose time complexities are as follows. The part of (1) is $O(m)$, where m is the length of the query play; that of (2) is $\alpha(N)$, where $\alpha(N)$ is the time complexity of a k-nearest neighbor (kNN) search algorithm on a set of N items (e.g., $\alpha(N)$ could be $k \cdot \log(N)$ if the algorithm in [130] is adopted); and that of (3) is $O(r \cdot N \cdot \beta(n))$, where $\beta(n)$ is the time complexity of searching the most similar play of a game of length n to the query play (e.g., if the RLS-Skip algorithm is adopted, $\beta(n)$ is $O(n)$). In conclusion, the time complexity of ScoreSearch is $O(m + \alpha(N) + r \cdot N \cdot \beta(n))$.

8.4 Experiments

8.4.1 Experimental Setup

Dataset. Our experiments are conducted on real-world soccer player tracking data. Detailed description of these measurements can be found in Section 7.4.1.

Algorithms. For the play similarity measurement, we study our proposed play2vec as well as the following four measurements, namely DTW [38], Frechet [65], Chalkboard [41], and EMDT [82].

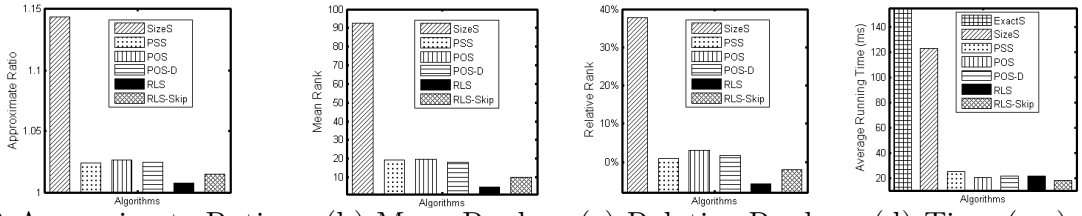
For the problem of searching for the most similar play to a query play within a game, since it is a new problem and no existing algorithms can be used, we evaluate the algorithms proposed in this chapter, namely ExactS, SizeS, PSS, POS, POS-D, RLS, and RLS-Skip. Note that we did not consider the method proposed in the work [41], which materializes all plays of lengths from 1s to 5s, for two reasons: (1) the range from 1s to 5s is ad-hoc and may not serve well longer query plays and there is no clear clue about how to set the range; and (2) the SizeS algorithm could be regarded as a similar method to this one, which considers all those plays with similar lengths to the query play's, i.e., the range is adaptive.

Parameter Setting. The parameter setting of play2vec is shown in Section 7.4.1. For SizeS and POS-D, we use the setting $\xi = 5$ and $D = 5$ by default, respectively. We will study the effects of these two parameters. For training the RL-based models (i.e., RLS and RLS-Skip), we use a feedforward neural network with 2 layers. In the first layer, we use the tanh function with 25 neurons, and in the second layer, we use the softmax function with $2 + k$ neurons as the output, corresponding to different actions, where for RLS, we use the setting $k = 0$ and for RLS-Skip, we use the setting $k = 3$ by default. We will study the effect of the skipping parameter k. In the training process, the size of replay memory for the DQN method is set at 2000. We train our model on 25k pairs of plays chosen randomly, using the Adam stochastic gradient descent with an initial learning rate of 0.001. The parameter ϵ is set at 0.1 with decay 0.99 for the ϵ-greedy strategy in the DQN method, and the reward discount rate γ is set as 0.99.

Evaluation Platform. All the methods are implemented in Python 3.6. The implementation of play2vec and RLS are based on tensorflow 1.8.0. The experiments are conducted on a machine with Intel(R) Xeon(R) CPU E5-1620 v2 @3.70GHz 16.0GB RAM and one Nvidia GeForce GTX 1070 GPU.

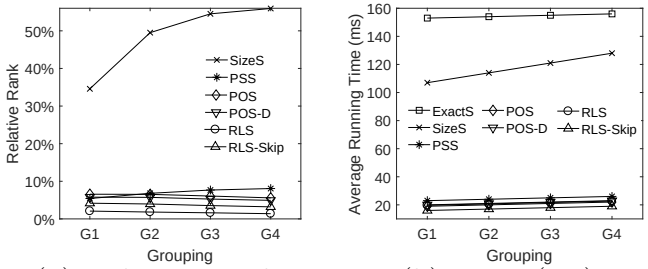
8.4.2 Evaluation of Searching Similar Plays Within a Game

We randomly sample 10,000 pairs of sports games/plays from the dataset, and for each pair, we use the short one as the query play to search the most similar play from the other one. We report the average results under each setting.



(a) Approximate Ratio (b) Mean Rank (c) Relative Rank (d) Time (ms)

Figure 8.2: Results of AR, MR and RR (a)-(c) and running time (d).

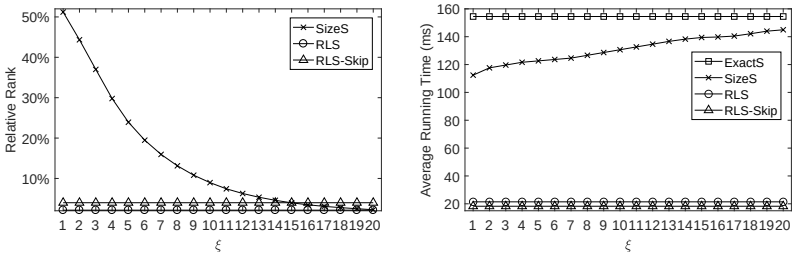


(a) Relative Rank (b) Time (ms)

Figure 8.3: Results of RR and running time for varying query play lengths.

Evaluation Metrics To evaluate the effectiveness of an approximate algorithm for the SimPlay problem, we adopt three metrics in the recent study [47]. (1) Approximate Ratio (AR): It is defined as the ratio between the dissimilarity of the play returned by an approximate algorithm and that of the play returned by an exact algorithm. (2) Mean Rank (MR): We sort all the plays of a game in an ascending order of their dissimilarities wrt a query. MR is defined as the rank of the solution returned by an approximate algorithm. (3) Relative Rank (RR): RR is a normalized version of MR by the total number plays of a game. Note that for all measurements, a smaller value indicates a better algorithm.

Effectiveness Evaluation Figure 8.2 (a)-(c) show the effectiveness results. We observe that learning-based algorithms (i.e., RLS and RLS-Skip) consistently outperform all other non-learning based approximate algorithms in terms of all three metrics. For example, RLS outperforms PSS, the best non-learning based algorithm, by 75% (resp. 60%) in terms of Mean Rank (resp. Relative Rank). As expected, POS-D slightly outperforms POS because it checks more plays for choosing split points during the search process. Additionally, we observe that the effectiveness of SizeS is much worse, probably because the length of the most similar play may not have similar length as the query play. RLS-



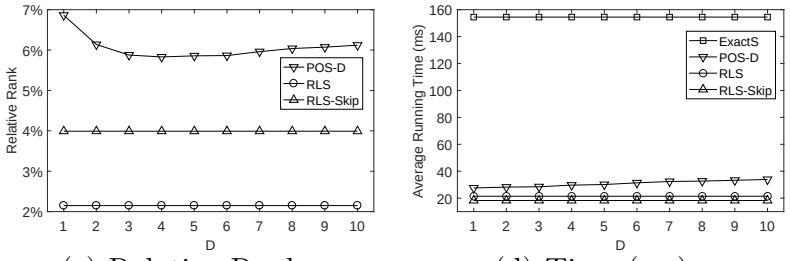
(a) Relative Rank (b) Time (ms)

Figure 8.4: The effect of soft margin ξ for SizeS.

Skip is a bit less effective than RLS, but still better than those non-learning based algorithms owing to the benefit of its learned policy for decision making.

Furthermore, we partition the query plays into four groups, namely G1, G2, G3, and G4, each with 10,000 plays, such that the lengths of the plays in a group are as follows: G1 = [100, 150), G2 = [150, 200), G3 = [200, 250) and G4 = [250, 300). Then, we randomly select 10,000 games and use them as a database for each group. We report the average results for each group in terms of RR in Figure 8.3 (a).

The results of AR and MR provide similar clues and thus are omitted due to the page limit. Overall, these approximate algorithms remain stable except for SizeS. We observe that SizeS gets worse as the query length increases. This is because for a longer query play, a larger range may be required to retain the effectiveness.



(c) Relative Rank (d) Time (ms)

Figure 8.5: The effect of delay D for POS-D.

Efficiency Evaluation The results of running time are shown in Figure 8.2 (d). We observe that RLS-Skip runs the fastest because it skips some frames and the cost of maintaining the states and making decisions for these frames is saved. ExactS has the largest running time. For example, it is usually around 7-9 times slower than PSS, POS, POS-D, RLS and RLS-Skip. SizeS runs faster than ExactS since it explores fewer plays, and PSS is slightly slower than POS, POS-D, and RLS. This is because PSS needs to

compute the similarities of suffixes to make the splitting decision while the other three only compute the similarities of prefixes. We show the results of running time when varying the length of query plays in Figure 8.3 (b). We notice that the average running time of all the algorithms except SizeS is almost not affected by the query length. This is because the time complexity of computing a similarity is constant once the vector of the query is learned (we did not count the time for learning the representation of the query play since it is shared by all algorithms). For SizeS, the time grows as the query length increases because the candidate plays that are explored are longer and it costs more time learn their representations for measuring their similarities. Overall, the efficiency results are consistent with their time complexities as shown in Table 8.1.

Parameter Study We evaluate the effect of the soft margin ξ for SizeS. Figure 8.4 shows the results of RR and runing time. The results of AR and MR provide similar clues and thus are omitted due to the page limit. As expected, the effectiveness of SizeS becomes better (i.e., it approaches and exceeds RLS and RLS-Skip gradually) when ξ becomes larger. However, its running time increases and approaches that of the ExactS. This is because a larger setting of ξ indicates a larger search space, which approaches the search space of the ExactS as ξ grows.

In Figure 8.5, we study the effects of the parameter D for POS-D. We vary the D from 1 to 10. Overall, the D would not greatly affect the POS-D on both effectiveness and efficiency. As D increases, the effectiveness improves at the beginning because it checks more frames to perform a split operation; however, the effectiveness would drop slightly with a larger setting of D since some prefixes that would have been formed and considered if D is 0 or small would be destroyed. The efficiency drops slightly as D increases since it checks more plays during the search. We thus adopt the setting $D = 5$ as the default one since it provides a good trade-off between effectiveness and efficiency.

We further study the effects of skipping steps k for RLS-Skip in Table 8.2. As expected, the general trend is that the effectiveness drops but its efficiency improves as k increases since with a larger k, it tends to skip more frames, which may exclude more plays from being considered and save some computation costs. In addition, we report the statistics of the portion of skipped frames. Note that RLS-Skip with $k = 0$ corresponds

Table 8.2: The effect of skipping steps k for RLS-Skip.

Metrics	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
AR	1.007	1.010	1.012	1.015	1.027	1.034
MR	4.794	6.554	9.299	10.163	23.159	33.959
RR	2.1%	2.9%	3.5%	4.0%	6.9%	10.4%
Time (ms)	21.407	20.236	19.364	18.252	15.181	12.377
Skipped Frs	0%	0.5%	1.0%	1.4%	6.5%	9.7%

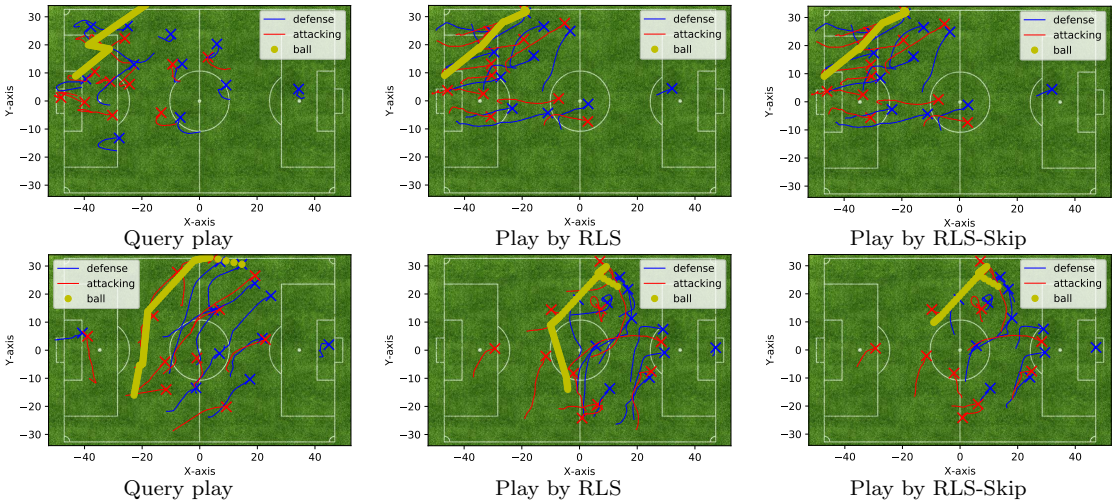


Figure 8.6: Case study on similar play retrieval.

to RLS. We adopt the setting $k = 3$ as the default one since it provides a reasonable trade-off between effectiveness and efficiency.

Case Study We illustrate a case study to show the most similar play returned by RLS and RLS-Skip for two random query plays in Figure 8.6, where the blue, red and yellow lines denote the trajectories of defense players, attacking players and ball in the play, and the small "x" is the end point of the movement. In general, we observe the returned plays by RLS and RLS-Skip match the query plays very well, which is due to their data-driven nature to find a similar play.

8.4.3 Evaluation of Searching Similar Plays Within a Database

Compared Methods We compare the ScoreSearch with two baseline methods. The ScoreSearch algorithm uses the RLS-Skip algorithm for finding the most similar play of a game to a query play.

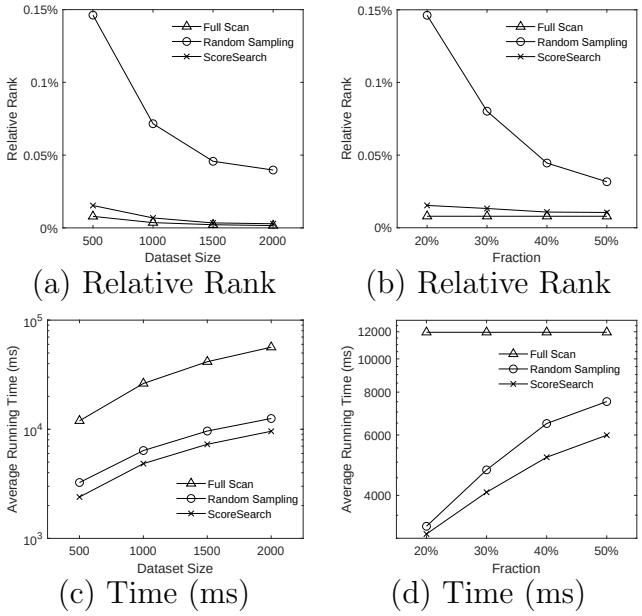
(1) Full Scan. It finds the most similar play of each game in the database to a query play using RLS-Skip and then returns the one with the greatest similarity to the query play among the found plays.

(2) Random Sampling. It randomly samples a fraction of $r \cdot N$ games from the database, finds the most similar play of each sampled game to a query play using RLS-Skip, and then returns the one with the greatest similarity to the query play among the found plays.

Parameter Settings and Evaluation Metrics For training the triplet network in ScoreSearch, we randomly sample 30k triplets from the dataset and set $\delta = 1.0$ in the triplet loss based on empirical findings. To evaluate the effectiveness of ScoreSearch, we reuse the evaluation metrics of AR, MR and RR, which are calculated based on the results returned by Full Scan.

Effectiveness Evaluation We study the ScoreSearch algorithm for searching similar plays in a database. We vary two parameters, i.e., dataset size and the fraction parameter r. We report the average effectiveness results in Figure 8.7 (a)-(b). We observe the effectiveness improves as the fraction parameter r increases. This is because with a larger r, more games that contain similar plays would be considered for searching for the similar play. In addition, the effectiveness improves slightly as the dataset size increases. This is because with a larger N, more games would be considered for searching for the similar play. As expected, ScoreSearch is much better than the Random Sampling since it filters games guided by a learning model instead of a simple random process. The results provided by AR and MR show similar trends and thus they are omitted.

Efficiency Evaluation The running times of Full Scan, Random Sampling and ScoreSearch are presented in Figure 8.7 (c)-(d). As expected, the running time increases with a larger fraction parameter r since more games are selected from the database for conducting the search. In addition, we observe that ScoreSearch is faster than the Random Sampling. This could be explained by the fact that ScoreSearch selects shorter games compared with Random Sampling and the process of searching for a similar play from a shorter game takes less time than from a longer game.



(a) Relative Rank (b) Relative Rank

(c) Time (ms) (d) Time (ms)

Figure 8.7: Results of RR (a)-(b) and running time (c)-(d) for varying dataset size and fraction.

8.5 Summary

In this chapter, we study the similar play retrieval problem. This problem is to find a fragment of a game in a database, which is the most similar to a query play. We make the following technical contributions. We develop a suite of algorithms for the problem of searching for the most similar play to a query play within a game. Third, we develop a novel algorithm based on deep metric learning, ScoreSearch, for searching for the most similar play to a query play within a database. One interesting future research direction could be to conduct other analytic tasks such as play clustering based on the proposed play2vec similarity measurement.

Part III

Socioeconomic Status Inference from Trajectory Data

Chapter 9

On Inferring User Socioeconomic Status with Mobility Records

9.1 Overview

In this chapter, we give the preliminaries in Section 9.2 and present the data in Section 9.3. We present our `DeepSEI` model in Section 9.4, and report our experimental results in Section 9.5. We conclude this chapter in Section 9.6.

9.2 Preliminaries

One of the major goals of this chapter is to infer users' socioeconomic status with their mobility records. The hypothesis is that users' mobility records reflect users' economic statuses, which has been observed in the literature. Specifically, this chapter will explore the solution with deep neural networks, which take the features extracted in terms of three aspects as inputs, namely spatiality, temporality and activity, and output a tag indicating users' socioeconomic classes (e.g., a binary tag indicating two socioeconomic classes). In this chapter, we use the house price data collected from Lianjia [131] to define users' socioeconomic labels (the details will be presented in Section 9.3.3). With the labels, we train our model in a supervised manner, and use the trained model for predicting other data, whose label is unknown. In addition, we collect the learned embeddings from the model. We expect that the embeddings have incorporated economic context well, and we use them to explore more economically related tasks, such as clustering.

Table 9.1: Dataset statistics III.

Statistics	Geolife
# of trajectories	17,621
Total # of points	24,876,978
Ave. # of points per trajectory	1,412
Sampling rate	1s ~ 5s
Average distance	9.96m

9.3 Datasets

The dataset used in this work contains three parts, the first part is the mobility data (i.e., GPS trajectories) generated by users. The second part is POI contexts, which are used to capture users' activity features. The third part is house price considered as the economic information, which is used to construct labels to reflect the socioeconomic status of users for evaluation. We collect the last two parts of data through web map services and web crawlers.

9.3.1 Mobility Data

The mobility records correspond to a sequence of time-stamped locations sampled by the GPS device. For each record, it captures the location (i.e., latitude and longitude) of a user at a timestamp. We use the Geolife dataset [63] for the mobility data. It collects the trajectories of 182 users in a period of five years, where the trajectory data is distributed in over 30 cities of China, and the majority part was generated in Beijing. The detailed statistic of the dataset is summarized in Table 9.1.

9.3.2 POI Data

We collect Point-of-Interest (POI) data through Amap Map API [132], which contains 156,653 records in Beijing. For each POI record, it provides the following attributes, including venue name, category, location with latitude and longitude information. We consider 11 major POI categories in this work, namely working, residence, food and drink, attractions, community, shopping, education, hospitals, lodging, traffic, and recreation. We collect the POI data between 2011 and 2012, corresponding to the period of the mobility records on Geolife.

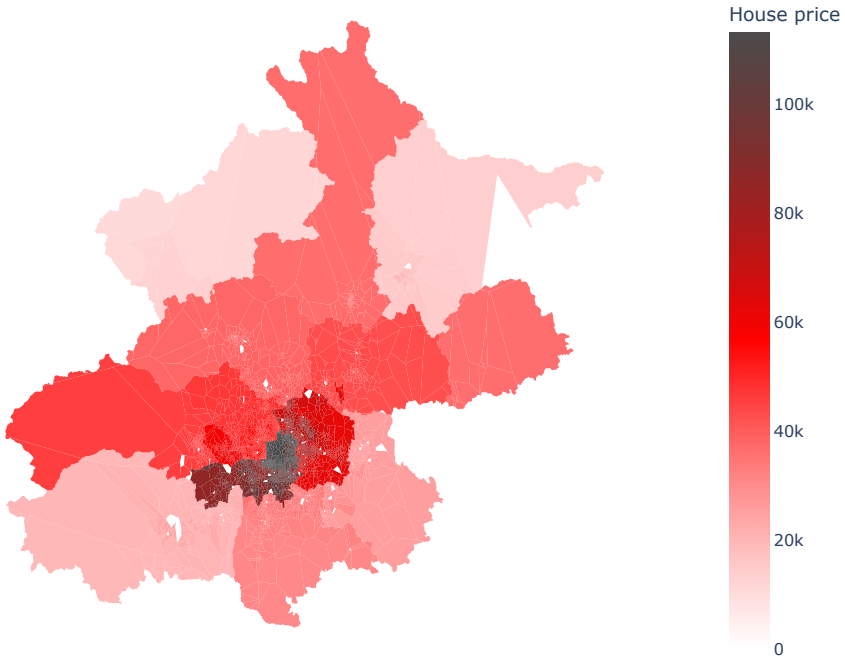


Figure 9.1: House price of Beijing, where the Voronoi polygons are generated based on the spatial distribution of the collected residential locations from Lianjia.

9.3.3 House Price Data

For house price, we collect the data by crawling online housing agents, i.e., Lianjia [131], which is the largest Chinese real-estate brokerage company that provides a comprehensive coverage of housing properties. We crawl 8,124 residential sale prices in Beijing. For each transaction, it records the residential name, sale price, floor size and residential address with latitude and longitude information we collected from Amap Map [132]. Here, the house prices correspond to the average prices with the unit of rmb/m^2. We illustrate the collected house price in Figure 9.1.

By following [53], we use the house price data to denote users' socioeconomic status. In particular, the range of collected average house price is from 10,588 to 113,224 in Beijing. To study multiple-class classification, we take the binary classification for example. We define a binary label with the median threshold of the range (i.e., $\frac{10{,}588+113{,}224}{2} = 61,891$), we set label 0 for the users, whose house prices are smaller than the threshold; label 1 otherwise. Here, we find users' home locations by following the previous studies [53, 133, 134], the home of a user is inferred as the location visited the most frequently during nighttime from 22:00 to 07:00.

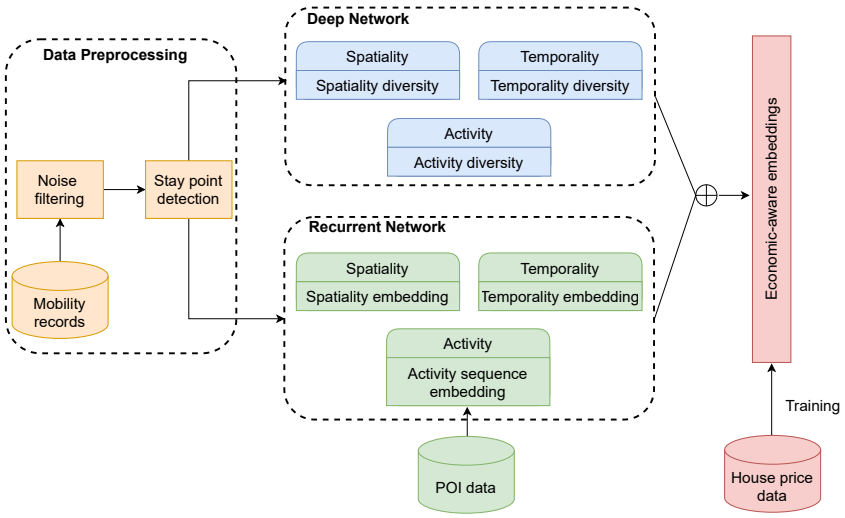


Figure 9.2: The overall framework of `DeepSEI`, where $\bigoplus$ denotes the concatenation operation.

9.4 Proposed Method

9.4.1 Overview

To predict users' socioeconomic classes with their mobility records, we propose a novel deep model for the Socioeconomic Inference, called `DeepSEI`. `DeepSEI` model consists of three components, including data preprocessing (Section 9.4.2), deep network (Section 9.4.3) and recurrent network (Section 9.4.4). The deep network aims to capture some statistics based on users' mobility records and the recurrent network aims to capture the sequential patterns behind users' mobility records.

For deep network, it is designed to quantify three important aspects of users' mobility characteristics: spatiality, temporality and activity. In spatiality, we consider users' radius of gyration extracted from the their trajectories, which describe the typical spatial range of users' activities. In temporality, it is intuitive that people with different professions (e.g., government officials or IT engineers) would travel differently, and the characteristics can be reflected on their temporality. We explore an entropy-based temporality indicator for capturing the regularity of users' activities. In activity, it is intuitive that individual daily activities that could be used for inferring users' economic statuses, e.g., rich people have diverse daily activities and travel more places in general, and thus we use an entropy-based activity indicator to capture that aspect.

For recurrent network, we focus on the sequential activity data extracted from users' mobility records, where each activity involves the spatial, temporal, semantic information. We design a hierarchical LSTM for the recurrent network, the hierarchical structure can reduce the mobility sequence length, and extract periodic mobility behaviours of human, which provides necessary information to infer the condition of users' socioeconomic. In particular, spatial and temporal features are two informative and fundamental ones to record where and when the user's activities are conducted. Semantic information of the mobility records provide the context for understanding the intended activities that a user conducts, e.g., working or dining. Intuitively, the contextual semantics allow us to profile the user's economic statuses, e.g., a user visits the fast food restaurants frequently and rarely visits recreational venues such as fitness centers, he may have poor economic conditions with health risks.

We concatenate the outputs of the two networks into a long vector, which is further fed into a fully connected layer to produce the predicted users' classes. We train the `DeepSEI` model in a supervised manner (Section 9.4.5). Figure 9.2 illustrates the overall framework of `DeepSEI` and we discuss the detailed designs of each component as follows.

9.4.2 Data Preprocessing

Recall that we collect three types of data in our task, i.e., mobility data, POI data and house price data. Mobility data records the location of a user's movement at a timestamp and we use the data for constructing spatial and temporal features. We use POI data for constructing semantic features (i.e., working, food and drink), which describe the categories of the places visited by users. We use the house price data as economic contexts for constructing the labels indicating users' socioeconomic classes. In data preprocssing, it involves two steps (i.e., Noise Filtering and Stay Point Detection) for preprocessing the mobility data.

Noise Filtering. For mobility records data (or trajectory data), we first perform the trajectory segmentation by dividing the data into one-week trajectory instances, which are used to train and test the model. Within each segmented trajectory, we filter those noisy data points and then detect those stay points, where a stay point corresponds to one activity such as working, shopping, and staying at home. In particular, trajectory data

usually contains noises due to the way how it is collected, e.g., a sampled location might be several hundred meters away from a true location. Such noisy points will affect the quality of stay point detection. We adopt a heuristic-based approach proposed in [135] for filtering noisy points in trajectories. It sequentially calculates the traveling speed for each point in a trajectory based on its precursor and itself. If the speed is larger than a threshold, the current examined point is removed from the point sequence.

Stay Point Detection. Based on the cleaned trajectories, we extract all stay points from them. Specifically, we adopt the stay point detection algorithm proposed in [136] for detecting stay points from trajectories. The algorithm first checks if the distance between an anchor point and its successors in a trajectory is larger than a given threshold S_d. It then calculates the duration between the anchor point and the last successor within S_d. If the duration is larger than a temporal threshold S_t, a stay point is detected, and the anchor point moves to the next point after the current stay point. Otherwise, the anchor point moves forward by one. This process is repeated until the anchor point moves to the end of the sequence so that all stay points in a trajectory are extracted.

9.4.3 Deep Network

One previous study [53] reveals that users' socioeconomic status can be reflected by their trajectories. Inspired by this, we use some indicators that are computed from users' mobility records for the task. These indicators capture users' spatiality, temporality and activity. Those captured features are generally static with fixed values. We embed those features via a deep network and concatenate those embedded features for the task.

Spatiality Diversity. To capture the mobility features from users' trajectory data, we consider radius of gyration, which is widely used as a spatial indicator to capture users' mobility characteristics [52, 53]. Given a trajectory data $T =< (p_1, t_1), (p_2, t_2), ..., (p_n, t_n) >$, where p_i and t_i $(1 \leq i \leq n)$ denote the location p_i of a moving object at time t_i. The radius of gyration R_g is defined as follows.

$$R_g = \sqrt{\frac{\sum_{i=1}^{n}(p_i - p_c)^2}{n}}, \quad p_c = \frac{\sum_{i=1}^{n} p_i}{n}, \tag{9.1}$$

where p_c denotes the center of the sampled locations. The rationale of radius of gyration is to capture the spatial dispersion of a user' movement. Intuitively, a small radius indicates that the user's activities are mainly in a small area.

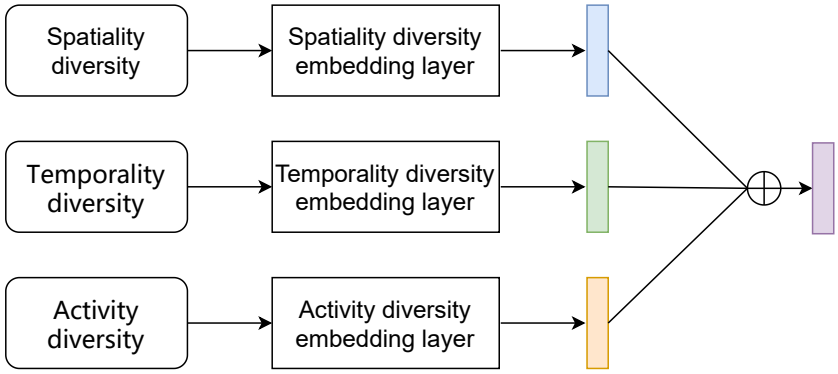


Figure 9.3: The architecture of deep network, where $\bigoplus$ denotes the concatenation operation.

Temporality Diversity. We consider the feature in temporal aspect. It is intuitive that people with similar professions would travel similarly in terms of temporality. For example, office staff would commute between home and office regularly on weekdays while self-employers would mainly stay at home and only go out occasionally. In addition, some users (e.g., those work at a government department) would commute more regularly than others (those work at an IT company). These temporality patterns embed rich information that could be used for inferring economic statuses of users. Therefore, we explore a temporality indicator, namely temporality diversity.

Given a user's stay points extracted from his/her mobility records $(s_1, \Delta t_1), (s_2, \Delta t_2), ..., (s_n, \Delta t_n)$, where s and Δt denote the stay location and the duration at that location, respectively. By following [53, 137], we calculate the temporality diversity via cross-entropy, which captures the duration distribution among those stay locations. Let $p_i = \frac{\Delta t_i}{\sum_{j=1}^{n} \Delta t_j}$ denote the proportion of stay duration at location s_i, the Temporality Diversity (TD) is defined as follows:

$$TD = -\sum_{i=1}^{n} p_i \log(p_i), \tag{9.2}$$

where $\sum_{i=1}^{n} p_i = 1$. The rationale of the indicator is to consider the human daily regularity. For example, for some people (e.g., IT programmers), whose daily lives are mainly concentrated at home and work places, they would have a high regularity reflected as a low cross-entropy value.

Activity Diversity. Previous studies [53, 138] exhibit that the diversity of individual daily activities has a strong correlation with their socioeconomic status. Intuitively, a

well-developed city (e.g., Beijing in our study) provides for a wide range of activities for citizens, and richer people tend to have a higher activity diversity in daily life, e.g., their jobs are generally with higher diversification.

Inspired by this, we explore the features for describing activity diversity in the model. Given a sequence of user's stay points $(s_1, \Delta t_1), (s_2, \Delta t_2), ..., (s_n, \Delta t_n)$, where each stay location s is potentially associated with an activity such as working. We define the source and destination pair $e = (s_{i-1}, s_i)(0 < i \leq n)$ as two consecutive stay locations in the trip, and let E denote a set of all possible source and destination pairs extracted from the whole stay points, where the direction of movement could be ignored. For each pair $e \in E$, $p(e)$ denotes the proportion of observing the movement corresponding to the pair e wrt the total number of movements (i.e., $n-1$) in the records. For example, given a user's stay points $(a, \Delta t_1), (b, \Delta t_2), (c, \Delta t_3), (d, \Delta t_4), (c, \Delta t_5), (b, \Delta t_6), (a, \Delta t_7)$, there are 6 movements in the records and 4 source and destination pairs in E without considering direction, i.e., (a, b), (b, c) and (c, d). For the pair $e = (a, b)$, $p(e)$ is calculated as $p(e) = \frac{2}{6} = 0.33$. Note that $\sum_{e \in E} p(e) = 1$, and we define the Activity Diversity (AD) via cross-entropy as follows:

$$AD = -\sum_{e \in E} p(e) \log(p(e)). \tag{9.3}$$

The rationale is that for a user, whose trips are concentrated on a few locations, he/she would have a high cross-entropy value.

Network Architecture. Figure 9.3 illustrates the architecture of the deep network. To feed the above features into the `DeepSEI` model, we first tokenize them. In particular, spatiality diversity, temporality diversity and activity diversity are naturally in the form of continuous float values, we tokenize them by partitioning the range with a predefined granularity, e.g., we partition the range 0.09-8,143.3 (resp. 0-5.73 and 0.02-5.36) of spatiality diversity (resp. temporality diversity and activity diversity) with a 100 (resp. 0.5 and 0.5) granularity, and thus we obtain 82 (resp. 11 and 10) tokens.

After obtaining the tokens of above features, we then embed each feature via an embedding layer, and denote the embedded features (i.e., 32-dimensional vectors) to be fed to the deep network for spatiality diversity, temporality diversity and activity diversity as x^{ds},x^{dt} and x^{da}, respectively. We concatenate the embedded feature vectors, and thus obtain a long vector with dimension $32 * 3$, which is further used for training. To

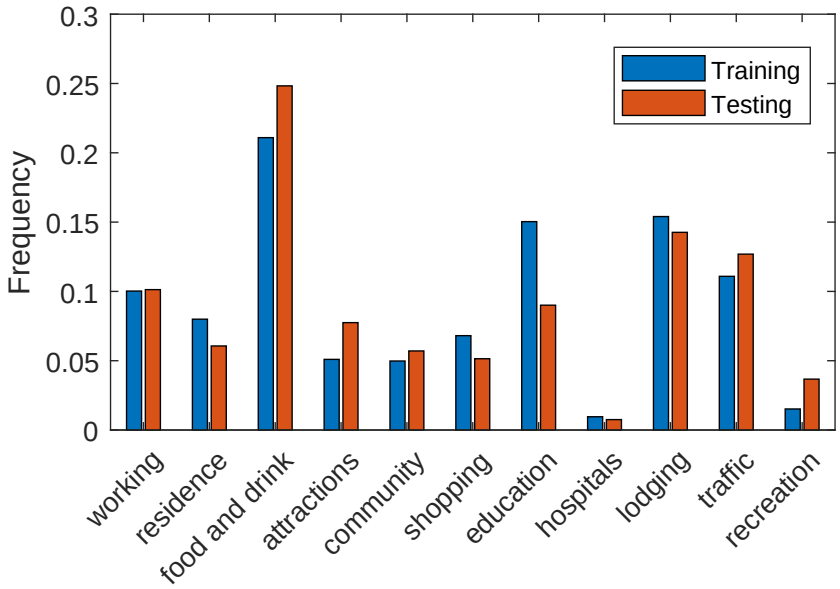


Figure 9.4: Activity distribution of Beijing.

facilitate the training process, we pre-train the tokens of spatiality diversity, temporality diversity and activity diversity via Skip-Gram model [139]. After pre-training, the tokens with similar context (e.g., two tokens correspond to two similar values for a feature) will be embedded into similar vectors in a latent space. We then use the pre-trained vectors to initialize the embedding layer of the deep network, and those embeddings can still be optimized with the model training.

9.4.4 Recurrent Network

Recurrent network adopts a hierarchical LSTM based network. Compared with the implementation of vanilla RNN based models, hierarchical LSTM is more capable of capturing the sequential and periodic information from users' trajectory data. Here, for each stay point in the trajectory, we capture the features in terms of spatiality, temporality and activity aspects.

Spatiality Embedding. Recall that we partition a geographical space into grids, and map the GPS coordinates of a stay point to the grids. Naturally, the mapped grid that encompasses the coordinates of the stay point can represent its spatial information. We then obtain the vector for the stay point by embedding its grid id as a one-hot vector, which is further fed into the hierarchical LSTM.

Temporality Embedding. For each extracted stay point, it is associated with a timestamp to record the starting time of the stay. We take the timestamp as a temporal feature, and embed it into the model. Intuitively, although the users are with different kinds of lifestyles, their temporalities are generally periodic and stable during their daily lives. By following the previous study [49], we split a one-week trajectory data into two parts, i.e., one for weekdays and one for weekends. For each part, we further split a day into 24 hourly time bins, and therefore we obtain $24 * 2 = 48$ time bins in total to tokenize the temporal feature, where we take the temporality for weekdays and weekends differently. This is because users generally have different lifestyles for these two parts. Similarly, we obtain the temporality vectors by embedding their time bins as one-hot vectors, and fed them into the hierarchical LSTM.

Activity Sequence Embedding. Recall that each detected stay point is potentially associated with an activity, we investigate whether users' activity sequences are informative for inferring users' economic statuses. Note that these sequential patterns are not captured by the mobility indicators or the temporality indicators that are based on some statistics. To this end, we use Recurrent Neural Networks (RNN) [140] for embedding users' activity sequences, where each activity is represented by the learned hidden vector of the stay point that the activity is associated with. The hidden vector that is outputted by the network will be used for the task.

Based on the stay points extracted from users' mobility records, an immediate task is to infer the most relevant activity for each stay point. To do this, we adopt a simple yet effective distance-based method to infer the activity. For each stay point, we check 8 neighbouring grids of the grid, where the stay point is located in. Then, the activity associated with the stay point is inferred as the most frequent POI category among those POIs in the 8 neighbouring grids. In addition, if no POI can be matched within the grids, we assign the activity with a special tag called "other". In Figure 9.4, we illustrate the inferred activity distribution for Beijing in terms of training and testing (details will be presented in the sequel). We notice some other methods such as Markov based inference models [141–143] are also applicable for the task. Note that the POI categories are in the form of discrete tokens. Similarly, we can obtain the activity vectors by embedding their tokens as one-hot vectors.

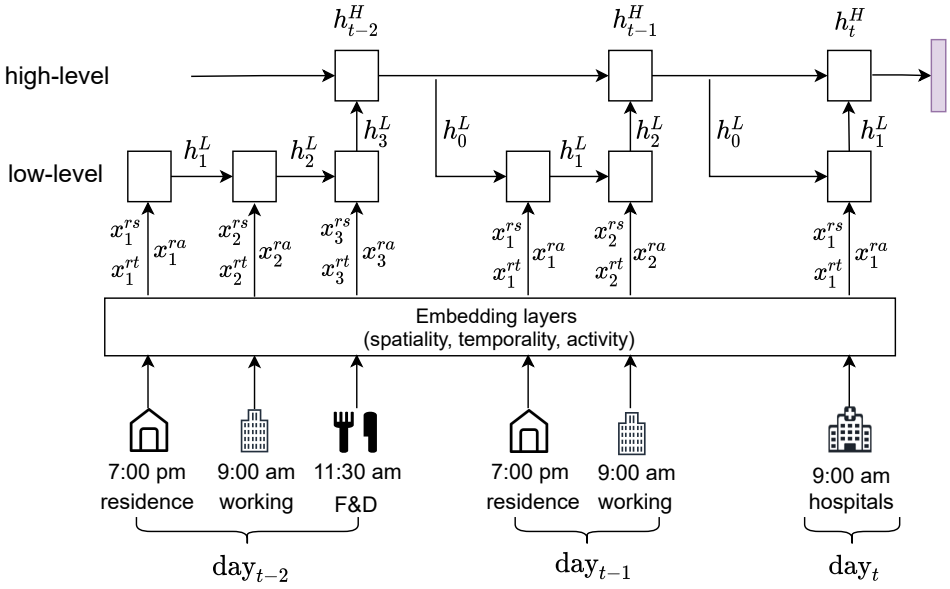


Figure 9.5: The architecture of recurrent network.

Network Architecture. Figure 9.5 illustrates the architecture of the recurrent network, which is implemented with a hierarchical LSTM, where the LSTM units [140] are employed for both low-level and high-level structures. Take an one-week trajectory for example, the low-level LSTM is to capture the intra-transitions within a day and the high-level LSTM is to capture the inter-transitions across days.

For the low-level LSTM, we denote the embeddings used in the recurrent network for spatiality, temporality and activity as x^{rs}, x^{rt} and x^{ra}. Then, we concatenate those embeddings at each time step i, denoted by $x_i^{rs} \oplus x_i^{rt} \oplus x_i^{ra}$, and feed the concatenation to low-level LSTM to obtain the hidden state h_i^L at this time step.

$$h_i^L = \text{LSTM}^L(x_i^{rs} \oplus x_i^{rt} \oplus x_i^{ra}, h_{i-1}^L). \tag{9.4}$$

We take the last hidden state within the day denoted by h_n^L, as a latent representation for capturing the intra-transitions within the day, and it is further fed into high-level LSTM.

$$h_j^H = \text{LSTM}^H(h_n^L, h_{j-1}^H), \tag{9.5}$$

where h_j^H denotes the high-level hidden state at the j^{th} day, which is then fed to initialize the hidden state h_0^L for the low-level LSTM. We feed the last hidden state at the high-level LSTM (i.e., suppose h_7^H is the last day of a week) into a fully connected layer as the output of recurrent network.

With the hierarchical LSTM, it brings two advantages. First, compared with vanilla RNN, modeling the user's mobility records in a hierarchical way is able to reduce the sequence length, where the low-level LSTM is for handling the records within a day and the high-level LSTM is for handling the records across days. The design helps alleviate the issue of degraded model performance for modeling long mobility sequences. Second, the hierarchical structure is able to capture both sequential and periodic information, where the low-level LSTM captures the sequential transitions from users' mobility records, and high-level LSTM preserves the periodic information on a daily basis.

9.4.5 Jointly Training Deep and Recurrent Networks

We jointly train the two networks (i.e., deep and recurrent networks) for predicting users' socioeconomic classes. Following the previous study [49], we partition the whole Geolife dataset into one-week trajectories. We randomly sample 70% for training (i.e., 1,360 one-week trajectories) and the remaining for testing (i.e., 584 one-week trajectories). For each trajectory, it is associated a ground truth label indicating the user's socioeconomic class, which is obtained using house price data described in Section 9.3.3

To train the `DeepSEI`, we first pre-train the deep and recurrent networks, separately. The pre-training provides a warm-start before the joint training. During the joint training, we concatenate the outputs by the two networks as a long vector, which is further fed into a fully connected layer with the softmax function to produce the probability of indicating each class. We generate 50 episodes for the pre-training and joint training. We use Binary Cross Entropy Loss (BCELoss) for the task, where the Adam stochastic gradient descent is adopted to optimize the network parameters.

9.5 Experiments

9.5.1 Experimental Setup

Tasks. We explore two tasks with the `DeepSEI` model. One is classification, we consider the number of classes from 2 to 5 by evenly partitioning the house price range to 2 - 5 equal intervals, and define the corresponding labels as described in Section 9.3.3. The other is clustering, we collect the concatenated embedding outputted by deep network

and recurrent network. We consider the learned embeddings, which have incorporated economic context from the users, and can serve other economically related tasks. We explore k-means clustering in this chapter, and vary the k from 2 to 5. Similarly, the constructed labels are used as the ground truth for the clustering.

Baselines. The following baselines are adapted for comparison.

• SES [53]. The study verified users' socioeconomic status can be reflected by their mobility patterns. Inspired by the study, we extract the mobility indicators including (1) radius of gyration, (2) number of activity locations, (3) activity entropy, (4) travel diversity, (5) K-radius of gyration, (6) unicity. These indicators capture how regularly, broadly, frequently, and intensively a user would travel within a city. Based on the features, we explore the following 10 classifiers for the classification task, including RBFSVM, LinearSVM, Logistic Regression, K-Nearest Neighbors (KNN), Decision Tree, Random Forest, Bayes, Adaboost, Gradient Boost and XGBoost. Among them, we select two classifiers with the best effectiveness, namely SES (Random Forest) and SES (XGBoost). In addition, we concatenate those mobility indicators as a vector for the clustering task.

• DIF [52]. The study proposes a framework of inferring users' demographics (e.g., gender, martial status or age) from their trajectories and geographical context. Other than the feature (1), (2), (3), (4), (6) studied in SES, it further incorporates (7) number of unique stay points, (8) centroid of stay points, (9) number of travels and (10) land use. Similarly, we explore the aforementioned 10 classifiers based on the features, and DIF (Random Forest) and DIF (XGBoost) dominate others in terms of the effectiveness. For clustering, we concatenate these features as the input.

• L2P [49]. The study investigates users' demographics and proposes a general location-to-profile (L2P) framework. L2P extracts the features from users' check-ins in terms of spatiality, temporality, and location knowledge (e.g., POI categories). It constructs a three-way tensor based on the extracted features, and adopts Tucker tensor decomposition to obtain a feature vector for each user. Based on the feature vectors, we still explore the aforementioned 10 classifiers, and L2P (Random Forest) and L2P (XGBoost) stand out. Those feature vectors can also be used for clustering.

Evaluation Metrics. This chapter involves two tasks: classification and clustering. For classification, we use the F_1-score and accuracy as the evaluation metric by following [49].

The two metrics are widely used for classification task and can evaluate the imbalanced cases effectively. For clustering, we use the metrics of Adjusted Rand Index (ARI) and Adjusted Mutual Information (AMI). They measure the correlation between the clustered result and the ground truth. Their values lie in the range of $[-1, 1]$, and we normalize their values in $[0, 1]$ for the ease of reading, where a higher ARI or AMI indicates a better result.

Parameter Setting. Our `DeepSEI` model consists of deep network and recurrent network. For deep network, we embed the features of (1) spatiality diversity, (2) temporality diversity and (3) activity diversity into 32-dimensional vectors, and concatenate them as a long vector with dimension $32 * 3 = 96$ as the output. For recurrent network, we embed the features of (4) spatiality, (5) temporality and (6) activity into 32-dimensional vectors, and feed the concatenation as a 96-dimensional vector (i.e., $32 * 3 = 96$), into a hierarchical LSTM. To implement the hierarchical LSTM, we use a low-level LSTM with 64 hidden units and a high-level LSTM with 64 hidden units. The hidden vector at the last step of the high-level LSTM is then fed into a fully connected layer with 32 neurons. Thus, the output of recurrent network is a 32-dimensional vector. Further, the outputs of deep network and recurrent network are concatenated as a 128-dimensional vector (i.e., $96 + 32 = 128$), which is fed into a fully connected layer with the softmax function as the activation function. We train the networks with Adam stochastic gradient descent and an initial learning rate of 0.001.

The default parameters of stay point radius S_d and stay point duration S_t are set to 100m and 60 min, respectively. Here, we vary the parameter S_d from 100m to 300m, since the results are similar and we use the setting of S_d = 100m. The setting of the parameter S_t will be studied in experiments. For extracting the above features from (1) to (6), the following parameters are involved: the cell size for the feature (4), the spatiality granularity for the feature (1) and the temporality granularity and activity granularity for the feature (2,3). We use the settings of cell size, spatiality granularity, temporality granularity and activity granularity as 200m, 100, 0.5 and 0.5, respectively. The results of their effects are shown in experiments.

Evaluation Platform. We implement `DeepSEI` and other baselines in Python 3.6 and Tensorflow 2.3.0. The experiments are conducted on a desktop with Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz and a 32 GB memory.

Table 9.2: Effectiveness evaluation and running time.

Method	Classification								Clustering							
Metric	2		3		4		5		2		3		4		5	
	F1	Acc	F1	Acc	F1	Acc	F1	Acc	ARI	AMI	ARI	AMI	ARI	AMI	ARI	AMI
SES(RF)	59.3	68.9	42.4	44.6	32.1	37.2	25.6	30.3	49.7	50.2	50.0	50.2	49.4	50.7	49.9	50.3
SES(XGBoost)	60.0	60.3	41.0	43.3	32.6	36.4	23.9	27.8								
DIF(RF)	69.1	69.3	57.1	59.6	50.3	53.0	47.1	48.5	60.3	56.9	57.8	55.9	54.0	52.6	50.7	50.4
DIF(XGBoost)	70.3	75.5	61.6	63.5	52.6	55.7	42.6	49.3								
L2P(RF)	67.8	72.2	55.2	56.4	46.1	48.2	41.5	44.3	56.6	55.5	53.8	53.4	51.2	50.8	49.7	50.0
L2P(XGBoost)	68.2	67.8	58.4	59.3	47.7	49.0	40.2	42.2								
DeepSEI	**86.1**	**90.2**	**80.2**	**80.4**	**63.9**	**64.2**	**53.3**	**58.8**	**83.2**	**77.5**	**80.9**	**71.9**	**70.6**	**70.3**	**69.2**	**64.4**

Table 9.3: Ablation study for `DeepSEI`.

Method	Classification	Clustering
`DeepSEI`	86.1	83.2
w/o Deep Network	73.7	78.2
w/o Spatiality Diversity	78.8	80.0
w/o Temporality Diversity	82.5	81.9
w/o Activity Diversity	82.8	81.9
w/o Recurrent Network	33.4	60.4
w/o Spatiality Embedding	75.7	78.1
w/o Temporality Embedding	82.8	80.5
w/o Activity Embedding	68.6	77.9

9.5.2 Experimental Results

(1) Effectiveness evaluation (comparison with different classifiers). We compare the `DeepSEI` model with the baselines. In Table 9.2, we report their effectiveness in terms of F_1-score(%) and accuracy(%) for classification and ARI(%) and AMI(%) for clustering. Overall, our `DeepSEI` model consistently outperforms the baselines, e.g., in binary classification and clustering, it outperforms the best baseline (i.e., DIF) by 22.5% and 37.9%, respectively. The reasons are mainly two-fold: 1) the `DeepSEI` model is with more comprehensive features to infer the users' socioeconomic status from three aspects, i.e., spatiality, temporality and activity; 2) we study a deep learning based solution to incorporate those features with two well-designed networks, i.e., deep network and recurrent network. The two networks are capable to capture the context and mobility records generated with GPS instead of the mobile phone data that are used as the existing studies.

(2) Ablation study. We conduct an ablation study to evaluate the effect of each network (i.e., deep network or recurrent network) and feature in the `DeepSEI` model, and the comparing results are reported in Table 9.3. Overall, we can see all these networks and

features contribute to the final result. For Recurrent Network, w/o Recurrent Network corresponding to only Deep Network is kept, the result performs the worst with F_1-score of 33.4% and ARI of 60.4%. This is because it captures a sequence of users' daily activities, which is essential to infer users' socioeconomic status. For Deep Network, we observe the spatiality embedding is with the most effect, e.g., when the spatiality is removed, the F_1-score is 78.8, which drops 9.3%. This is because users' economic statuses are highly linked to the range of their activity territory, which has been verified in previous studies [53].

Table 9.4: Impacts of stay point duration (mins) for `DeepSEI`.

Parameter	30	60	90	120	150
# Training instances	1,485	1,396	1,360	1,336	1,315
# Testing instances	637	599	584	573	564
Classification	74.3	78.9	**86.1**	82.6	81.9
Clustering	80.9	81.9	**83.2**	81.9	81.8

(3) Parameter study (varying stay point duration S_t). The stay point duration parameter S_t controls the time threshold of the stay point detection algorithm, where the points in a trajectory will be merged as one stay point if the duration of the first point (called anchor point) and the last point in the trajectory is within S_t. We vary the S_t from 30 minutes to 150 minutes, and the results are reported in Table 9.3. As expected, with the increase of S_t, less stay points are detected, which corresponds to less training and testing instances are generated. We observe that our model performs the best when S_t is set to 90 minutes with the F_1-score of 86.1% and ARI of 83.2%. This is because with a small S_t, the detected stay points cannot accurately reflect the users' activities, it falsely takes some trivial behaviors such as "walking" as the users' activities, which prevents the model to learn useful information. With a large S_t, many stay points cannot be detected, which causes the model performance degrades as many features that are associated with the stay points are missing.

(4) Parameter study (varying cell size). In Table 9.5, we study the effect of cell size by varying the size from 100 meters to 500 meters. Here, the effect of cell size is mainly in two aspects: 1) for spatiality, each grids contains a set of stay points; with a large

Table 9.5: Impacts of cell size (m) for DeepSEI.

Parameter	100	200	300	400	500
# Location tokens	4,460	3,080	2,378	1,995	1,707
Classification	82.3	**86.1**	83.6	81.5	79.8
Clustering	82.0	**83.2**	81.2	80.5	79.4

size, more stay points will be indexed to the same grid, which results many stay points share the same mobility embedding; 2) for activity, we infer the activity of a stay point using the POIs in 8 neighbouring grids of the grid, where the stay point is located; then, a large cell size, corresponds to a large grid, which takes more POIs into consideration. Based on the results, we observe that the cell size 200 meters fits our model best. With the smallest cell size (100 meters), the model degrades. This is because a small cell size generates more tokens, which makes the model difficult to train. On the other hand, a large cell size causes a lower resolution of the stay points, and overlooks the differences of mobility features. In addition, as more POIs are considered in a large grid, those POIs may generate noise to interfere with POI inference and this is in line with our intuition.

Table 9.6: Impacts of spatiality granularity for DeepSEI.

Parameter	100	200	300	400	500
# Spatiality tokens	81	40	27	20	16
Classification	84.1	85.9	**86.1**	85.6	83.8
Clustering	82.1	81.9	**83.2**	82.7	82.0

(5) Parameter study (varying spatiality diversity granularity). We study the effect of spatiality granularity in Table 9.6. We vary the granularity from 100 to 500, and report the effectiveness in terms of classification and clustering. The parameter captures the resolution of the spatial range of users' daily activities. A smaller value provides a higher resolution but incurs more tokens, which affects the model training. With a larger value, the capability of model to distinguish different spatial diversities will degrade. We set it to 300, which leads to the best effectiveness.

(6) Parameter study (varying temporality and activity diversity granularity). We study the effect of diversity granularity for two entropy-based indicators, i.e., temporality diversity and activity diversity. We vary the granularity parameter from 0.1 to 0.9,

Table 9.7: Impacts of temporality and activity granularity for DeepSEI.

Parameter	0.1	0.3	0.5	0.7	0.9
# Temporality tokens	57	28	11	8	6
# Activity tokens	53	27	10	7	5
Classification	72.3	78.6	**86.1**	83.8	81.6
Clustering	80.4	81.5	**83.2**	81.8	80.8

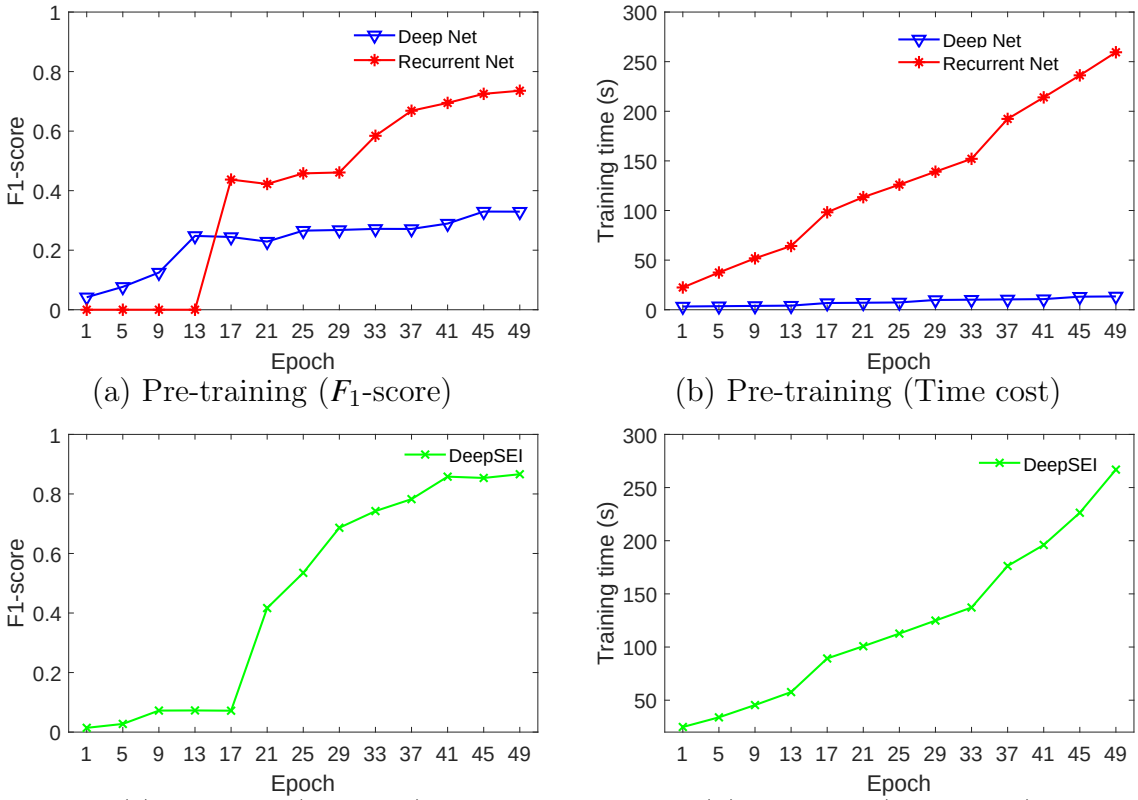
Table 9.8: Case study, DS, DT and DA denote the features captured via the deep network for spatiality diversity, temporality diversity and activity diversity; RT and RA denote the features captured via the recurrent network for temporality (the time bin) and activity.

Case	User 1 (richer)		User 2 (richer)		User 3 (poorer)		User 4 (poorer)	
DS, DT and DA	9.62, 1.30 and 2.16		5.52, 3.14 and 2.20		14.29, 4.45 and 2.77		10.38, 4.89 and 5.12	
Stay points	RT	RA	RT	RA	RT	RA	RT	RA
s_1	18	residence	20	residence	9	working	32	hospital
s_2	7	recreation	12	food and drink	10	traffic	35	traffic
s_3	9	education	12	traffic	12	food and drink	36	food and drink
s_4	10	working	14	working	13	working	40	community
s_5	13	education	19	lodging	18	residence	41	residence
s_6	17	residence	37	residence	7	traffic	34	attractions
s_7	9	working	8	working	8	working	43	residence

and the results are reported in Table 9.7. Temporality diversity and activity diversity are two features with continuous numerical values, dividing their ranges with a small granularity will generate too many tokens, and causes the model difficult to learn the features; with a large granularity, the capability of the model to identify the diversity from users' mobility patterns will degrade, which is as expected. overall, a moderate setting with the value of 0.5 provides the best effectiveness.

(7) Training time. In Figure 9.6, we report the times and the corresponding effectiveness with the default setup in Section 9.5.1. We generate 50 epochs for pre-training and training, respectively. We observe that the effectiveness improves with the number of epochs and the corresponding training time increases almost linearly. In pre-training, the Recurrent Network takes more time because it has a more complex network architecture (i.e., hierarchical LSTM). In training, the DeepSEI model incorporates the two networks and obtains a further improvement after 32 epochs. We observe that the DeepSEI model converges after 41 epochs, and we use the trained model for other experiments.

(8) Case study. We conduct a case study. We select four cases for the study, where User 1 and 2 are identified as the richer users in the same class 1, and User 3 and 4 are identified as the poorer users in another class 2. In Figure 9.7, we visualize the locations



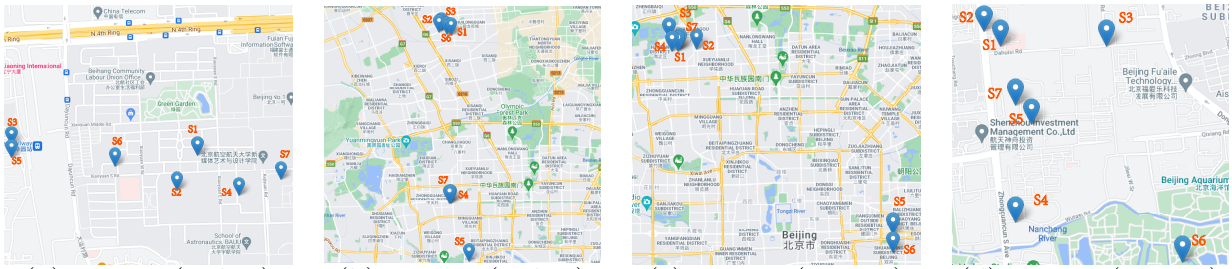
(a) Pre-training (F_1-score)
(b) Pre-training (Time cost)
(c) Training (F_1-score)
(d) Training (Time cost)

Figure 9.6: Training cost on Geolife.

of their stay points on the map. In Table 9.8, we list the features captured by the deep network and recurrent network. We observe the following insights that may explain the relationship between their mobility patterns and socioeconomic status.

Insight 1: Richer users tend to travel shorter. In Table 9.8, we observe the richer users (User 1 and User 2) are generally with the smaller spatiality diversities (e.g., 9.62 and 5.52) than poorer users. This insight is in line with the intuition from the previous study [53], and the reason could be that rich people are busy with work and have limited time for travelling.

Insight 2: Richer users are generally with lower temporality/activity diversity of daily activities. Temporality/activity diversity is an entropy-based feature to reveal the regularity of users' daily activities. In Table 9.8, we observe the regularity of User 1 and User 2 are high, corresponding to the smaller values. For example, in Figure 9.7, User 1 mainly commutes between home ("residence" POI) and office ("working" POI) regularly, and



(a) User 1 (richer) (b) User 2 (richer) (c) User 3 (poorer) (d) User 4 (poorer)

Figure 9.7: Illustration of stay points for four users with different socioeconomic status.

he/she is with the least temporality diversity 1.30. In contrast, the User 4 is irregular, e.g., he/she visits many places instead of staying somewhere and working.

Insight 3: Richer users are with secure jobs. We infer the users' employment status based on the data extracted from their stay points. We infer that User 1 is with a steady job since he/she works (at 09:00 am - 05:00 pm) and stay homes (at 05:00 pm - 07:00 am) regularly. In this situation, he/she has a stable source of income (e.g., we infer that he/she may be a faculty at an university based on the stay points on the map), and the status is reflected on his/her house price data accordingly.

9.6 Summary

In this chapter, we study user socioeconomic status inference, and propose a novel deep model called `DeepSEI` for the task. `DeepSEI` is composed of two neural networks, i.e., deep network and recurrent network. We extract features from three aspects of users' mobility records, namely spatiality, temporality and activity, and feed them to the two networks. We conduct the experiments on Geolife dataset with collected contexts, and the results show that the `DeepSEI` model achieves a noticeable improvement over the baselines.

Chapter 10

Conclusions and Future Work

In this thesis, we study trajectory simplification, similarity search and socioeconomic status inference from trajectory data with different deep learning models: (1) error-bounded online trajectory simplification with multi-agent reinforcement learning; (2) size-bounded trajectory simplification with reinforcement learning; (3) query accuracy driven trajectory simplification with reinforcement learning; (4) subtrajectory similarity search with reinforcement learning; (5) multi-trajectory similarity measurement with representation learning; (6) similarity search within a database with metric learning, and (7) inferring user socioeconomic status from their trajectories with deep learning. We conclude the thesis as follows.

10.1 Conclusion

The first study is regarding error-bounded online trajectory simplification (EB-OTS). EB-OTS is a popular research problem, whose aim is to drop as many points as possible from a trajectory in a streaming fashion and keep the information loss, captured as the "erro", bounded by an error tolerance. We propose a multi-agent RL-based method called MARL4TS. Compared with existing algorithms, MARL4TS is data-driven and can adapt to different error measurements. We conduct extensive experiments on real-world trajectory datasets, which show that MARL4TS simplifies trajectories with consistently lower compression ratios and runs comparably fast, compared with existing methods.

The second study is regarding the size-bounded trajectory simplification problem, which is to drop some points of a trajectory when they are being collected and/or after

they are accumulated in order to save storage cost and more importantly make the query processing efficiently on these data. In this work, we propose a new solution based on reinforcement learning for the trajectory simplification problem. We formalize the objective of trajectory simplification and show that it can be perfectly captured by the RL algorithms with some theoretical justifications. In addition, the proposed algorithms are data-driven, generic, effective and efficient based on the extensive experimental study.

The third study is regarding the query accuracy driven trajectory simplification problem, which aims to find a simplified trajectory database, such that the difference between query results on the original database and simplified database is minimized. We develop the first solution to the problem, called RL4QDTS, which is based on reinforcement learning. Compared with existing trajectory simplification algorithms, `RL4QDTS` is able to optimize the objective of the QDTS problem directly. Extensive experiments on two real-world trajectory datasets show that our solution is able to consistently outperform the existing EDTS algorithms for five query processing operations.

The fourth study is regarding the similar subtrajectory search (SimSub) problem, which aims to return a portion of a trajectory (i.e., a subtrajectory) that is the most similar to a query trajectory. Subtrajectory similarity is a fundamental concept in spatiotemporal data management systems. SimSub could capture trajectory similarity in a finer-grained way and many real applications take subtrajectories as basic units for analysis; however, the problem has been mostly disregarded. To this end, we develop a suite of algorithms including both exact and approximate ones for the SimSub problem, and conduct experiments to show the performances of these algorithms. Among them, two algorithms that are based on deep reinforcement learning stand out and outperform others in terms of both effectiveness and efficiency on three real-world trajectory datasets.

The fifth study aims at improving the effectiveness and efficiency of similar sports play (i.e., multi-trajectory) retrieval utilizing deep representation learning techniques. It becomes a common practice to track the moving agents in a sports game (e.g., the players and the ball in a soccer game). The data collected by these tracking systems can be represented as the trajectories of the players and the ball in a game, i.e., it embeds both spatial and temporal features of a game. Hence, it is usually termed spatiotemporal sports data and has been widely used in many sports analytics tasks. In this work, we propose

the first deep learning-based approach to learn the representations of spatiotemporal sports data, called play2vec, which is robust against noise and fast enough to compute the similarity of a given pair of sports plays in linear time. The experiments show that our learning-based solution performs more effectively and efficiently compared with the state-of-the-art methods on real-world soccer games.

The sixth study is regarding the similar play retrieval problem. This problem is to find a fragment of a game in a database, which is the most similar to a query play. We make the following technical contributions. Firstly, we develop a suite of algorithms for the problem of searching for the most similar play to a query play within a game. Secondly, we develop a novel algorithm based on deep metric learning, ScoreSearch, for searching for the most similar play to a query play within a database.

The seventh study is regarding the user socioeconomic status inference from the user's trajectory data, and we propose a novel deep model called `DeepSEI` for the task. `DeepSEI` is composed of two neural networks, i.e., deep network and recurrent network. We extract features from three aspects of users' mobility records, namely spatiality, temporality and activity, and feed them to the two networks. We conduct the experiments on Geolife dataset with collected contexts, and the results show that the `DeepSEI` model achieves a noticeable improvement over the baselines.

10.2 Future Work

10.2.1 Aging-Preserving Trajectory Simplification

Nowadays, massive amounts of trajectory data are being accumulated and used in diverse applications, and the accumulation of trajectory data causes challenges that data storage is expensive and query processing on the data is time-consuming. To solve these challenges, trajectory simplification techniques have been introduced to reduce the sizes of trajectories, while preserving as much information as possible. The underlying rationale of trajectory simplification is that not all points in a trajectory carry equally important information, and thus dropping unimportant ones may be acceptable. We plan to explore aging-friendly trajectory simplification techniques [25]. The intuition is that as time evolves, the trajectory data gets older, and less precision (measured by

an error) becomes acceptable. We say that a trajectory simplification algorithm $\mathcal{A}$ (it receives an original trajectory, and outputs its simplified trajectory) wrt an error measurement M is aging friendly, if for every error tolerance e_1 and e_2, where $e_1 < e_2$, it has $\mathcal{A}(T, e_2, M) = \mathcal{A}(\mathcal{A}(T, e_1, M), e_2, M)$ for every trajectory T. We aim to develop an aging-friendly trajectory simplification algorithm, which drops as many points as possible under a given error tolerance. We call the problem Aging-Preserving Trajectory Simplification. Specifically, the existing solution Douglas-Peucker (DP) [57] is aging-friendly under PED and SED measurements. However, the effectiveness of DP can be improved in the aging scenario, given that some heuristics are involved in DP. We plan to explore other solutions with the power of deep learning techniques to handle the aging problem, while achieving superior effectiveness.

10.2.2 Learned Index on Spatial Data

There has been an explosion in the amount of spatial data in recent years. The big volume data makes it very challenging to manage and analyze the data. To support the efficient processing of spatial queries, such as range queries and KNN queries, spatial databases have relied on delicate indices. Recently, machine learning powered learned index (i.e., RMI [144]) has been introduced to replace classic index structures like B-Tree, whose idea is to learn a recursive model for indexing 1-dimensional data by learning a cumulative distribution function to map a search key to a rank in a list of ordered data objects. Further, several learned indices [145, 146] have been proposed based on the RMI, and the idea of learned indices is also extended to spatial data and multi-dimensional data. However, those learned indices require replacing both the indices and query processing algorithms currently deployed by the databases, and such a radical departure is likely to encounter challenges and obstacles. In contrast, we plan to develop novel ML indexing techniques on classic spatial indices such as R-Tree. R-Tree is arguably the most popular spatial index that prunes irrelevant data for queries, and is widely used in commercial databases such as PostgreSQL and MySQL. Different from previous studies in this area which focus on learning a cumulative distribution function, we plan to consider a fundamentally different approach to use machine learning techniques to construct an R-Tree in a data-driven way for better query efficiency in a dynamic environment, where

updates occur frequently and bulk loading is not viable. With our proposal, the basic structure of the R-Tree is retained and thus all the currently deployed query processing algorithms in a spatial database will still be applicable to our index.

10.2.3 Spatio-Temporal Analytics

Some research plans on spatio-temporal analytics are summarized below. (1) anomalous subtrajectory detection, detecting anomalous trajectory has become an important concern in many location-based applications. However, most of the existing studies for this problem can only give a vague judgment (e.g., an anomaly score) of whether a given trajectory is anomalous, and fail at indicating which part of the trajectory is anomalous, i.e., detecting anomalous subtrajectory. We plan to develop techniques for detecting anomalous subtrajectory, which is important for real-world applications, since it can more accurately locate and indicate the anomalousness in a trajectory in a finer-grained way. (2) Trajectory-User Linking (TUL), which links the POI trajectories to their generators, is essential for many tasks in location-based social networks, such as recommendation, prediction, and user profiling. By empirical findings, existing works usually cannot have satisfactory results on short trajectories, e.g., in many cases, a short trajectory involves only a few POIs (maybe just 2 or 3 POIs), which may not help to infer the user who generates the trajectory. This is called the data sparsity problem. We plan to explore recently emerging learning techniques to alleviate the data sparsity problem.

Appendix - List of Publications

(* indicates that Zheng is the co-first author, # indicates that Zheng is the corresponding author)

International Conferences

- Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, **Zheng Wang**, Sheng Wang. "Effectively Learning Spatial Indexes with a Support for Updates", In Proceedings of the 2023 ACM SIGMOD International Conference on Management of Data (**SIGMOD 2023**), Regular Paper, Accepted.
- Qianru Zhang*, **Zheng Wang***, Cheng Long, Chao Huang, Siu-Ming Yiu, Yiding Liu, Gao Cong, Jieming Shi, "Online Anomalous Subtrajectory Detection on Road Networks with Deep Reinforcement Learning". The 39th IEEE International Conference on Data Engineering (**ICDE 2023**), Regular Paper, Accepted.
- **Zheng Wang**, Mingrui Liu, Cheng Long, Qianru Zhang, Jiangneng Li, Chunyan Miao. "On Inferring User Socioeconomic Status with Mobility Records", The 2022 IEEE International Conference on Big Data (**BigData 2022**), Regular Paper, Accepted.
- Qianru Zhang*, **Zheng Wang***, Cheng Long, Siu-Ming Yiu. "On Predicting and Generating a Good Break Shot in Billiards Sports", In SIAM International Conference on Data Mining (**SDM 2022**), Pages 109-117.
- **Zheng Wang**, Cheng Long, Gao Cong, Qianru Zhang. "Error-Bounded Online Trajectory Simplification with Multi-agent Reinforcement Learning", In Proceedings of the 27th SIGKDD conference on Knowledge Discovery and Data Mining (**KDD 2021**), Pages 1758-1768.

- Fei Li, **Zheng Wang**#, Siu Cheung Hui, Lejian Liao, Dandan Song, Jing Xu, Guoxiu He, Meihuizi Jia. "Modularized Interaction Network for Named Entity Recognition", The 59th Annual Meeting of the Association for Computational Linguistics (**ACL 2021**), Pages 200-209.

- **Zheng Wang**, Cheng Long, Gao Cong. "Trajectory Simplification with Reinforcement Learning", The 37th IEEE International Conference on Data Engineering (**ICDE 2021**), Pages 684-695.

- Fei Li, **Zheng Wang**#, Siu Cheung Hui, Lejian Liao, Dandan Song, Jing Xu. "Effective Named Entity Recognition with Boundary-aware Bidirectional Neural Networks", In Proceedings of The 2021 Web Conference (**WWW 2021**), Pages 1695-1703.

- **Zheng Wang**, Cheng Long, Gao Cong, Yiding Liu. "Effective and Efficient Subtrajectory Similarity Computation with Deep Reinforcement Learning", In Proceedings of the VLDB Endowment (**PVLDB 2020**), Volume 13, Number 12, Pages 2312-2325.

- Ce Ju*, **Zheng Wang***, Cheng Long, Xiaoyu Zhang, Dong Eui Chang. "Interaction-aware Kalman Neural Networks for Trajectory Prediction", IEEE Intelligent Vehicles Symposium (**IEEE IV 2020**), Pages 1793-1800.

- **Zheng Wang**, Cheng Long, Gao Cong, Ce Ju. "Effective and Efficient Sports Play Retrieval with Deep Representation Learning", In Proceedings of the 25th SIGKDD conference on Knowledge Discovery and Data Mining (**KDD 2019**), Pages 499-509.

- Jia Zhang, **Zheng Wang**, Qian Li, Jialin Zhang, Yanyan Lan, Qiang Li, Xiaoming Sun. "Efficient Delivery Policy to Minimize User Traffic Consumption in Guaranteed Advertising", In Proceedings of the 31st AAAI Conference (**AAAI 2017**), Pages 252–258.

International Journals

- **Zheng Wang**, Cheng Long, Gao Cong. "Similar Sports Play Retrieval with Deep Reinforcement Learning", IEEE Transactions on Data Engineering (**TKDE 2021**), Volume 1: Regular Paper, Accepted.
- Yixiang Fang, **Zheng Wang**, Reynold Cheng, Xiaodong Li, Siqiang Luo, Jiafeng Hu, Xiaojun Chen. "On Spatial-Aware Community Search", IEEE Transactions on Data Engineering (**TKDE 2018**), Volume 31, Number 4, Pages 783-798.

Appendix - Additional Proofs

Quality Analysis of the SizeS Algorithm in Section 6.3.1.2

There usually exists a space, within which the objects move. Therefore, we assume the points of trajectories are all located in a $d_{max} \times d_{max}$ rectangle, where d_{max} is a large number that captures the extent of the space. We use a coordinate system whose origin is at the middle of this rectangle.

Case 1: DTW. Consider a problem input with a query trajectory with m points $T_q =< p'_1, p'_2, ..., p'_m >$ and a data trajectory with $n = m^2$ points $T =< p_{1,1}, p_{1,2}, ..., p_{1,m}, p_{2,1}, p_{2,2}, ..., p_{2,m}, ..., p_{m,1}, p_{m,2}, p_{m,m} >$. We assume that m is an even number and $l = m/2$. In addition, we let $d = d_{max}/m$. The locations of the points in these trajectories are provided as follows.

- $p'_i = (-(l - i + 1/2) \cdot d, 0)$ for $i = 1, 2, ..., l$;
- $p'_i = ((i - l - 1/2) \cdot d, 0)$ for $i = l + 1, l + 2, ..., m$;
- $p_{i,j}$'s $(1 \leq j \leq m)$ are evenly located at the circle with its center at p'_i and its radius equal to ϵ, where ϵ is a very small real number for $i = 1, 2, ..., m$;

Consider the optimal solution. Its DTW distance, denoted by D_o, is at most the distance between T and T_q, which is equal to $m^2 \cdot \epsilon$ (points $p_{i,j}$ for $1 \leq j \leq m$ are aligned with point p'_i for $1 \leq i \leq m$). That is, we have $D_o \leq m^2 \cdot \epsilon$.

Consider the approximate solution returned by SizeS. Suppose that $\xi = 0$. That is, we only consider subtrajectories with the length exactly equal to m. We further know that each such sub-trajectory consists of points, either all located at a circle with the center at one of the point of T_q or some located at a circle with the center at a point

p'_i and others located at a circle with the center at a point p'_{i+1} $(1 \leq i < m)$. It could be verified that among these subtrajectories, the one with some located at the circle with the center at p'_l and others at p'_{l+1} has the smallest DTW distance from T_q and would be returned. We denote its DTW distance to T_q by D_a. It could be verified that $D_a > 2 \times (\Sigma_{i=1}^{l-1}((l-i) \cdot d - \epsilon) + \epsilon)$, where the lower bound of the Euclidean distance from a point of T_q to its aligned point of the returned subtrajectory is (1) $((l-i) \cdot d - \epsilon)$ for $1 \leq i \leq l-1$ and (2) ϵ for $i = l$; and (3) that of the point symmetric to p'_i (w.r.t. the origin) for $l+1 \leq i \leq m$ due to the symmetry. Therefore, we have

$$\begin{aligned}\frac{D_a}{D_o} &\geq \frac{2 \times (\Sigma_{i=1}^{l-1}((l-i) \cdot d - \epsilon) + \epsilon)}{m^2 \cdot \epsilon} \\ &= \frac{m^2/4 \cdot d - m/2 \cdot d - m \cdot \epsilon + 4 \cdot \epsilon}{m^2 \cdot \epsilon} \\ &= \frac{1/4 \cdot d - 1/(2m) \cdot d - \epsilon/m + 4/m^2 \cdot \epsilon}{\epsilon}\end{aligned}$$

which approaches infinity when m approaches infinity and ϵ approaches 0. In summary, the solution returned by SizeS could be arbitrarily worse than the optimal one when DTW distance is used.

Case 2: Frechet and t2vec. Consider the case when Frechet distance is used. The optimal solution and the approximate solution returned would be the sames as the case when DTW distance is used, but the distances would be different. D_o would be equal to ϵ and D_a would be at least $((l-1) \cdot d - \epsilon)$. Therefore, we have

$$\frac{D_a}{D_o} \geq \frac{(l-1) \cdot d - \epsilon}{\epsilon}$$

which approaches infinity when ϵ approaches 0.

Consider the case when t2vec is used. Since it is a learning-based distance - in theory, it may reduce to any possible distance metric such as DTW and Frechet. Thus, the analysis for DTW or Frechet could be carried over for t2vec.

Quality Analysis of Spliting-based Algorithms in Section 6.3.1.3

Case 1: PSS with DTW. Consider a problem input with a data trajectory with $n+3$ points $T =< p'_1, p'_2, p_1, p_2, ..., p_n, p'_3 >$ (n is a positive integer) and a query trajectory

$T_q =< p' >$. Let $d = d_{max}/2$. The locations of the points in these trajectories are provided as follows.

- $p'_1 = (-d/2, 0)$, $p'_2 = (-d, 0)$;
- $p_i = (0, 0)$ for $i = 1, 2, ..., n$;
- $p'_3 = (d, 0)$;
- $p' = (0, \epsilon)$, where ϵ is a very small non-negative real number;

Consider the optimal solution. It could be any subtrajectory $< p_i >$ $(1 \leq i \leq n)$ and the corresponding DTW distance is equal to ϵ, which we denote by D_o.

Consider the approximate solution returned by PSS. It is the subtrajectory $< p'_1 >$, which is explained as follows. When it scans the first point p'_1, it would split the trajectory at p'_1 and update the best-known subtrajectory to be $< p'_1 >$ with the DTW distance equal to $\sqrt{d^2/4 + \epsilon^2}$. It then continues to scan the following points $p'_2, p_1, ..., p_n, p'_3$ and would not perform any split operations at these points due to the fact that p'_2 and p'_3 are farther away from p' than p'_1. As a result, $< p'_1 >$ would be returned as a solution. We denote the DTW distance of this solution by D_a, i.e., $D_a = \sqrt{d^2/4 + \epsilon^2}$.

Consider the approximation ratio (AR). We have

$$\frac{D_a}{D_o} = \frac{\sqrt{d^2/4 + \epsilon^2}}{\epsilon} > \frac{d/2}{\epsilon}$$

which approaches infinity when ϵ approaches zero.

Consider the mean rank (MR). We know that the rank of the approximate solution $< p'_1 >$ is at least $\frac{n(n+1)}{2} + 1$ since any subtrajectory of $< p_1, p_2, ..., p_n >$ has a smaller DTW distance than $< p'_1 >$ (assuming $\epsilon = 0$). Therefore, the mean rank would approach infinity when n approaches infinity.

Consider the relative rank (RR). Based on the analysis of mean rank, we know that the relative rank of the approximate solution $< p'_1 >$ is at least $\frac{\frac{n(n+1)}{2}+1}{\frac{(n+3)(n+4)}{2}}$, which approaches 1 when n approaches infinity.

In conclusion, the solution returned by PSS could be arbitrarily worse than the optimal one in terms of AR, MR, and RR, when the DTW distance is used.

Case 2: Other Algorithms and Similarity Measurements Consider the other algorithms, i.e., POS and POS-D. It could be verified that they would run exactly in the same way as PSS on the problem input provided in the Case 1. Therefore, the conclusion would be carried over. Consider the other measurements, namely, Frechet and t2vec. For Frechet, it could be verified that the optimal solution and the approximate solution returned by the algorithms are the same as those in the Case 1 and their Frechet distances and DTW distances to T_q are equal since both solutions and T_q involve one single point. As a result, the conclusion for DTW distance could be carried over for Frechet distance. For t2vec, since it is a learning-based distance - in theory, it may reduce to any possible distance metric such as DTW and Frechet. Thus, the analysis for DTW in the Case 1 could be carried over for t2vec.

Appendix - Code and Data

The codes and datasets in this thesis can be downloaded via the following links.

- Chapter 3
  `https://github.com/zhengwang125/EB-OTS`
- Chapter 4
  `https://github.com/zhengwang125/RLTS`
- Chapter 5
  `https://www.dropbox.com/sh/ui2yegn2wmok8hs/AAAXRfeKH1R7AHATmMmsW3Cga?dl=0`
- Chapter 6
  `https://github.com/zhengwang125/SimSub`
- Chapter 7
  `https://github.com/zhengwang125/play2vec`
- Chapter 8
  `https://github.com/zhengwang125/SimPlay`
- Chapter 9
  `https://github.com/zhengwang125/DeepSEI`

References

[1] Haitao Yuan, Guoliang Li, Zhifeng Bao, and Ling Feng. An effective joint prediction model for travel demands and traffic flows. In *ICDE*, pages 348–359. IEEE, 2021.

[2] Shanshan Feng, Lucas Vinh Tran, Gao Cong, Lisi Chen, Jing Li, and Fan Li. Hme: A hyperbolic metric embedding approach for next-poi recommendation. In *SIGIR*, pages 1429–1438, 2020.

[3] Torsten Seemann, Courtney R Lane, Norelle L Sherry, Sebastian Duchene, Anders Gonçalves da Silva, Leon Caly, Michelle Sait, Susan A Ballard, Kristy Horan, Mark B Schultz, et al. Tracking the covid-19 pandemic in australia using genomics. *Nature communications*, 11(1):1–9, 2020.

[4] Chao Huang, Junbo Zhang, Yu Zheng, and Nitesh V Chawla. Deepcrime: Attentive hierarchical recurrent networks for crime prediction. In *CIKM*, pages 1423–1432, 2018.

[5] Dongxiang Zhang, Mengting Ding, Dingyu Yang, Yi Liu, Ju Fan, and Heng Tao Shen. Trajectory simplification: an experimental study and quality analysis. *PVLDB*, 11(9):934–946, 2018.

[6] Sheng Wang, Zhifeng Bao, J Shane Culpepper, and Gao Cong. A survey on trajectory data management, analytics, and learning. *CSUR*, 54(2):1–36, 2021.

[7] Zheng Wang, Cheng Long, and Gao Cong. Trajectory simplification with reinforcement learning. In *ICDE*, 2021.

[8] Nirvana Meratnia and A Rolf. Spatiotemporal compression techniques for moving point objects. In *EDBT*, pages 765–782. Springer, 2004.

[9] Xuelian Lin, Jiahao Jiang, Shuai Ma, Yimeng Zuo, and Chunming Hu. One-pass trajectory simplification using the synchronous euclidean distance. *VLDBJ*, 28(6): 897–921, 2019.

[10] Jiajun Liu, Kun Zhao, Philipp Sommer, Shuo Shang, Brano Kusy, and Raja Jurdak. Bounded quadrant system: Error-bounded trajectory compression on the go. In *ICDE*, pages 987–998. IEEE, 2015.

[11] Jiajun Liu, Kun Zhao, Philipp Sommer, Shuo Shang, Brano Kusy, Jae-Gil Lee, and Raja Jurdak. A novel framework for online amnesic trajectory compression in resource-constrained environments. *IEEE TKDE*, 28(11):2827–2841, 2016.

[12] Xuelian Lin, Shuai Ma, Han Zhang, Tianyu Wo, and Jinpeng Huai. One-pass error bounded trajectory simplification. *PVLDB*, 10(7):841–852, 2017.

[13] Cheng Long, Raymond Chi-Wing Wong, and HV Jagadish. Direction-preserving trajectory simplification. *PVLDB*, 6(10):949–960, 2013.

[14] Bingqing Ke, Jie Shao, Yi Zhang, Dongxiang Zhang, and Yang Yang. An online approach for direction-based trajectory compression with error bound guarantee. In *APWeb*, pages 79–91. Springer, 2016.

[15] Bingqing Ke, Jie Shao, and Dongxiang Zhang. An efficient online approach for direction-preserving trajectory simplification with interval bounds. In *MDM*, pages 50–55. IEEE, 2017.

[16] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *IJRR*, 32(11):1238–1274, 2013.

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[18] Martin L Puterman. *Markov Decision Processes.: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 2014.

[19] Michalis Potamias, Kostas Patroumpas, and Timos Sellis. Sampling trajectory streams with spatiotemporal criteria. In *SSDBM*, pages 275–284. IEEE, 2006.

[20] Jonathan Muckell, Jeong-Hyon Hwang, Vikram Patil, Catherine T Lawson, Fan Ping, and SS Ravi. Squish: an online approach for gps trajectory compression. In *Proceedings of the 2nd International Conference on Computing for Geospatial Research & Applications*, page 13. ACM, 2011.

[21] Jonathan Muckell, Paul W Olsen, Jeong-Hyon Hwang, Catherine T Lawson, and SS Ravi. Compression of trajectory data: a comprehensive evaluation and new approach. *GeoInformatica*, 18(3):435–460, 2014.

[22] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[23] Andrew Ilyas, Logan Engstrom, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. A closer look at deep policy gradients. *International Conference on Learning Representations*, 2020.

[24] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[25] Hu Cao, Ouri Wolfson, and Goce Trajcevski. Spatio-temporal data reduction with deterministic error bounds. *The VLDB Journal*, 15(3):211–228, 2006.

[26] Xuelian Lin, Shuai Ma, Jiahao Jiang, Yanchen Hou, and Tianyu Wo. Error bounded line simplification algorithms for trajectory compression: An experimental evaluation. *TODS*, 46(3):1–44, 2021.

[27] John Edward Hershberger and Jack Snoeyink. *Speeding up the Douglas-Peucker line-simplification algorithm*. University of British Columbia, Department of Computer Science, 1992.

[28] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. An online algorithm for segmenting time series. In *ICDM*, pages 289–296. IEEE, 2001.

[29] Cheng Long, Raymond Chi-Wing Wong, and HV Jagadish. Trajectory simplification: on minimizing the direction-based error. *PVLDB*, 8(1):49–60, 2014.

[30] Pankaj K Agarwal, Kyle Fox, Kamesh Munagala, Abhinandan Nath, Jiangwei Pan, and Erin Taylor. Subtrajectory clustering: Models and algorithms. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 75–87. ACM, 2018.

[31] Jae-Gil Lee, Jiawei Han, and Kyu-Young Whang. Trajectory clustering: a partition-and-group framework. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 593–604. ACM, 2007.

[32] Kevin Buchin, Maike Buchin, Joachim Gudmundsson, Maarten Löffler, and Jun Luo. Detecting commuting patterns by clustering subtrajectories. *International Journal of Computational Geometry & Applications*, 21(03):253–282, 2011.

[33] Lei Chen and Raymond Ng. On the marriage of lp-norms and edit distance. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 792–803. VLDB Endowment, 2004.

[34] Lei Chen, M Tamer Özsu, and Vincent Oria. Robust and fast similarity search for moving object trajectories. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 491–502. ACM, 2005.

[35] Min Xie. Eds: a segment-based distance measure for sub-trajectory similarity search. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 1609–1610. ACM, 2014.

[36] Sayan Ranu, P Deepak, Aditya D Telang, Prasad Deshpande, and Sriram Raghavan. Indexing and matching trajectories under inconsistent sampling rates. In *2015 IEEE 31st International Conference on Data Engineering*, pages 999–1010. IEEE, 2015.

[37] Xiucheng Li, Kaiqi Zhao, Gao Cong, Christian S Jensen, and Wei Wei. Deep representation learning for trajectory similarity computation. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 617–628. IEEE, 2018.

[38] Byoung-Kee Yi, HV Jagadish, and Christos Faloutsos. Efficient retrieval of similar time sequences under time warping. In *ICDE*, pages 201–208. IEEE, 1998.

[39] Panagiotis Tampakis, Christos Doulkeridis, Nikos Pelekis, and Yannis Theodoridis. Distributed subtrajectory join on massive datasets. *arXiv preprint arXiv:1903.07748*, 2019.

[40] Panagiotis Tampakis, Nikos Pelekis, Christos Doulkeridis, and Yannis Theodoridis. Scalable distributed subtrajectory clustering. *arXiv preprint arXiv:1906.06956*, 2019.

[41] Long Sha, Patrick Lucey, Yisong Yue, Peter Carr, Charlie Rohlf, and Iain Matthews. Chalkboarding: A new spatiotemporal query paradigm for sports play retrieval. In *Proceedings of the 21st International Conference on Intelligent User Interfaces*, pages 336–347, 2016.

[42] Zheng Wang, Cheng Long, Gao Cong, and Ce Ju. Effective and efficient sports play retrieval with deep representation learning. In *SIGKDD*, pages 499–509, 2019.

[43] Tom Decroos, Jan Van Haaren, and Jesse Davis. Automatic discovery of tactics in spatio-temporal soccer match data. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 223–232. ACM, 2018.

[44] Younghoon Kim and Kyuseok Shim. Efficient top-k algorithms for approximate substring matching. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 385–396. ACM, 2013.

[45] Zhenjie Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, and Divesh Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 915–926. ACM, 2010.

[46] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[47] Zheng Wang, Cheng Long, Gao Cong, and Yiding Liu. Efficient and effective similar subtrajectory search with deep reinforcement learning. *PVLDB*, 13(12):2312–2325, 2020.

[48] Mahmut Kaya and Hasan Şakir Bilge. Deep metric learning: A survey. *Symmetry*, 11(9):1066, 2019.

[49] Yuan Zhong, Nicholas Jing Yuan, Wen Zhong, Fuzheng Zhang, and Xing Xie. You are where you go: Inferring demographic attributes from location check-ins. In *Proceedings of the eighth ACM international conference on web search and data mining*, pages 295–304, 2015.

[50] Christopher Riederer. *Location Data: Perils, Profits, Promise.* Columbia University, 2020.

[51] Yang Zhang and Tao Cheng. A deep learning approach to infer employment status of passengers by using smart card data. *IEEE Transactions on Intelligent Transportation Systems*, 21(2):617–629, 2019.

[52] Lun Wu, Liu Yang, Zhou Huang, Yaoli Wang, Yanwei Chai, Xia Peng, and Yu Liu. Inferring demographics from human trajectories and geographical context. *Computers, Environment and Urban Systems*, 77:101368, 2019.

[53] Yang Xu, Alexander Belyi, Iva Bojic, and Carlo Ratti. Human mobility and socioeconomic status: Analysis of singapore and boston. *Computers, Environment and Urban Systems*, 72:51–67, 2018.

[54] Shichang Ding, Hong Huang, Tao Zhao, and Xiaoming Fu. Estimating socioeconomic status via temporal-spatial mobility analysis-a case study of smart card data. In *ICCCN*, pages 1–9. IEEE, 2019.

[55] Hengfeng Li, Lars Kulik, and Kotagiri Ramamohanarao. Spatio-temporal trajectory simplification for inferring travel paths. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 63–72, 2014.

[56] Richard Bellman. On the approximation of curves by line segments using dynamic programming. *Communications of the ACM*, 4(6):284, 1961.

[57] David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: the international journal for geographic information and geovisualization*, 10(2): 112–122, 1973.

[58] Yunheng Han, Weiwei Sun, and Baihua Zheng. Compress: A comprehensive framework of trajectory compression in road networks. *TODS*, 42(2):1–49, 2017.

[59] Renchu Song, Weiwei Sun, Baihua Zheng, and Yu Zheng. Press: A novel framework of trajectory compression in road networks. *Proceedings of the VLDB Endowment*, 7(9):661–672, 2014.

[60] Tianyi Li, Ruikai Huang, Lu Chen, Christian S Jensen, and Torben Bach Pedersen. Compression of uncertain trajectories in road networks. *PVLDB*, 13(7):1050–1063, 2020.

[61] Tianyi Li, Lu Chen, Christian S Jensen, and Torben Bach Pedersen. Trace: real-time compression of streaming trajectories in road networks. *Proceedings of the VLDB Endowment*, 14(7):1175–1187, 2021.

[62] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[63] Yu Zheng, Xing Xie, Wei-Ying Ma, et al. Geolife: A collaborative social networking service among user, location and trajectory. *IEEE Data Eng. Bull.*, 33(2):32–39, 2010.

[64] Shuang Wang and Hakan Ferhatosmanoglu. Ppq-trajectory: Spatio-temporal quantization for querying in large trajectory repositories. *PVLDB*, 14(2):215–227, 2020.

[65] Helmut Alt and Michael Godau. Computing the fréchet distance between two polygonal curves. *International Journal of Computational Geometry & Applications*, 5(01n02):75–91, 1995.

[66] Sheng Wang, Zhifeng Bao, J Shane Culpepper, Zizhe Xie, Qizhi Liu, and Xiaolin Qin. Torch: A search engine for trajectory data. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 535–544. ACM, 2018.

[67] Haitao Yuan and Guoliang Li. Distributed in-memory trajectory similarity search and join on road network. In *2019 IEEE 35th international conference on data engineering (ICDE)*, pages 1262–1273. IEEE, 2019.

[68] Sheng Wang, Zhifeng Bao, J Shane Culpepper, Timos Sellis, and Xiaolin Qin. Fast large-scale trajectory clustering. *PVLDB*, 13(1):29–42, 2019.

[69] Di Yao, Gao Cong, Chao Zhang, and Jingping Bi. Computing trajectory similarity in linear time: A generic seed-guided neural metric learning approach. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1358–1369. IEEE, 2019.

[70] Chunyang Ma, Hua Lu, Lidan Shou, and Gang Chen. Ksq: Top-k similarity query on uncertain trajectories. *IEEE Transactions on Knowledge and Data Engineering*, 25(9):2049–2062, 2012.

[71] Eamonn Keogh and Chotirat Ann Ratanamahatana. Exact indexing of dynamic time warping. *Knowledge and information systems*, 7(3):358–386, 2005.

[72] Michail Vlachos, George Kollios, and Dimitrios Gunopulos. Discovering similar multidimensional trajectories. In *Proceedings 18th international conference on data engineering*, pages 673–684. IEEE, 2002.

[73] Thanawin Rakthanmanon, Bilson Campana, Abdullah Mueen, Gustavo Batista, Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 262–270. ACM, 2012.

[74] Thanawin Rakthanmanon, Bilson Campana, Abdullah Mueen, Gustavo Batista, Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn Keogh. Addressing big data time series: Mining trillions of time series subsequences under dynamic time warping. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 7(3): 10, 2013.

[75] Xueyuan Gong, Simon Fong, and Yain-Whar Si. Fast fuzzy subsequence matching algorithms on time-series. *Expert Systems with Applications*, 116:275–284, 2019.

[76] Vassilis Athitsos, Panagiotis Papapetrou, Michalis Potamias, George Kollios, and Dimitrios Gunopulos. Approximate embedding-based subsequence matching of time series. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 365–378. ACM, 2008.

[77] Wook-Shin Han, Jinsoo Lee, Yang-Sae Moon, and Haifeng Jiang. Ranked subsequence matching in time-series databases. In *Proceedings of the 33rd international conference on Very large data bases*, pages 423–434. VLDB Endowment, 2007.

[78] Yang-Sae Moon, Kyu-Young Whang, and Woong-Kee Loh. Duality-based subsequence matching in time-series databases. In *Proceedings 17th International Conference on Data Engineering*, pages 263–272. IEEE, 2001.

[79] Christos Faloutsos, Mudumbai Ranganathan, and Yannis Manolopoulos. *Fast subsequence matching in time-series databases*, volume 23. ACM, 1994.

[80] Sanghyun Park, Wesley W Chu, Jeehee Yoon, and Chihcheng Hsu. Efficient searches for similar subsequences of different lengths in sequence databases. In *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*, pages 23–32. IEEE, 2000.

[81] Sang-Wook Kim, Sanghyun Park, and Wesley W Chu. An index-based approach for similarity search supporting time warping in large sequence databases. In *Proceedings 17th International Conference on Data Engineering*, pages 607–614. IEEE, 2001.

[82] Dan He, Boyu Ruan, Bolong Zheng, and Xiaofang Zhou. *Trajectory Set Similarity Measure: An EMD-Based Approach*, pages 28–40. 05 2018. ISBN 978-3-319-92012-2. doi: 10.1007/978-3-319-92013-9_3.

[83] Long Sha, Patrick Lucey, Yisong Yue, Xinyu Wei, Jennifer Hobbs, Charlie Rohlf, and Sridha Sridharan. Interactive sports analytics: An intelligent interface for utilizing trajectories for interactive sports play retrieval and analytics. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 25(2):13, 2018.

[84] Susan Hanson and Perry Hanson. Gender and urban activity patterns in uppsala, sweden. *Geographical Review*, pages 291–299, 1980.

[85] Susan Hanson and Perry Hanson. The travel-activity patterns of urban residents: dimensions and relationships to sociodemographic characteristics. *Economic geography*, 57(4):332–347, 1981.

[86] Susan Hanson. The determinants of daily travel-activity patterns: relative location and sociodemographic factors. *Urban Geography*, 3(3):179–202, 1982.

[87] Mei-Po Kwan. Gender, the home-work link, and space-time patterns of nonemployment activities. *Economic geography*, 75(4):370–394, 1999.

[88] Narisra Limtanakool, Martin Dijst, and Tim Schwanen. The influence of socioeconomic characteristics, land use and travel time considerations on mode choice for medium-and longer-distance trips. *Journal of transport geography*, 14(5):327–341, 2006.

[89] Yang Xu, Shih-Lung Shaw, Ziliang Zhao, Ling Yin, Zhixiang Fang, and Qingquan Li. Understanding aggregate human mobility patterns using passive mobile phone location data: a home-based approach. *Transportation*, 42(4):625–646, 2015.

[90] Joshua Blumenstock, Gabriel Cadamuro, and Robert On. Predicting poverty and wealth from mobile phone metadata. *Science*, 350(6264):1073–1076, 2015.

[91] Qunying Huang and David WS Wong. Activity patterns, socioeconomic status and urban spatial structure: what can social media data tell us? *International Journal of Geographical Information Science*, 30(9):1873–1898, 2016.

[92] Daniel Kelly, Barry Smyth, and Brian Caulfield. Uncovering measurements of social and demographic behavior from smartphone location data. *IEEE Transactions on Human-Machine Systems*, 43(2):188–198, 2013.

[93] Anna Monreale, Fabio Pinelli, Roberto Trasarti, and Fosca Giannotti. Wherenext: a location predictor on trajectory pattern mining. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 637–646, 2009.

[94] Chao Zhang, Jiawei Han, Lidan Shou, Jiajun Lu, and Thomas La Porta. Splitter: Mining fine-grained sequential patterns in semantic trajectories. *Proceedings of the VLDB Endowment*, 7(9):769–780, 2014.

[95] Yile Chen, Cheng Long, Gao Cong, and Chenliang Li. Context-aware deep model for joint mobility and time prediction. In *Proceedings of the 13th International Conference on Web Search and Data Mining*, pages 106–114, 2020.

[96] Jie Feng, Yong Li, Chao Zhang, Funing Sun, Fanchao Meng, Ang Guo, and Depeng Jin. Deepmove: Predicting human mobility with attentional recurrent networks. In *Proceedings of the 2018 world wide web conference*, pages 1459–1468, 2018.

[97] Yingzi Wang, Nicholas Jing Yuan, Defu Lian, Linli Xu, Xing Xie, Enhong Chen, and Yong Rui. Regularity and conformity: Location prediction using heterogeneous mobility data. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1275–1284, 2015.

[98] Shuai Xiao, Mehrdad Farajtabar, Xiaojing Ye, Junchi Yan, Le Song, and Hongyuan Zha. Wasserstein learning of deep generative point process models. *arXiv preprint arXiv:1705.08051*, 2017.

[99] Shuai Xiao, Junchi Yan, Xiaokang Yang, Hongyuan Zha, and Stephen Chu. Modeling the intensity function of point process via recurrent neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[100] Nan Du, Hanjun Dai, Rakshit Trivedi, Utkarsh Upadhyay, Manuel Gomez-Rodriguez, and Le Song. Recurrent marked temporal point processes: Embedding event history to vector. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1555–1564, 2016.

[101] Junchi Yan, Xin Liu, Liangliang Shi, Changsheng Li, and Hongyuan Zha. Improving maximum likelihood estimation of temporal point process via discriminative and adversarial learning. In *IJCAI*, pages 2948–2954, 2018.

[102] Sung Nok Chiu, Dietrich Stoyan, Wilfrid S Kendall, and Joseph Mecke. *Stochastic geometry and its applications*. John Wiley & Sons, 2013.

[103] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[104] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, pages 415–432. ACM, 2019.

[105] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *PVLDB*, 12(12):2118–2130, 2019.

[106] Immanuel Trummer, Samuel Moseley, Deepak Maram, Saehan Jo, and Joseph Antonakakis. Skinnerdb: regret-bounded query evaluation via reinforcement learning. *PVLDB*, 11(12):2074–2077, 2018.

[107] Yansheng Wang, Yongxin Tong, Cheng Long, Pan Xu, Ke Xu, and Weifeng Lv. Adaptive dynamic bipartite graph matching: A reinforcement learning approach. In *ICDE*, pages 1478–1489. IEEE, 2019.

[108] Ronen I Brafman and Moshe Tennenholtz. R-max-a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3(Oct):213–231, 2002.

[109] Michael Kearns and Satinder Singh. Near-optimal reinforcement learning in polynomial time. *Machine learning*, 49(2-3):209–232, 2002.

[110] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4): 279–292, 1992.

[111] Kaixiang Lin, Renyu Zhao, Zhe Xu, and Jiayu Zhou. Efficient large-scale fleet management via multi-agent deep reinforcement learning. In *SIGKDD*, pages 1774–1783, 2018.

[112] Hyun-Rok Lee and Taesik Lee. Improved cooperative multi-agent reinforcement learning algorithm augmented by mixing demonstrations from centralized policy. In *AAMAS*, pages 1089–1098, 2019.

[113] Yexin Li, Yu Zheng, and Qiang Yang. Efficient and effective express via contextual cooperative reinforcement learning. In *SIGKDD*, pages 510–519, 2019.

[114] Yexin Li, Yu Zheng, and Qiang Yang. Cooperative multi-agent reinforcement learning in express system. In *CIKM*, pages 805–814, 2020.

[115] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[116] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International Conference on Machine Learning*, pages 1188–1196, 2014.

[117] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

[118] Zheng Wang, Ce Ju, Gao Cong, and Cheng Long. Representation learning for spatial graphs. *arXiv preprint arXiv:1812.06668*, 2018.

[119] Zheng Wang, Cheng Long, Gao Cong, and Qianru Zhang. Error-bounded online trajectory simplification with multi-agent reinforcement learning. In *SIGKDD*, 2021.

[120] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[121] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[122] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[123] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[124] Yasushi Sakurai, Christos Faloutsos, and Masashi Yamamuro. Stream monitoring under the time warping distance. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 1046–1055. IEEE, 2007.

[125] Abdullah Mueen and Eamonn Keogh. Extracting optimal performance from dynamic time warping. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2129–2130, 2016.

[126] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[127] Han Su, Kai Zheng, Haozhou Wang, Jiamin Huang, and Xiaofang Zhou. Calibrating trajectory data for similarity-based analysis. In *Proceedings of the 2013 ACM SIGMOD international conference on management of data*, pages 833–844. ACM, 2013.

[128] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. The earth mover's distance as a metric for image retrieval. *International journal of computer vision*, 40(2): 99–121, 2000.

[129] Naomi S Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[130] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[131] Homelink. `https://www.lianjia.com`.

[132] Amap Map Service. `https://lbs.amap.com/api/webservice/summary/`.

[133] Balázs Cs Csáji, Arnaud Browet, Vincent A Traag, Jean-Charles Delvenne, Etienne Huens, Paul Van Dooren, Zbigniew Smoreda, and Vincent D Blondel. Exploring the mobility of mobile phone users. *Physica A: statistical mechanics and its applications*, 392(6):1459–1473, 2013.

[134] Santi Phithakkitnukoon, Zbigniew Smoreda, and Patrick Olivier. Socio-geography of human mobility: A study using longitudinal mobile phone data. *PloS one*, 7(6): e39253, 2012.

[135] Yu Zheng. Trajectory data mining: an overview. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(3):1–41, 2015.

[136] Quannan Li, Yu Zheng, Xing Xie, Yukun Chen, Wenyu Liu, and Wei-Ying Ma. Mining user similarity based on location history. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, pages 1–10, 2008.

[137] Joachim Scheiner. The gendered complexity of daily life: effects of life-course events on changes in activity entropy and tour complexity over time. *Travel Behaviour and Society*, 1(3):91–105, 2014.

[138] Luca Pappalardo, Dino Pedreschi, Zbigniew Smoreda, and Fosca Giannotti. Using big data to study the link between human mobility and socio-economic development. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 871–878. IEEE, 2015.

[139] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[140] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[141] Fei Wu and Zhenhui Li. Where did you go: Personalized annotation of mobility records. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 589–598, 2016.

[142] Zhixian Yan, Dipanjan Chakraborty, Christine Parent, Stefano Spaccapietra, and Karl Aberer. Semantic trajectories: Mobility data computation and annotation. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 4(3):1–38, 2013.

[143] Zhixian Yan, Dipanjan Chakraborty, Christine Parent, Stefano Spaccapietra, and Karl Aberer. Semitri: a framework for semantic annotation of heterogeneous trajectories. In *Proceedings of the 14th international conference on extending database technology*, pages 259–270, 2011.

[144] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*, pages 489–504, 2018.

[145] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. Lisa: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, pages 2119–2133, 2020.

[146] Jianzhong Qi, Guanli Liu, Christian S Jensen, and Lars Kulik. Effectively learning spatial indices. *Proceedings of the VLDB Endowment*, 13(12):2341–2354, 2020.