NEIST: A Neural-Enhanced Index for Spatio-Temporal Queries

Sai Wu, Zhifei Pang, Gang Chen, Yunjun Gao, Cenjiong Zhao, and Shili Xiang

Abstract—Previous work on the spatio-temporal index often adopts a simple linear model to predict the future positions of moving objects, which may generate numerous errors for complex road networks and fast moving objects. In this paper, we propose NEIST, a neural-enhanced index to process spatio-temporal queries with enhanced efficiency and accuracy, by intelligently leveraging the movement patterns among moving objects. NEIST applies a Recurrent Neural Network (RNN) model to predict future positions of moving objects based on observed trajectories. To reduce the prediction overhead, a suffix-tree is further built to index trajectories with similar suffixes, and thus similar objects within a given similarity bound are grouped together to share the same prediction result. A prediction result in NEIST represents possible positions of a group of moving objects in the next t time slots. Inside each time slot, traditional linear prediction model is then adopted and a TPR-Tree is built to support spatio-temporal queries. We use Singapore and Porto taxi trajectory datasets to evaluate NEIST. Compared to previous approaches, NEIST achieves a much more efficient query performance and is able to produce about 70 percent more accurate results.

Index Terms—Neural networks, spatio-temporal index, deep learning, query

1 Introduction

In spatio-temporal database, timestamp range query is widely used to answer questions like "return the objects within 1KM^2 after 10 minutes". Such queries can be expressed as a triplet $< t_s, t_e, R >$, where t_s is the query start time, t_e is the query end time and R is a multi-dimensional polygon. To process timestamp queries, a set of time parameterized indexes, such as TPR-Tree [1] and TPR*-Tree [2], are proposed. Those indexes are optimized for timestamp queries in $[t_c, t_c+ H]$, where t_c is the current update time and H is a parameter called the horizon, denoting how far the index can "see" in the future.

Such queries are widely used in Singapore's e-city project for smart traffic management. For example, the project will wisely adjust the traffic light cycle to avoid the traffic congestion, if we can precisely predict the future positions of vehicles. We simply increase the green light duration for the roads that may receive a high volume of traffic in the near future. The strategy can effectively reduce the possibility of traffic jam. We also employ the technique to manage taxis over the city. By predicting the future pedestrian density and taxi positions, we know where taxis are insufficient and can do some pre-scheduling. Some other applications are like the ERP[1] management in Singapore, MRT-train scheduling and pedestrian diversion. To predict the future positions of a moving object, previous approaches assume that the object follows its current direction and velocity. Such linear model, although significantly reducing the complexity of index maintenance and query processing, is unrealistic and leads to very inaccurate results in some circumstances, such as in the context of urban road networks. However, if we individually model each moving object with its own moving patterns, the cost of index maintenance will be so high that we cannot efficiently support timestamp queries over a large-scale moving object sets.

In this paper, we propose NEIST, a neural-enhanced index to process spatio-temporal queries. NEIST is originally designed to process timestamp queries for vehicles in road networks. So we use the road network application as an example to show our idea. The design philosophy of NEIST is based on the following considerations:

- In a road network, vehicles may show different moving patterns at different locations. It is impossible to use one or a few functions to catch all those patterns. On the other hand, deep neural network can simulate any complex functions with millions of parameters. It also provides a much more precise prediction, compared to other learning techniques.
- Vehicles may show similar trajectory patterns, if their last few positions are almost the same. This is because vehicles always prefer the "optimal" path (the shortest path or path with the fewest traffic lights) and the possible paths are constrained by the road networks. So we can group them together and reduce the prediction cost.
- Ad-hoc prediction VS snapshot prediction. Objects will report their current positions periodically. However, due to unstable transmission, the updates are

1. https://www.lta.gov.sg/content/ltaweb/en/roads-and-motoring/managing-traffic-and-congestion/electronic-road-pricing-erp.html

- *S. Wu, Z. Pang, G. Chen, Y. Gao, and C. Zhao are with the College of Computer Science, Zhejiang University, Hangzhou, Zhejiang 310027, China. E-mail: {wusai, zhifeipang, cg, gaoyj, zhcj}@zju.edu.cn.*
- *S. Xiang is with the Institute for Infocomm Research, A*STAR, Singapore 138632. E-mail: sxiang@i2r.a-star.edu.sg.*

Manuscript received 16 Dec. 2018; revised 21 Aug. 2019; accepted 2 Oct. 2019. Date of publication 7 Oct. 2019; date of current version 5 Mar. 2021. (Corresponding author: Zhifei Pang.)
Recommended for acceptance by G. Li.
Digital Object Identifier no. 10.1109/TKDE.2019.2945947

TABLE 1
Prediction Accuracy of Linear Model

Error Bound	t_1	t_2	t_3	t_4	t_5
0	22.5	13.7	9.1	6.5	5.5
1	44.1	28.2	18.8	14.7	11.9
2	57.6	37.2	25.6	20.1	16.3

TABLE 2
Prediction Accuracy of Seq2Seq Model

Error Bound	t_1	t_2	t_3	t_4	t_5
0	75.6	74.0	72.1	70.0	68.5
1	98.4	98.0	95.6	92.5	90.2
2	99.8	99.7	98.5	96.0	95.4

received at different frequencies. For example, in the taxi dataset, our vehicles are required to report their GPS locations every 2 minutes, but we may receive a new report after 10 minutes since the last one. In the approach of ad-hoc prediction, it is to update the prediction for every object when receiving a new report. This approach is inefficient and thus tends to generate the inaccurate prediction, because the prediction for each individual object via neural network is costly and the old prediction will drift away from the real situation if a new prediction is only invoked after a long time. So in the NEIST, we adopt the snapshot prediction. Namely, after a period, we will generate new predictions for all objects. This is a computation expensive operation, but it is well-supported by the parallelism of GPU. We manage to update the prediction of ten thousands objects in less than 0.03 second. Accordingly, our index applies the batch update strategy when new snapshot is generated.

In the NEIST, we predict the moving patterns using the Seq2Seq model [3] which is one of RNN variants. The input of our model is a vector of a moving object, denoting its current position, velocity, timestamp and other attributes. To simplify the prediction, we split the time into equal-size slots (like 2-minute slot). Each time, our model will generate a prediction for a specific object, indicating its possible positions in the next t time slots. We also partition the space into equal-size grids. For example, in Porto road map (18KM×48KM), we set the default grid size to be 250M×250M. If the predicted position and the ground-truth result are in the same grid, the prediction is correct. Tables 1 and 2 compare the prediction accuracy of linear model and Seq2Seq. We can see that for the next time slot t_1, Seq2Seq achieves 75.6 percent accuracy, higher than the linear model (22.5 percent). In reality, the prediction within vicinity is also valuable, especially considering the traffic lights and small grid size. So we relax our accuracy requirement by defining a parameter *error bound*. Error bound e allows the prediction result and ground truth to differ by at most e grids (in any axis). We observe that the results improve a lot even with error bound 1.

One disadvantage of using Seq2Seq model is that the training and prediction become intractable, if there are too many objects involved. However, based on the observation from the real taxi trajectories, we find: 1) in a small area, most trajectories can be classified into a few groups based on their similarities. This is because vehicles try to find the optimal path in the road network and few possible paths are left. 2) Objects sharing the last few GPS reports will get the same prediction for their future locations. This is a feature of the Seq2Seq model.

Motivated by the above findings, we propose a novel Seq2seq model enhanced with context information. To reduce prediction overhead, we adopt suffix tree to share prediction results among similar trajectories. Additionally, we devise a new snapshot-based index TS-TPR to process timestamp queries. Different from TPR-Tree where updates are processed one by one, our index adopts the batch update model and results in a better index structure and less accumulative update cost. We evaluate our approach using a real taxi dataset and compare it with TPR-Tree. The remaining part of the paper is organized as follows. Section 2 briefly reviews previous work. Section 3 introduces the details of our prediction model and index structure. Section 4 evaluates our approach and Section 5 concludes the paper.

2 Related Work

2.1 Indexes of Moving Objects

Indexing moving objects to support various types of queries has been a hot topic for years. Novel index structures for trajectory data were proposed [4], [5]. However, they were designed to answer queries for historical trajectories. Without the help of any prediction model, it is not clear how to use those indexes to support predictive queries. TPR-Tree [1] is the first one that supports predictive spatio-temporal queries. To further improve query performance, Tao et al. [2] proposed TPR*-Tree, a variant of TPR-Tree, which optimizes the index maintenance by applying different algorithms. As suggested by [6], the two indexes show almost the same performance when processing predictive queries. So in our paper, we use TPR-Tree as our baseline.

Many other works try to reuse B^+-Tree structure to index moving objects. Indexes, such as ST^2B-Tree [7], B^x-Tree [8], are built on top of B^+-Tree. They map the records of moving objects into 1-dimensional space and then index them through B^+-Tree structure. To the best of our knowledge, all mentioned indexes for moving objects are under assumption that objects follow a linear motion model, so their future positions can be calculated by using current locations and velocities. However, this assumption is obviously not valid for most real scenarios, because existing indexes fail to deliver a high-precision result for predictive query.

Tao [9] proposed a general framework for monitoring and indexing moving objects where each GPS device is required to generate an individual function (i.e., linear, quadratic, curving, etc.) to accurately capture its own movements. By collecting the functions and corresponding parameters, their index server can predict the moving patterns of each object. However, different GPS devices are composed of different hardware and interfaces. Due to a lack of computation capability, it is impractical to assume that they can generate a complicated movement function.

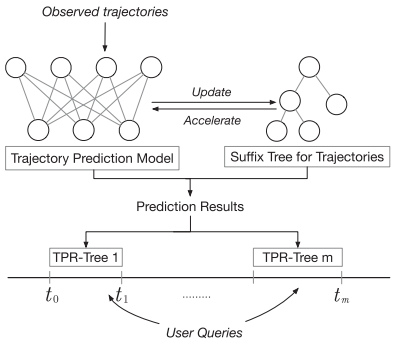


Fig. 1. The architecture of NEIST.

TABLE 3
Frequent Used Notions

Symbol	Definition
T (, T_o or T_s)	Trajectory
$\hat{T}$	Predicted trajectory by neural network
Φ_λ	Grid mapping function with granularity λ
M_λ	Matrix represents the grid map
H	The furthest prediction by neural network
$\mathcal{T}$	A single TPR-Tree
Δt	Time interval of TS-TPR
m	The number of TPR-Trees of TS-TPR. $H = \Delta t \times m$
Q	Query types for TS-TPR
l	The length of query window
$\mathcal{G}$ (or $\hat{\mathcal{G}}$)	Trajectory in grid ID representation

2.2 Neural Models for Moving Objects

Deep learning has been shown a great success in a variety of areas including speech recognition, computer vision and natural language processing [10]. Since our target is to predict the trajectories in the future, we narrow our scope into the learning techniques for trajectories. Alexandre et al. [11] and Federico et al. [12] proposed models to predict the pedestrian trajectory by taking the environment influence into consideration. These models trained for low-speed pedestrians in a crowded space are not applicable to the environments where objects are moving in a large area with various velocities. Rajiv et al. [13] devised a model to predict the basketball trajectories which is essentially a regression to a specific motion pattern. Hao et al. [14] and Di et al. [15] utilized a variant RNN (Recurrent Neural Network) to model taxi trajectories. They generated an embedding for each trajectory so that it can be further utilized by other applications. Alexandre et al. [16] proposed prediction models for taxi trajectories which can predict the destination of an arbitrary trip. Some works [17], [18], [19], [20] adopt pattern-based strategies to perform the prediction. However, such methods do not work well in a real complicated scenario because moving objects show varieties of patterns which cannot be fully captured by a single model. As a consequence, such models fail to predict the trajectories if a pattern has not been observed before. Additionally, some works use algorithms to calculate the embedding of trajectories. Ningnan et al. [21] and Zhao et al. [22] proposed the sophisticated models to generate the multi-context embedding for trajectories. The research of trajectory representation is orthogonal to our work. We can also use the vector representation of trajectory to improve the prediction accuracy. Unlike previous approaches, our work adopts the same philosophy of [23] to employ neural models to build and tune the performance of indexes.

3 Neural-Enhanced Index

This section presents the overview of NEIST. As illustrated in Fig. 1, NEIST mainly consists of three components: trajectory prediction model, suffix-tree index of trajectories and TS-TPR (time-slots based TPR-Trees). For a specific moving object, its existing trajectory is used as input to our neural model which will generate prediction to the trajectory's extension for the near future. The suffix tree indexes the prediction results for moving objects sharing similar trajectories. If a new object has a similar trajectory to the one indexed by the suffix tree, instead of performing the prediction via neural model, we directly return the prediction result which significantly reduces the prediction overhead. Finally, the prediction results generated by the neural model or suffix-tree index will be inserted into TS-TPR to answer the predictive spatio-temporal queries.

TS-TPR supports four spatio-temporal query types. We first review these query types.

Type 1 Prediction query for a single trajectory: $Q_1 = (T, t_s, t_e)$ specifies a trajectory T and time interval $[t_s, t_e]$. This query retrieves the future path in the corresponding interval.

Type 2 Time slice query: $Q_2 = (R, t)$ specifies a region R at the timestamp t.

Type 3 Window query: $Q_3 = (R, t_s, t_e)$ specifies a region R that crosses time interval $[t_s, t_e]$.

Type 4 Moving window query: $Q_4 = (R_1, R_2, t_s, t_e)$ specifies the region R_1 and R_2 that cross time interval $[t_s, t_e]$. This query retrieves the points in a hyper trapezoid by connecting the two regions. Note that this query is a general form of Q_1 and Q_2.

As shown in Fig. 4, the four query types are denoted by Q_1 to Q_4 respectively. The axes x and y are two dimensions of the spatial coordinates. t_0 is the time when last position is reported. Time range $(t_0, t_5]$ is the future time. For ease of presentation, we list the frequent used notions at Table 3.

This section is organized as the following. In Section 3.1, we explain our neural model for trajectory prediction. Section 3.2 introduces a time-slot based TPR-Tree to perform spatio-temporal queries by using the predicted trajectories. To reduce the prediction overhead, Section 3.3 proposes an error-bounded approximate prediction approach and a suffix-tree index to share the prediction results among similar trajectories.

3.1 Trajectory Prediction

Trajectory can be considered as a sequence of coordinates. So previous work adopts RNN model to perform the prediction [11], [13]. More specifically, LSTM (Long Short-Term Memory) network achieves better performances in sequence learning tasks, like speech recognition and hand writing generation. Given an input sequence $x = (x_1, \ldots, x_t)$, LSTM generates the sequence $y = (y_1, \ldots, y_t)$ by applying the following equation iteratively,

$$h_t = f(h_{t-1}, x(t)) \tag{1}$$

$$y_t = W^{yh} h_t + b^y, \tag{2}$$

where f is a non-linear function provided by a typical LSTM cell, W^{yh} and b^y are the weight and bias of the

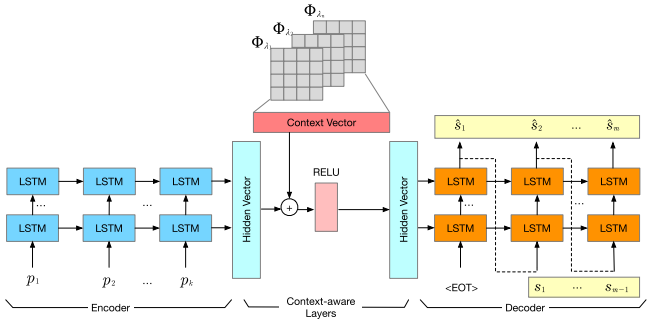


Fig. 2. The Architecture of Trajectory Learning Model. We use multi-stacked LSTM cells as the encoder and decoder, respectively. The context-aware layers take into account the context effect to the current trajectory. $(\Phi_{\lambda_1}, \ldots, \Phi_{\lambda_n})$ are the grid mapping functions with different granularity. The symbol $<\text{EOT}>$ means the end of a trajectory and $\oplus$ is to concatenate two vectors.

linear transformation from the hidden state to the output respectively.

In this paper, we adopt a variant of sequence-to-sequence (Seq2Seq) model [3]. The core idea behind Seq2Seq is to use two different LSTM [24] networks, encoder and decoder, to perform data feature extraction and result prediction respectively. The encoder maps the input sequence to a fixed-size hidden vector and the decoder then maps the hidden vector back to the output sequence. Essentially, it learns a conditional distribution of a variable-length sequence conditioned by another variable-length one.

As illustrated in Fig. 2, the trajectory prediction model consists of three components: the encoder, the context-aware layers, and the decoder. The encoder is a multi-layered LSTM network. The point $p_i (p_i \in T_o, 1 \leq i \leq k)$ of the input trajectory T_o with length k, is sequentially fed into the encoder. The hidden state of LSTM cells is recursively updated by Eq. 1 until the end symbol of trajectory (EOT) is read (EOT is represented by a zero vector in our implementation). Then the last state of encoder is the result hidden vector V, which can be considered as features of the input trajectory. The decoder takes the hidden vector V as its initial state. To simplify our presentation, we use $T_s = (s_1, \ldots, s_m)$ and $\hat{T}_s = (\hat{s}_1, \ldots, \hat{s}_m)$ to denote the ground truth and estimated results given by the neural model for the future trajectory respectively. The decoder is trained to generate $\hat{s}_t (\hat{s}_t \in \hat{T}_s, 1 \leq t \leq m)$ conditioned by V and the hidden state h_{t-1}, where h_{t-1} is computed as:

$$h_t = f(h_{t-1}, s_{t-1}, V). \tag{3}$$

Specially, when $t = 0$, the input is $<\text{EOT}>$ and the hidden state is represented as a zero vector. During the training stage, we use the ground truth of the future trajectory T_s as the input of the decoder. While at the testing stage, as the dash line shows in Fig. 2, the predicted point $\hat{s}_t$ at timestamp t will be used as the input at timestamp $t + 1$. The hidden state is then computed by

$$h_t = f(h_{t-1}, \hat{s}_{t-1}, V). \tag{4}$$

When the decoder completes, we calculate the loss of T_s and $\hat{T}_s$ by using minimum square error method, namely $loss = \sum^m (\hat{T}_s - T_s)^2$.

Context-aware Layers. The influence of context to the trajectory prediction has been discussed by previous works [11], [12]. In our scenario, a taxi driver can change his/her decision based on the context information, such as traffic jam at some roads or a specific road being blocked. Moreover, once he/she changes his/her decision, such decision also affects the other drivers, e.g., creating new traffic jams. To represent the context information, we divide the area into equal-size grids and collect all reported positions of taxis for every time slot. We map these locations into the grid map and calculate the total number of taxis in each grid. Such a taxi density map can be a context for the taxi trajectory prediction. If a grid contains too many taxis, the other taxis should adjust their routine to avoid such an area.

Based on this observation, we propose the context-aware layers. Given a spatial coordinate p, we define the grid mapping function $\Phi_\lambda = \Phi(p, \lambda)$ to get the corresponding grid coordinate with a certain granularity λ. The granularity λ, denoted by a tuple of positive integers $\lambda = (g_x, g_y)$, determines how to partition a map. For example, if $\lambda = (10, 20)$, the map will be equally partitioned into 10×20 grids. A grid map is represented by a matrix M_λ with size (g_x, g_y). We apply Φ_λ to the input trajectory T_o and get a sequence of grid coordinates. The value of M_λ is calculated by counting the number of taxis that share the same grid coordinate.

To extract the correlations among different grids, we vary the value of λ. Namely, we generate a set of grid mapping functions by changing the granularity. Typically, a fine-grained partitioning can effectively model the city area where small roads cross each other, while a coarse-grained partitioning is able to find the trajectory trends from a global view. Here, we vary the granularity from λ_1 to λ_n to build a sequence of the grid map $(M_{\lambda_1}, \ldots, M_{\lambda_n})$.

For example, we apply 3 CNNs (Convolutional Neural Networks) for each grid map. Take an arbitrary M_λ with shape (g_x, g_y) for example. Here, the grid map is treated as an image with 1 color channel. Namely, we can reshape M_λ into size $(g_x, g_y, 1)$. We first apply a convolutional layer with kernel size (2,2), stride 1 and output channel 32 for M_λ. Then a max pooling layer comes with another convolutional layer with kernel size (2,2) and stride 2. The shape of intermediate feature with M_λ is now $(g_x/2, g_y/2, 32)$. Next, we apply two convolutional layers to flatten the features by using kernel sizes $(g_x/2, 1)$ and $(1, g_y/2)$. In this way, we obtain an output vector with shape (1,128).

We assemble all outputs of CNNs for each grid map into a context vector V_c by the following equation,

$$V_c = \sum^n \oplus f_{\text{CNNs}}(M_\lambda), \tag{5}$$

where $\sum \oplus$ is the operation of vector concatenation along the last dimension and f_{CNNs} is the function of multiple CNNs. We further concatenate V_c and the hidden vector V_h as input for another layer with a ReLU activation function to generate a new hidden vector V_h' by Eq. (6).

$$V_h' = f_{\text{ReLU}}((V_c \oplus V_h)W^{h+c,h} + b^h). \tag{6}$$

The ReLU layer guarantees that the shapes of two hidden vectors remain the same. Finally, the new hidden vector will be fed into the decoder as its initial state.

We tune the number of context layers with a simple NAS (Neural Architecture Search) approach. In particular, we start with a model with a single context layer. Then, we gradually increase the number of context layers and double the length of grid cells. For each new model, we train and test its

precision with the same set data. Finally, the one with highest precision is adopted. We will give more detail evaluations in the experiment section.

3.2 Time-Slots Based TPR-Tree

In Section 3.1, we propose a neural network based model to predict the future positions of moving objects. In this section, we explain how NEIST answers the predictive spatio-temporal queries with Seq2Seq model.

The predictive spatio-temporal query is employed to retrieve a set of moving objects that will intersect a query window during a future time interval. Lots of data structures have been proposed to process such queries. Among them, TPR-Tree [1] is the most widely used one. Before delving into the details of our approach, we first review TPR-Tree briefly. The object movements in TPR-Tree are represented by a linear motion function. For example, suppose r is a moving object indexed in TPR-Tree at time t. The current location of r is r_l and the velocity is r_v. Based on that information, the location of this object at any future time t_f can be calculated as $r_l + (t_f - t) \times r_v$. However, TPR-Tree is incapable of handling our taxi scenario. The reason is twofold. 1) By using the neural model for prediction, the future trajectory is no longer a linear trajectory. Namely, let $T_s = (s_1, \ldots, s_m)$ be our prediction result, where $s_i (1 \leq i \leq m)$ is the position of the moving object at a future time t_i. Given an arbitrary point s_i and a time interval Δ, TPR-Tree would fail to infer the future position of $s_{i+\Delta}$ at time $t_{i+\Delta}$. 2) If we just replace the linear motion model with ours, the performance of TPR-Tree could be even worse, because, every time a predictive query comes, the neural model has to predict the positions of all points involved in the query to compute the intersection. Since prediction cost of the neural model is much more expensive than the linear one, the online query time will be unacceptable.

To satisfy both requirements of prediction accuracy and query performance, we propose a time-slot based TPR-Tree (TS-TPR for short). We build one particular TPR-Tree $\mathcal{T}_i (1 \leq i \leq m)$ for each time slot t_i, which stands for a future time interval $[t_{i-1}, t_i)$. The length of each time slot Δt is equal. The predicted locations of all trajectories $T_s = (s_1, \ldots, s_m)$ will be inserted into the corresponding trees based on its time slot. The valid query window for a TS-TPR is $\Delta t \times m$. For example, if we set $\Delta t = 2$ minutes and the number of TPR-Tree $m = 5$, the valid query window is $[t_0, t_0 + 10]$. If a user issues a query beyond our valid query window, we will use the TPR-Tree of time-slot $[t_{m-1}, t_m)$ and apply the linear model for prediction. This makes our approach degrade to the conventional TPR-Tree algorithm, since based on our experiment results, the neural model achieves a similar bad prediction result as the linear model for far future.

One assumption of TS-TPR is that the motions of objects are almost linear during a short time (a time slot). Note that there is a trade-off between prediction accuracy and space-time cost in this assumption. Suppose the furthest prediction of our neural model is made for time H, then we have $H = \Delta t \times m$. If Δt is small, objects follow a linear motion with a high probability which improves the accuracy of position estimation. But by increasing the number of time slots, more TPR-Trees have to be built which incurs higher overheads. On the other hand, if Δt is large, both the prediction accuracy and processing cost of TS-TPR decrease. Especially, when $\Delta t = H$, TS-TPR will become a regular TPR-Tree.

By default, time interval Δt is equal to the report frequency. However, due to unstable network communication, the report frequency varies and may be quite high (like a few seconds), which incurs high overheads for both construction and query of TS-TPR. We discuss how to set a proper Δt and m. Suppose the report frequency is f and a predefined accuracy threshold θ_{acc}, we first determine the maximum positive integer k by solving the following equation.

$$\arg\max_k \left\{ |P_j - \mathrm{NN}_j| < \theta_{acc} | \forall j \in \left[1, \frac{H}{k \times f}\right] \right\}, \tag{7}$$

P_j is the prediction accuracy at the timestamp of jth TPR-Tree by applying linear motion model to the former tree, while NN_j is accuracy from neural network at the same timestamp. The evaluation of accuracy will be detailed in Section 4.2. θ_{acc} is used to tune the accuracy degradation when time interval varies. After k is determined, we get $\Delta t = k \times f$ and $m = \lfloor \frac{H}{\Delta t} \rfloor$.

Creation. A TS-TPR consists of a series of TPR-Trees where the leaf nodes maintain entries for spatial coordinates and velocities of all moving objects. For a newly generated prediction trajectory $T_s = (s_1, \ldots, s_m)$, each point except the last one will be inserted into the corresponding TPR-Tree $\mathcal{T}$. Let s_0 denote the latest reported location of the moving object at current timestamp t_0. In $\mathcal{T}_i$, the object coordinate is s_{i-1} and the velocity is calculated by the equation $v_i = \frac{s_i - s_{i-1}}{\Delta t}, 1 \leq i \leq m$. So the record of this object in $\mathcal{T}_i$ is represented by (s_{i-1}, v_i).

For example, assume that a prediction trajectory of a moving object is $T_s = [(1,3), (6,8), (5,3)]$, the latest reported coordinate is $s_0 = (1,1)$ and the interval of each time slot is $\Delta t = 2$. The current TS-TPR consists of three TPR-Trees $(\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3)$. In $\mathcal{T}_1$, the object coordinate is represented by $s_0 = (1,1)$ and the velocity is calculated by $v_1 = \frac{(1,3) - (1,1)}{2} = (0,1)$. As a result, we insert (1,1,0,1) into $\mathcal{T}_1$ for this object. Similarly, we insert (1,3,2.5,2.5) into $\mathcal{T}_2$ and $(6, 8, -0.5, -2.5)$ for $\mathcal{T}_3$.

Update. When time elapses to t_1, for the newly observed position, the neural model generates a new predicted trajectory $T'_s = (s'_2, \ldots, s'_{m+1})$. We use T'_s to update the current TS-TPR. The first TPR-Tree $\mathcal{T}_1$ is discarded, and a new TPR-Tree $\mathcal{T}'_{m+1}$ is built. Since the object location at the same time slot may differ in T_s and T'_s, the corresponding indexes in $\mathcal{T}_2$ to $\mathcal{T}_m$ should be updated accordingly.

In real systems, we continuously receive the updates of positions from moving objects. To model the scenario as discrete time slots, we maintain a buffer to collect all updates between two consecutive updates of TS-TPR. As shown in Fig. 3, the update buffer is a hash table where the key is the id of moving objects (denoted by *oid*) and the value is a pair of timestamp and position. When time elapses to the next time slot, the neural network loads the tuples from the buffer and generates the predicted trajectories for future. Given the new prediction results, the outdated TPR-Tree will be discarded and a new one will be created by using bulk loading technique which is discussed in the following section. Note that the queries issued during the update will be processed by the old index until the new one is fully built. As shown in Fig. 3, query Q_{t_1} is issued during the update of TS-TPR at time slot t_1. If $\mathcal{T}'_2$ is not completely built, the query will be answered by $\mathcal{T}_2$. When $\mathcal{T}'_2$ is ready, it will take over the queries and replace $\mathcal{T}_2$.

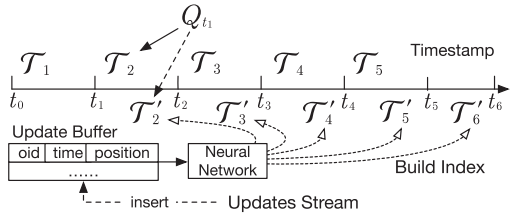


Fig. 3. Update process of TS-TPR.

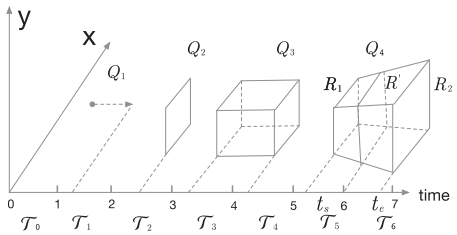


Fig. 4. Four query types of TS-TPR.

Queries. When TS-TPR receives a prediction or timeslice query (namely the query type Q_1 or Q_2), it is straightforward to assign a proper TPR-Tree to answer the corresponding query indicated by the query timestamp t. For instance, in Fig. 4, Q_1 and Q_2 will be dispatched to $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively. However, some queries, such as window queries Q_3 and Q_4, may overlap with multiple time slots. In TS-TPR, we split such queries into multiple sub-queries. Each sub-query is assigned to a time slot and processed by the corresponding TPR-Tree. Note that Q_4 is a general form of Q_3, so we use Q_4 as an example to explain how the TS-TPR processes window queries.

As shown in Fig. 4, the query window $[t_s, t_e]$ of Q_4 overlaps with two time slots, $[t_5, t_6)$ and $[t_6, t_7)$. We split the query window of Q_4 into two intervals $[t_s, t_6)$ and $[t_6, t_e]$. A new region R' at time t_6 is calculated by the following equation.

$$R' = \frac{R_2 - R_1}{t_e - t_s}(t_6 - t_s) + R_1. \tag{8}$$

As a result, Q_4 is converted into two sub-queries $Q_4^1 = (R_1, R', t_s, t_6)$ and $Q_4^2 = (R', R_2, t_6, t_e)$, which are sent to TPR-Trees of slots $[t_5, t_6)$ and $[t_6, t_7)$ respectively. The results of Q_4^1 and Q_4^2 will be merged together.

Since queries may access multiple trees, it is necessary to consider the time complexity in such circumstances. As pointed out by [2], the average time complexity of queries for TPR-Tree is

$$C_q = \sum_{\text{eachnode}} A_{SR}(o, q),$$

where $A_{SR}(o, q)$ is the intersected area of node o and query region q. As for TS-TPR, we should determine the number of trees queried averagely. We directly give the upper bound n_{up} and leave the proof in the appendix section.

$$n_{up} = \left\lfloor \frac{l}{\Delta t} \right\rfloor + 2.$$

Based on that, the query time complexity for TS-TPR is computed as follows.

$$C_{TS} = O\left(\frac{l}{\Delta t} C_q\right).$$

Algorithm 1. Bulk Loading for TPR-Trees

```
1: Func bulkLoading()
2: Sorter crt_sorter //Sorter is to sort and store records
3: dim = 0, level = 0
4: crt_sorter.insert(d) for d in Dataset
5: crt_sorter.sort(dim)
6: while True do
7:   createLevel(crt_sorter, next_sorter, b,
8:   level++, dim)
9:   crt_sorter = next_sorter
10:  if(crt_sorter.getTotalEntries() == 1) then break
11:  crt_sorter.sort(dim);
12: end while
13: EndFunc
14:
15: Func createLevel (crt_sorter, next_sorter, b, level, dim)
16: P = ceil(crt_sorter.getTotalEntries/b)
17: S = ceil(sqrt(double(P)))
18: if dim == (lastDim + 1) or S == 1 then
19:   while crt_sorter.hasNext do
20:     n = createNode(crt_sorter.getNext(b), level)
21:     write(n)
22:     next_sorter.insert(n.data)
23:   end while
24: else
25:   while crt_sorter.hasNext do
26:     //insert S × b rocords to tmp_sorter each time
27:     tmp_sorter.insert(crt_sorter.getNext(S × b));
28:     tmp_sorter.sort(dim + 1);
29:     createLevel(tmp_sorter, next_sorter,
30:     b, level, dim + 1)
31:   end while
32: end if
33: EndFunc
```

TS-TPR Bulk Loading. If we adopt the conventional TPR-Tree construction process, the TS-TPR incurs roughly m times more overhead than a single TPR-Tree. To address the problem, we propose a novel bulk loading technique based on Sort-Tile-Recursive (STR) [25] algorithm. STR is originally designed to handle R-Tree bulk loading problems. We modify STR so that it can be applied to TS-TPR. The experiment shows that our method reduces the loading time of TS-TPR to 1/30.

Let r and b be the dataset size and the capacity of an R-Tree node respectively. The number of leaf nodes is $P = \lceil \frac{r}{b} \rceil$. Let $S = \lceil \sqrt{P} \rceil$. The STR algorithm is performed in a two-step way:

1) Sort r points by x-coordinate and partition points into slices. Each slice, except the last one, contains $S \times b$ consecutive points. Inside each slice, we sort the points by y-coordinate and pack them into groups of length b. The first group is formed into the first leaf node, the next into the second node and so on. 2) Recursively pack the leaf nodes into the next level nodes, proceeding upwards, until the root node is created.

Before creating a TS-TPR, the data of moving objects for each TPR-Tree are well-prepared. However, vanilla STR algorithm cannot be applied to our situation directly. Because in TPR-Tree, even though two points share the same spatial coordinate, they may not be inserted into the same tree node due to different velocities. As a result, we modify

the sort metric of STR described in step 1 by taking the velocity into consideration. Suppose the spatial coordinate of a point is (x, y) and the corresponding velocity is (v_x, v_y) . Let Δt be the time interval between two time slots. We define A_x and A_y in the following equations, $A_x = \int_{\Delta t} x + v_x \Delta t$ and $A_y = \int_{\Delta t} y + v_y \Delta t$. We first use A_x to sort the data and then in each slice, we further sort the data by A_y.

The bulk loading algorithm, as shown in Algorithm 1, is applied to construct a new TPR-Tree. The algorithm consists of two functions, *bulkLoading* (lines 1-14) and *createLevel* (lines 16-34). In *bulkLoading*, we first insert all data into a list (line 4) which is used to store and sort the records. Then, the data are sorted by the integration of the first dimension (x-coordinate) on time (line 6). With the sorted data, the algorithm invokes *createLevel* to bulk-load the first level of nodes. The variable $next\_sorter$ maintains the nodes inserted in current iteration and will be used as the initial value of crt_sorter in the next iteration. When the current sorter crt_sorter has only one element (namely, the root node has been created), the algorithm terminates.

In function *createLevel*, records in the same level are packed into nodes. The algorithm first packs $S \times b$ records into small segments each time (line 28), and then sorts each dimension by invoking itself recursively. When all segments have been sorted in all dimensions, the algorithm wraps consecutive b segments into a node and then write it into the tree (line 19-24).

Time Complexity of Bulk Loading. As a variant R-Tree, TPR-Tree adds the velocity to each record and remains the insertion scheme. Based on that, the time complexity of insertion and construction for TPR-Tree are $O(r)$ and $O(r^2)$ respectively. In the following, we analyze the complexity of bulk loading (BL for short). For ease of expression, we first define related symbols. The number of ith level (from bottom to top) entries is r_i. As shown in Algorithm 1, we can easily get $r_{i+1} = r_i/b$ and $r_1 = r$. The height of TPR-Tree is $h = \log_b r$ which is equal to the number of iterations of bulk loading algorithm. Let $S_i = \sqrt{r_i/b}$. In each level, BL sorts records in the current level twice and the corresponding costs C_{s1} and C_{s2} are as follows.

$$C_{s1} = r_i \log r_i$$
$$C_{s2} = S_i^2 b \log S_i b = r_i \log \sqrt{r_i b}.$$

According to the algorithm, $b < r$ holds in all levels because if $b \geq r$, the algorithm terminates. So, an upper bound of sorting cost can be $C_s = 2 r_i \log r_i$. Based on that, we can easily infer the total cost C_B by

$$C_B = \sum_{i=1}^{h} C_s = 2 \sum_{i=1}^{h} r_i \log r_i. \tag{9}$$

So, the time complexity of BL is $C_B = O(\sum_{i=1}^{h} r_i \log r_i)$. Next, we use Theorem 1 to show that BL algorithm outperforms the TPR-Tree construction in time complexity.

Theorem 1. *Given the number of entries r, the node capacity b, when $r > b$, the time complexity of TPR-Tree construction $C_R = O(r^2)$ is an upper bound for bulk loading algorithm.*

Proof. For conciseness, we leave out the big O notation. The proof is shown by Equation 10 .

$$\begin{aligned} C_B &= \sum_{i=1}^{h} r_i \log r_i \\ &< \sum_{i=1}^{h} r \log r_i = r \sum_{i=1}^{h} \log r^h / b^{h(h-1)/2} \\ &< rh \log r \; < \; r(\log_b r)^2 \; < \; r^2 = C_R \\ &\text{s.t. } r > b. \end{aligned} \tag{10}$$

Note that $(log_b r)^2 < r, \text{s.t. } r > b$ can be easily inferred by calculus, hence we omit the proof. □

In the following, we consider the I/O cost for both TPR-Tree construction and BL. When an entry is inserted, the corresponding index nodes of TPR-Tree have to be updated and written back to the disk. So, the number of I/Os is $N_R = hr$. However, the BL packs nodes from bottom to up and writes them back to disk when all nodes in current level are well-prepared. The I/O cost for BL is estimated as: $N_B = \sum_{i=1}^{h} r_i/b$. It is easy to infer that N_B is significantly fewer than N_R which validates BL algorithm is much efficient with respect to the I/O cost.

3.3 Error-Bounded Approximate Prediction

One major problem of applying neural models for prediction is its low performance due to complex tensor computations. In order to improve the performance, we cache recent prediction results of the neural network using a key-value store where the key is input trajectory and the value is corresponding prediction result. Furthermore, locality-sensitive hashing [26], [27] (LSH for short) is applied to cluster trajectories to generate an approximate prediction. Similar trajectories sharing the same hash value will be grouped as a cluster. For each cluster, we use the center trajectory to represent the cluster and its corresponding prediction results accordingly. Furthermore, we prove that the prediction result of each cluster can be approximately bounded by predefined parameters with given confidences.

Additionally, we build a suffix tree to index the center trajectories for clusters. When the suffix of a new trajectory hits an indexed one, the prediction result of indexed trajectory is reused without applying the neural model to generate new predictions.

The adoption of suffix tree is based on two observations: 1) If two trajectories share the same suffixes, they actually follow the same path in the recent past and probably stick to the same path in the near future. We intend to find such trajectories and share the prediction results among them. 2) This is also consistent with the design of Seq2Seq model, which relies on the most recent memories (suffix for the trajectory) to perform the predictions. To provide a performance bound for the result sharing, we define a similarity function to measure the distance of two trajectories. For a new trajectory, we select its nearest neighbor from the suffix tree and reuse its prediction result if their distance is smaller than a predefined threshold Θ. Otherwise, the new trajectory will be forwarded to the neural model.

In our work, we reduce the complexity of measuring the distance of two trajectories by using grids. A grid function Φ_λ with granularity $\lambda = (g_x, g_y)$ is used to convert the input trajectory into a grid coordinate sequence. Each grid coordinate is assigned an id in range $[0, g_x \times g_y)$. As a result, each trajectory is represented by a sequence of grid ids, denoted as $\mathcal{G}$. For example, if we set $\lambda = (3, 3)$, the map will be partitioned

into 9 grids. We simplify the coordinates of grids as following.

$$\begin{bmatrix} (0,0) & (0,1) & (0,2) \\ (1,0) & (1,1) & (1,2) \\ (2,0) & (2,1) & (2,2) \end{bmatrix}.$$

And we generate a unique id for each grid sequentially. E.g., grids with coorinates (1,1) and (0,2) will be named as the 4th and 3rd grid. Furthermore, we adopt L1-norm as our distance metric between two grid sequences $\mathcal{G}_1, \mathcal{G}_2$ with equal length n.

$$dist(\mathcal{G}_1, \mathcal{G}_2) = \sum_{i=1}^{n} (|\mathcal{G}_1^i.x - \mathcal{G}_2^i.x| + |\mathcal{G}_1^i.y - \mathcal{G}_2^i.y|).$$

Error-Bounded Prediction. To describe how good the approximate prediction result is, we set a distance threshold and only when the distance between two trajectories is less than the threshold, shall we share their prediction results. The threshold is a tunable parameter. A strict threshold reduces the possibility of sharing, while a loose one decreases the prediction precision. In the following discussion, we will show formally that the threshold is correlated with the prediction error bound.

Let $\mathcal{G}_1$, $\mathcal{G}_2$ be two arbitrary trajectories (represented as two grid sequences) with the same length in dataset $S_\mathcal{G}$. We use $\mathcal{G}_1^{t_i}$ and $\hat{\mathcal{G}}_2^{t_i}$ to represent the predicted positions at timestamp $t_i, i \in [1, n]$ respectively, where n is the length of predicted trajectory. Let e and p denote the constant for error bound and probability respectively. If $dist(\mathcal{G}_1, \mathcal{G}_2) < d$, and $Pr(|\hat{\mathcal{G}}_1^{t_i} - \hat{\mathcal{G}}_2^{t_i}| < e) > p$, where $d, e \in N$, $s, e \in [1, n]$ and $0 \leq p \leq 1$, we say $S_\mathcal{G}$ has a (d, e, p) error-bounded prediction in timestamp interval $[t_s, t_e]$, $\forall t_i \in [t_s, t_e]$,

For a specific error bound e, we define a function of distance to measure the probability of prediction error between two trajectories at timestamp t_i:

$$Pr(|\hat{\mathcal{G}}_1^{t_i} - \hat{\mathcal{G}}_2^{t_i}| < e) = f_{t_i}(dist(\mathcal{G}_1, \mathcal{G}_2)).$$

We can generate the function set for time interval $[t_s, t_e]$ based on the data distribution using neural models.

$$S_{f_t} = \{f_{t_i} | \forall t_i \in [t_s, t_e]\}.$$

Note that since the prediction is neural model dependent, it is hard to infer the precise analytic expressions of the mappings. In our work, the mappings are obtained after the neural model is built, which is similar to the learned index approach [23]. The distance of samples are calculated pairwise, and then the probability of predictions difference less than e at timestamp t_i under the same distance will be estimated. This procedure can be done offline without incurring any online overhead. Now, we are able to calculate distance d by the following equation with error bound e, probability p and timestamp interval $[t_s, t_e]$ given.

$$d = min(\{\lfloor f_{t_i}^{-1}(p) \rfloor | \forall t_i \in [t_s, t_e]\}). \tag{11}$$

Trajectory Clustering. It incurs high maintenance cost to cache all recent trajectories and their prediction results. Even if we do so, it is costly to compare a new trajectory with all cached ones to decide whether to share the results or not. To address this problem, we adopt locality-sensitive hashing [26] (LSH) to group the cached trajectories into clusters, and use the center trajectories to represent the cluster and their results. Additionally, we prove that trajectories in one cluster are still error-bounded for predictions.

The problems of trajectory clustering have been studied for years [28]. For the unsupervised algorithm of trajectory clustering, density-based spatial clustering of applications with noise (DBSCAN) is the most representative one [29]. In DBSCAN, maximum distance threshold ε and minimal number of points in a cluster $minPts$ are given in advance. A point p is a core point if there are at least $minPts$ points surrounding p within distance e. Such points are called *directly reachable* to p. Points set $\{q_1, q_2, \ldots, q_n\}$ are said *reachable* to p if q_1 is *directly reachable* to p and every q_{i+1} is *directly reachable* to q_i. All points not reachable from any other point are outliers which do not belong to any cluster. However, DBSCAN cannot be applied in our scenario, as it keeps expanding a cluster until it cannot find more than $minPts$ directly reachable points. This strategy will violate our assumption about the maximum diameter of a cluster, and as a result, predictions of trajectories in the same cluster are not error-bounded.

In our work, we choose LSH to cluster trajectories for the reason that unlike DBSCAN, the maximum distance within a cluster is guaranteed by a given probability. We briefly review LSH algorithm and then prove that trajectories in each cluster are error-bounded. LSH is proposed to solve the (R, c)-NN problem. Given a metric space $\mathcal{M} = (M, D)$, a thershold $R > 0$ and a factor $c \geq 1$, a family $\mathcal{F} = \{h : M \to S\}$ is called (R, cR, p_1, p_2)-sensitive for D if for any two elements $v, x \in \mathcal{M}$:

- if $D(v, x) \leq R$ then $Pr(h(v) = h(x)) \geq p_1$;
- if $D(v, x) \geq cR$ then $Pr(h(v) = h(x)) \leq p_2$;

To cluster trajectories, we need two more parameters k and L to make LSH algorithm more robust so that the similar trajectories are more likely to have the same hash value. First, we define a new hash family $G = (g_1, g_2, \ldots, g_L)$ where each hash function g is generated by concatenating random k functions from $\mathcal{F}$, namely $g = (h_1, h_2, \ldots, h_k)$. Trajectories that have the same hash value regarding to family G will be grouped into a cluster. Note that L1-norm is used as our distance metric. As the following theorem indicates, trajectories in the same cluster share the error-bounded prediction.

Theorem 2. *Given a (d, e, p_e) error-bounded dataset $S_\mathcal{G}$ in timestamp interval $[t_s, t_e]$, a (d, cd, p_1, p_2)-sensitive hash family $\mathcal{F}$ and G for any $\mathcal{G}_1, \mathcal{G}_2 \in S_\mathcal{G}$ and $t_i \in [t_s, t_e]$, if $G(\mathcal{G}_1) = G(\mathcal{G}_2)$, then $Pr(|\hat{\mathcal{G}}_1^{t_i} - \hat{\mathcal{G}}_2^{t_i}| < e | D(\mathcal{G}_1, \mathcal{G}_2) \leq d) \geq p_e \times (1 - (1 - p_1^k)^L)$.*

Proof. Based on the property of LSH, the algorithm succeeds in finding a similar trajectory, $\mathcal{G}_1$ within distance d from $\mathcal{G}_2$ with probability at least $(1 - (1 - p_1^k)^L)$, noted as P_d. Given the (d, e, p_e) error-bounded dataset, we have $P_e = Pr(|\hat{\mathcal{G}}_1^{t_i} - \hat{\mathcal{G}}_2^{t_i}| < e) \geq p_e$. Since P_d and P_e are independent, $Pr(|\hat{\mathcal{G}}_1^{t_i} - \hat{\mathcal{G}}_2^{t_i}| < e | D(\mathcal{G}_1, \mathcal{G}_2) \leq d) = P_d \times P_e \geq p_e \times (1 - (1 - p_1^k)^L)$. □

Based on the theorem, we can use the cluster center as the representation of each cluster to generate approximate prediction for trajectories.

Suffix Tree For Similarity Search. To generate predictions for a new trajectory, it is intuitive to use LSH to find the corresponding cluster and return the prediction. However, such method has a limitation due to the constraint of distance metric. For example, if a center trajectory is [(1,2),(3,4),(5,6),(7,8)] and the new trajectory is [(11,2),(3,4),(5,6),(7,8)], it is very unlikely that these two trajectories fall into the same cluster

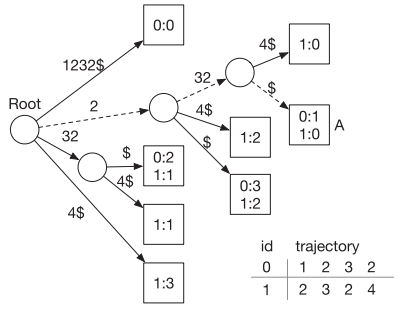


Fig. 5. An Example of the Suffix-tree Index. We use circles and squares to denote internal nodes and leaf nodes, respectively. We insert two trajectories whose ids are 0, 1, respectively.

since the distance 10 is rather large, even though they have a long common suffix. Based on the fact we have observed, if two trajectories are sharing a long common suffix, it is likely that they also have a similar path in the near future. The intuition is that when we use RNN as our neural model, the prediction result is mainly determined by the recent past. In order to relax the constraint of LSH, we adopt a suffix-tree to index the cluster centers.

As shown in Fig. 5, we illustrate a suffix tree indexing two center trajectories. Each edge is labeled with a non-empty sub-trajectory of grid id. Let symbol \$ be the end of a sub-trajectory. Every leaf node represents a trajectory suffix which consists of the edge labels along the path to the leaf node. The entry within a leaf node is denoted as a tuple $<$id, offset$>$ to describe the sub-trajectory location. For the path denoted by the dash line in Figure 5, the trajectory suffix represented by leaf node A is (2,3,2). Leaf node A contains two records $< 0:1 >, < 1:0 >$ which indicates that trajectories 0 and 1 have the same suffix (2,3,2) with offset 1 and 0 respectively.

To retrieve similar trajectories to an input trajectory T_o, we query n suffixes of T_o in the suffix tree. Each suffix $T_o[i:](0 \leq i < n)$ is a slice of T_o. For example, if a trajectory is $T_o = (1, 2, 3, 4)$ and $n = 3$, the query suffixes are (1,2,3,4), (2,3,4) and (3,4). The query results of suffixes are collected together as the candidates. Note that, if two candidates have the same trajectory id, we only take the one with a smaller offset (e.g., we prefer the longer suffix). As an example, suppose the input trajectory is $T_o = (3, 2, 4)$ and n is set to 2. The result for query suffixes (3,2,4) and (2,4) are (1,1) and (1,2) respectively. Since we intend to get a longer matched sequence, the final candidate result is (1,1).

Similarity Function. We need a function to measure the similarity between two trajectories in the suffix tree, so that only very similar trajectories can share their prediction results. Let S_i and S_o denote the suffixes of input and the indexed trajectory respectively. We use function f_t to denote the observed time of each point in a trajectory. The output of f_t is a sequence of timestamps. Then the time difference is calculated by $t_{\text{diff}} = \sum |f_t(S_i) - f_t(S_o)|$. We assume that a smaller t_{diff} indicates that two trajectories are more similar to each other.

Based on that, we give our similarity function, $\theta = \frac{|S_i|}{t_{\text{diff}}}$ where $|S_i|$ is the length of query suffix S_i. We evaluate all candidate results by the similarity function. If the highest score is smaller than the predefined threshold Θ, the input trajectory will be sent to the neural network to perform the prediction task. Otherwise, the prediction result of the nearest neighbor will be reused. Θ is another tunable parameter, which is learned by a simple two layer full-connected neural network to reduce the prediction error. Due to space limitation, we will not discuss it in details.

4 Experiments

In this section, we show the effectiveness of NEIST. All experiments are conducted on our in-house server which is equipped with a 32-core CPU, 64GB DDR3 memory and a GTX TITAN X graphic card (with 12GB memory). We describe our dataset in Section 4.1. We show the effectiveness of our trajectory prediction model in Section 4.2. The performance and accuracy of queries performed by NEIST are evaluated in Section 4.3. Finally, we show the effectiveness of suffix indexing in Section 4.4.

4.1 Data Collection

We use two taxi datasets to validate the effectiveness of NEIST. Thr first one is the Singapore taxi trajectory dataset collected over the whole month of August, 2008. This dataset contains more than 15000 taxis and 140 millions trajectories. The format of a trajectory follows $\{id, [(t_1, x_1, y_1), \ldots, (t_n, x_n, y_n)]\}$ where t is the timestamp and x, y are longitude and latitude respectively. Due to the unstable transmission, the reporting frequency of taxis varies from seconds to minutes. The average updating frequency is 1.9 minutes. In our experiments, we fix the frequency to 2 minutes. So we first clean the data as below. Given a trajectory, we determine the number of updates by $l = \lfloor (t_n - t_1)/2 \rfloor$. For an arbitrary timestamp $t_1 + i \times 2 (0 \leq i < l)$, we find two nearest reports and apply the linear motion model to recover the corresponding position. The second dataset[2] describes about a million trajectories for all the 442 taxis running in Porto from 01/07/2013 to 30/06/2014. We merely extract the coordinates and timestamp following the format described above. The transmission frequency in this dataset is determined. Each taxi reports its position every 0.25 minutes. Additionally, we filter out the trajectories whose duration are less than 20 minutes for both datasets. Namely, we use trajectories in the 10 past minutes to predict the ones in the next 10 minutes. We randomly choose 80 percent trajectories for training our neural model and the remaining 20 percent are used for testing.

4.2 Effect of Trajectory Prediction Model

Our trajectory prediction model is implemented using TensorFlow[3] with version 1.4. We deploy the Adam Optimizer to optimize all parameters with learning rate 0.03 and dropout 0.3. For both decoder and encoder, we set the number of LSTM layers to 3 with hidden size 256 to each cell (The number of layers is determined empirically, since the loss will not decrease significantly when we add more LSTM layers). Batch size is set to 256.

We first find the optimal grid size which achieves the lowest loss for neural model. We fix the number of context layers to 1 and inspect the loss in different granularities. The length of grid cells starts from 100M, increased by 50M each time. As depicted in Fig. 6, when the length is 250M, the optimal loss is achieved on both datasets. Based on that, the default grid size is set to be 250M$\times$250M. Then, to obtain the overview traffic in the whole road network, we select an optimal number of context layers. The

2. https://www.kaggle.com/crailtap/taxi-trajectory/version/1
3. https://www.tensorflow.org/

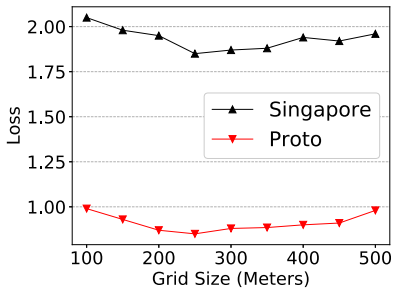


Fig. 6. Loss at different granularity.

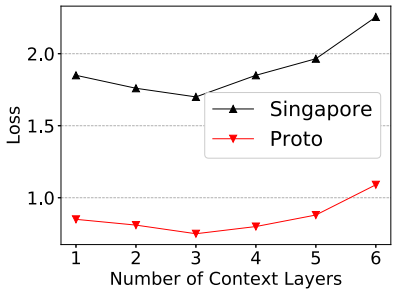


Fig. 7. Loss at different number of context layers.

length of grid cells is doubled when we add one more layer. For example, if the number is 2, context layers should have 2 different granularities, i.e., 250M×250M and 500M×500M. From Fig. 7, we can see that when the number of layers is 3, the lowest loss is the achieved. The reason why the loss does not decrease further when more layers are added is that the number of parameters increases rapidly due to CNNs adopted in the context layers which leads to a hard convergence.

In the following, we evaluate two models. One is just a Seq2Seq-based model without the context-aware layers, called *NN*; while the other one is equipped with context-aware layers, called *NN-CTX*.

Note that the road network of Singapore and Porto can be roughly described as rectangles with size 30KM×40KM and 18KM×48KM respectively. By default, we use a grid function with granularity 250M×250M to partition the whole map. Therefore, the granularities of 3 grid maps in *NN-CTX* for both datasets are (120,160),(60,80),(30,40) and (72,192),(36,96),(18,48) respectively. Parameters for CNNs are set to values as we explained in context-aware layers of Section 3.1.

Evaluation Metric. To evaluate the effectiveness of both the trajectory prediction model and the linear motion model, we use the default grid map to measure the accuracy. Given a grid map, we convert both the ground truth and the predicted coordinate into grid coordinates, denoted as $(G(x), G(y))$ and $(G(\hat{x}), G(\hat{y}))$. Additionally, the error bound e is defined. We assume that, if $G(\hat{x}) \in [G(x) - e, G(x) + e]$ and $G(\hat{y}) \in [G(y) - e, G(y) + e]$, we say this prediction is correct.

In Table 4, we show the accuracy of our trajectory prediction model compared with the linear motion model. (*LM* for short). Besides, we also illustrate the results of traditional polynomial regression models with degree 2 (*PR-2* for short) and *ARMA* (Auto Regressive Moving Average) model [30], [31]. ARMA model attempts to take both AR(p) and MA(q) into consideration when modeling time series. AR tries to capture the momentum and trend of the series, while the other is responsible for the shock observed in noises. Note that, in *PR*, the positions of moving objects in last 10 minutes are used to fit the

TABLE 4
Prediction Accuracy of Five Models: LM, PR-2, ARMA, NN, and NN-CTX

Dataset	Error Bound	Method	Timestamp 2	4	6	8	10
Singapore	0	LM	12.3	2.7	1.0	0.4	0.1
		PR-2	3.8	0.8	0.2	0.1	0.0
		ARMA	13.7	1.8	1.1	0.5	0.1
		NN	28.7	15.1	6.0	2.1	1.5
		NN-CTX	30.1	17.0	10.6	6.3	3.1
	1	LM	34.2	12.7	3.5	1.4	0.9
		PR-2	19.0	4.7	1.5	0.5	0.2
		ARMA	36.6	16.4	7.5	4.9	2.1
		NN	65.1	36.2	20.2	11.9	6.2
		NN-CTX	71.2	40.0	25.2	16.8	10.2
	2	LM	50.5	22.3	9.2	4.3	2.1
		PR-2	34.8	10.3	3.7	1.5	0.7
		ARMA	55.4	27.7	12.1	7.2	3.3
		NN	82.1	50.6	32.9	20.7	10.8
		NN-CTX	88.7	57.1	38.2	25.1	15.6
Porto	0	LM	22.5	13.7	9.1	6.5	5.5
		PR-2	10.5	6.4	3.1	1.5	0.9
		ARMA	30.6	16.8	12.7	8.9	7.5
		NN	71.9	70.4	66.5	65.7	64.2
		NN-CTX	75.6	74.0	72.1	70.0	68.5
	1	LM	44.1	28.2	18.8	14.7	11.9
		PR-2	15.2	10.0	5.2	2.9	1.3
		ARMA	55.4	30.7	26.3	17.5	15.5
		NN	95.1	93.3	90.0	88.6	85.7
		NN-CTX	98.4	98.0	95.6	92.5	90.2
	2	LM	57.6	37.2	25.6	20.1	16.3
		PR-2	24.4	15.6	8.7	5.5	2.7
		ARMA	60.5	46.8	36.8	27.5	18.6
		NN	97.5	96.6	95.4	93.6	90.9
		NN-CTX	99.8	99.7	98.5	96.0	95.4

model by minimum mean square error method, while in *LM*, we merely use the current locations and velocities to perform predictions. We carefully tune the parameters p and q for *ARMA*, and the best accuracy is achieved when $p = 2$ and $q = 1$. We range error bound e from 0 to 2 and inspect the results at timestamps from 2 to 10 for both datasets. Generally, as t increases, the accuracy drops. But when e increases, more results can be found so that the accuracy also increases. The linear model *LM* outperforms *PR-2* in all settings. Since the future path is heavily dependent on the latest positions and *LM* is able to capture the current motion status, it can produce a better prediction result. We notice that the accuracy of *LM* and *ARMA* is very sensitive to the prediction timestamp t and drops significantly when t is getting large. The accuracy of neural network based model can produce about three times more accurate predictions. With the time increase, all these models cannot return the satisfied results. Because, in a real scene, velocities of taxis vary a lot and the road network is extremely intricate. In the comparison with *NN* and *NN-CTX*, We can see that the accuracy of *NN-CTX* is roughly 3 percent higher than that of *NN* which proves the effectiveness of context-aware layers. We can see the accuracy for Porto dataset is far better than that for Singapore. The reasons can be twofold: 1) The updating frequency for Singapore dataset varies a lot so that missing values at some timestamps are recovered using a linear model. However, the Porto dataset has a stable report frequency and no missing values; 2) The area of Singapore is

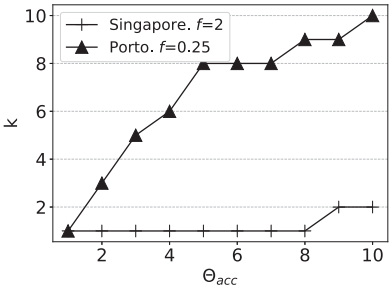


Fig. 8. Maximum k with different θ_{acc}.

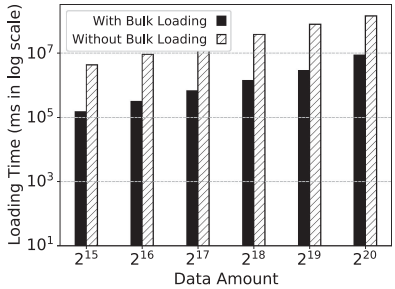


Fig. 9. Effectiveness of bulk loading algorithm.

roughly twice larger than Porto, so the road network in Singapore is more intricate which is more challenging for training and prediction.

Note that, the problem of trajectory prediction is dynamic in nature. As time goes by, the quality of neural model could decrease so that periodically retraining is necessary. However, training a new neural model is offline and it will not draw any extra online overhead.

4.3 Evaluation on TS-TPR

With the trajectory prediction generated by our neural model, NEIST can provide a more accurate query result and a better query performance compared with TPR-Tree. We adopt the disk-based TPR-Tree implementation from libspatialindex[4] library.

We first examine how to set a proper time interval when the parameter θ_{acc} is given according to Eq. (7). In our setting, the reporting frequency f for Singapore and Porto dataset are 2 and 0.25 minutes respectively. The parameter Horizon $H = 10$ minutes. After the training of neural model is finished, the correlation between θ_{acc} and maximum k is shown in Fig. 8. We can see that k is much more sensitive to θ_{acc} when f is small. Based on that, with a slight sacrifice of accuracy, we can increase the time interval of TS-TPR so that fewer trees will be built and incur less overhead in creation and queries. We choose $\theta_{acc} = 5\%$ in our following experiments which means the time interval for both datasets are 2 minutes and 5 TPR-Trees are build for the next 10 minutes. By default, the TPR-Tree in libspatialindex library buffers 10 nodes into memory. The size of each node is less than 10KB with node capacity 100. The buffer size for a single TPR-Tree is less than 100KB. So TS-TPR will not incur too much space cost of memory.

In the following, we evaluate the bulk loading and update performance by using Singapore dataset. In order to show the feasibility of our algorithm with a very large-scale data collection, we randomly pick spatial points from our taxi trajectory dataset with the size ranging from 2^{15} to 2^{20}. As shown in Fig. 9, our method outperforms the regular insertion over 30 times, because the way of packing nodes upwards reduces the splitting time cost of the regular insertion operations. Besides, the effectiveness of bulk loading algorithm also validates the performance of creating a TS-TPR. In our experiments, we build a TS-TPR with 5 TPR-Trees by the bulk loading algorithm. However, the time cost is still just 1/6 of creating a regular TPR-Tree.

At each time slot, both TPR-Tree and TS-TPR should update their records. To be specific, TPR-Tree just updates the position and velocity at the current timestamp while TS-TPR has to update all TPR-Trees with prediction trajectories generated by the neural model. We randomly select 2^9 to 2^{14} trajectories and insert them into TPR-Tree and TS-TPR. Note that we insert points of trajectories into TPR-Tree step by step and calculate the update time for each next time slot. The time cost of neural model prediction is also included. In Fig. 10, when the number of trajectory is small, e.g., 2^9 and 2^{10}, the time cost of both indexes are roughly equal. But when the number of trajectories increases, processing cost of TS-TPR is significantly less than that of TPR-Tree. Since the update of TPR-Tree follows the delete-reinsertion strategy, TPR-Tree incurs higher update overhead for a large size of data. On the other hand, in TS-TPR, we remove the outdated TPR-Tree and bulk load the new data to reduce the cost of updates.

In Table 5, we list parameters for query evaluations. Note that parameters in boldface are default settings. The query window length, noted as l, ranges from [1,10] and its start timestamp is randomly selected from $[0, 10 - l]$. Query type Q_1 is not evaluated in this section. Due to lack of a specific region, the precision and recall are not applicable for this query type. We first evaluate the query performance and accuracy during updating by using default settings. In Fig. 11, we range the number of trajectories from 2^9 to 2^{14}. The experiment shows that the QPS (Queries Per Second) of TS-TPR is twice as the TPR-Tree. This is because when the query time is far from the current, the bounding box tends to expand too much and overlaps with many tree nodes. As a result, the query performance of TPR-Trees drops dramatically. While in TS-TPR, each tree is just responsible for one time slot and can avoid such problems.

During the update, queries will be performed by the old index until the new one is fully built. We have conducted the experiment to simulate the situation. When the next time slot comes, all TPR-Trees are updating and queries are still handled by the old index. We compare the precision and recall of an updating TS-TPR and the one which has accomplished the update. As shown in Fig. 12, the precision and recall drop about 10 percent at most in each timestamp. In our experiment, suppose all taxis are updated simultaneously, the update time will be less than 8 seconds (occupying 6.7 percent of a time interval) according to Fig. 10.

Now, we examine the precision and recall of queries for both datasets. The query loads for each query type are equally generated and the overall number of workloads is 10^4. Figs. 13, 14, and 15, and Figs. 16, 17, and 18 show the precision and recall evaluation of both structures for dataset Singapore and Porto respectively. We can see that as the query region becomes larger, the precision and recall increase. Our method outperforms TPR-Tree in both precision and recall, especially for the long future queries. When timestamp is 2,

4. http://libspatialindex.github.io/

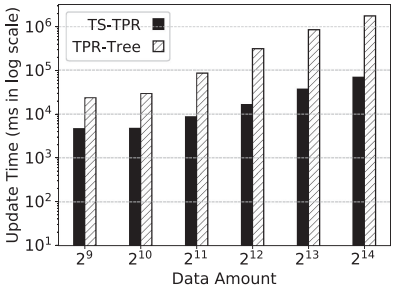


Fig. 10. Update of TS-TPR and TPR-Tree.

TABLE 5
Query Parameters

Parameter	Value
H (minutes)	10
#Query Load	10^4
Dataset	**Singapore**, Porto
Query Type	Q_2, $\boldsymbol{Q_3}$, Q_4
Query Region Size	0.25%, $\mathbf{1\%}$, 2.25%
Query Window (minutes)	$1, 2, 3, 4, \mathbf{5}, 6, 7, 8, 9, 10$

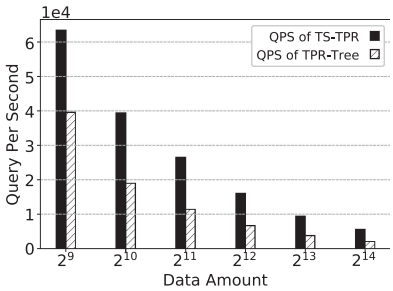


Fig. 11. QPS of TS-TPR and TPR-Tree.

TS-TPR and TPR-Tree share the same results, because both structures store the same data, namely current positions of moving objects. As explained before, the Singapore dataset is noisy and has various reporting frequency which incurs much difficulty for training and predictions. However, from the results, we can see that our method is very robust in such a piratical circumstance. When timestamps are in interval [6,10], our method can produce about 3 times more accurate results for Singapore dataset. The results for Porto dataset significantly outperform TPR-Tree in both precision and recall in all settings which validates the effectiveness of TS-TPR.

4.4 Effect of Error-Bounded Prediction

We share the prediction results among similar trajectories to improve the performance. Our approach can produce error-bounded approximate results. In this experiment, we vary our settings to test the performance of our approximate approach, namely the efficiency and accuracy using default query settings.

We first show the relationship between error bounds and distance thresholds. In Figs. 20 and 21, we vary the distance thresholds and test the probability that the prediction results are bounded by the pre-defined thresholds. In these experiments, we estimate the error bound and confidence based on the current neural and probability model, and apply them to predict the results for future data. As shown in both figures, the confidence is correctly bounded by our estimation (the black line). In our experiment, to get the best prediction results, we set the error bound to 1 and require the confidence greater than 90 percent. The distance threshold should hence be set as 1.

We apply the implementation of E^{2}LSH[5] [27] to cluster trajectories. Related parameters are listed as follows. We set Radius $R = 1$ (the distance threshold), success probability $\delta = 0.9$ and dimension $dim = 5$. The remaining parameters are set as default of E^{2}LSH. Suppose in each time slot, all taxis are updating their positions so that the maximum

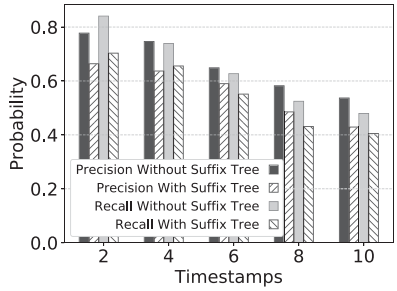


Fig. 12. Precision and recall during update.

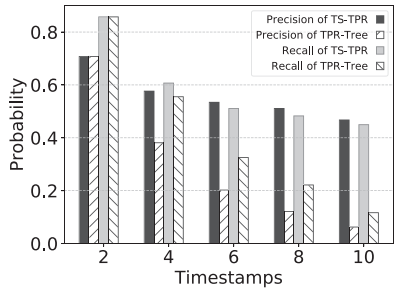


Fig. 13. Precision and recall of QR 0.25 percent (Singapore).

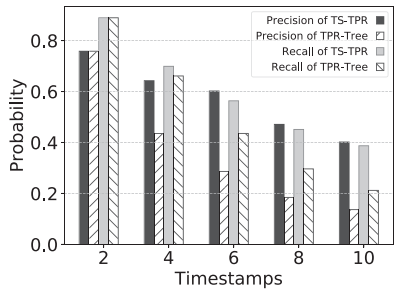


Fig. 14. Precision and recall of QR 1 percent (Singapore).

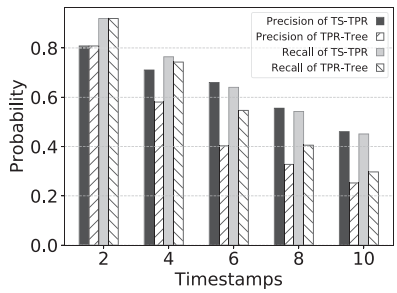


Fig. 15. Precision and recall of QR 2.25 percent (Singapore).

5. http://www.mit.edu/~andoni/LSH/

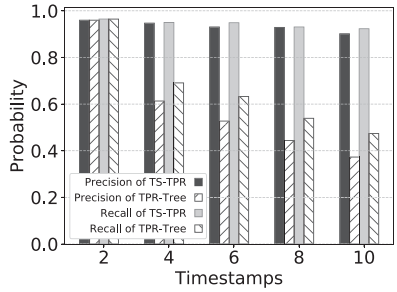


Fig. 16. Precision and recall of QR 0.25 percent (Porto).

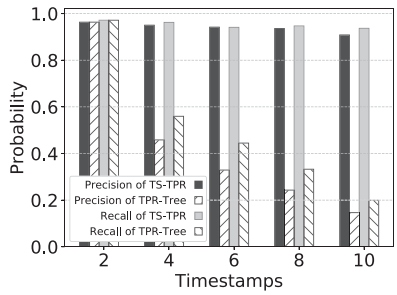


Fig. 17. Precision and recall of QR 1 percent (Porto).

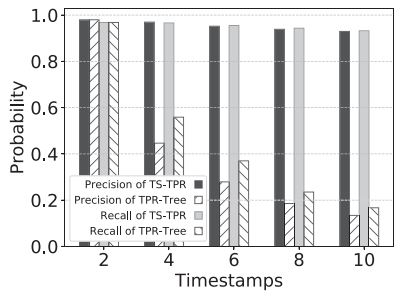


Fig. 18. Precision and recall of QR 2.25 percent (Porto).

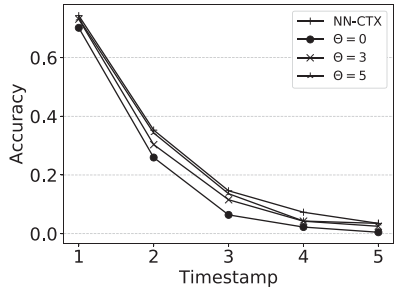


Fig. 19. Accuracy with different Θ.

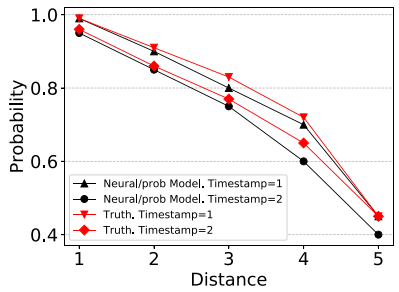


Fig. 20. Probability of prediction error less than 1.

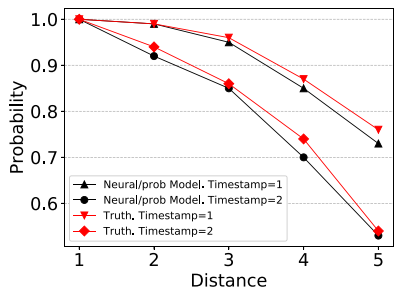


Fig. 21. Probability of prediction error less than 2.

number of trajectories is set to 15000. We use L1-norm as the distance metric. Given the above parameters, E^{2}LSH can find the optimal parameters for k and L. We just give the results, $k = 7$ and $L = 84$. As illustrated in Table 6, we range the number of trajectories from 5×10^3 to 15×10^3 and the result shows that the clustering is very time efficient(occupying 3 percent of a time interval at most), and we can reduce space cost by at most three times.

After clustering, we index the center trajectories by the suffix tree. The adoption of suffix index is based on the assumption that trajectories which share long suffixes, are likely to share their future paths. To verify this assumption, we calculate the probability of two trajectories sharing the same future gird location at timestamp t if they have the same suffix of length m. As shown in Fig. 22, if trajectories share more than 3 grids in the past, the probability of sharing a common future grid at timestamp 1 is more than 90 percent. Even at the timestamp 5, trajectories that share 5 grids in the past have about 40 percent probability to share the same future grid.

In Fig. 23, we range our predefined threshold Θ to illustrate the hitting ratio and processing time. We evaluate one thousand trajectory batches with size 2000 for each and calculate the average results. Hitting ratio indicates how many trajectories are predicted by using suffix-tree index. When $\Theta = 0$, about 35 percent trajectories are processed by the suffix tree and the processing time is about 8ms. As Θ increases, more trajectories are handled by the neural model which leads to a higher processing time cost. When $\Theta \geq 6$, all trajectories are predicted by the neural model and the processing time is about 12ms.

Fig. 19 depicts the relationship between different Θ and the prediction accuracy. Based on the result in Fig. 23, the prediction cost and accuracy increase with a larger Θ. At timestamp 2 to 4, the accuracy with $\Theta = 0$ is about 10 percent worse than that of *NN-CTX*. However, the accuracy is very close to *NN-CTX* at all 5 timestamps when $\Theta = 5$. This is essentially a trade-off between the prediction efficiency and accuracy. If the value is bigger enough, the suffix-tree index will be switched-off and all trajectories will go through the neural model. In our implementations, a simple two-layer fully connected neural network is built to tune Θ. For space limitation, we discard the details.

At last, we examine the query accuracy and efficiency with/without suffix tree index. Using the optimal parameters shown in previous experiments, we set distance threshold to 1 and Θ to 4.5. From Fig. 24, we can see that the accuracy and recall rate drop 8 percent and 5 percent respectively in average. The reason is that when $\Theta = 4.5$, about 20 percent (shown in Fig. 23) trajectories are fed into the suffix tree which incurs a few inaccurate predictions. However, as shown in Fig. 25, we save around 20 percent time for updating process since the prediction efficiency of suffix tree

TABLE 6
Effectiveness of LSH Clustering

#Trajectories ($\times 10^3$)	5	7.5	10	12.5	15
#Cluster ($\times 10^3$)	3.2	4.2	4.7	5.0	5.2
Time (second)	1.0	1.6	2.6	3.1	3.6

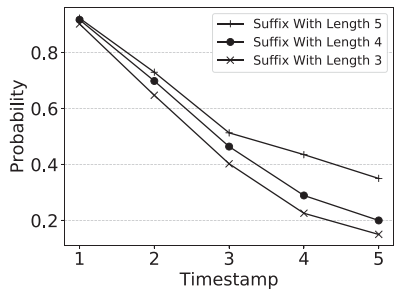


Fig. 22. Probability of sharing future locations.

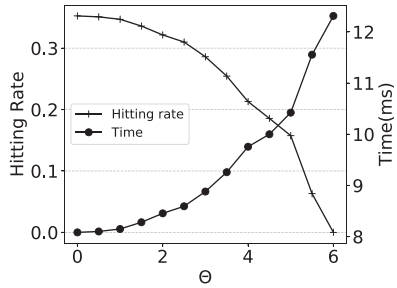


Fig. 23. Hitting ratio and time.

is much faster than the neural model, although the latter one can take the advantage of parallel acceleration by GPU.

5 Conclusion

In this paper, we propose a novel learning index approach, NEIST, to process spatio-temporal queries. A Seq2Seq model enhanced with context information is employed to predict the future trajectories for moving objects, achieving 70 percent more accurate results than the linear model. Moreover, to collaborate with the neural model, we design the time-slots based TPR-Tree to index the prediction results for query processing. To reduce the prediction overhead, objects are organized as groups to share the prediction results via a suffix-tree index. We use the real taxi trajectories to evaluate the NEIST, which shows significant improvements in precision, recall and query processing cost.

Appendix

The Expected Number of Trees Accessed

We use $le()$ and $re()$ to denote the left and right endpoints of either query Q or arbitrary time intervals t and t'. We suppose the location of $le(Q)$ follows a uniform distribution in interval $[0, H-l]$. There are total two situations ought to be considered.

The first case is that $le(Q)$ starts to slide from $le(t)$ to $re(t')$. In this process, the number of trees accessed is $n_1 = \lfloor \frac{l}{\Delta t} \rfloor + 1$. Total length for this case is $s_1 = (\lfloor \frac{l}{\Delta t} \rfloor (\Delta t + 1) - l)(\lfloor \frac{H-l}{\Delta t} \rfloor + 1)$.

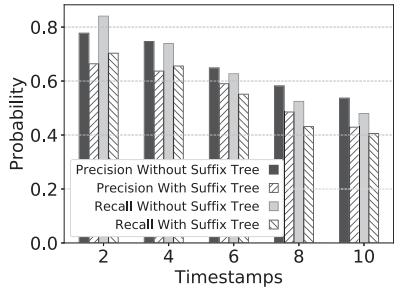


Fig. 24. Precision and recall of QR 1 percent with/without suffix tree.

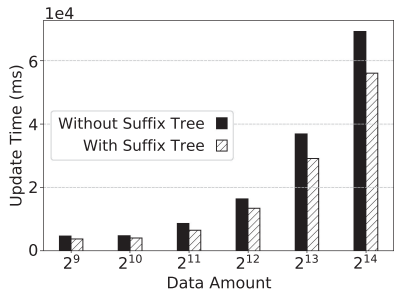


Fig. 25. Update efficiency of TS-TPR with/without suffix tree.

The other case is that $re(Q)$ starts to slide from $re(t)$ to $le(t')$. Accordingly, $n_2 = \lfloor \frac{l}{\Delta t} \rfloor + 2$, and $s_2 = (l - \lfloor \frac{l}{\Delta t} \rfloor \Delta t) \lfloor \frac{H-l}{\Delta t} \rfloor$.

Based on that, the expected number of trees accessed is

$$E(n) = \frac{s_1 n_1 + s_2 n_2}{H - l},$$

From the equation, it is easy to infer that the lower and upper bound of $E(n)$ is n_1 and n_2 respectively.

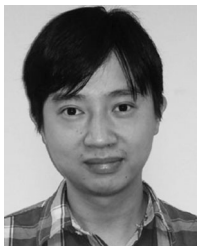
Acknowledgments

This work is supported by the Fundamental Research Funds for the Central Universities [grant number 2018FZA5 015], National Science Foundation of Zhejiang Province [grant number LY18F020005] and the NSFC [grant numbers 61661146001, 61872315]. This work is also supported by the National Research Foundation, Prime Ministers Office, Singapore, under NRF-NSFC Joint Research Grant Call on Data Science [grant number NRF2016NRFNSFC001-113].

References

[1] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. López, "Indexing the positions of continuously moving objects," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2000, pp. 331–342.
[2] Y. Tao, D. Papadias, and J. Sun, "The tpr*-tree: An optimized spatio-temporal access method for predictive queries," in *Proc. 29th Int. Conf. Very Large Data Bases*, 2003, pp. 790–801.
[3] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. 27th Int. Conf. Neural Inf. Process. Syst.* 2014, pp. 3104–3112.
[4] H. D. Chon, D. Agrawal, and A. El Abbadi, "Using space-time grid for efficient management of moving objects," in *Proc. 2nd ACM Int. Workshop Data Eng. Wireless Mobile Access*, 2001, pp. 59–65.
[5] R. Cai, Z. Lu, L. Wang, Z. Zhang, T. Z. J. Fu, and M. Winslett, "DITIR: Distributed index for high throughput trajectory insertion and real-time temporal range query," *Proc. VLDB Endowment*, vol. 10, pp. 1865–1868, 2017.
[6] S. Chen, C. S. Jensen, and D. Lin, "A benchmark for evaluating moving object indexes," *Proc. VLDB Endowment*, vol. 1, pp. 1574–1585, 2008.

[7] S. Chen, B. C. Ooi, K. Tan, and M. A. Nascimento, "St^{2}b-tree: A self-tunable spatio-temporal b$^+$-tree index for moving objects," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 29–42.
[8] C. S. Jensen, D. Lin, and B. C. Ooi, "Query and update efficient b +-tree based indexing of moving objects," in *Proc. 30th Int. Conf. Very Large Data Bases*, 2004, pp. 768–779.
[9] Y. Tao, C. Faloutsos, D. Papadias, and B. Liu, "Prediction and indexing of moving objects with unknown motion patterns," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2004, pp. 611–622.
[10] Y. LeCun, Y. Bengio, and G. E. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
[11] A. Alahi, K. Goel, V. Ramanathan, A. Robicquet, F. Li, and S. Savarese, "Social LSTM: Human trajectory prediction in crowded spaces," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 961–971.
[12] F. Bartoli, G. Lisanti, L. Ballan, and A. D. Bimbo, "Context-aware trajectory prediction," in *Proc. Int. Conf. Pat. Rec.*, 2018, pp. 1941–1946.
[13] R. Shah and R. Romijnders, "Applying deep learning to basketball trajectories," *arXiv preprint arXiv:1608.03793*, 2016.
[14] H. Wu, Z. Chen, W. Sun, B. Zheng, and W. Wang, "Modeling trajectories with recurrent neural networks," in *Proc. 26th Int. Joint Conf. Artif. Intell.*, 2017, pp. 3083–3090.
[15] D. Yao, C. Zhang, Z. Zhu, J. Huang, and J. Bi, "Trajectory clustering via deep representation learning," in *Proc. Int. Joint Conf. Neural Netw.*, 2017, pp. 3880–3887.
[16] A. de Brébisson, É. Simon, A. Auvolat, P. Vincent, and Y. Bengio, "Artificial neural networks applied to taxi destination prediction," in *Proc. Int. Conf. ECML PKDD Discovery Challenge*, 2015, pp. 40–51.
[17] G. Yavas, D. Katsaros, Ö. Ulusoy, and Y. Manolopoulos, "A data mining approach for location prediction in mobile environments," *Data Knowl. Eng.*, vol. 54, pp. 121–146, 2005.
[18] J. J. Ying, W. Lee, T. Weng, and V. S. Tseng, "Semantic trajectory mining for location prediction," in *Proc. 19th ACM SIGSPATIAL Int. Conf. Advances Geographic Inf. Syst.*, 2011, pp. 34–43.
[19] A. Sadilek and J. Krumm, "Far out: Predicting long-term human mobility," in *Proc. AAAI Conf. Artif. Intell.*, 2012, pp. 814–820.
[20] M. Morzy, "Prediction of moving object location based on frequent trajectories," in *Proc. Int. Symp. Comput. Inf. Sci.*, 2006, pp. 583–592.
[21] N. Zhou, W. X. Zhao, X. Zhang, J. Wen, and S. Wang, "A general multi-context embedding model for mining human trajectory data," *IEEE Trans. Knowl. Data Eng.*, 2016, pp. 1945–1958.
[22] W. X. Zhao, N. Zhou, A. Sun, J.-R. Wen, J. Han, and E. Y. Chang, "A time-aware trajectory embedding model for next-location recommendation," *Knowl. Inf. Syst.*, vol. 56, pp. 559–579, 2018.
[23] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," *CoRR*, vol. abs/1712.01208, 2017. [Online]. Available: http://arxiv.org/abs/1712.01208
[24] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735
[25] S. T. Leutenegger, J. M. Edgington, and M. A. López, "STR: A simple and efficient algorithm for R-tree packing," in *Proc. 13th Int. Conf. Data Eng.*, 1997, pp. 497–506.
[26] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proc. 25th Int. Conf. Very Large Data Bases*, 1999, pp. 518–529.
[27] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proc. 20th Annu. Symp. Comput. Geometry*, 2004, pp. 253–262.
[28] J. Bian, D. Tian, Y. Tang, and D. Tao, "A survey on trajectory clustering analysis," *CoRR*, vol. abs/1802.06971, 2018.
[29] J. Lee, J. Han, and K. Whang, "Trajectory clustering: A partition-and-group framework," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2007, pp. 593–604.
[30] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time Series Analysis: Forecasting and Control*. Hoboken, NJ, USA: Wiley, 2015.
[31] P. Whitle, *Hypothesis Testing in Time Series Analysis*. Stockholm, Sweden: Almqvist Wiksells, 1951.

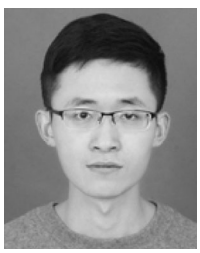


Sai Wu received the PhD degree from the National University of Singapore (NUS), in 2011, and now is an assistant professor in the College of Computer Science, Zhejiang University. His research interests include P2P systems, distributed database, cloud systems, and indexing techniques. He has served as a program committee member for VLDB, ICDE, SIGMOD, and CIKM.

Zhifei Pang is currently working toward the PhD degree in the College of Computer Science and Technology, Zhejiang University, China. His current research interests include indexing and in-memory distributed database.

Gang Chen received the BSc, MSc, and PhD degrees in computer science and engineering from Zhejiang University, in 1993, 1995, and 1998, respectively. He is currently a professor with the College of Computer Science, Zhejiang University. His research interests include database, information retrieval, information security, and computer supported cooperative work.

Yunjun Gao is a professor with the College of Computer Science, Zhejiang University, China. His primary research area is database, big data management, and AI interaction with db technology. He has published more than 90 papers on several premium/leading journals including the *ACM Transactions on Database Systems*, *VLDBJ*, *TKDE*, *TOIS*, *TFS*, *TITS*, and *DKE*, and various prestigious international conferences including SIGMOD, VLDB, ICDE, SIGIR, and EDBT.



Cenjiong Zhao is currently working toward the master's degree in the College of Computer Science and Technology, Zhejiang University, China. His current research interests include indexing and in-memory distributed database.

Shili Xiang received the BS and PhD degrees in computer science from the University of Science and Technology of China and National University of Singapore, respectively. She is currently a scientist and principal investigator in the Data Analytics department, Institute for Infocomm Research, Singapore. Her research interests include smart mobility, data mining, and machine learning.