KAMEL: A Scalable BERT-based System for Trajectory Imputation

Mashaal Musleh
University of Minnesota, USA

Mohamed F. Mokbel
University of Minnesota, USA

ABSTRACT

Numerous important applications rely on detailed trajectory data. Yet, unfortunately, trajectory datasets are typically sparse with large spatial and temporal gaps between each two points, which is a major hurdle for their accuracy. This paper presents KAMEL; a scalable trajectory imputation system that inserts additional realistic trajectory points, boosting the accuracy of trajectory applications. KAMEL maps the trajectory imputation problem to *finding the missing word* problem; a classical problem in the natural language processing (NLP) community. This allows employing the widely used BERT model for trajectory imputation. However, BERT, as is, does not lend itself to the special characteristics of trajectories. Hence, KAMEL starts from BERT, but then adds spatial-awareness to its operations, adjusts trajectory data to be closer to the nature of language data, and adds multipoint imputation ability to it; all encapsulated in one system. Experimental results based on real datasets show that KAMEL significantly outperforms its competitors and is applicable to city-scale trajectories, large gaps, and tight accuracy thresholds.

PVLDB Reference Format:
Mashaal Musleh and Mohamed F. Mokbel. KAMEL: A Scalable BERT-based System for Trajectory Imputation . PVLDB, 17(3): 525-538, 2023.
doi:10.14778/3632093.3632113

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/meshalawy/KAMEL.

1 INTRODUCTION

Numerous important applications heavily rely on trajectory data generated from GPS devices. Such applications include path finding (routing) [15, 26, 49, 76], traffic monitoring and forecasting [28, 31, 45, 67], location-based services [9, 22, 80]), contact tracing [2, 48, 72], map inference [7, 14, 58, 63], and urban planning [27, 36, 37]. Unfortunately, due to bandwidth, battery, and storage limitations, trajectory data are inherently sparse, i.e., there are frequent spatial and temporal gaps between every two consecutive readings. Such gaps present an inherent uncertainty of the object's whereabouts between each two GPS readings. The higher the sparsity (i.e., the larger such gaps spatially and temporally), the lower the accuracy and quality of all applications that rely on trajectory data.

As a means of boosting the accuracy of trajectory data and hence their applications, several techniques have been proposed to impute trajectory data by trying to predict the trajectory whereabouts within the gaps (e.g., see [8, 34, 40, 79]). The large majority of such techniques rely on matching the trajectories on the underlying road network, where the imputation process becomes mainly about finding the road network shortest path between each two consecutive trajectory points. Unfortunately, all such techniques have the implicit assumption that the underlying road network is available and reliable, which is not always true. Road networks, like any other type of data, suffer from all sorts of inaccuracy, and may not be even available in many places [44, 51, 64]. In fact, Microsoft has recently announced that it has found more than one million kilometers of roads missing from current maps [47], Amazon has its own map inference techniques [3], Uber and Lyft build their own map on top of OpenStreetMaps [43, 65], while Apple has recently completely rebuilt its map [4]. This is a very practical problem that triggered a multi-billion dollars industry for constructing accurate maps [17, 23] and a whole area of industrial and academic research of map inference, which aims to infer (all or missing parts) of a road network from trajectory data [7, 14, 58, 63]. This calls for new trajectory imputation techniques that do not rely on the underlying road network, and hence can generate accurate trajectories that can be used to infer the road network. This paper is concerned with imputing trajectories under this setting, which is when the road network is not used as an input in any way. There are only a couple of such approaches [20, 35], but, they are only applicable for small road networks and assume abundance of trajectory data. This makes them not applicable to large-scale road networks.

In this paper, we present KAMEL; a new framework for scalable trajectory imputation that is the first to combine the following distinct features: (1) does not require the knowledge of the underlying road networks as it makes its imputations solely based on input trajectories, (2) does not require prior highly *dense* trajectory data (i.e., large number of trajectories in a small area), (3) scales up to support large geographical areas beyond junction or small city areas, and (4) supports trajectory imputation in both offline bulk mode and online mode for incoming streams of trajectories.

The main idea of KAMEL is to map the trajectory imputation to the *"finding the missing word"* in Natural Language Processing (NLP), which is usually solved using the widely used BERT model [19]. Given a statement like "Paris is the ... of France", where "..." represents a missing word (due to speech recognition, translation, or typo), BERT finds out that the missing word is "capital". To do so, BERT is first trained by large numbers of statements. KAMEL deals with trajectories as statements, where a trajectory/statement is composed of an ordered set of points/words drawn from a finite pool of possible points/words. Also, points/words in a trajectory/statement are spatially/semantically related and are constrained by rules imposed by the underlying road network/language grammar. Hence, at its core, KAMEL is equipped with a BERT model trained by a set of trajectories, and then used to impute sparse trajectories.

This work is supported by NSF under grants IIS-1907855 and IIS-2203553.

Proceedings of the VLDB Endowment, Vol. 17, No. 3 ISSN 2150-8097.
doi:10.14778/3632093.3632113

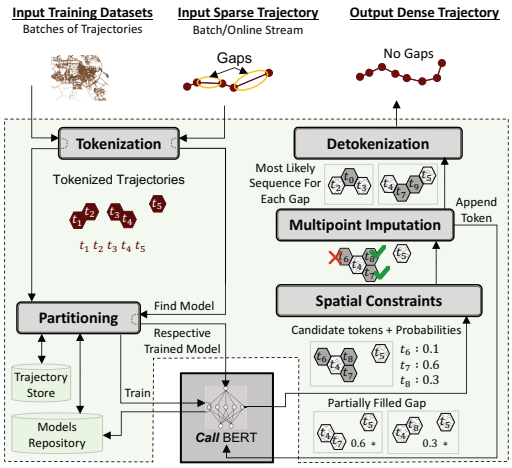


Figure 1: KAMEL **Architecture**

However, using BERT as is within KAMEL is not straightforward and results in a very poor accuracy and performance. This is mainly due to three main challenges: (1) *Spatial awareness*. BERT is not spatially aware, where it may poorly train its model by including datasets that are not spatially related and/or produce results that are not spatially feasible. (2) *Training Data factor*. This is the average number of times each word appears in the training dataset. The original BERT [19] is trained on ~3.3B word corpus composed of ~30K distinct words, so each word appears ~100K times on average in the training set. Meanwhile, trajectory data is no where close to these numbers. A typical trajectory dataset (e.g., Portland dataset [66]) would have ~2M GPS points with ~1.5M distinct points, so each point appears only once on average in the training dataset. Such very low training data factor significantly degrades the quality of BERT to be almost useless. (3) *Multiple missing points*. BERT usually aims to find one missing word in a statement, while trajectory imputation may need to find several missing points between two known points. Applying BERT repetitively to do so would significantly degrade the imputation quality and performance.

KAMEL overcomes all the challenges of using BERT through an architecture composed of five main modules, *Tokenization*, *Partitioning*, *Spatial Constraints*, *Multipoint Imputation*, and *Detokenization*. The *Partitioning* and *Spatial Constraints* modules address the *spatial awareness* challenge by injecting spatial awareness into both BERT training process and output result, respectively. The *Tokenization* and *Detokenization* modules address the *training data factor* challenge by clustering points into tokens to increase their appearance in the training dataset. The *Multipoint Imputation* module addresses the *multiple missing points* challenge by estimating the probability of multiple points together to form an imputed segment.

Our goal is to show that NLP models (e.g., BERT), trained with trajectory data, have the potential to push the state-of-the-art in trajectory imputation. We do not aim to find the best NLP model suited for trajectories. We have chosen BERT as it is one of the most commonly used models. Yet, other BERT variants or language models can be also used with different adaptations. We opted for a design that employs BERT *as is* as an architecture, trained on trajectories, rather than language, which makes KAMEL attractive for industrial and open-source market that are already familiar with it and would need less disturbance to their systems. The experimental evaluation based on system deployment of KAMEL and real datasets shows that KAMEL achieves a high recall score of 89% of the missing points from the ground truth trajectories, with high precision. Comparison with other approaches shows that KAMEL achieves nearly three times the score of the baselines and the state-of-the-art methods, and can work on various straight and curved trajectories, with gaps as large as 2.5 km.

The rest of this paper is organized as follows: Section 2 presents KAMEL architecture. The five modules of KAMEL are described in Section 3 to 7. Experimental results are presented in Section 8. Section 9 discusses related work. Section 10 concludes the paper.

2 KAMEL ARCHITECTURE

Figure 1 depicts the architecture of KAMEL, where it takes two types of input: (1) *training data*; new trajectory datasets which does not produce any output, instead KAMEL uses it to enrich its models. (2) *sparse trajectories*; the trajectories that the users wants to impute. KAMEL receives such data either in bulk offline mode or as a stream of incoming trajectories. In both cases, KAMEL outputs the set of imputed dense trajectories that correspond to the sparse input. KAMEL is a BERT-based system, where BERT is used as a core component and is depicted as a black box at the bottom of Figure 1. However, as mentioned earlier, using BERT as is results in a poor accuracy and performance, due to three main challenges, *spatial awareness*, *training data factor*, and *multiple missing points*. Hence, internally, KAMEL is composed of five main modules to address these challenges, namely: *Tokenization*, *Partitioning*, *Spatial Constraints*, *Multipoint Imputation*, and *Detokenization*, described briefly below:

Tokenization. This module is a gateway to KAMEL, where all input go through it first. It addresses the *training data factor* challenge by converting each input point to a token that covers a specific spatial area. This reduces the number of distinct items and increases their appearance in the training set. The output of this module is a set of tokens sent to the *Partitioning* module. Details are in Section 3.

Partitioning. This module addresses the *spatial awareness* challenge as it builds various BERT models based on the spatial coverage of the input data. It takes its input as a set of tokens produced from the *Tokenization* module. In case these tokens represent training data, the *Partitioning* module stores such data in its raw trajectory store. Then, it decides whether to enrich or expand one of its existing models, or even build a new model, and calls BERT accordingly. The new or updated models will be stored in a dedicated model repository. In case the input tokens represent sparse trajectories that need to be imputed, the *Partitioning* module finds out which BERT model needs to be used to impute each trajectory. For each trajectory gap represented by two end tokens, it calls the selected BERT model to impute one token between the two end tokens. The output of BERT is a set of candidate tokens (with probabilities) sent to the *Spatial Constraint* module. Details are in Section 4.

Spatial Constraints. This module addresses the *spatial awareness* in BERT output. Its input is the set of candidate tokens produced from BERT. Then, it drops off some of these tokens that do not satisfy a set of spatial constraints. Remaining tokens are passed to the *Multipoint Imputation* module. Details are in Section 5.

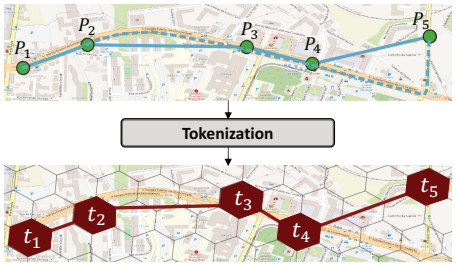


Figure 2: Example Output of the Tokenization Module

Multipoint Imputation. This module addresses the *multiple missing points* challenge. It converts its input from a candidate imputed token to a *sequence* of tokens through calling BERT iteratively. It outputs the most likely sequence of tokens for each trajectory gap and sends it to the *Detokenization* module. Details are in Section 6.

Detokenization. This module addresses the *training data factor* challenge, and to some extent it reverses the *Tokenization* module. The input to this module is the imputed trajectory, represented as tokens. The output is the imputed trajectory represented as points, which is the final output of Kamel. Details are in Section 7.

3 TOKENIZATION

This module is the gateway for Kamel where all input data (training data or sparse trajectories) have to go through before any other Kamel module. A typical trajectory input is composed of points, each represented by its latitude and longitude coordinates. With the high precision of such coordinates, a certain point does not often appear more than once or twice in a training dataset, which is not enough to train a BERT model. Contrast this to the language that BERT is used to work with, where each word may appear 100+K times during the training. This module aims to address this challenge by partitioning the whole space into a set of non-overlapping cells, where all points located in the same cell are given the same token, which is the cell ID. This means that a trajectory dataset will be presented as a set of tokens (cell IDs) instead of a set of points. So, different points will have the same token, and hence they can appear more often in the training dataset, which is also composed of tokens. To increase accuracy and efficiency, the *tokenization* module has to decide on how to partition the space into cells (Section 3.1) and the optimal cell size for each dataset (Section 3.2).

3.1 Hexagonal Space Partitioning

For its tokenization scheme, Kamel uses a flat *hexagonal* grid structure based on Uber's H3 Hexagonal Hierarchical Spatial Index [10, 24]. In this index, the whole world geographical area is divided into a set of non-overlapping hexagons, where each hexagon has a unique ID h_i. Figure 2 shows an example of the tokenization process using a hexagonal grid. The upper part of the figure shows an input trajectory that consists of five points P_1 to P_5, and the bottom part shows the output tokenized trajectory which consists of five tokens t_1 to t_5, where each token corresponds to one of the input points. It is important to note that the road network in the background is shown only for illustration, but the imputation process does not rely on (or even know about) it. Other tokenization alternatives such as Google S2 squares [59] partitioning, which is basically a form of a grid structure is also a possible alternative. However, hexagons help achieve more accurate imputations as confirmed by our experiments in Section 8.5.

(a) $\mathcal{H}$= 25m (b) $\mathcal{H}$= 75m (c) $\mathcal{H}$= 200m (d) Accuracy vs Cell Size

Figure 3: Selection Between Different Cell Sizes

The rationale of using hexagons over common traditional square or rectangular partitioning is that: With a hexagonal grid, all the six neighbors of each cell C have the same exact properties in terms of distance to C centroid and the length of the shared border with C. This is in contrast to rectangular partitioning, where each cell would have four corner neighbors sharing a point, two neighbors sharing the length, and two other neighbors sharing the width. Ensuring that all neighbors have the same properties makes it more suitable for BERT as going from one token (hexagon) to its neighbor would not be influenced by our partitioning. Meanwhile, we acknowledge that unlike rectangles, hexagons are not hierarchical, where we cannot fit a set of neighbor hexagonal cells into one bigger hexagonal cell. Yet, this is not a concern to Kamel, as there is no need for such hierarchy. Hexagons are needed only to tokenize points to cells, and later to detokenize cells to points (Section 7). The cost of mapping a point to its hexagonal cell has a constant time, as it is done through a series of coordinate system conversions [25].

3.2 Cell Size Optimization

Deciding on the right cell size would significantly impact Kamel accuracy. Figures 3(a)-(c) depict three different sizes, with edge length $\mathcal{H}$ = 25, 75, and 200 meters, respectively, laid on top of the same part of a road network. The cell size is inversely proportional to the number of distinct cells (tokens); the larger the size the less the number of distinct tokens, and hence token will appear more in the training set. Hence, large sizes would address the *training data factor* challenge and push for better imputation accuracy. Meanwhile, with large sizes, each cell becomes not really representative of the points inside it, as too many points would map to the same cell, which would negatively affect the accuracy of both the *tokenization* and *detokenization* modules (Section 7). In addition, large sizes make it difficult for BERT to learn distinguishable contexts between trajectories since a large cell would capture trajectories from many roads and directions. This may call for having small sizes, which would highly increase the number of distinct tokens in the dataset. This makes the decision on the right cell size an optimization problem, as illustrated in Figure 3(d), where both ends of the spectrum lead to low accuracy. Kamel is shipped with a default cell size obtained from our exhaustive experiments in Section 8. However, Kamel acknowledges that the optimal cell size might be different for each dataset, as it depends on both the trajectories and area characteristics. Hence, Kamel is equipped with an auto tuning module that sets the system cell size for a given *training* dataset. When the input to the *tokenization* module is a training dataset, we sample the input data and try training BERT models for various cell sizes, and then pick the size that achieves the highest accuracy.

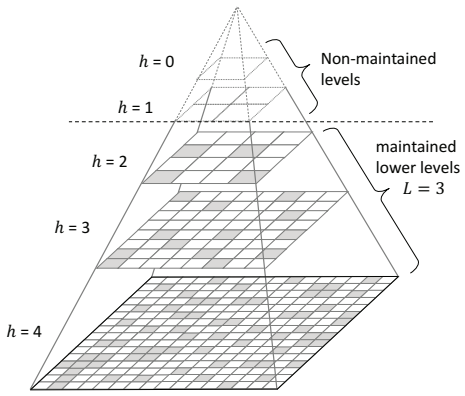


Figure 4: Pyramid Data Structure for Models Repository

4 PARTITIONING

The original BERT is trained separately for each language. For example, the BERT model used for English is pretty different from the BERT model used for Korean, as each is trained by totally different datasets. While the boundaries between languages are clear, it is not the case for spatial areas. Trajectory datasets may come from nearby, overlapping, or far spatial areas. The *Partitioning* module in Kamel is responsible for having spatial-awareness in BERT models by maintaining various models for various spatial areas even though their boundaries are not well defined. One can look at each BERT model in Kamel as a model for a different language. Hence, we maintain two data stores: A simple trajectory store [18, 62] that maintains the set of tokenized trajectories that Kamel received as input training dataset and a model repository that maintains all BERT models that Kamel has built for various spatial areas. Building and/or updating such models is completely done offline, where it may take hours depending on the number of models and trajectories. However, this does not affect the scalability of the imputation process itself, which is done online and only usees the precomputed models in its process. This way, Kamel scales to support large geographical areas. The tokenized input trajectories received by the *Partitioning* module either represent a set of sparse trajectories that need to be imputed or training trajectories. In the former case, the *Partitioning* module basically consults its model repository (Section 4.1) to retrieve the BERT model that is best suited for the imputation process. For training trajectories, the *Partitioning* module uses it to update its model repository (Section 4.2).

4.1 Models Repository Structure and Retrieval

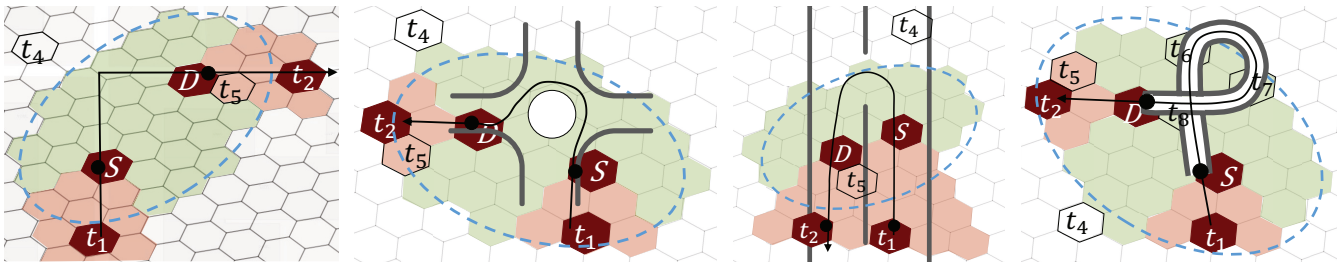
Kamel maintains its model repository in a disk-based hierarchical pyramid data structure of H levels [5], where each level h is decomposed to 4^h equal cells. The pyramid root is of height zero and has only one cell that covers the whole space. The pyramid is built bottom up, and not all levels have to be maintained. In fact, we only maintain the lowest L levels of the pyramid, because there is usually not enough trajectory data to build models for higher levels. The number of levels H and L are parameters that balance between the high resolution of the model and the maintenance overhead of the pyramid structure. Figure 4 gives an example of such structure with $H = 5$ and $L = 3$. Shaded cells are the ones that have models, while blank ones have nothing to maintain.

We maintain two kinds of models: (1) Single-cell models, which are built based on the contents of a single cell, and (2) Neighbor-cells models, which are built based on the contents of two neighboring cells sharing an edge, and stored in either the north or west cell of the two neighbors. The goal of neighbor-cell models is to help in boundary cases, where we need a model to cross the contents of two neighboring cells, especially, if they do not share a parent, or if their parent cell does not have a model. To ensure the model accuracy, we build a model at cell C at level l only if there are at least $k \times 4^{(H-l)}$ tokens in that cell, where k is a system parameter (default 20K) and H is pyramid height. This means that we need at least k tokens to build a model at a leaf node and $4k$ tokens to build a model at a cell just above the leaf level. For the neighbor cell models, we double that threshold. Each cell C contains one or more of the following: (1) The number of tokens in the trajectory store that lie within C, (2) A single-cell BERT model for the contents of C, along with its metadata, which include model statistics and last update date, (2) Up to two neighbor-cell BERT models (with their metadata) for the contents of C and its east and/or south neighbor cells, and (3) Up to two pointers to neighbor-cell BERT models stored at the north and/or west neighbor cells.

When the input to the *Partitioning* module is a sparse trajectory to be imputed, we find the smallest cell C or pair of neighbor cells C_i and C_j that fully enclose the trajectory minimum bounding rectangle, and pick BERT model accordingly for the imputation process. Calling the model does not scan or read any trajectory data after it has been trained offline, which makes Kamel highly scalable. In case there is no single or pair of cells with models to cover the trajectory, we split it into sub-trajectories that can be enclosed within a model. For those sub-trajectories that cannot fit in any of our models, we just impute them by a simple straight line.

4.2 Models Repository Maintenance

Whenever a new training trajectory dataset $\mathcal{T}$ is received, we first add it to our trajectory store. Then, we find the pyramid cell C as the smallest cell that fully encloses the minimum bounding rectangle of all trajectories in $\mathcal{T}$. Then, we enrich $\mathcal{T}$ by adding to it the set of trajectories in our trajectory store that are fully enclosed in C. With C, we do the following four steps: (1) If the number of tokens in $\mathcal{T}$ is above the model threshold, we build a BERT model and store it at C, which will either update an existing model or create a new one. (2) For each cell C_i among the four neighbors of C, if the total number of tokens in C and C_i is above our model threshold, we retrieve all trajectories in C_i and use it with $\mathcal{T}$ to build a neighbor-cell model. The model is stored in the north or west cell of C and C_i, with a pointer to it from the other cell. (3) If the total number of tokens in C and its three siblings is above the threshold to build a model at their parent, we do so. This would be done recursively for C's ancestors till we reach the lowest maintained level. (4) If C is a non-leaf node, we split the trajectories in $\mathcal{T}$ among the four children cells of C, and check if this would warrant building a new model at any of these children cells. We do so recursively till there are not enough tokens to build a model. Note that this does not need to happen for every single trajectory. Instead, it is scheduled as a background process when needed for a batch of new trajectories, without causing any downtime to the system.



(a) Right Turn (b) Roundabout Turn (c) U-Turn (d) Overpass Exit

Figure 5: Spatial Constraints Using Different Road Examples

5 SPATIAL CONSTRAINTS

The original BERT does not have any restrictions in its output as any word in the vocabulary can be the answer as long as it is the most likely one. However, in case of trajectory imputation, we would need to impose some constraints, due to two main reasons: (1) Imputed points have to respect physical movement constraints. For example, an imputed point p between two segment end points S and D should be within a spatial range that respects the locations and timestamps of both S and D. Unfortunately, BERT as is, may end up in predicting p outside that range as BERT may have seen that point following S for another segment that does not end at D. Moreover, unlike the original BERT that imputes only one point, and hence only picks the top possibility, we impute multiple points, and hence we pick the top k possible candidates (details in Section 6). With large value of k, it is more likely to have candidate imputed points that do not respect the physical movement constraints. (2) Cycles. As we aim to impute multiple points for each segment, BERT may end up producing points that go into cycles and hence do not converge to the segment end point.

The *Spatial Constraints* module takes the output from BERT as its input and filters out those tokens that (a) do not respect the physical movement constraints (Section 5.1), or (b) result in cycles in the imputed segment (Section 5.2). The output of the *Spatial Constraints* module is passed to the *Multipoint Imputation* module.

5.1 Physical Movement Constraints

The physical movement constraints mainly specify an area where any imputed token t between the two segment end tokens S and D must be located in. Figure 5 gives examples for applying such constraints for four road cases, namely, right turn, roundabout, U-turn, and overpass exit. In all cases, we need to impute a trajectory segment between tokens S and D, where t_1 and t_2 are the tokens that just come before S and after D, respectively. There are two types of physical movement constraints: *speed* and *direction* constraints. In all four road cases in the figure, the blue dashed ellipse presents an area computed per the *speed* constraints, where physically, a token cannot take place outside such area. The set of red hexagons (tokens) are picked per the *direction* constraints, where none of them can be an imputed token between S and D. This makes the set of green tokens are the only acceptable ones for any imputed token between S and D, where they present the set difference between the tokens within the blue ellipse and the red tokens. Below are the details of how to compute the area of each constraint.

Speed Constraints Area. The rationale behind such area is that travel distance correlates with duration, and a vehicle has a physical limit on where it can travel to, given a time span. This area is depicted as a blue dashed ellipse in the four cases of Figure 5, where the centers of tokens S and D are the ellipse foci points. The maximum total sum of distances from foci points to any point p in the ellipse is $speed_{max} \times TimeDiff(S,D)$, where $TimeDiff$ is the timestamp difference between S and D. While the token timestamps are already given as part of the input data, there are several ways to compute the maximum speed. One way is to use a fixed speed limit based on the city we are trying to impute trajectories in. Another way is to consider the speed of the preceding imputed segment multiplied by a conservative factor. KAMEL currently uses a fixed speed inferred from its training trajectory data. In the four cases of Figure 5, token t_4 would be rejected as it is outside the dashed ellipse, and hence it is physically impossible to reach there.

Direction Constraints Area. The rationale behind such area is that an imputed token should respect the direction from S to D, and not jump ahead of D towards t_2 or before S towards t_1. This area is depicted as a set of red hexagonal tokens in Figure 5, where it is computed as all tokens deviated within a certain angle (default 45°) from S towards its previous token t_1 and from D towards its next token t_2. In the four cases of Figure 5, token t_5 is rejected as it is within 45°angle from the direction of D to t_2.

5.2 Cycles Prevention

It is important to note that when imputing a gap between two tokens S and D, it is most likely that we will need to insert multiple tokens in between. Hence, during the imputation of one segment, BERT will be called many times, which may result in having cycles. A cycle is formed if the sequence of the last x tokens are repeated. A trivial cycle would take place when x=1, which means that BERT has returned the same token as the last imputed token. In this case, the *Spatial Constraints* module would just reject this outcome. For a non-trivial case of $x > 1$, the *Spatial Constraints* module always checks the last x imputed tokens, and rejects all of them together if a cycle is detected. KAMEL uses a default value of x=6, which is very reasonable to detect almost all possible cycles.

Figure 5(d) gives an interesting case for an overpass route, where four imputed tokens t_8, t_6, t_7, t_8 are added between S and D. Although t_8 appears twice, but this is not considered a cycle as there is no repeated sequence of any length x. This shows the ability of KAMEL to impute such non-trivial trajectory.

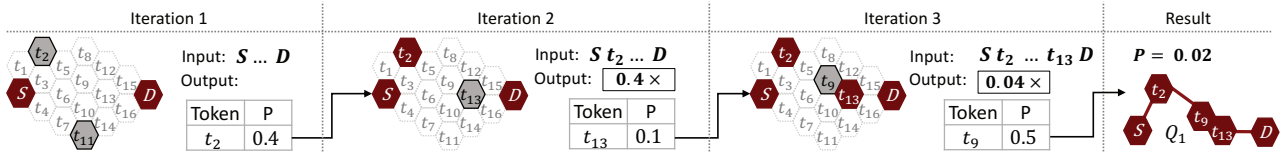


Figure 6: Iterative BERT Calling

6 MULTIPOINT IMPUTATION

For each trajectory segment, the input to this module is a set of candidate tokens returned from BERT and filtered out from the *Spatial Constraints* module. Each token is associated with a probability indicating how likely it is to be the imputed token. Since BERT is designed to predict only *one* missing word in a statement, it just picks the token with the highest probability. However, in case of trajectories, one token is not enough to impute a segment. Hence, the goal of the *Multipoint Imputation* module is to adapt BERT to support imputing multiple tokens between each two consecutive trajectory tokens, such that the distance between any two consecutive tokens is less than a certain maximum gap distance, max_{gap}. KAMEL employs two approaches: (1) iterative BERT calling (Section 6.1), which resembles calling BERT, as is, iteratively and (2) Bidirectional beam search (Section 6.2), which employs a spatially modified version of the classical Beam search algorithm [56] to guide multiple BERT calls. In both approaches, we put a hard limit on the maximum number of times that BERT can be called. If exceeded, we declare failure and resort to linear line imputation between the segment end points. For each segment, the output of the *Multipoint Imputation* module is a complete imputed trajectory segment as a sequence of hexagonal tokens.

6.1 Iterative BERT Calling

This approach basically calls BERT iteratively to fill in multiple imputed tokens between the segment end tokens S and D, such that the distance between any two consecutive tokens is less than a certain gap distance, max_{gap}. Algorithm 1 gives the pseudo code for this approach, where it starts by initializing the output segment to its two end tokens, and a *GapPointer* variable to the starting token, indicating where to insert our next imputed token. Then, it calls BERT, which returns a set of candidate tokens to pick one of them as the next imputed one. The candidate tokens are passed by the *Spatial Constraints* module (*SConstraints*) to filter out tokens that do not satisfy the *speed* and *direction* constraints as described in Section 5. Then, we insert the candidate token with the highest probability just after S. Finally, we call the function *FindFirstGap* to find the first gap in the segment with a length more than max_{gap}. If such gap exists, we repeat the cycle by calling BERT to find the next imputed token and so on. The algorithm concludes when there is no such gap, and returns the complete imputed segment.

Example. Figure 6 gives an example of the iterative BERT calling approach to impute the trajectory segment between tokens S and D with maximum allowed gap distance max_{gap}=2. In the first iteration, token t_2 is returned from BERT as the candidate token with highest probability (0.4) and passed all spatial constraints. Hence, it is inserted as the first imputed point between S and D. Then, we try to find if there is still any gap in the segment. Apparently, S and t_2 are within two tokens from each other, which means that there is no gap in between to fill. However, t_2 and D are still of distance more than two tokens from each other. Hence, the *FindFirstGap* function will return t_2 as the token that we still need to add (at least) one more point after it. This means that we need to go through a second iteration, which returns token t_{13} as the top candidate valid token to be inserted between t_2 and D. Then, calling the *FindFirstGap* function will again return t_2 as the distance between t_2 and t_{13} is still within two tokens. Then, in a third iteration, t_9 will be added between t_2 and t_{13}. Then, *FindFirstGap* will return *null* as all tokens are within a distance less than two tokens from each other. Hence, the algorithm returns the imputed segment as S, t_2, t_9, t_{13}, D.

Algorithm 1 Iterative BERT Calling

Procedure IterativeBERT(SourceToken S, DestinationToken D)
1: *Segment* ← $\{S, D\}$; *GapPointer* ← S
2: **while** *GapPointer* **is not** null **do**
3: *CandTokens* ← BERT(*Segment*, *GapPointer*)
4: *CandTokens* ← SConstraints(*GapPointer*, *CandTokens*)
5: *Segment* ← Insert(*Segment*, **Top**(*CandTokens*), *GapPointer*)
6: *GapPointer* ← FindFirstGap(*Segment*)
7: **end while**
8: **return** *Segment*

6.2 Bidirectional Beam Search

The iterative calling approach is kind of a greedy approach as it only considers the topmost probable token in each iteration. That may not be the right decision as it may lead to less probable later options. Hence, we introduce the *bidirectional beam search* approach that aims to pick the most probable *sequence* of tokens between the source and destination tokens.

Main Idea. Our main idea is to adapt the classical Beam Search [56], designed for searching graphs, to work for trajectory imputation. Instead of searching all possible paths in a graph, beam search limits its search to only the top B promising paths, where a higher B gives more accurate results, but more expensive search. We then employ the following to adopt the beam search idea for trajectory imputation: (1) Instead of searching in only one direction, we make the beam search bidirectional, as trajectory imputation could go in several directions to impute a given segment. So, we keep track with the most probable B sequence of tokens of all directions that can contribute to the final imputed segment. (2) As the probability of a sequence of tokens is computed by multiplying each token probability, longer segments are penalized by having smaller probabilities. Since we are looking for the most probable path, not the shortest one, we counterweight this penalty via length normalization [71]. In particular, a segment S with a probability P would have its normalized probability as $P \times |S|^{\alpha}$ where $0 <= \alpha <= 1$ is the normalization strength, which we set to 1 as a default, and $|S|$ is the number of tokens in segment S.

Algorithm 2 Bidirectional Beam Search

Procedure BeamBERT(Token S, Token D, BeamSize B)

```
1:  Gap.Segment ← {(S, D), 1}; Gap.Pointer ← S
2:  AllGaps ← {Gap}; ProbLimit ← ∞; AnswerSet ← ϕ;
3:  while AllGaps is not null do
4:      NewSegments ← ϕ
5:      for each Gap in AllGaps do
6:          CandTokens ← BERT(Gap.Segment, Gap.Pointer)
7:          CandTokens ← SConstraints(Gap.Pointer,CandTokens)
8:          for each token T in Top(CandTokens,B) do
9:              Segment ← Insert(Gap.Segment,T,Gap.Pointer,Prob(T))
10:             NewSegments ← NewSegments ∪ Segment
11:         end for
12:     end for
13:     NewSegments ← TopB(NewSegments,B,ProbLimit)
14:     AllGaps ← ϕ
15:     for each Segment in NewSegments do
16:         Gaps ← FindGaps(Segment)
17:         if Gaps is empty then
18:             AnswerSet ← AnswerSet ∪ Normalized(Segment)
19:             ProbLimit ← Min(ProbLimit, Segment.Probability)
20:         else
21:             AllGaps ← AllGaps ∪ Gaps
22:         end if
23:     end for
24: end while
25: return Top(AnswerSet)
```

Algorithm. Algorithm 2 gives the pseudo code for our bidirectional beam search approach. It starts by initializing the list *AllGaps* by one gap for the source and destination tokens and a pointer to the source token, indicating where we need to insert our next token. Unlike the case for iterative calling, the gap segment is associated with a probability, initialized by 1. Then, for each gap in *AllGaps*, we call BERT followed by the *Spatial Constraints* module to get the list of candidate tokens, along with their probabilities. Unlike the case of iterative calling where we only care about the top token, here we care about the top B probable tokens (Line 8 in Algorithm 2). Hence, for each such token T, we construct a new segment by inserting T in the gap, along with updating the segment probability. Among all such new segments constructed for all gaps, we pick only the top B of them according to their probability, bounded by an upper limit (initially set to ∞) (Line 13 in Algorithm 2). For each of the remaining B segments, we aim to find all the gaps that are still there to address in the next iteration. If any of such segments has no gaps, we add it to our answer set with its *normalized* probability score, and use that score to adjust our upper limit of considering any further segments. Once there are no gaps in any of the segments, we conclude by returning the segment with the highest probability.

Example. Figure 7 gives an example of the bidirectional beam search algorithm to impute the gap between S and D with beam size $B = 3$. In the *first iteration*, we call BERT to get the top-B probable tokens, which returns t_2, t_{11}, and t_{13}, with probabilities 0.4, 0.3, and 0.2, respectively. Having t_2 between S and D leaves one gap between t_2 and D as the distance between them is more than our max_{gap} threshold. Meanwhile, having t_{11} leaves two gaps between S and t_{11} and between t_{11} and D. Then, having t_{13} leaves only one gap between S and t_{13}. So, in total, we have four gaps to consider further. Hence, in the *second iteration*, we make four BERT calls, one for each gap. The first returns three tokens t_8, t_{13}, and t_{15} with a probability of 0.1 each. The segment probability is 0.04 as they will be multiplied by 0.4, which is the probability of the previous iteration. The second gap (between S and t_{11}) returns t_4, t_6, and t_7 with probabilities 0.1, 0.1, and 0.6, respectively. Multiplied by 0.3, the segment probability for these tokens are 0.03, 0.03, and 0.18, respectively. Similarly, The third call returns tokens t_{13} t_{15}, t_{16} with segment probabilities 0.24, 0.03, 0.03, and the fourth call returns tokens t_2 t_6, t_{15} with segment probabilities 0.04, 0.06, 0.04. Out of the 12 new segments, we only pick the top 3 probable ones, namely, segments $\{S, t_7, t_{11}, D\}$, $\{S, t_{11}, t_{13}, D\}$, and $\{S, t_6, t_{13}, D\}$.

In the *third iteration* to find if there are still gaps in the three segments we still have. The first segment has a gap between t_{11} and D. Similarly, the second segment has a gap between S and t_{11}. Meanwhile, there are no gaps in the third segment, and hence it is concluded as one possible complete imputed segment, with an updated normalized probability $0.06 \times 2^1 = 0.12$. We then use this probability as a lower limit of any segment to be considered further. Hence, we call BERT for only two gaps. The first call suggests t_{13}, t_{15}, and t_{16} with probabilities 0.2, 0.1, and 0.2. To get the segment probabilities, we multiply each by 0.18 as the probability given from the previous iteration. To get the normalized probability, we multiply each segment probability by three as each segment would have three imputed tokens. This makes the normalized segment probabilities for these tokens are 0.11, 0.05, and 0.11. Similarly, the second call suggests tokens t_7, t_3, and t_6 with normalized probabilities 0.36, 0.14, 0.07. Among these six segments, we pick only the top three ones that are above our lower limit of 0.12. There are only two such segments, namely, $\{S, t_7, t_{11}, t_{11}, D\}$ with probability 0.36 and $\{S, t_7, t_{11}, t_{13}, D\}$ with probability 0.14. In the *fourth iteration*, we find out that the first segment has no more gaps, hence we add it to the answer set with its probability 0.36. We call BERT for the second segment to get a concluded one with a normalized score of 0.15. Finally, among the three concluded segments, we return the one with the highest probability of 0.36. Contrast this to what we get from the *Iterative Calling* approach (Figure 6), where we end up with a segment with a normalized probability of 0.06, which shows how powerful is our bidirectional beam search approach.

7 DETOKENIZATION

The output of the *Multipoint Imputation* module is an imputed trajectory presented as a set of hexagonal tokens. Then, the objective of the *Detokenization* module is to take this output and convert each of its tokens to be a point. The output of the *Detokenization* module is a complete imputed trajectory presented as a sequence of GPS points. The functionality of the *Detokenization* module is composed of the below *offline* and *online* operations:

Offline Operations. When training data is uploaded to Kamel and passed through its *Tokenization* module, Kamel triggers the classical DBSCAN clustering algorithm [21] to spatially cluster the contents of each token, based on each point's direction. Since we do not have any information on the underlying road network, the clustering would help in understanding where the points in each

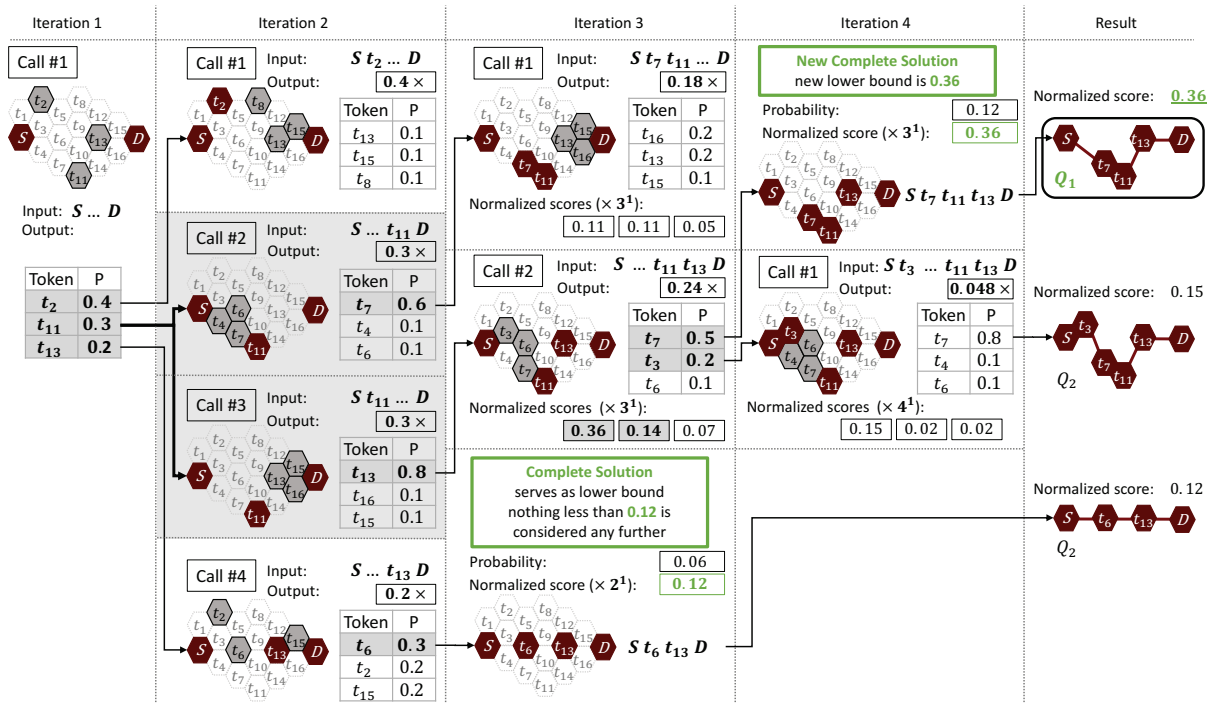
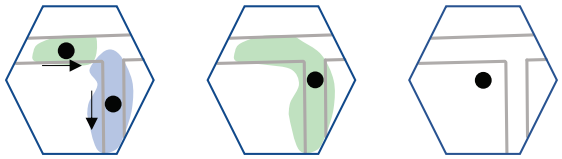


Figure 7: Bidirectional Beam Search

token are usually located within the token. Figure 8 gives three outcome cases of running a DBSCAN clustering algorithm on a hexagonal token where the road network within the hexagon has a right turn. It is important to note that the road network here is just drawn for illustration, but it is not actually available to KAMEL. In the first case (Figure 8(a)), there were enough data points that made the clustering algorithm identify two separate clusters, one going in a horizontal direction, and one in a vertical direction. In the second case (Figure 8(b)), there was not enough data to get two clusters, so, all data in the hexagon is considered as one cluster. In the third case (Figure 8(c)), there were almost no data in the token to run a clustering algorithm. In the first two cases, the centroid of each cluster is presented by a solid dot. In the third case, the solid dot represents the hexagon centroid. The information for all clusters along with their centroids are stored per each token as its metadata, to be used later in the online part.

Online Detokenization. When receiving the output of the *Multi-point Imputation* module, the *Detokenization* module goes through each token, and replaces it with a centroid point using the following procedure: If the token T has multiple clusters (Figure 8(a)), we first compute the token direction angle as the average of the incoming angle between T and its preceding token and the outgoing angle between T and its next token. Then, we pick the cluster that has a closer direction angle to T, and return its centroid point. If the token has only one cluster (Figure 8(b)), we just return the centroid point of that cluster. Finally, if there are no clusters in the token (Figure 8(c)), we just return the hexagon centroid. It is important to note that this latter case is unlikely to take place, as BERT is very unlikely to recommend a hexagon token that has not appeared much in its training data, due to the lack of points there.



(a) Cluster Centroid (b) Data Centroid (c) Hexagon Centroid

Figure 8: Three Outcomes for Clustering Points in a Token

8 EXPERIMENTAL EVALUATION

Baselines. KAMEL is designed as a pre-processing step for map inference applications, a practical case where the road network is not used as input either because it is not available or not trusted. Hence, we compare KAMEL, based on a real system implementation [50] against TrImpute [20] as the state-of-the-art and the only trajectory imputation technique that does not rely on an underlying map and scale up to large networks. For a baseline, we also compare against linear interpolation, where trajectories are imputed by a simple linear line. For complete analysis, we include Map Matching [74] results as an example of techniques that rely on road networks.

Datasets. (1) Porto, Portugal [54]: 1.7M trajectories for real taxi trips, composed of ~83M GPS points, driven for a total length of ~8.8M km spanning an area of ~500 km^2, and (2) Jakarta, Indonesia [30]: 56K trajectories for real ride-sharing trips, composed of ~56M GPS points, driven for a total length of ~500K km spanning an area of ~660 km^2. For each dataset, we use 80% for training and 20% for testing. We sparsify testing trajectories by imposing gaps where we keep the first trajectory point, then, remove all points within distance $Sparse_{distance}$, keep the next point, and so on.

Performance metrics. (1) *Recall.* We discretize ground truth trajectories by placing points P as one point every max_{gap} distance (same threshold used in the imputation process). The *Recall* is the ratio of points in P that the algorithm has correctly recalled within the accuracy threshold δ from the imputed trajectory. (2) *Precision.* We discretize imputed trajectories by placing points Q as one point every max_{gap} distance. *Precision* is the ratio of points in Q that are within accuracy threshold δ from the ground truth. (3) *Failure Rate.* An imputation technique fails to impute a trajectory segment when it just inserts a linear line between the segment end points. *Failure rate* is the ratio of segments imputed by a linear line. (4) *Time Overhead.* Training and imputation time overhead of each algorithm.

Default values and parameter tuning. Unless mentioned otherwise, we set the maximum allowed gap max_{gap} to 100m, the accuracy threshold δ to 50m for Porto and 25m for Jakarta, and the imposed sparsity distance $Sparse_{distance}$ to 1km. For Kamel parameter tuning, our experiments for both datasets (omitted for space constraints) show that setting hexagon size $\mathcal{H}$ to 75m, beam size $\mathcal{B}$ to 10, and pyramid levels L and H to 3 and 10 give the best trade-offs between accuracy and overhead. For Porto, we only needed to build three single-cell models, as one in each of the lowest three pyramid levels. For Jakarta, we had to build 20 models as 7 single-cell and 7 neighbor-cells models in the lowest level, two single-cell and one neighbor-cell models in each of the other two levels. We use BERT original architecture as described in [19] and its open-source implementation [6], with 768 hidden dimensions, 12 attention heads, and 12 hidden layers. The average vocabulary size in our experiments is ~80K, which creates ~165M trainable parameters.

Experiments Design. Sections 8.1 and 8.2 study the impact of sparse distance $Sparse_{distance}$ and accuracy threshold δ, respectively. The overhead of training and imputation time is discussed in Section 8.3. Section 8.4, 8.5, and 8.6 study the impact of road type, grid type, and training data properties, respectively. Section 8.7 performs an ablation study for Kamel components. Training was conducted using a single Google Cloud TPU. All other experiments were conducted using an Intel(R) Xeon(R) Silver 4112 CPU running @ 2.60GHz, with 196GB of memory and 1TB of SSD storage.

8.1 Impact of Data Sparseness

Figure 9 gives the impact of data sparseness on the recall, precision, and failure rate of Kamel and its competitors for both datasets. For reference, we also report the performance of map matching techniques that have the knowledge of the full underlying network, however, we do not consider map matching as a competitor, as Kamel, and its competitors, work on the environments where road network is not reliable. For all experiments, we vary $Sparse_{distance}$ from 500 to 4,000m. For Porto data in Figures 9(a) and 9(b), Kamel consistently achieves much higher accuracy than competitors. It is even very close to the performance of map matching, which shows how powerful is Kamel where it acts *as if* it knows the underlying road network, while it really has no knowledge about it.

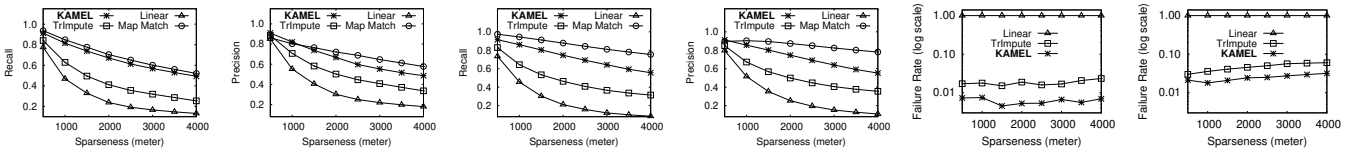
For lower gaps (e.g., 500m), Kamel gives the best performance, but other techniques, even linear interpolation, still give acceptable performance, as it is not that hard to impute small gaps. For medium gaps (e.g., 1,500 - 2,500m), Kamel is 1.5 to 3 times better than its competitors. TrImpute was unable to cope with such gaps as it only works when there are highly dense prior trajectories, which is not practical. It is important to note that these medium gap sizes are the most practical ones, given the current sampling rates of location-tracking devices. We stretched our experiments to very large sparse gaps for up to 4km, which is analogous to asking BERT to predict a full 200 words paragraph given only the first and last words. Even with this, Kamel still gives more than 50% recall and precision, while TrImpute and linear interpolation gives around 25% and 10% recall, respectively. This shows that Kamel is still able to do some useful imputation even for unusually very large gaps.

Figures 9(c) and 9(d) repeat the same experiments for Jakarta. Similarly, Kamel consistently outperforms its competitors. However, Kamel has a better performance than Porto, both in terms of absolute recall and precision and in its relative performance to its competitors. The main reason is that even though Jakarta dataset has significantly less number of trajectories than Porto, each trajectory, on average, has 1,000 points compared to 50 points in Porto. Longer trajectories give more semantics on what points could come after others, and only Kamel takes full advantage of this.

Figures 9(e) and 9(f) give the impact of data sparseness on the failure rate for both datasets. By definition, linear interpolation has a 100% failure rate. Kamel significantly outperforms TrImpute in both datasets and for all sparse gaps. For Porto, Kamel has consistently less than 1% failure rate, compared to up to 2.5% for TrImpute. For Jakarta, as the data is more sparse, failure rate is higher for both with up to 3% for Kamel and 6% for TrImpute. In all cases, failure rate is still within 2% for Kamel for the medium practical gaps, which is highly acceptable for the trajectory imputation.

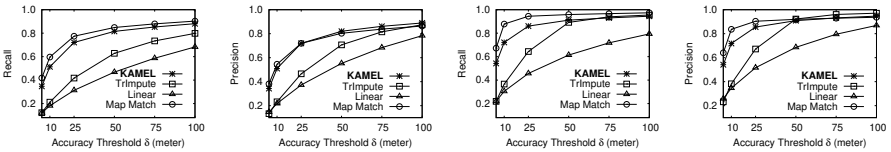
8.2 Impact of Accuracy Threshold

Figure 10 gives the impact of varying the accuracy threshold δ from 5 to 100m on the recall and precision of Kamel and its competitors for both Porto and Jakarta datasets. We also plot the performance of map matching as a point of reference. The objective is to understand the accuracy of each method when used for either relaxed applications that can entertain large thresholds or sensitive applications that require few meters of accuracy. For Porto dataset, Kamel consistently achieves higher accuracy than its competitors, with a performance that is almost identical to map matching. For high δ (e.g., 75-100m), though Kamel gives higher recall and precision than its competitors, the difference is not that much. The main reason is that by increasing δ, lower accuracy techniques like TrImpute would still be able to catch. However, a 100m accuracy threshold may not be practical to consider, especially in dense cities where 100m distance may just lead to a different road. For medium δ (e.g., 25-50m), which are more practical and acceptable by the large majority of trajectory applications, Kamel has ~80% recall and precision, which is significantly higher than that of TrImpute and linear interpolation. We stretched our experiments to very tight accuracy threshold less than 10m. While TrImpute and linear interpolation become almost useless, Kamel is still able to cope with ~40% recall and precision. For Jakarta dataset, we get a similar analysis and conclusion. The only difference is that all techniques give higher accuracy for the corresponding δ for Porto. The main reason is that Jakarta spans a large area, hence a higher value of δ would be acceptable as roads are not as close to each other as in Porto. However, for lower δ, Kamel has significantly higher accuracy.



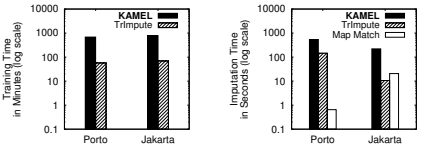
(a) Porto – Recall (b) Porto – Precision (c) Jakarta – Recall (d) Jakarta – Precision (e) Porto – Failure (f) Jakarta – Failure

Figure 9: Impact of Data Sparseness on Recall, Precision, and Failure Rate



(a) Porto – Recall (b) Porto – Precision (c) Jakarta – Recall (d) Jakarta – Precision

Figure 10: Impact of Accuracy Threshold on Recall and Precision



(a) Training Time (b) Imputation Time

Figure 11: Timing Analysis

8.3 Training and Imputation Time

Figure 11 gives the training and imputation time for Porto and Jakarta datasets. For training time, apparently Kamel takes more time than TrImpute, with 680 minutes for Porto and 780 minutes for Jakarta. This is mainly because Kamel inherits the complex training model from BERT, while TrImpute training basically computes a simple set of stats and lookup indices. However, we do not see this as a major issue since training is an offline process and only happens periodically whenever a bulk of new trajectories is received, and without affecting the imputation or causing any system downtime. Figure 11(b) gives the average imputation time for trajectories. Kamel takes longer with 540 seconds for Porto and 220 seconds for Jakarta, which is mainly due to the Multipoint Imputation module that trades-off accuracy with imputation time. Since our main goal in Kamel is higher accuracy, we tune our modules to achieve higher accuracy, while accepting the imputation time overhead.

8.4 Impact of Road Type

This section aims to understand the behavior of Kamel and its competitors for straight and curved road segments.Hence, as we have the ground truth for our test trajectories, we classify each test trajectory segment into two types: *straight* and *curved*. A segment is identified as *straight* if the Euclidean distance between its two end points is within a very small threshold (5m by default) from their road network distance, otherwise, the segment is identified as *curved*. Figures 12-I and 12-II repeat the same experiments of Sections 8.1 and Section 8.2, when only focusing on *straight* and *curved* segments, respectively. Experiments are only shown for Jakarta dataset. Porto dataset gives a similar analysis and conclusion.

For *straight* segments (Figure 12-I), Kamel outperforms its competitors in all measures, sparsity gaps, and accuracy thresholds. For sparsity gaps (Figures 12-I(a)-(b)), TrImpute gives the worst performance, as it gets distracted with various directions of surrounding GPS points and misses the easiest case of going in a straight direction. Kamel outperforms linear interpolation, mainly due to our definition of a *straight* segment, which allows for a 5m threshold. For failure rate (Figure 12-I(c), Kamel significantly outperforms others. For various thresholds δ (Figures 12-I(d)-(e)), both Kamel and linear interpolation exhibit similar high performance, while TrImpute has a much worse performance for lower δ.

For *curved* segments (Figure 12-II), Kamel consistently has the highest performance in all measures. For the most practical medium gaps, Kamel significantly outperforms TrImpute, which shows the resilience of Kamel towards curved segments. For failure rates (Figure 12-II(c)), Kamel consistently outperforms its competitors. For various thresholds δ (Figures 12-II(d)-(e)), it is expected that the performance of TrImpute and linear interpolation catch with Kamel for large δ, which is a very relaxed value that is not practical in most applications. For practical medium δ, Kamel is clearly better, showing its applicability to most trajectory applications. For very tight values of $\delta < 10m$, both TrImpute and linear interpolation are useless, while Kamel has more than 50% recall and precision.

8.5 Impact of Grid Type

Figure 12-III compares two tokenization alternatives for Kamel, namely, Uber H3 Hexagons [24] and Google S2 Squares [59]. For S2, we set the edge length to 120m to ensure a similar area coverage as that of hexagons with edge length of 75m. In all measures, a hexagonal grid gives better performance due to its unique properties. In particular, all neighbors of a certain hexagonal cell have identical properties in terms of the length of shared borders and the distance between their centroids, which makes the transition patterns between cells more consistent and easier to learn.

8.6 Impact of Training Data Properties

Figure 12-IV studies the impact of the training data size for Jakarta dataset (Porto dataset gives similar behavior), where we compare four variants of Kamel trained on 100%, 75%, 50%, and 25% of the available training trajectories. In all metrics, the three variants 100%, 75%, and 50% perform almost identically with only some minor differences. Only the 25% variant shows a noticeable reduction in performance. This shows that Kamel can still achieve its good results even with as little as half of the data it has. Figure 12-V studies the impact of the density of training data. We train Kamel using four variations of Jakarta dataset, all have the same number of trajectories but differ in their GPS density. Those variants include the original dense dataset which has a sampling rate of 1 second, and three other sparse variants with sampling rates 15, 30, and 60 seconds. For all metrics, Kamel still achieves almost the same performance for the 1 and 15 seconds sampling rates. It is important

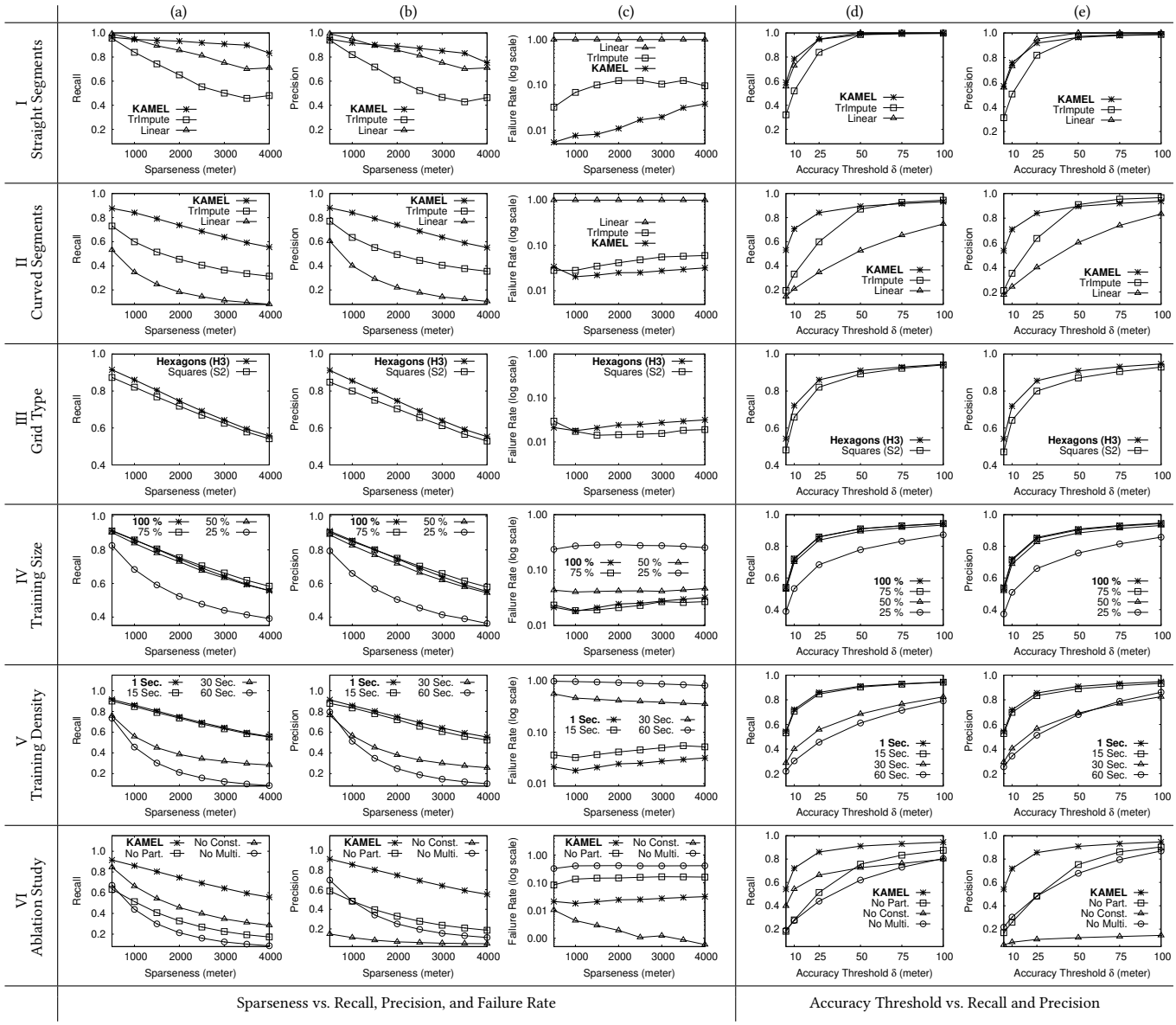


Figure 12: Performance Analysis Under Various Scenarios

to note that the 15 seconds sampled data is basically 15 times less (7%) of the original data. While having dense data is much better for training, this experiment shows that KAMEL can still work perfectly fine with only 7% of its available data for Jakarta.

8.7 Ablation Analysis

Figure 12-VI performs an ablation study of KAMEL using Jakarta dataset, to understand the impact of each of its components by disabling them one at a time and then measure all performance metrics. In particular, we compare four system variants: (a) KAMEL: our full system as explained in this paper, (b) *No Part.*: disables the Spatial Partitioning module (Section 4) by training one BERT model for the entire data, (c) *No Const.*: disables the Spatial Constraints module (Section 5) by accepting any BERT prediction, and (d) *No Multi.*: disables the Multipoint Imputation module (Section 6) by calling BERT only once to get a single imputation point.

For recall and precision across various $Sparse_{distance}$ (Figures 12-VI(a) and (b)), removing any of KAMEL components significantly reduces its accuracy. For the recall, removing the multipoint imputation affects the performance the most because the system now predicts only one point for each gap and leaves the rest of it, hence reduces the recall. In contrast, removing the spatial constraints affects the precision the most while affecting the recall the least. This is because removing these constraints does not prevent the system from still predicting some accurate imputations, but will allow it to predict so many noisy points. For the failure rate (Figure 12-VI(c)), removing any module causes KAMEL to fail more except when removing the spatial constraints, which causes less failure rate. This

is because BERT is now allowed to make any prediction regardless of how imprecise it is. Removing the multipoint imputation module causes Kamel to fail the most as this makes Kamel predict only one point for each gap. Similarly, in Figures 12-VI(d) and (e), the recall and precision for the full Kamel for all accuracy thresholds δ is higher than it is when removing any of its components. For the recall, one observation here is that the variant with no spatial partitioning starts lower than other variants at lower thresholds δ, but then outperforms them with the increase of δ. This shows the maximum effect of the spatial partitioning is at lower thresholds. This is where the locally trained models are mostly needed to improve the accuracy. For precision, without the spatial constraints module, the precision is consistently low around 10% or less regardless of δ.

9 RELATED WORK

Trajectory Imputation with Road Network. The immense need for high resolution trajectories motivated the efforts to insert points between each two consecutive trajectory points. Such a process had various names, including trajectory interpolation [39, 79], completion [35], cleaning [77], restoration [34], recovery [69, 70], and imputation [20, 50]. The large majority of such work mainly rely on matching trajectory points on the underlying road network [8, 40], followed by a shortest path algorithm between consecutive points. Unfortunately, none of this work is applicable to us as they all rely on an underlying road network. In our case, we aim for imputation without road network knowledge as our target applications include map inference, which needs to infer the *unknown* road network.

Trajectory Imputation without Road Network. Motivated by the unreliability of road networks [17, 23] and the need to develop trajectory-based map inference techniques [1], it becomes highly important to develop trajectory imputations techniques that do not rely on the road network, and hence can be used as a preparation step before any map inference technique [7, 14, 58, 63]. Unfortunately, existing techniques suffer from one or more of the following: (1) only impute one or two points between each two trajectory points [69], (2) only applicable to impute high granularity zones [32, 55], (3) only applicable to junction-level road network [35]. Our proposed Kamel system overcomes all of these issues as it scales up to impute tens of points between each two trajectory points, applicable to a very fine granularity of GPS points, and applicable to a city-scale road network. The closest work to ours is TrImpute [20], which uses historical data to find the most likely imputed points for each trajectory segment. However, it only works when there are significant amounts of highly dense historical data, which is not a practical case. We consider TrImpute as our direct competitor and compare against it in our experiments.

NLP Models and Trajectories. The BERT language model was introduced in 2018 [19] as an infrastructure for a myriad of complex linguistic tasks, including sentence completion. It utilizes the Transformer neural network [68], trained on vast textual corpora by removing random words from sentences and tasks the model with predicting each missing word based on surrounding context (left and right words). Since then, several new models and variations were proposed, including XLNet [75], RoBERTa [38], DistilBERT [60], ALBERT [33], and GPT [11], each focusing on a certain aspect such as training size, objectives, and masking procedures. Research efforts in employing language models for the spatial domain mainly utilize it as a *pretrained* model in its lingual form by verbally asking it questions of spatial nature [29, 53, 73]. Our system Kamel does not use a pretrained BERT model. Instead, it trains BERT with trajectory data through its system architecture and five components built around BERT. We have chosen the BERT model as it is the first and most commonly used language model, yet other BERT variants can be also used. Our goal is not to find which language model is best suited for trajectories. Instead, our goal is to show that language models, in general, trained with trajectory data, can be adopted to solve trajectory imputation with higher accuracy than state-of-the-art techniques. We opted for a design that employs BERT as is. Another design alternative would be to embed spatial awareness inside the core of each internal BERT component, as outlined in our recent vision paper [52]. Our rationale is that our design is more attractive to a large sector of systems that already deploy the BERT model and would need less disturbance to their systems. Future work would explore the second design option by carefully looking into the internal components of BERT one by one.

Time-series Imputation. In the same way we have mapped the trajectory imputation problem to the missing word problem and used NLP models to solve it, one can also consider mapping it to the multivariate time series imputation problem, and use any of its solutions that are based on Generative Adversarial Networks [41, 42, 61], Generative Learning Models [13, 46, 57], Recurrent Neural Networks [12], Graph Neural Networks [16], or attention mechanisms [78]. Similar to the case of language models, solutions to the multivariate time series imputation: (a) assume only few missing data points, hence to support trajectory imputation, it would need to go through a multipoint imputation module similar to that of Section 6, (b) mainly impute sensor readings, which have a very limited domain of values, hence to support trajectory imputation, it would need Tokenization and Detokenization modules similar to that of Sections 3 and 7, (c) are not spatially-aware, hence to support trajectory imputation, it would need spatial partitioning and spatial constraints modules similar to that of Sections 4 and 5. In essence, an orthogonal approach to ours is to start from a multivariate time series imputation algorithm and adopt it in a similar way to what we have done in Kamel for NLP models.

10 CONCLUSION

This paper has presented Kamel; a scalable BERT-based system for trajectory imputation. Kamel maps the trajectory imputation problem to *finding the missing word* problem in natural language processing (NLP). Hence, Kamel starts from using BERT, the widely used language model for NLP problems. However, BERT, as is, does not lend itself to the special characteristics of the trajectory imputation problem. Hence, Kamel architecture is composed of five main modules to make BERT applicable for trajectory imputation. These modules adapt the nature of trajectory data to be more fit for BERT, and inject the spatial-awareness in both the input and output of BERT. Experimental results based on real system implementation and real datasets show that Kamel significantly outperforms its competitors. In addition, unlike all related approaches, Kamel was able to impute city-scale trajectories, large sparse gaps that need tens of imputed points, all within tight accuracy thresholds.

REFERENCES

[1] S. Abbar, M. Alizadeh, F. Bastani, S. Chawla, S. He, H. Balakrishnan, and S. Madden. The Science of Algorithmic Map Inference (Tutorial). In *KDD*, 2018.

[2] R. Alseghayer. Racoon: Rapid Contact Tracing of Moving Objects Using Smart Indexes. In *MDM*, 2021.

[3] Road Inference From GPS Trajectories. https://www.amazon.science/publications/ring-net-road-inference-from-gps-trajectories-using-a-deep-segmentation-network/.

[4] Apple rolls out all-new map across Belgium, Liechtenstein, Luxembourg, the Netherlands, and Switzerland. https://www.apple.com/cm/newsroom/2022/12/apple-rolls-out-all-new-map-across-belgium-liechtenstein-luxembourg-the-netherlands-and-switzerland/.

[5] W. G. Aref and H. Samet. Efficient Processing of Window Queries in The Pyramid Data Structure. In *PODS*, 1990.

[6] Bert implementation by google. https://github.com/google-research/bert/.

[7] J. Biagioni and J. Eriksson. Inferring Road Maps from Global Positioning System Traces: Survey and Comparative Evaluation. *Transportation Research Record: Journal of the Transportation Research Board*, 2291(1), 2012.

[8] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk. On Map-Matching Vehicle Tracking Data. In *VLDB*, 2005.

[9] I. R. Brilhante, J. A. F. de Macêdo, F. M. Nardini, R. Perego, and C. Renso. Planning Sightseeing Tours Using Crowdsensed Trajectories. *ACM SIGSPATIAL Special*, 7(1), 2015.

[10] I. Brodsky. Uber's Hexagonal Hierarchical Spatial Index. https://eng.uber.com/h3.

[11] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language Models are Few-Shot Learners. In *NeurIPS*, 2020.

[12] Z. Che, S. Purushotham, K. Cho, D. Sontag, and Y. Liu. Recurrent Neural Networks for Multivariate Time Series with Missing Values. *Nature Scientific Reports*, 2018.

[13] Z. Che, S. Purushotham, M. G. Li, B. Jiang, and Y. Liu. Hierarchical Deep Generative Models for Multi-Rate Multivariate Time Series. In *Proceedings of the International Conference on Machine Learning, ICML*, 2018.

[14] C. Chen, C. Lu, Q. Huang, Q. Yang, D. Gunopulos, and L. J. Guibas. City-Scale Map Creation and Updating using GPS Collections. In *KDD*, 2016.

[15] L. Chen, S. Shang, C. S. Jensen, B. Yao, Z. Zhang, and L. Shao. Effective Efficient Reuse of Past Travel Behavior for Route Recommendation. In *KDD*, 2019.

[16] A. Cini, I. Marisca, and C. Alippi. Filling the G_ap_s: Multivariate Time Series Imputation by Graph Neural Networks. In *International Conference on Learning Representations, ICLR*, 2022.

[17] CNN Buisness. The Billion Dollar War over Maps. https://money.cnn.com/2017/06/07/technology/business/maps-wars-self-driving-cars/index.html.

[18] P. Cudré-Mauroux, E. Wu, and S. Madden. TrajStore: An Adaptive Storage System for Very Large Trajectory Data sets. In *ICDE*, 2010.

[19] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: Pre-training Deep Bidirectional Transformers for Language Understanding. *CoRR*, abs/1810.04805, 2018.

[20] M. M. Elshrif, K. Isufaj, and M. F. Mokbel. Network-less trajectory imputation. In *SIGSPATIAL*, 2022.

[21] M. Ester, H. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD*, 1996.

[22] Q. Gao, G. Trajcevski, F. Zhou, K. Zhang, T. Zhong, and F. Zhang. DeepTrip: Adversarially Understanding Human Mobility for Trip Recommendation. In *SIGSPATIAL*, 2019.

[23] Grand View Research. Abolute Reports. Global High Accuracy Map Market Size, Status and Forecast 2021-2027, Mar. 2020. https://www.grandviewresearch.com/industry-analysis/digital-map-market.

[24] H3: Hexagonal Hierarchical Geospatial Indexing System. https://h3geo.org/.

[25] Conversion from latitude/longitude to containing H3 Cell Index. https://h3geo.org/docs/core-library/latLngToCellDesc/.

[26] S. He, F. Bastani, S. Abbar, M. Alizadeh, H. Balakrishnan, S. Chawla, and S. Madden. Roadrunner: Improving the Precision of Road Network Inference from GPS Trajectories. In *SIGSPATIAL*, 2018.

[27] T. He, J. Bao, S. Ruan, R. Li, Y. Li, H. He, and Y. Zheng. Interactive Bike Lane Planning Using Sharing Bikes' Trajectories. *TKDE*, 32(8), 2020.

[28] B. Hossain, K. A. Adnan, M. F. Rabbi, and M. E. Ali. Modelling Road Traffic Congestion from Trajectories. In *DSIT*, 2020.

[29] J. Huang, H. Wang, Y. Sun, Y. Shi, Z. Huang, A. Zhuo, and S. Feng. ERNIE-GeoL: A Geography-and-Language Pre-trained Model and its Applications in Baidu Maps. In *KDD*, 2022.

[30] X. Huang, Y. Yin, S. Lim, G. Wang, B. Hu, J. Varadarajan, S. Zheng, A. Bulusu, and R. Zimmermann. Grab-posisi: An extensive real-life GPS trajectory dataset in southeast asia. In *PredictGIS@SIGSPATIAL*, 2019.

[31] B. B. Krogh, O. Andersen, E. Lewis-Kelham, N. Pelekis, Y. Theodoridis, and K. Torp. Trajectory Based Traffic Analysis. In *SIGSPATIAL*, 2013.

[32] J. Krumm. Maximum Entropy Bridgelets for Trajectory Completion. In *SIGSPATIAL*, 2022.

[33] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In *ICLR*, 2020.

[34] B. Li, Z. Cai, M. Kang, S. Su, S. Zhang, L. Jiang, and Y. Ge. A Trajectory Restoration Algorithm for Low-sampling-rate Floating Car Data and Complex Urban Road Networks. *International Journal of GIS*, 35(4), 2021.

[35] Y. Li, Y. Li, D. Gunopulos, and L. J. Guibas. Knowledge-based Trajectory Completion from Sparse GPS Samples. In *SIGSPATIAL*, 2016.

[36] Y. Li, J. Luo, C. Chow, K. Chan, Y. Ding, and F. Zhang. Growing Charging Station Network For Electric Vehicles With Trajectory Data Analytics. In *ICDE*, 2015.

[37] F. Lin and H. Hsieh. An Intelligent And Interactive Route Planning Maker For Deploying New Transportation Services. In *SIGSPATIAL*, 2018.

[38] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR*, abs/1907.11692, 2019.

[39] J. A. Long. Kinematic Interpolation of Movement Data. *International Journal of Geographical Information Science*, 30(5), 2016.

[40] Y. Lou, C. Zhang, Y. Zheng, X. Xie, W. Wang, and Y. Huang. Map-Matching for Low-Sampling-Rate GPS Trajectories. In *SIGSPATIAL*, 2009.

[41] Y. Luo, X. Cai, Y. Zhang, J. Xu, and X. Yuan. Multivariate Time Series Imputation with Generative Adversarial Networks. In *Advances in Neural Information Processing Systems, NeurIPS*, 2018.

[42] Y. Luo, Y. Zhang, X. Cai, and X. Yuan. E^2GAN: End-to-End Generative Adversarial Network for Multivariate Time Series Imputation. In *IJCAI*, 2019.

[43] Lyft Engineering. How Lyft Creates Hyper-Accurate Maps from Open-Source Maps and Real-Time Data. https://eng.lyft.com/how-lyft-creates-hyper-accurate-maps-from-open-source-maps-and-real-time-data-8dcf9abdd46a.

[44] Mapillary. Unveiling the Mapping in Logistics Report: The Impact of Broken Maps on Last-Mile Deliveries. https://blog.mapillary.com/update/2020/02/14/mapping-in-logistics.html.

[45] C. Meng, X. Yi, L. Su, J. Gao, and Y. Zheng. City-wide Traffic Volume Inference with Loop Detector Data and Taxi Trajectories. In *SIGSPATIAL*, 2017.

[46] X. Miao, Y. Wu, J. Wang, Y. Gao, X. Mao, and J. Yin. Generative Semi-supervised Learning for Multivariate Time Series Imputation. In *AAAI*, 2021.

[47] Discover New Roads with Bing Maps. https://blogs.bing.com/maps/2022-12/Bing-Maps-is-bringing-new-roads/.

[48] M. F. Mokbel, S. Abbar, and R. Stanojevic. Contact Tracing: Beyond the Apps. *ACM SIGSPATIAL Special*, 12(2), 2020.

[49] M. Musleh, S. Abbar, R. Stanojevic, and M. F. Mokbel. QARTA: An ML-based System for Accurate Map Services. *PVLDB*, 14(11), 2021.

[50] M. Musleh and M. Mokbel. A Demonstration of KAMEL: A Scalable BERT-based System for Trajectory Imputation. In *SIGMOD*, 2023.

[51] M. Musleh and M. F. Mokbel. RASED: A Scalable Dashboard for Monitoring Road Network Updates in OSM. In *MDM*, 2022.

[52] M. Musleh, M. F. Mokbel, and S. Abbar. Let's Speak Trajectories. In *SIGSPATIAL*, Seattle, WA, USA, 2022.

[53] V. K. Penumadu, N. Methani, and S. Sohoney. Learning Geospatially Aware Place Embeddings via Weak-supervision. In *SIGSPATIAL*, 2022.

[54] Taxi Service Trajectory. Prediction Challenge. ECML PKDD 2015. http://www.geolink.pt/ecmlpkdd2015-challenge/dataset.html.

[55] K. K. Qin, Y. Ren, W. Shao, B. Lake, F. Privitera, and F. D. Salim. Multiple-level Point Embedding for Solving Human Trajectory Imputation with Prediction. *ACM TSAS*, 2023.

[56] D. R. Reddy et al. Speech Understanding Systems: A Summary of Results of the Five-Year Research Effort. *Department of Computer Science. Carnegie Mellon University, Pittsburgh, PA*, 17, 1977.

[57] X. Ren, K. Zhao, P. J. Riddle, K. Taskova, Q. Pan, and L. Li. DAMR: Dynamic Adjacency Matrix Representation Learning for Multivariate Time Series Imputation. *Proceeding of ACM Management of Data*, 1(2), 2023.

[58] S. Ruan, C. Long, J. Bao, C. Li, Z. Yu, R. Li, Y. Liang, T. He, and Y. Zheng. Learning to Generate Maps from Trajectories. In *AAAI*, 2020.

[59] S2 Spherical Geometry Library. https://s2geometry.io/.

[60] V. Sanh, L. Debut, J. Chaumond, and T. Wolf. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108, 2019.

[61] R. Shahbazian and S. Greco. Generative Adversarial Networks Assist Missing Data Imputation: A Comprehensive Survey and Evaluation. *IEEE Access*, 11, 2023.

[62] Z. Shang, G. Li, and Z. Bao. DITA: Distributed In-Memory Trajectory Analytics. In *SIGMOD*, 2018.

[63] R. Stanojevic, S. Abbar, S. Thirumuruganathan, S. Chawla, F. Filali, and A. Aleimat. Robust Map Inference through Network Alignment of Trajectories. In *SDM*, 2018.

[64] Traffic Technology Today. Poor maps costing delivery companies US $6bn annually. https://www.traffictechnologytoday.com/news/mapping/poor-maps-costing-delivery-companies-us6bn-annually.html.

[65] Uber engineering. enhancing the quality of uber maps with metrics computation. https://eng.uber.com/maps-metrics-computation/.

[66] UCR STAR: The UCR Spatio-temporal Active Repository. OSM/GPS Dataset for Portland, OR, USA. https://star.cs.ucr.edu/?OSM/GPS#center=45.5428,-122.6544.

[67] A. Vahedian, X. Zhou, L. Tong, Y. Li, and J. Luo. Forecasting Gathering Events through Continuous Destination Prediction. In *SIGSPATIAL*, 2017.
[68] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is All you Need. In *NeurIPS*, 2017.
[69] J. Wang, N. Wu, X. Lu, W. X. Zhao, and K. Feng. Deep Trajectory Recovery with Fine-Grained Calibration using Kalman Filter. *TKDE*, 33(3), 2021.
[70] H. Wu, J. Mao, W. Sun, B. Zheng, H. Zhang, Z. Chen, and W. Wang. Probabilistic Robust Route Recovery with Spatio-Temporal Dynamics. In *KDD*, 2016.
[71] Y. Wu et al. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation, 2016.
[72] L. Xiong, C. Shahabi, Y. Da, R. Ahuja, V. Hertzberg, L. Waller, X. Jiang, and A. Franklin. REACT: Real-Time Contact Tracing and Risk Monitoring Using Privacy-enhanced Mobile Tracking. *ACM SIGSPATIAL Special*, 12(2), 2020.
[73] H. Xue, B. P. Voutharoja, and F. D. Salim. Leveraging Language Foundation Models for Human Mobility Forecasting. In *SIGSPATIAL*, 2022.
[74] C. Yang and G. Gidofalvi. Fast Map Matching, An Algorithm Integrating Hidden Markov Model With Precomputation. *IJGIS*, 32(3), 2018.
[75] Z. Yang, Z. Dai, Y. Yang, J. G. Carbonell, R. Salakhutdinov, and Q. V. Le. XLNet: Generalized Autoregressive Pretraining for Language Understanding. In *NeurIPS*, 2019.
[76] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang. T-Drive: Driving Directions Based on Taxi Trajectories. In *SIGSPATIAL*, 2010.
[77] A. Zhang, S. Song, J. Wang, and P. S. Yu. Time Series Data Cleaning: From Anomaly Detection to Anomaly Repairing. *PVLDB*, 10(10), 2017.
[78] J. Zhao, C. Rong, C. Lin, and X. Dang. Multivariate time series data imputation using attention-based mechanism. *Neurocomputing*, 542, 2023.
[79] K. Zheng, Y. Zheng, X. Xie, and X. Zhou. Reducing Uncertainty of Low-Sampling-Rate Trajectories. In *ICDE*, 2012.
[80] F. Zhou, H. Wu, G. Trajcevski, A. A. Khokhar, and K. Zhang. Semi-supervised Trajectory Understanding with POI Attention for End-to-End Trip Recommendation. *ACM TSAS*, 6(2), 2020.