Network-less Trajectory Imputation

Mohamed M. Elshrif
Qatar Computing Research Institute
Doha, Qatar
melshrif@hbku.edu.qa

Keivin Isufaj
Qatar Computing Research Institute
Doha, Qatar
keisufaj@hbku.edu.qa

Mohamed F. Mokbel*
University of Minnesota, Twin Cities
Minneapolis, MN, USA
mokbel@umn.edu

ABSTRACT

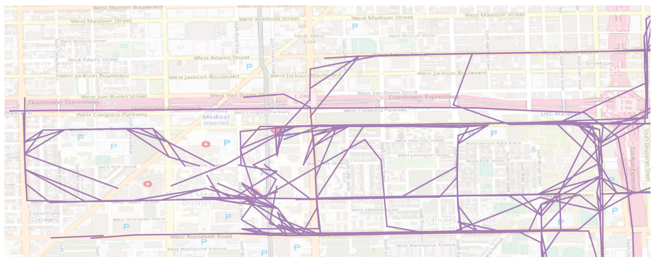
The ability to collect large numbers of trajectory data through GPS-enabled devices have enabled a myriad of very important applications that are widely used on a daily basis. This includes urban computing, transportation, and map APIs for routing and navigation. Unfortunately, a major hinder for all these applications is the accuracy of collected trajectories. Due to low sampling rates, trajectories are usually sparse in terms of the large spatial and temporal distances between each two consecutive collected points. This paper presents TrImpute; a novel framework for trajectory imputation that inserts artificial GPS points between the real ones in a way that the imputed trajectories end up to be very similar to the case if such trajectories were collected with a much higher sampling rate. Unlike all prior trajectory imputation techniques, TrImpute does not assume the knowledge of the underlying road network. This makes it more practical when the underlying road network is not available or inaccurate. Experimental results on real datasets and a real deployment of TrImpute show that it is highly scalable, accurate, and can significantly boost the performance of trajectory applications by feeding them highly accurate trajectories.

1 INTRODUCTION

Trajectory data, produced from a variety of GPS-enabled devices (e.g., vehicles, smart phones, and wearable devices) and represented by a sequence of spatio-temporal GPS points, have enabled a myriad of highly important applications and fundamental operations. Examples of such applications include urban computing [57], transportation [29], map inference [25, 44], data-driven routing [24, 40], and traffic prediction [11, 34, 48]. Examples of fundamental trajectory operations that empower such applications include trajectory similarity [19, 54], clustering [30, 51], and analytics [10, 50, 58]. Due to the importance of such applications and operations, several systems are built for scalable and accurate trajectory data management [18, 20, 32, 43].

Though all such trajectory systems, applications, and operations have been enabled by the sheer sizes of trajectory data (in terms of the number of trajectories or GPS points), they all significantly suffer from the sparsity of each single trajectory. As trajectory data comes from devices geared towards battery savings, trajectories are usually sparse with large spatial and temporal gaps between consecutive points within the same trajectory. This severely degrades the accuracy of the applications that rely on trajectory data. To address such important accuracy gap, several recent efforts were dedicated to increase the accuracy of collected trajectory data through a process that had various names, e.g., trajectory interpolation [35, 56], trajectory completion [33], trajectory data cleaning [55], trajectory restoration [31], and trajectory imputation [14]. Without loss of generality, in this paper, we use the term ”trajectory imputation”. The main goal of all these approaches is to insert artificial points between each two consecutive trajectory points, with the promise that the artificially imposed points are as accurate as if there were actual readings of trajectory data. The large majority of such techniques assume the existence of the underlying road network. Hence, the trajectory imputation process becomes mainly a map matching process of sparse trajectory points [7, 12, 27, 41, 52], followed by inserting artificial points that match the underlying road network.

*The work of this author is partially supported by the National Science Foundation, USA, under Grants IIS-1907855 and IIS-2203553.


(a) Map Inference with Raw trajectories

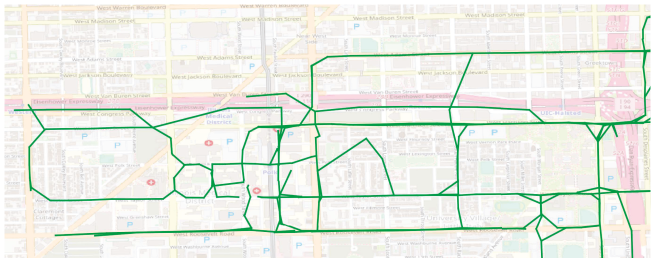

(b) Map Inference with TrImpute trajectories

Figure 1: Effect of TrImpute on Map Inference Application

Unfortunately, the assumption of having the underlying network is not always valid. The recent explosion in using map services have actually shown that current maps are not as accurate as it looks [3, 36, 47]. This triggered a whole multi-billion dollars industry for constructing accurate maps [17, 23]. As a result, several recent research efforts are dedicated to map inference (a.k.a map making), which basically aim for constructing accurate road network either from trajectory data [6, 15, 42, 44] or satellite images [5, 16, 45]. With such inaccuracy in maps, the process of trajectory imputation can be seen as first constructing the map with any of the existing map inference algorithms, then, imputing the trajectories. However, this results in a very low accuracy, as the accuracy of map inference itself is highly dependent on the trajectory sampling rate. Up to the authors’ knowledge, the only trajectory imputation work that does not assume or build a road network [33] still builds a skeleton of the road network, and is applicable only to a junction-level road network. This makes it inherently inaccurate and does not scale to a city-level trajectory imputation.

In this paper, we propose TrImpute; a novel approach for Network-less Trajectory Imputation. Unlike all other trajectory imputation approaches, TrImpute neither relies on or builds its own underlying road network, and hence the term "Network-less". Instead, TrImpute relies on the crowd wisdom by taking advantage of neighboring GPS points to guide its imputation process for each sparse trajectory. TrImpute serves as a preprocessing step for various trajectory applications, operations, and systems, to significantly boost their accuracy. For example, Figure 1 shows the effect of TrImpute on map inference algorithms. In particular, Figure 1(a) gives the result of running a map inference algorithm [44] using raw sparse trajectory data in the area of downtown Chicago[1]. The inferred map is plotted in blue, while the background map is shown as an image for reference only to visually assess the quality of the generated map. Meanwhile, Figure 1(b) gives the result of running the same exact map inference algorithm [44], plotted in green, but after imputing the raw sparse trajectories using TrImpute. It is visually clear that TrImpute did significantly boost the map inference algorithm accuracy. The green road network generated by TrImpute trajectories is way more accurate than the blue road network generated by raw sparse trajectories.

The main idea of TrImpute is to rely on the crowd wisdom to guide its imputation process as a substitute of the lack of knowledge of underlying road network. For TrImpute, any arbitrary GPS point can consult its neighboring points (regardless of their trajectories) to recommend a set of artificial *candidate points*, where each of these points could be possibly the next imputation point towards the segment destination. *Candidate points* must follow a set of strict properties (defined by TrImpute) to ensure that the imputed segment would be very similar to the case of a real high sampling rate of these segments. With the concept of *candidate points*, the spatial imputation process of TrImpute, which is done on one trajectory segment at a time, becomes finding a set of consecutive candidate points between the trajectory segment end points. Unlike other trajectory imputation techniques, TrImpute does not stop at spatial imputation, but it also goes for temporally imputing each segment. The TrImpute temporal imputation process basically annotates the spatially imputed points by temporal information that match the traffic conditions of these imputed points.

Extensive experimental results based on real trajectory data, collected from a real deployment of TrImpute (within the QARTA system [2, 37]), in 4,000 Taxis in the state of Qatar show that: (a) TrImpute is highly scalable to a city-scale trajectory imputation, (b) TrImpute imputed trajectories are very similar to the originally raw dense trajectories that were made sparse for experimental evaluation, (c) TrImpute imputed points have high accuracy when matched to a ground truth map, obtained from OpenStreetMap [39], and (d) TrImpute was able to significantly boost the performance of a map inference algorithm [44], when it was used as its a preprocessing step between the raw data and the map inference algorithm.

The rest of the paper is organized as follows: Section 2 discusses related work. TrImpute framework is described in Section 3. TrImpute main components, namely, spatial and temporal imputation, are described in Sections 4 and 5. Section 6 provides extensive experimental evaluation of TrImpute. Section 7 concludes the paper.

[1]The raw dataset can be found at https://www.cs.uic.edu/bin/view/Bits/Software.

2 RELATED WORK

Trajectory data has always been crucial to several important applications, including urban computing [57], transportation [29], map inference [25, 44], and routing [24, 40]. However, they all behave very poorly when trajectory data is relatively sparse. For example, a sparse trajectory data set will make map inference algorithms [6, 15, 25, 42, 44] give inaccurate maps, trajectory clustering techniques [4, 8, 30, 51] give inaccurate clusters, trajectory similarity search [19, 54] give inaccurate results, while trajectory data analysis in general [10, 50, 58] yield wrong results.

Hence, many efforts were dedicated to increase the accuracy of raw sparse trajectories by inserting artificial points, turning them into dense trajectories. The promise is that these newly inserted points are as accurate as if the trajectories were originally dense. Such efforts were performed under various names, e.g., trajectory interpolation [35, 46, 49], trajectory data cleaning [55], trajectory restoration [31], trajectory imputation [14], path inference [9, 22, 53, 56] and trajectory completion [33]. However, many of these efforts were not directed to road network and traffic analysis. This includes generating trajectories for inter-continental scale animal and bird movements, where trajectories may span hundreds or thousands of kilometers [46], interpolating cyclists or athletes trajectories using their kinematics (velocity and acceleration) information sampled at a high temporal resolution [35], interpolating trajectories of geolocated objects in video frames used for driving simulators [49], and enriching trajectory information with semantic meaning to infer the trip purpose [14]. None of this work is applicable to our case of road network vehicle movement.

When it comes to road network, some of the existing efforts actually address the trajectory imputation problem when the data is dense, rather than sparse. In this case, the objective is not to insert new artificial points to densify sparse trajectories. Instead, it is to remove noisy and outlier data as a means to make trajectories only include accurate points. This has taken various forms, including framing the problem as a time series data cleaning to detect and remove anomaly trajectory points [55], focusing on finding popular routes rather than constructing the routes [53], and removing points that do not clearly belong to certain clusters formed by other points [9, 22]. None of these approaches is applicable to our case, as our objective is trying to insert new points rather than removing existing points. Meanwhile, existing work in densifying raw sparse trajectories for vehicle data [31, 56] mainly start with a map matching process [7, 12, 27, 41, 52], where all GPS points are matched to the underlying road network. Once the road segments are identified, they are used to find intermediate points to densify raw trajectories. Unfortunately, none of this work is applicable to our case, as they assume the complete knowledge of the underlying network, which is not available in our case. .

Up to our knowledge, there is only one prior attempt that aims to impute raw trajectories without the knowledge of the road network [33]. However, it relies on the $L1$−medial skeleton extraction algorithm [28], which is inherently designed to only support small regions. Hence, the trajectory imputation process has a very limited applicability and can only support junction-level trajectories, e.g., trajectories in an intersection. This cannot be applied to our case where we need to impute trajectories in a city scale.

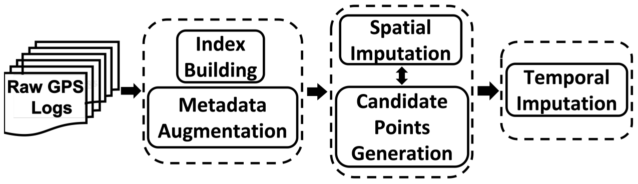


Figure 2: TrImpute Framework

3 TRIMPUTE FRAMEWORK

Figure 2 gives the architecture of our proposed TrImpute framework. The input to TrImpute is a set of raw GPS points, where each point P has the format $<$*TrajID*, *PointID*, *latitude*, *longitude*, *timestamp* $>$. The output would have the same format, yet, with much more points, as more artificial imputed points are inserted between each two raw points of the same trajectory. Within TrImpute, the input goes through three main components, namely, *preprocessing*, *spatial imputation*, and *temporal imputation*, as follows:

Preprocessing. TrImpute preprocesses input data with two tasks: (1) *Index Construction.* We bulk load input data to two index structures; a hierarchical quad-tree index based on the points locations [26], and an inverted list for trajectory IDs where each trajectory ID points to a list of its temporally ordered GPS points. One scan over all points is enough for both bulk loading operations. Both index structures will be used by the spatial imputation process to efficiently retrieve trajectory data. (2) *Angle and Speed Inference.* TrImpute relies on the angle and speed information of input GPS points to guide its imputation process. However, such information is not directly available. Hence, we utilize the trajectory inverted list to enrich each trajectory point with its angle and speed. Figure 3 gives an example of a trajectory $Traj_1$ with three points P_{11}, P_{12}, and P_{13}, each has its *latitude*, *longitude*, and *timestamp* information. We then define the angle of each point P_i as the angle between the east direction and the line connecting P_i to its consecutive point P_{i+1}. The last point P_j in each trajectory inherits the same angle of its preceding point. Hence, the angles of the three trajectory points are computed and stored as 30^o, 345^o, and 345^o. We then calculate and store the speed of each point P_i as the ratio of the distance between P_i and its consecutive point P_{i+1} to the time difference between these two points. The last trajectory point inherits the same speed of its preceding point. Hence, the speed of the three trajectory points are 15, 25, and 25 m/s.

Spatial Imputation. The spatial imputation component is responsible on adding artificially imputed points to all trajectories with the promise that these points are as accurate as if the trajectory data was collected with a higher sampling rate. Since TrImpute is a network-less imputation, it does so without the knowledge of the underlying network. Instead, it relies on the wisdom of the crowd composed of nearby GPS points of many other trajectories to suggest some *candidate points* that will contribute to the imputed path. Details are in Section 4.

Temporal Imputation. The temporal imputation component follows the spatial imputation to annotate the imputed points with timestamps that would match the temporal readings if these points were collected with a higher sampling rate. It does so by taking timely traffic conditions into account. Details are in Section 5.

Input Trajectory Attributes					Metadata	
GPS point		Lat	Lon	timestamp	angle	Speed (m/s)
$Traj_1$	P_{11}	51.493	25.282	1517504575	30°	15
	P_{12}	51.506	25.315	1517504578	345°	25
	P_{13}	51.629	25.374	1517504581	345°	25

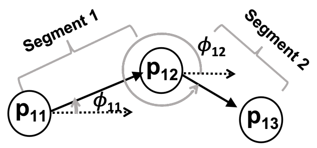


Figure 3: Example of angle and speed inference

4 TRIMPUTE SPATIAL IMPUTATION

This section describes the spatial imputation process, which is done per each trajectory segment that goes from point P to point Q. The objective is to insert a certain number of artificial points between P and Q that would mimic the situation if there are more accurate real readings between P and Q. The main idea of TrImpute spatial imputation is to define the concept of candidate points (Section 4.1) as the set of possible next artificial point(s) for any given point in the space. Then, the spatial imputation process (Section 4.2) becomes finding the shortest path from P to Q composed of a set of consecutive candidate points. It is worth emphasizing here that the modules of finding candidate points and spatial imputation are performed without any knowledge of the underlying network. Both modules rely on the wisdom of the crowd drawn from the set of GPS points of a large set of sparse trajectories.

4.1 Candidate Points

This section defines the concept of *candidate points* and describes an algorithm that gets such points for any given point.

Properties of Candidate Points. Given a start and end points P and Q, P_{Cand} is a set of artificial points (P_C) where any of them could possibly be the next imputed point after P towards Q. Each P_C has five attributes: latitude, longitude, angle, timestamp, and speed. Only one of the P_{Cand} points would finally contribute to the imputed path from P to Q. To ensure accurate imputation, we aim to find P_{Cand} that would satisfy the following four properties, where all properties include parameters that achieve a trade-off between accuracy and computational overhead.

(1) *Property 1: Number of candidates N.* The number of points in P_{Cand} should be capped by N. A higher value of N would suggest more alternatives, which gives more flexibility to come up with a more accurate imputation. Yet, this would come with more computational overhead that may not be needed.
(2) *Property 2: Crowd ratio threshold α.* According to the spirit of the TrImpute framework, all points in P_{Cand} should follow the crowd wisdom, which means avoid going in a direction that no one is going for. Hence, a parameter α is defined as the minimum possible crowd ratio for a direction to be considered. A lower value of α would mean exploring more directions, which provides more alternatives to find a more accurate imputation. Yet, this would come with more computational overhead that may not be needed.

(3) *Property 3: Angle threshold δ.* Each point P_C in P_{Cand} should not change the angle of P by more than a certain threshold δ. Recall that the angle of P is computed per its direction toward Q. Since P_C will be inserted between P and Q, it should not significantly disturb that angle. For example, P_C should not change P's angle to an opposite direction. The higher the value of δ, the more accuracy we can get as more options will be considered, however, more computations will be needed then.

(4) *Property 4: Distance threshold d.* All points in P_{Cand} should have a distance d from P. The lower the d, the more accurate the imputation, as there will be more points between P and Q. Yet, this will also result in a more computational overhead, where more points will be inserted between P and Q.

Main Idea. To come up with the set of points P_{Cand} that satisfy the above four properties, we aim to smartly use our budget of N candidate points to get a set of representative points. Our main idea is to avoid having candidate points that are close to each other. Instead, the N points should be diverse enough to cover a spectrum of imputation possibilities. To achieve this, (1) we divide the space around P into N buckets, where each bucket i (from 0 to $N-1$) covers the space between the angles $\frac{360}{N}i$ and $\frac{360}{N}(i+1)$. For example, if N=4, then, the space around P is divided into four quadrants. Then, each bucket would contribute, at most, one candidate point to P_{Cand}, which means that we will have at most N candidate points (*Property 1*). (2) Honoring the crowd wisdom (*Property 2*) with all the raw trajectory data points may not be practical. To resolve this, we will only consider the nearby crowd to point P, represented by the set of points R that fall within a circular range query centered at P with radius d. We then populate our N buckets with the points in R where each bucket B_i includes the set of points in R with angles that fall within the bucket angle range. Based on the number of points that land in each each bucket, we exclude any bucket B_i that has a crowd ratio (i.e., number of points in B_i divided by number of points R) below our threshold α. The rational here is that a low ratio of the crowd following the angles of these buckets gives an indication that these points could be outliers or paths that are not usually used by the large majority of trajectories. (3) To honor the angle threshold constraint (*Property 3*), we first assign an angle to each bucket B_i as the *average* angle of all points within B_i. Then, we exclude any bucket B_i with an angle $B_i.Angle$ outside the range $[P.Angle - \delta, P.Angle + \delta]$. The rational is that such buckets will end up in a candidate point that does not satisfy Property 3 above. (4) Finally, for each bucket B_i in the set of remaining buckets, we generate one candidate point with distance d from P (*Property 4*) in the direction of B_i's angle.

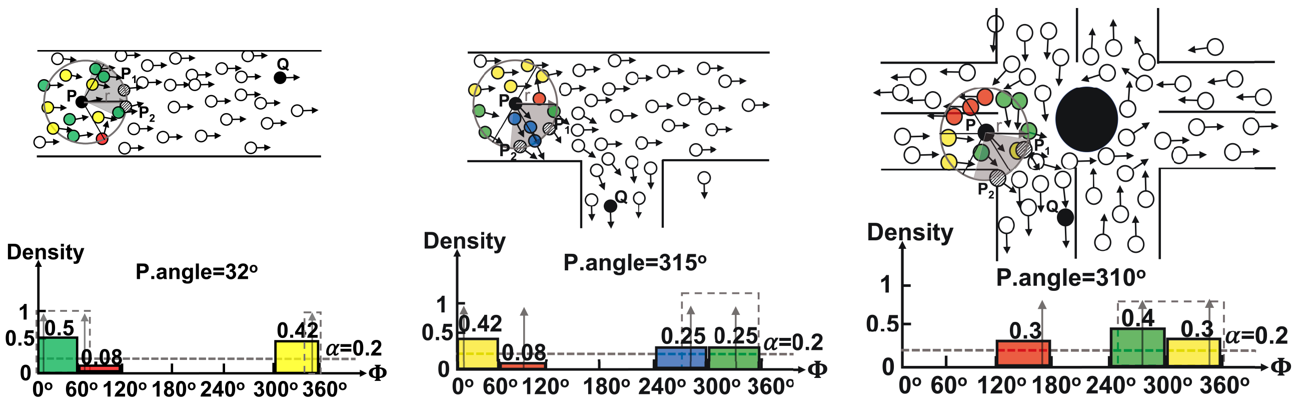
Algorithm. Algorithm 1 gives the pseudo code for generating the set of candidate points P_{Cand} to any given point P towards a destination point Q. In addition to P and Q, the algorithm takes, as input, the four parameters N, α, δ, and d that would ensure that all points in P_{Cand} satisfy the four properties mentioned above. The algorithm starts by using the two parameters N and d to build a histogram H of N buckets where each bucket i includes the set of points within distance d from P and have an angle within the range $\frac{360}{N}i$ and $\frac{360}{N}(i+1)$. Then, we use the parameters α and δ to exclude the buckets that would not satisfy the second and third properties, i.e., buckets that have crowd ratio less than α or their average angle is not within δ from P's angle. For the remaining buckets, we generate one candidate point per bucket within distance d from P in the direction of the bucket angle. We set the angle of that candidate point towards the direction of the destination point Q (line 8).

Algorithm 1 Candidate Points

```
1: procedure CandidatePoints($P$, $Q$, $N$, $\alpha$, $\delta$, $d$)
2:     $H \leftarrow BuildAngleHistogram(N, d)$
3:     $B \leftarrow QualifiedHistogramBuckets(P, H, \alpha, \delta)$
4:     $P_{Cand} \leftarrow \{\ \}$;
5:     for each bucket $b$ in $B$ do
6:         $P_C.lat \leftarrow P.lat + d \times \sin(b.Angle)$
7:         $P_C.lon \leftarrow P.lon + d \times \cos(b.Angle)$
8:         $P_C.angle \leftarrow Angle(P_C, Q)$
9:         $P_{Cand} \leftarrow P_{Cand} \cup P_C$
10:    end for
11:    return $P_{Cand}$
12: end procedure
```

Example: Figure 4 gives three examples of finding the set of candidate points for point P as the first step to impute the trajectory segment between P and Q. The examples show three different road topologies, depicted in the top part of the figure, namely, straight road (Figure 4(a)), T-shape intersection (Figure 4(b)), and roundabout (Figure 4(c)), where points P and Q are depicted as black circles, and the angle of point P is 32°, 315°, and 310°, respectively. These angels were calculated using the horizontal line (East direction) as the reference direction (For details see Section 3 and Figure 3). The bottom part of the figures depict the angle histogram of the points that lie within the circle of radius d around point P, which indicate the trajectory points that we will consider and take their wisdom. For illustration, we use a different color for each histogram bucket that matches the color of its corresponding points that contribute to it. Within each histogram bucket, an upward arrow is placed in the location of the bucket angle, which is average angle of all points within that bucket. For all examples, we set N=6 (i.e., we will only have 6 histogram buckets), α=0.2, and δ=45°.

In Figure 4(a), there are 12 points within distance d from P, where 6 of them (ratio 0.5) fall within the first histogram bucket (plotted in green) with an average angle of 10°, one point (ratio 0.08) falls within the second bucket (plotted in red) with angle 70°, and 5 points (ratio 0.42) fall within the sixth bucket (plotted in yellow) with average angle 355°. This means that only the first and sixth buckets are above the crowd ratio threshold α=0.2. Since both buckets also satisfy the angle threshold as their angles are within $\pm\delta$ (45°) from P's angle (32°) (depicted as grey sector in the top part of the figure and dotted rectangle in the bottom part), we generate one candidate point for each bucket (depicted as dashed circles) within distance d from P and on the direction of each bucket angle. In Figure 4(b), 12 points are within distance d from P, where 5, 1, 3, and 3 of them fall within the first, second, fifth, and sixth histogram buckets with average angles, 10°, 95°, 285°, and 330°. Since the second histogram bucket has a crowd ratio that is lower than our threshold α=0.2, we do not consider it any further. For the angle threshold, the first bucket angle is outside the allowed range as it has a difference of 55° from P's angle (315°), which is more than our allowed threshold of δ=45°. Hence, only the fifth and



(a) Straight road

(b) T-Shape intersection

(c) Roundabout

Figure 4: Candidate Points Example.

sixth buckets are qualified to produce candidate points, where we generate one for each. In Figure 4(c), 10 points are within distance d from P, where 3, 4, and 3 of them fall within the third, fifth, and sixth histogram buckets with average angles, 175°, 275°, and 350°. All buckets satisfy the crowd ratio threshold α=0.2. However, only the fifth and sixth buckets satisfy the angle threshold as their angles are within ±45° from P's angle . Hence, we generate two candidates as one for each of these two buckets.

4.2 Spatial Imputation Process

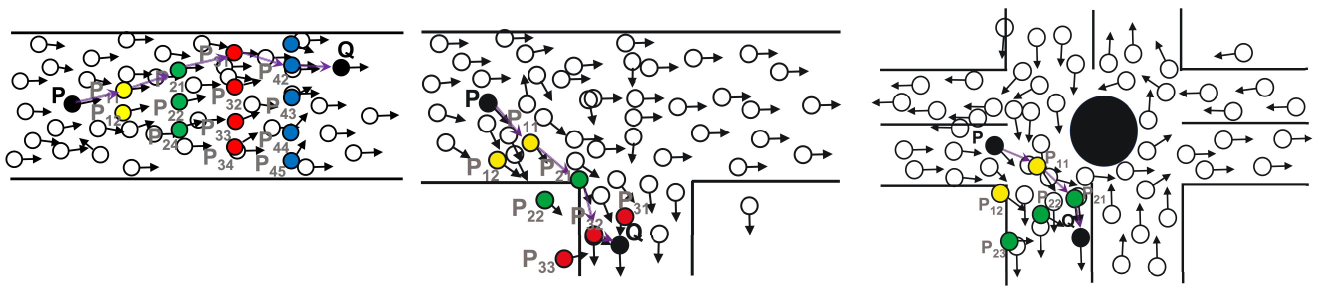
The spatial imputation process is basically trying to fill in the distance between P and Q with a set of consecutive candidate points such that the total distance from P to Q (and through all added candidate points) is the shortest possible path.

Main Idea. Since all candidate points are within distance d from their previous points, then the shortest possible path from P to Q would include the least possible number of candidate points. Hence, our main idea is to start exploring possible paths with c candidate points before we explore any path with c+1 candidate points. A shortest path is concluded if one of the latest candidate points is within distance d from the destination point Q. If there are multiple paths with the same number of candidate points, we pick the shortest one of them. To avoid exploiting an excessive number of paths, if a candidate point is within distance ϵ from a previously visited point, we do not consider it any further.

Algorithm. Algorithm 2 gives the pseudo code for the spatial imputation process, where the input is the source and destination points P and Q, and the output is a path, composed of a set of candidate points, from P to Q. We start by initializing the set of visited points, which will be used to store the visited points of all candidate paths, by the source and destination points, the current candidate paths (*CandPaths*) by only one path of length one composed of the source point, and the set of final paths by empty. For each iteration i, we consider those paths with length i candidate points. In each iteration, we call the *CandidatePoints* procedure (Algorithm 1) once per path, using the last point in every path in our current list of candidate paths (*CandPaths*). Note that *CandPaths[i]* will pass the first two parameters P and Q to *CandidatePoints( )* procedure and Algorithm 1 will add its own system parameters N, α, δ, and d to come up with the set of accurate candidate points P_{Cand}. Then, we call a procedure, called *FilterCandPoints* to: (a) remove from P_{Cand} those points that are within distance ϵ from any previously visited point and (b) populate a new list P_{Final} by the set of points in P_{Cand} that are within distance d from the destination Q. If P_{Final} is still empty, then we know that we still need at least one more candidate point to be closer to Q. Hence, we generate new $|CandPaths|$ candidate paths, as one for each candidate points P_C in P_{Cand} by augmenting the current path of i points with P_C, making a new path with i+1 points. Meanwhile, if P_{Final} is not empty, we know that we have a possible imputed path. Hence, we form the set *FinalPath*, composed of $|P_{Final}|$ paths, each formed by augmenting the current path with each point P_F in P_{Final}. The algorithm concludes by returning the shortest path among the ones in *FinalPath*.

Algorithm 2 Spatial Imputation

```
1: procedure TrImputeSpatial($P$, $Q$)
2:     $Visited \leftarrow [P, Q]$; $CandPaths \leftarrow [[P]]$; $FinalPath \leftarrow \{ \}$
3:     while $FinalPath$ is empty do
4:         $NewCandPaths \leftarrow \{ \}$
5:         for $i$ = 1 to $|CandPaths|$ do
6:             $P_{Cand} \leftarrow CandidatePoints(LastPoint(CandPaths[i]))$
7:             $P_{Cand}, P_{Final} \leftarrow FilterCandPoints(P_{Cand}, Visited)$
8:             if $P_{Final}$ is empty then
9:                 for each $P_C$ in $P_{Cand}$ do
10:                    Add $(CandPaths[i] + P_C)$ to $NewCandPaths$
11:                end for
12:            else
13:                for each $P_F$ in $P_{Final}$ do
14:                    Add $(CandPaths[i] + P_F)$ to $FinalPath$
15:                end for
16:            end if
17:        end for
18:        $CandPaths \leftarrow NewCandPaths$
19:    end while
20:    return the shortest distance path in $FinalPath$
21: end procedure
```



(a) Straight road (b) T-Shape intersection (c) Roundabout

Figure 5: The Spatial Imputation Process.

Example: Figure 5 builds on the example of Section 4.1 to show the full imputed path from P to Q for the straight road, T-shape intersection, and roundabout, given in Figure 4. For the straight road case (Figure 5(a)), the first iteration would be executed on point P and will be exactly as the case of Figure 4(a), where two candidate points P_{11} and P_{12} are returned (depicted in yellow). Now, we have two possible paths to consider in the second iteration, namely, [P, P_{11}] and [P, P_{12}]. Applying the candidate point procedure for each of these paths returns two candidate points for the first path and one candidate point for the second path (depicted in green). This leaves us with three possible paths so far, [P, P_{11}, P_{21}], [P, P_{11}, P_{22}], and [P, P_{12}, P_{24}]. Going on the same way, the third iteration gives four candidate points (depicted in red) and the fourth iteration gives five candidate points (depicted in blue), which leaves us with five possible paths. There is no need for a sixth iteration as we already have two candidate points P_{41} and P_{42} within distance d from Q. This gives us two possible imputed path, and we pick the shortest of them as: [P, P_{11}, P_{21}, P_{31}, P_{42}, Q]. For the T-shape intersection (Figure 5(b)), three iterations were enough to come up with an imputed path [P, P_{11}, P_{21}, P_{32}, Q]. We note that some of the imputed points (P_{22} and P_{33}) were off-road, which is still OK as the algorithm runs with no knowledge of the underlying road network. Yet, the algorithm was able to end up with an imputed path with only points that lie on the road. For the roundabout (Figure 5(c)), two iterations were enough to come up with an imputed path [P, P_{11}, P_{21}, Q]. Both the T-shape and roundabout cases show how TrImpute has nicely followed the crowd wisdom to come up with an imputed path that follows the complex road topology, even though it has no idea about the underlying road network.

5 TRIMPUTE TEMPORAL IMPUTATION

As discussed in Section 3, the *spatial* imputation process between points P and Q would need to be followed by a *temporal* imputation process, where the objective is to annotate the imputed points by timestamp and speed information. Given that the timestamp and speed are known for both end points P and Q, one straightforward way is to evenly split the time difference between Q and P over the set of imputed points. However, this may not be accurate as parts of the imputed segments may have higher traffic than others. Hence, there is a need for a more accurate temporal imputation.

Main Idea. Our main idea is to start by calculating the timestamp and speed for each imputed point P_i, based on the timestamp and speed of the previous point P_{i-1} and distance d. While this would respect the traffic at each part of the imputed segment, it would likely result in an obvious timestamp margin error δ_{ts}, where the timestamp of the last imputed point may not match the timestamp of Q. Hence, we take this margin error and evenly distribute it over all segments, and adjust the speed accordingly. So, in a nutshell, rather than evenly splitting the timestamp between P and Q, we evenly split the error margin among the segments, which still respect the difference in traffic along the segment. For example, if we have four imputed points, which means five segments from P and Q, we would decrease the time taken of each segment by one fifth of the timestamp margin error δ_{ts}.

Algorithm 3 Temporal Imputation

```
1:  procedure TrImputTemporal($Path$, $Q$)
2:      for $i$ = 1 to $|Path|$ do
3:          $Path[i].time \leftarrow Path[i-1].time + \frac{d}{Path[i-1].speed}$
4:          $Path[i].speed \leftarrow \frac{Distance(Path[i],Q)}{Q.time - Path[i].time}$
5:      end for
6:      $PathError \leftarrow Q.time - Path[|Path|].time$
7:      $\delta_{ts} \leftarrow \frac{PathError}{|Path|-1}$
8:      for $i$ = 1 to $|Path|$ do
9:          $Path[i].time \leftarrow Path[i].time + i \times \delta_{ts}$
10:         $Path[i].speed \leftarrow \frac{d}{Path[i].time - Path[i-1].time}$
11:     end for
12:     return $Path$
13: end procedure
```

Algorithm. Algorithm 3 gives the pseudo code of our temporal imputation procedure. The input to the algorithm is the spatially imputed *Path* between P and Q, where the first point (*Path[0]*) is P, and the last point of the path is Q. The algorithm goes through two main iterations over all points in the given path. The first iteration goes from the second till the last point of the given path, where for each point, we compute: (a) timestamp, as the timestamp of the previous point plus the distance between the two points (d) divided by the speed at the previous point, and (b) the speed, which is based on the distance and time difference from the destination point Q. Then, we calculate the error margin *PathError* as the time difference between the real value we have in Q and the last point estimated value from the first iteration. We then calculate the δ_{ts}

as the delta error that would evenly split over all the path segments. The second iteration deploys this error margin over the imputed points, where the first imputed point will be adjusted by adding the δ_{ts} to it. Next, the second imputed point will be adjusted by adding twice of δ_{ts} to it, which takes into account the errors in the first segment in addition to the error of the second segment. Generally speaking, the ith imputed point will be adjusted by adding $i \times \delta_{ts}$ to it. Speed information will be adjusted accordingly.

Example. Following up on the T-Shape intersection example of Figure 5(b), where the spatially imputed path is: $[P, P_{11}, P_{21}, P_{32}, Q]$. Assume that distance d=50 meters, and $P.time$=01:23:09, $P.speed$=25 m/s, $Q.time$=01:23:17, and $Q.speed$=25m/s. The first iteration will start by calculating $P_{11}.time$ as 01:23:09+50/25 = 01:23:11, which will make $P_{11}.speed$=16.67m/s. Then, we estimate $P_{21}.time$ as 01:23:11 + 50/16.67 as 01:23:14, which will make $P_{21}.speed$=21m/s. Next, we estimate $P_{32}.time$ as 01:23:14+50/21 = 01:23:16, which will make $P_{32}.speed$=10m/s. Finally, we estimate the $Q.time$ as 01:23:16+50/10 = 01:23:21. Now, we compute the path error exploiting the original time of Q, as $PathError$=01:23:17-01:23:21=-4 seconds, which means that each segment needs to be adjusted by δ_{ts}=-4/4=-1 sec. Hence, we add δ_{ts} to the first imputed points, twice of δ_{ts} to the second imputed points, and triple of δ_{ts} to the third imputed point. This will end up in having $P_{11}.time$ = 01:23:10, $P_{21}.time$ = 01:23:12, $P_{32}.time$ = 01:23:13, and $Q.time$ = 01:23:17.

6 EXPERIMENTAL EVALUATION

This section extensively evaluates our TrImpute framework based on real data obtained from a real deployment of the QARTA system [2, 37], which runs in all taxis (~4K) in the State of Qatar. Since our data is already dense, we impose our own sparsification over the data, where we sample the data of each trajectory according to some sparsification length. Unless mentioned otherwise, we use 200K trajectories (~23 millions of GPS points) with total length of trajectories ~766,000 Km that spanning an area of 64Km^2 from city of Doha, Qatar. We set sparsification length (spatial gap between two consecutive trajectory points) to 1Km. The TrImpute parameters are: number of candidate points N=12, crowd ratio threshold α=0.01, angle threshold δ=120^o, and distance threshold d=40 meters. In addition, we used ϵ=10 meters, as a tolerance distance to reduce the number of redundant candidate paths.

Baseline Algorithms: As mentioned in Section 2, all trajectory imputation techniques assume the knowledge of the underlying road network. Hence, none of them is applicable to our case, where we do not have the road network. The only exception is the *knowledge-based trajectory completion* framework [33] that can work without underlying road network. However, as confirmed by their own (and our) experiments, it is applicable only to very small networks, which can represent a small junction scale. Hence, it cannot be applied to a large city-scale setting, like Doha, Qatar, which we use to evaluate TrImpute. Hence, we are not considering [33] any further in the paper. We are then left with only one option that we can use to compare against, which is the simple linear interpolation method, where all imputed points lie on the straight line that connects the two end points P and Q.

Evaluation Metrics: To provide an objective evaluation of TrImpute, we use the following four evaluation metrics: (1) *Completion Rate*, which is the ratio of trajectory segments that TrImpute was able to successfully impute to all segments. It is important to note that the basic linear interpolation always has 100% completion rate, as it is just simply a straight line between the two end points. For TrImpute, whenever it fails to impute a certain segment, we report it as failure, and we just use liner interpolation for that segment, and proceed to the next one. (2) *Fréchet Accuracy*, which is the Fréchet distance [21] between each imputed segment and its ground truth obtained from raw trajectories. We then use a threshold distance (default 50m) where any Fréchet distance below that threshold is considered an accurate imputation. This is similar to the accuracy defined in [33], though the threshold distance had a default value of 125m. By using a smaller threshold in TrImpute, we are striving for a much higher accuracy. (3) *OSM Accuracy*, which indicates how the imputed trajectories match a ground truth road network, obtained from OpenStreetMap (OSM) [39]. We use the same threshold distance (default 50m) as Fréchet accuracy to indicate that an imputed point has successfully matched an actual road segment. (4) *Application Accuracy*, which shows how the imputed trajectories affect the performance of an application that rely on the accuracy of its input trajectories.

Experimental Design: We start our experiments by a sensitivity analysis (Section 6.1) to decide on the best values for TrImpute parameters, N, α, δ, and d. We then evaluate TrImpute against linear interpolation with respect to Fréchet accuracy (Section 6.2) and OSM accuracy (Section 6.3). Finally, we evaluate the impact of TrImpute vs linear interpolation, when they both used as a preprocessing step for the map inference algorithm [44] (Section 6.4).

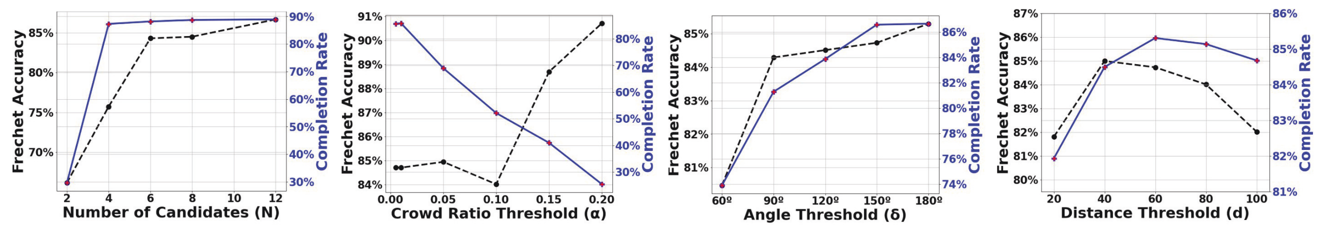
6.1 TrImpute Sensitivity Analysis

Figure 6 studies the impact of varying TrImpute main four parameters, N, α, δ, and d on Fréchet accuracy and trajectory completion rate, as follows:

Number of candidates points (N): Figure 6(a) depicts the impact of varying the number of candidate points from 2 to 12. Both accuracy and completion rate are getting better, and reaching up to 90% accuracy, with the increase of the number of candidate points. The main reason is that more candidate points give TrImpute more options to pick from, and hence increases the chance to find an accurate path. With this, we use 12 as out default number of candidate points. The reason we are not going for more than 12 is mainly due to the computational overhead that significantly increases with the increase of the number of candidates.

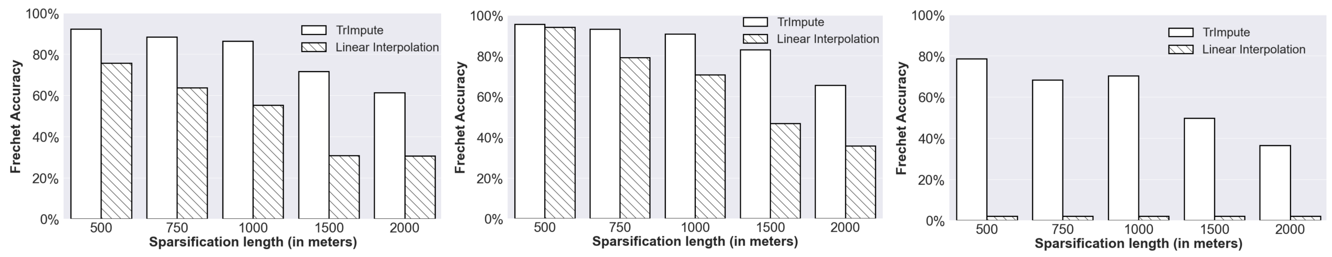
Crowd ratio threshold (α): Figure 6(b) depicts the impact of varying the crowd ratio threshold (α) from 0 to 0.2. Though the best accuracy is achieved with higher α, the completion ratio becomes the lowest. The main reason is that higher α would exclude several options that could help in imputing trajectory segments. Meanwhile, a crowd ratio of 0.01 would still give a very reasonable accuracy with 85%and 90+% of completion rate. So, we use this value as our default. It is important to note that though the value of α looks very small, it is still effective in excluding outlier paths, which significantly enhances the computational overhead.

Angle threshold (δ): Figure 6(c) depicts the impact of varying the angle threshold (δ) from 60° to 180°. Similar to the effect of the number of candidates, both accuracy and completion rate increases



(a) Number of Candidates Points (b) Crowd Ratio Threshold (c) Angle Threshold (d) Distance Threshold

Figure 6: TrImpute Sensitivity Analysis



(a) All segments (b) Straight segments (c) Curved segments

Figure 7: TrImpute vs Linear Interpolation for various Road Types and Sparsification Values.

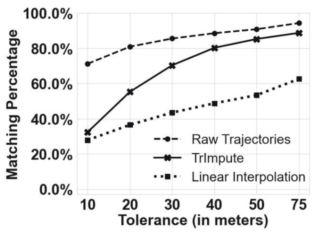
with δ, as more paths are considered. However, though not shown in the figure, this comes with a huge computational overhead as a value of 180° means exploiting all possible directions, which could be redundant. Hence, we settle on a default value of δ=120°, which still gives very high accuracy and completion rate that are very close to the case of 180°.

Distance threshold (d): Figure 6(d) depicts the impact of varying the distance threshold (d) from 20m to 100m. A small value of d would reduce both accuracy and completion rate as this would increase the chance of not being able to find good points within distance d from the current point, and hence we will not be able to continue our imputation process, and we would lean to linear interpolation then. Meanwhile, a large value of d would definitely reduce the accuracy as no enough points will be inserted between the source and destination points. Hence, and based on this experiment, we settle on a default value of d=40m, which achieves the peak accuracy point and very close to the highest completion rate.

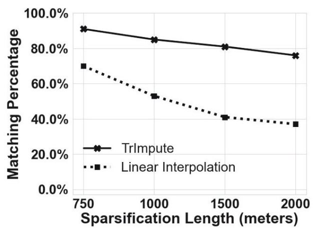
6.2 Trajectory Similarity

Figure 7(a) depicts the impact of varying the sparsification length from 500m to 2,000m on the Fréchet accuracy of both TrImpute and linear interpolation. As expected, the accuracy is reduced with the sparsification length, as it becomes much harder to impute the trajectories. Yet, there are two important things to note here: (a) In all sparsification values, TrImpute is significantly more accurate than linear interpolation, and (b) The performance gap between TrImpute and linear interpolation is significantly increasing with the increase in sparsification. This shows that TrImpute can sustain high sparsification values, while linear interpolation dramatically fail to 30% accuracy with 1,500 m sparsification.

Figures 7(b) and 7(c) provide a closer look (and justification) on the performance gap between TrImpute and linear interpolation, where the Fréchet accuracy is computed based on categorizing road segments into straight and curved ones, respectively, across different sparsification lengths. We use the original raw trajectory data to identify curved segments, where the angle of a set of five consecutive points has a difference more than a certain threshold. If the angle of these consecutive points is more or less the same, we consider that segment as straight. Otherwise, the segment is consider to be curved. Figure 7(b) compares the Fréchet accuracy for both TrImpute and linear interpolation when only considering straight segments. Though, TrImpute is still consistently better than linear interpolation for all sparsification values, but linear interpolation is not doing bad. In fact, linear interpolation still works well, especially for low sparsification values, with 90+% accuracy for 500m sparsification. This is expected as linear interpolation would be an acceptable solution for straight roads. The low performance of 30% accuracy of linear interpolation for 2,000m sparsification is due to lane changes in the road, where linear interpolation cannot capture this, while TrImpute would still be able to get it. This exhibits the robustness of TrImpute against high sparsification lengths. Figure 7(c) runs the same experiment exclusively on curved road segments. It is clear that linear interpolation dramatically fails with only 2% accuracy even for the lowest sparsification length of 500m. Meanwhile, TrImpute still keeps high accuracy, which intuitively gets lower with high sparsification. Overall, the experiments in Figure 7 show the power (and robustness) of TrImpute where it can accommodate high sparsification values and curved segments, while linear interpolation performance is not even acceptable in such settings.

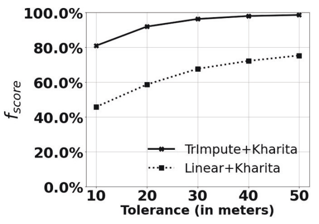


(a) Tolerance

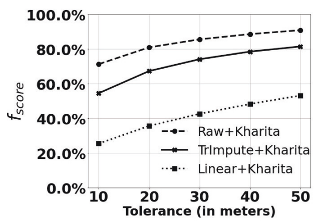


(b) Sparsification

Figure 8: OSM Accuracy



(a) Reference: Raw Trajectories



(b) Reference: OSM

Figure 9: Map Inference Application.

6.3 OSM Accuracy

Figure 8 evaluates OpenStreetMap (OSM) accuracy for both TrImpute and linear interpolation. The objective is to see how the imputed points get matched to the ground truth map, represented by OpenStreetMap [39]. In this case, the matching criteria is done by looking for the nearest segment from each GPS point. The more matching the better as it shows that the imputed points are true points. We assume that a point is well matched to the map if it is within a tolerance distance of 50m, which is way more strict than the 125m that was considered in [33]. Figure 8(a) gives the impact of varying the tolerance value from 10m to 75m. For reference, we also plot the OSM accuracy of our raw trajectory data, where apparently, even our raw data does not match 100% for any tolerance. This is expected due to the inherent noise with any such collected real data. Apparently, with more tolerance, both TrImpute and linear interpolation would do better. However, we can see that TrImpute consistently outperforms linear interpolation, and the gap is actually widening with higher tolerance values. With our default tolerance value of 50m, around 85% of the TrImpute imputed point match well with OSM, which is very close to the accuracy of our raw trajectory data. This shows that TrImpute was able to find out imputed points that are very likely close to what the raw data already had. This also shows that if our real data was free of noise (i.e., 100% accuracy), TrImpute would also have reached close to 100% accuracy as it follows the raw data given to it. Meanwhile, only 50% of the linear interpolation points would match even with a 50m, tolerance, which is considered as a very low accuracy. Figure 8(b) gives the impact of varying the sparsification distance from 750m to 2,000m. More sparsification would definitely lower the accuracy. However, we can see that the accuracy decrease in TrImpute is reasonable, compared to the linear interpolation trend. In particular, even with 1,500m sparsification, TrImpute is still able to get 80% accuracy, which is double the 40% accuracy of linear interpolation.

6.4 Map Inference Application

This section shows the impact of TrImpute over the map inference applications, as an example of trajectory-based applications. Map inference algorithms are concerned with inferring the underlying road network map from a set of GPS trajectories. Empowered by the availability of trajectory data, the inaccuracy of publicly available road networks (e.g., OSM) [38], and the immense need of having accurate underlying maps, map inference has been a rich area of study over the last few years, e.g., see [13] and [1] for a detailed survey and tutorial for current state-of-the-art map inference algorithms. All such algorithms are highly sensitive to their input trajectory data. If any of the algorithms is fed with sparse trajectories, it will result in an inaccurate map. Our objective here is to show the impact of applying TrImpute as a preprocessing step to map inference algorithms. TrImpute would be able to densify sparse trajectories in a way that will make map inference algorithms way more accurate than working directly with sparse trajectories.

Figure 9 gives the impact of TrImpute on one of the state-of-the-art map inference algorithms, Kharita [44]. We run Kharita three times with three different inputs: (a) *Raw+Kharita*, raw trajectory data, (b) *Linear+Kharita*, sparse data imputed using linear interpolation, and (c) *TrImpute+Kharita*, sparse data imputed by TrImpute. To quantify the output map quality of each of these runs, we use the F_1-score measure over all road network segments with respect to some reference road network. We use a match distance threshold (tolerance), where segments that are far from their original ones within this threshold will be considered correct.

Figure 9(a) shows the impact of varying the tolerance from 10m to 50m on the F_1-score of the map generated by TrImpute+Kharita and Linear+Khalita, when the reference map is the one generated by *Raw+Khaita*. The result is actually impressive showing that with a tolerance distance of 50m, the result of running *TrImpute+Kharita* is 99% accurate compared to the result of *Raw+Kharita*. This means that TrImpute is able to empower map inference algorithms even if the input data is sparse. TrImpute will make such algorithms work with similar quality as if they have raw dense data. Even with a very strict low tolerance of 10m, TrImpute results in 80% of raw data. Meanwhile, *Linear+Kharita*, is not helping, where the accuracy falls below 50% for 10m tolerance, and its best is 75% for 50m threshold. This shows that map inference algorithms would not really function if their input is sparse, and at their best are linearly interpolated. Figure 9(b) gives the same experiment of Figure 9(a), yet, when considering that the OpenStreetMap (OSM) [39] is the reference map. With a tolerance of 50m, running Kharita over raw data gives 90% accuracy, while running Kharita over TrImpute data gives 80+% accuracy. In all cases, *TrImpute+Kharita* gives close accuracy to *Raw+Kharita*, while *Linear+Kharita* consistently gives unacceptable performance. This again confirms that TrImpute has the power to remove a major hurdle to all map inference algorithms. These algorithm can only work with dense trajectories. With TrImpute, these algorithms can now work with sparse trajectories, which is much easier to obtain than dense ones.

7 CONCLUSION

This paper presented TrImpute; a novel framework for trajectory imputation that has the ability to impute sparse trajectory data, *without* the knowledge of the underlying road network, and hence it is considered a *network-less* trajectory imputation framework. In lieu of the lack of knowledge of road network, TrImpute relies on the nearby crowd wisdom to guide its imputation process. Basically, for each point in the space, nearby points suggest a list of candidate points where any of them could possibly be the next imputed point. Then, TrImpute formalizes its spatial imputation process for each trajectory segment as to find the shortest set of consecutive candidate points between the end points of the trajectory segment. TrImpute follows up by doing temporal imputation, where the spatially imputed points are annotated by timestamp information that respect the traffic conditions of the trajectory segment end points. Extensive experimental results based on real data and deployment show that TrImpute is highly scalable, accurate, and significantly boost the performance of trajectory applications.

REFERENCES

[1] S. Abbar, M. Alizadeh, F. Bastani, S. Chawla, S. He, H. Balakrishnan, and S. Madden. The Science of Algorithmic Map Inference (Tutorial) https://sites.google.com/view/algorithmic-map-making/home. In *KDD*, London, UK, 2018.

[2] S. Abbar, R. Stanojevic, M. Musleh, M. M. Elshrif, and M. F. Mokbel. A Demonstration of QARTA: An ML-based System for Accurate Map Services. *PVLDB*, 14(12), 2021.

[3] S. Abbar, R. Stanojevic, S. Mustafa, and M. Mokbel. Traffic Routing in the Ever-Changing City of Doha. *CACM*, 64(4), 2021.

[4] P. Agarwal, K. Fox, K. Munagala, A. Nath, J. Pan, and E. Taylor. Subtrajectory clustering: Models and algorithms. In *PODS*, 2018.

[5] F. Bastani, S. He, S. Abbar, M. Alizadeh, H. Balakrishnan, S. Chawla, S. Madden, and D. J. DeWitt. RoadTracer: Automatic Extraction of Road Networks From Aerial Images. In *CVPR*, 2018.

[6] 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.

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

[8] C. Brunsdon. Path Estimation from GPS Tracks. In *Proceedings of the 9th International Conference on GeoComputation*, 2007.

[9] L. Cao and J. Krumm. From GPS Traces to a Routable Road Map. In *SIGSPATIAL*, 2009.

[10] X. Cao, G. Cong, and C. S. Jensen. Mining Significant Semantic Locations From GPS Data. *PVLDB*, 3(1), 2010.

[11] P. S. Castro, D. Zhang, and S. Li. Urban Traffic Modelling and Prediction Using Large Scale Taxi GPS Traces. In *PerCom*, 2012.

[12] E. W. Chambers, B. T. Fasy, Y. Wang, and C. Wenk. Map-Matching Using Shortest Paths. *TSAS*, 6(1), 2020.

[13] P. Chao, W. Hua, R. Mao, J. Xu, and X. Zhou. A Survey and Quantitative Study on Map Inference Algorithms From GPS Trajectories. *TKDE*, 34(1), 2022.

[14] C. Chen, S. Jiao, S. Zhang, W. Liu, L. Feng, and Y. Wang. TripImputor: Real-Time Imputing Taxi Trip Purpose Leveraging Multi-Sourced Urban Data. *IEEE Transactions on Intelligent Transportation Systems, TTIT*, 19(10), 2018.

[15] 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.

[16] G. Cheng, Y. Wang, S. Xu, H. Wang, S. Xiang, and C. Pan. Automatic Road Detection and Centerline Extraction via Cascaded End-to-End Convolutional Neural Network. *IEEE Trans on Geoscience and Remote Sensing*, 55(6), 2017.

[17] 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] R. S. de Sousa, A. Boukerche, and A. A. F. Loureiro. Vehicle Trajectory Similarity: Models, Methods, and Applications. *ACM Computing Surveys*, 53(5), 2020.

[20] X. Ding, L. Chen, Y. Gao, C. S. Jensen, and H. Bao. UlTraMan: A Unified Platform for Big Trajectory Data Management and Analytics. *PVLDB*, 11(7), 2018.

[21] T. Eiter and H. Mannila. Computing discrete fréchet distance. Technical report, Technische Universität Wien, 1994.

[22] T. Goren-Bar and J. Greenfeld. A Method for Constructing 3D Traveling Routes from GPS Navigation Data. In *SIGSPATIAL GeoStreaming Workshop*, 2012.

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

[24] C. Guo, B. Yang, J. Hu, and C. Jensen. Learning to Route with Sparse Trajectory Sets. In *ICDE*, 2018.

[25] 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.

[26] G. R. Hjaltason, H. Samet, and Y. J. Sussmann. Speeding up Bulk-Loading of Quadtrees. In *ACM GIS*, 1997.

[27] G. Hu, J. Shao, F. Liu, Y. Wang, and H. T. Shen. IF-Matching: Towards Accurate Map-Matching with Information Fusion. In *ICDE*, 2017.

[28] H. Huang, S. Wu, D. Cohen-Or, M. Gong, H. Zhang, G. Li, and B. Chen. L1-Medial Skeleton of Point Cloud. *ACM Transactions on Graphics*, 32(4), 2013.

[29] C. S. Jensen. Value Creation from Massive Data in Transportation? The Case of Vehicle Routing. *IEEE Data Engineering Bulletin*, 42(3), 2019.

[30] J. Lee, J. Han, and K. Whang. Trajectory clustering: a partition-and-group framework. In *SIGMOD*, 2007.

[31] 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. *IJGIS*, 35(4), 2021.

[32] R. Li, H. He, R. Wang, S. Ruan, Y. Sui, J. Bao, and Y. Zheng. TrajMesa: A Distributed NoSQL Storage Engine for Big Trajectory Data. In *ICDE*, 2020.

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

[34] Y. Li, R. Yu, C. Shahabi, and Y. Liu. Diffusion Convolutional Recurrent Neural Network: Data-Driven Traffic Forecasting. In *International Conference on Learning Representations, ICLR*, 2018.

[35] J. A. Long. Kinematic Interpolation of Movement Data. *IJGIS*, 30(5), 2016.

[36] 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.

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

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

[39] OpenStreetMap (OSM). https://www.openstreetmap.org/.

[40] S. A. Pedersen, B. Yang, and C. S. Jensen. Fast Stochastic Routing under Time-varying Uncertainty. *VLDB J.*, 29(4), 2020.

[41] E. Rappos, S. Robert, and P. Cudré-Mauroux. A Force-directed Approach for Offline GPS Trajectory Map Matching. In *SIGSPATIAL*, 2018.

[42] 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.

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

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

[45] T. Sun, Z. Di, P. Che, C. Liu, and Y. Wang. Leveraging Crowdsourced GPS Data for Road Extraction From Aerial Imagery. In *CVPR*, 2019.

[46] G. Technitis, W. Othman, K. Safi, and R. Weibel. From A to B, randomly: A Point-to-point Random Trajectory Generator for Animal Movement. *IJGIS*, 29(6), 2015.

[47] 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.

[48] L. Tran, M. Mun, M. Lim, J. Yamato, N. Huh, and C. Shahabi. DeepTRANS: A Deep Learning System for Public Bus Travel Time Estimation using Traffic Forecasting. *PVLDB*, 13(12), 2020.

[49] R. Vishen, M. C. Silaghi, and J. Denzinger. GPS Data Interpolation: Bezier Vs. Biarcs for Tracing Vehicle Trajectory. In *ICCSA*, 2015.

[50] S. Wang, Z. Bao, J. S. Culpepper, and G. Cong. A Survey on Trajectory Data Management, Analytics, and Learning. *ACM Computing Surveys*, 54(2), 2021.

[51] S. Wang, Z. Bao, J. S. Culpepper, T. Sellis, and X. Qin. Fast Large-Scale Trajectory Clustering. *PVLDB*, 13(1), 2019.

[52] H. Wei, Y. Wang, G. Forman, and Y. Zhu. Map Matching: Comparison of Approaches using Sparse and Noisy Data. In *SIGSPATIAL*, 2013.

[53] L.-Y. Wei, Y. Zheng, and W.-C. Peng. Constructing Popular Routes from Uncertain Trajectories. In *KDD*, 2012.

[54] D. Xie, F. Li, and J. M. Phillips. Distributed Trajectory Similarity Search. *PVLDB*, 10(11), 2017.

[55] A. Zhang, S. Song, J. Wang, and P. S. Yu. Time Series Data Cleaning: From Anomaly Detection to Anomaly Repairing. *PVLDB*, 10(10), 2017.

[56] K. Zheng, Y. Zheng, X. Xie, and X. Zhou. Reducing Uncertainty of Low-Sampling-Rate Trajectories. In *ICDE*, 2012.

[57] Y. Zheng, L. Capra, O. Wolfson, and H. Yang. Urban computing: Concepts, Methodologies, and Applications. *ACM Trans. on Intel. Sys. and Tech.*, 5(3), 2014.

[58] Y. Zheng, L. Zhang, X. Xie, and W. Ma. Mining Interesting Locations and Travel Sequences from GPS Trajectories. In *WWW*, 2009.